mpmath documentation

Release 0.20-git

Fredrik Johansson **mpmath contributors**

October 20, 2015

Contents

Mpmath is a Python library for arbitrary-precision floating-point arithmetic. For general information about mpmath, see the project website http://mpmath.org/

These documentation pages include general information as well as docstring listing with extensive use of examples that can be run in the interactive Python interpreter. For quick access to the docstrings of individual functions, use the index listing, or type `help(mpmath.function_name)` in the Python interactive prompt.

CHAPTER 1

Introduction

1.1 Setting up mpmath

1.1.1 Download and installation

Installer

The mpmath setup files can be downloaded from the mpmath download page or the Python Package Index. Download the source package (available as both .zip and .tar.gz), extract it, open the extracted directory, and run

```
python setup.py install
```

If you are using Windows, you can download the binary installer

```
mpmath-(version).win32.exe
```

from the mpmath website or the Python Package Index (PyPI). Run the installer and follow the instructions.

Using pip

Releases are registered on PyPI, so you can install latest release of the Mpmath with pip

```
pip install mpmath
```

or some specific version with

```
pip install mpmath==0.19
```

Using setuptools

If you have setuptools installed, you can download and install mpmath in one step by running:

```
easy_install mpmath
```

or

```
python -m easy_install mpmath
```

If you have an old version of mpmath installed already, you may have to pass `easy_install` the `-U` flag to force an upgrade.

Debian/Ubuntu

Debian and Ubuntu users can install mpmath with

```
sudo apt-get install python-mpmath
```

See debian and ubuntu package information; please verify that you are getting the latest version.

OpenSUSE

Mpmath is provided in the "Science" repository for all recent versions of openSUSE. To add this repository to the YAST software management tool, see http://en.opensuse.org/SDB:Add_package_repositories

Look up http://download.opensuse.org/repositories/science/ for a list of supported OpenSUSE versions and use http://download.opensuse.org/repositories/science/openSUSE_11.1/ (or accordingly for your OpenSUSE version) as the repository URL for YAST.

Current development version

See http://code.google.com/p/mpmath/source/checkout for instructions on how to check out the mpmath Subversion repository. The source code can also be browsed online from the Google Code page.

Checking that it works

After the setup has completed, you should be able to fire up the interactive Python interpreter and do the following:

```
>>> from mpmath import *
>>> mp.dps = 50
>>> print mpf(2) ** mpf('0.5')
1.4142135623730950488016887242096980785696718753769
>>> print 2*pi
6.2831853071795864769252867665590057683943387987502
```

Note: if you have are upgrading mpmath from an earlier version, you may have to manually uninstall the old version or remove the old files.

1.1.2 Using gmpy (optional)

By default, mpmath uses Python integers internally. If gmpy version 1.03 or later is installed on your system, mpmath will automatically detect it and transparently use gmpy integers intead. This makes mpmath much faster, especially at high precision (approximately above 100 digits).

To verify that mpmath uses gmpy, check the internal variable `BACKEND` is not equal to 'python':

```
>>> import mpmath.libmp
>>> mpmath.libmp.BACKEND
'gmpy'
```

The gmpy mode can be disabled by setting the MPMATH_NOGMPY environment variable. Note that the mode cannot be switched during runtime; mpmath must be re-imported for this change to take effect.

1.1.3 Running tests

It is recommended that you run mpmath's full set of unit tests to make sure everything works. The tests are located in the `tests` subdirectory of the main mpmath directory. They can be run in the interactive interpreter using the `runtests()` function:

```
import mpmath
mpmath.runtests()
```

Alternatively, they can be run from the `tests` directory via

```
python runtests.py
```

The tests should finish in about a minute. If you have psyco installed, the tests can also be run with

```
python runtests.py -psyco
```

which will cut the running time in half.

If any test fails, please send a detailed bug report to the mpmath issue tracker. The tests can also be run with py.test. This will sometimes generate more useful information in case of a failure.

To run the tests with support for gmpy disabled, use

```
python runtests.py -nogmpy
```

To enable extra diagnostics, use

```
python runtests.py -strict
```

1.1.4 Compiling the documentation

If you downloaded the source package, the text source for these documentation pages is included in the `doc` directory. The documentation can be compiled to pretty HTML using Sphinx. Go to the `doc` directory and run

```
python build.py
```

You can also test that all the interactive examples in the documentation work by running

```
python run_doctest.py
```

and by running the individual `.py` files in the mpmath source.

(The doctests may take several minutes.)

Finally, some additional demo scripts are available in the `demo` directory included in the source package.

1.1.5 Mpmath under Sage

Mpmath is a standard package in Sage, in version 4.1 or later of Sage. Mpmath is preinstalled a regular Python module, and can be imported as usual within Sage:

```
----------------------------------------------------------------------
| Sage Version 4.1, Release Date: 2009-07-09                         |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
sage: import mpmath
sage: mpmath.mp.dps = 50
sage: print mpmath.mpf(2) ** 0.5
1.4142135623730950488016887242096980785696718753769
```

The mpmath installation under Sage automatically use Sage integers for asymptotically fast arithmetic, so there is no need to install GMPY:

```
sage: mpmath.libmp.BACKEND
'sage'
```

In Sage, mpmath can alternatively be imported via the interface library `sage.libs.mpmath.all`. For example:

```
sage: import sage.libs.mpmath.all as mpmath
```

This module provides a few extra conversion functions, including `call()` which permits calling any mpmath function with Sage numbers as input, and getting Sage `RealNumber` or `ComplexNumber` instances with the appropriate precision back:

```
sage: w = mpmath.call(mpmath.erf, 2+3*I, prec=100)
sage: w
-20.829461427614568389103088452 + 8.6873182714701631444280787545*I
sage: type(w)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: w.prec()
100
```

See the help for `sage.libs.mpmath.all` for further information.

1.2 Basic usage

In interactive code examples that follow, it will be assumed that all items in the `mpmath` namespace have been imported:

```
>>> from mpmath import *
```

Importing everything can be convenient, especially when using mpmath interactively, but be careful when mixing mpmath with other libraries! To avoid inadvertently overriding other functions or objects, explicitly import only the needed objects, or use the `mpmath.` or `mp.` namespaces:

```
from mpmath import sin, cos
sin(1), cos(1)

import mpmath
mpmath.sin(1), mpmath.cos(1)

from mpmath import mp     # mp context object -- to be explained
mp.sin(1), mp.cos(1)
```

1.2.1 Number types

Mpmath provides the following numerical types:

Class	Description
`mpf`	Real float
`mpc`	Complex float
`matrix`	Matrix

The following section will provide a very short introduction to the types `mpf` and `mpc`. Intervals and matrices are described further in the documentation chapters on interval arithmetic and matrices / linear algebra.

The mpf type is analogous to Python's built-in float. It holds a real number or one of the special values inf (positive infinity), -inf (negative infinity) and nan (not-a-number, indicating an indeterminate result). You can create mpf instances from strings, integers, floats, and other mpf instances:

```
>>> mpf(4)
mpf('4.0')
>>> mpf(2.5)
mpf('2.5')
>>> mpf("1.25e6")
mpf('1250000.0')
>>> mpf(mpf(2))
mpf('2.0')
>>> mpf("inf")
mpf('+inf')
```

The mpc type represents a complex number in rectangular form as a pair of mpf instances. It can be constructed from a Python complex, a real number, or a pair of real numbers:

```
>>> mpc(2,3)
mpc(real='2.0', imag='3.0')
>>> mpc(complex(2,3)).imag
mpf('3.0')
```

You can mix mpf and mpc instances with each other and with Python numbers:

```
>>> mpf(3) + 2*mpf('2.5') + 1.0
mpf('9.0')
>>> mp.dps = 15       # Set precision (see below)
>>> mpc(1j)**0.5
mpc(real='0.70710678118654757', imag='0.70710678118654757')
```

1.2.2 Setting the precision

Mpmath uses a global working precision; it does not keep track of the precision or accuracy of individual numbers. Performing an arithmetic operation or calling mpf() rounds the result to the current working precision. The working precision is controlled by a context object called mp, which has the following default state:

```
>>> print mp
Mpmath settings:
  mp.prec = 53                [default: 53]
  mp.dps = 15                 [default: 15]
  mp.trap_complex = False     [default: False]
```

The term **prec** denotes the binary precision (measured in bits) while **dps** (short for *decimal places*) is the decimal precision. Binary and decimal precision are related roughly according to the formula prec = 3.33*dps. For example, it takes a precision of roughly 333 bits to hold an approximation of pi that is accurate to 100 decimal places (actually slightly more than 333 bits is used).

Changing either precision property of the mp object automatically updates the other; usually you just want to change the dps value:

```
>>> mp.dps = 100
>>> mp.dps
100
>>> mp.prec
336
```

When the precision has been set, all mpf operations are carried out at that precision:

```
>>> mp.dps = 50
>>> mpf(1) / 6
mpf('0.16666666666666666666666666666666666666666666666666666656')
>>> mp.dps = 25
>>> mpf(2) ** mpf('0.5')
mpf('1.4142135623730950488801688713')
```

The precision of complex arithmetic is also controlled by the `mp` object:

```
>>> mp.dps = 10
>>> mpc(1,2) / 3
mpc(real='0.3333333333321', imag='0.6666666666642')
```

There is no restriction on the magnitude of numbers. An `mpf` can for example hold an approximation of a large Mersenne prime:

```
>>> mp.dps = 15
>>> print mpf(2)**32582657 - 1
1.24575026015369e+9808357
```

Or why not 1 googolplex:

```
>>> print mpf(10) ** (10**100)
1.0e+100000000000000000000000000000000000000000000000000...
```

The (binary) exponent is stored exactly and is independent of the precision.

Temporarily changing the precision

It is often useful to change the precision during only part of a calculation. A way to temporarily increase the precision and then restore it is as follows:

```
>>> mp.prec += 2
>>> # do_something()
>>> mp.prec -= 2
```

Since Python 2.5, the `with` statement along with the mpmath functions `workprec`, `workdps`, `extraprec` and `extradps` can be used to temporarily change precision in a more safe manner:

```
>>> from __future__ import with_statement  # only need this in Python 2.5
>>> with workdps(20):
...     print mpf(1)/7
...     with extradps(10):
...         print mpf(1)/7
...
0.14285714285714285714
0.14285714285714285714285714285714
>>> mp.dps
15
```

The `with` statement ensures that the precision gets reset when exiting the block, even in the case that an exception is raised. (The effect of the `with` statement can be emulated in Python 2.4 by using a `try/finally` block.)

The `workprec` family of functions can also be used as function decorators:

```
>>> @workdps(6)
... def f():
...     return mpf(1)/3
...
```

```
>>> f()
mpf('0.33333331346511841')
```

Some functions accept the `prec` and `dps` keyword arguments and this will override the global working precision. Note that this will not affect the precision at which the result is printed, so to get all digits, you must either use increase precision afterward when printing or use `nstr`/`nprint`:

```
>>> mp.dps = 15
>>> print exp(1)
2.71828182845905
>>> print exp(1, dps=50)       # Extra digits won't be printed
2.71828182845905
>>> nprint(exp(1, dps=50), 50)
2.7182818284590452353602874713526624977572470937
```

Finally, instead of using the global context object `mp`, you can create custom contexts and work with methods of those instances instead of global functions. The working precision will be local to each context object:

```
>>> mp2 = mp.clone()
>>> mp.dps = 10
>>> mp2.dps = 20
>>> print mp.mpf(1) / 3
0.3333333333
>>> print mp2.mpf(1) / 3
0.33333333333333333333
```

Note: the ability to create multiple contexts is a new feature that is only partially implemented. Not all mpmath functions are yet available as context-local methods. In the present version, you are likely to encounter bugs if you try mixing different contexts.

1.2.3 Providing correct input

Note that when creating a new `mpf`, the value will at most be as accurate as the input. *Be careful when mixing mpmath numbers with Python floats*. When working at high precision, fractional `mpf` values should be created from strings or integers:

```
>>> mp.dps = 30
>>> mpf(10.9)   # bad
mpf('10.900000000000000355271367880050')
>>> mpf('10.9')  # good
mpf('10.899999999999999999999999999997')
>>> mpf(109) / mpf(10)   # also good
mpf('10.899999999999999999999999999997')
>>> mp.dps = 15
```

(Binary fractions such as 0.5, 1.5, 0.75, 0.125, etc, are generally safe as input, however, since those can be represented exactly by Python floats.)

1.2.4 Printing

By default, the `repr()` of a number includes its type signature. This way `eval` can be used to recreate a number from its string representation:

```
>>> eval(repr(mpf(2.5)))
mpf('2.5')
```

Prettier output can be obtained by using `str()` or `print`, which hide the `mpf` and `mpc` signatures and also suppress rounding artifacts in the last few digits:

```
>>> mpf("3.14159")
mpf('3.1415899999999999')
>>> print mpf("3.14159")
3.14159
>>> print mpc(1j)**0.5
(0.707106781186548 + 0.707106781186548j)
```

Setting the `mp.pretty` option will use the `str()`-style output for `repr()` as well:

```
>>> mp.pretty = True
>>> mpf(0.6)
0.6
>>> mp.pretty = False
>>> mpf(0.6)
mpf('0.59999999999999998')
```

The number of digits with which numbers are printed by default is determined by the working precision. To specify the number of digits to show without changing the working precision, use *mpmath.nstr()* and *mpmath.nprint()*:

```
>>> a = mpf(1) / 6
>>> a
mpf('0.16666666666666666')
>>> nstr(a, 8)
'0.16666667'
>>> nprint(a, 8)
0.16666667
>>> nstr(a, 50)
'0.16666666666666665741480812812369549646973609924316'
```

CHAPTER 2

Basic features

2.1 Contexts

High-level code in mpmath is implemented as methods on a "context object". The context implements arithmetic, type conversions and other fundamental operations. The context also holds settings such as precision, and stores cache data. A few different contexts (with a mostly compatible interface) are provided so that the high-level algorithms can be used with different implementations of the underlying arithmetic, allowing different features and speed-accuracy tradeoffs. Currently, mpmath provides the following contexts:

- Arbitrary-precision arithmetic (`mp`)
- A faster Cython-based version of `mp` (used by default in Sage, and currently only available there)
- Arbitrary-precision interval arithmetic (`iv`)
- Double-precision arithmetic using Python's builtin `float` and `complex` types (`fp`)

Most global functions in the global mpmath namespace are actually methods of the `mp` context. This fact is usually transparent to the user, but sometimes shows up in the form of an initial parameter called "ctx" visible in the help for the function:

```
>>> import mpmath
>>> help(mpmath.fsum)
Help on method fsum in module mpmath.ctx_mp_python:

fsum(ctx, terms, absolute=False, squared=False) method of mpmath.ctx_mp.MPContext instance
    Calculates a sum containing a finite number of terms (for infinite
    series, see :func:`~mpmath.nsum`). The terms will be converted to
...
```

The following operations are equivalent:

```
>>> mpmath.mp.dps = 15; mpmath.mp.pretty = False
>>> mpmath.fsum([1,2,3])
mpf('6.0')
>>> mpmath.mp.fsum([1,2,3])
mpf('6.0')
```

The corresponding operation using the `fp` context:

```
>>> mpmath.fp.fsum([1,2,3])
6.0
```

2.1.1 Common interface

`ctx.mpf` creates a real number:

```
>>> from mpmath import mp, fp
>>> mp.mpf(3)
mpf('3.0')
>>> fp.mpf(3)
3.0
```

`ctx.mpc` creates a complex number:

```
>>> mp.mpc(2,3)
mpc(real='2.0', imag='3.0')
>>> fp.mpc(2,3)
(2+3j)
```

`ctx.matrix` creates a matrix:

```
>>> mp.matrix([[1,0],[0,1]])
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
>>> _[0,0]
mpf('1.0')
>>> fp.matrix([[1,0],[0,1]])
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
>>> _[0,0]
1.0
```

`ctx.prec` holds the current precision (in bits):

```
>>> mp.prec
53
>>> fp.prec
53
```

`ctx.dps` holds the current precision (in digits):

```
>>> mp.dps
15
>>> fp.dps
15
```

`ctx.pretty` controls whether objects should be pretty-printed automatically by `repr()`. Pretty-printing for `mp` numbers is disabled by default so that they can clearly be distinguished from Python numbers and so that `eval(repr(x)) == x` works:

```
>>> mp.mpf(3)
mpf('3.0')
>>> mpf = mp.mpf
>>> eval(repr(mp.mpf(3)))
mpf('3.0')
>>> mp.pretty = True
>>> mp.mpf(3)
3.0
>>> fp.matrix([[1,0],[0,1]])
matrix(
[['1.0', '0.0'],
```

```
 ['0.0', '1.0']])
>>> fp.pretty = True
>>> fp.matrix([[1,0],[0,1]])
[1.0  0.0]
[0.0  1.0]
>>> fp.pretty = False
>>> mp.pretty = False
```

2.1.2 Arbitrary-precision floating-point (mp)

The mp context is what most users probably want to use most of the time, as it supports the most functions, is most well-tested, and is implemented with a high level of optimization. Nearly all examples in this documentation use mp functions.

See Basic usage for a description of basic usage.

2.1.3 Arbitrary-precision interval arithmetic (iv)

The iv.mpf type represents a closed interval $[a, b]$; that is, the set $\{x : a \le x \le b\}$, where a and b are arbitrary-precision floating-point values, possibly $\pm\infty$. The iv.mpc type represents a rectangular complex interval $[a, b] + [c, d]i$; that is, the set $\{z = x + iy : a \le x \le b \wedge c \le y \le d\}$.

Interval arithmetic provides rigorous error tracking. If f is a mathematical function and $\hat{f}$ is its interval arithmetic version, then the basic guarantee of interval arithmetic is that $f(v) \subseteq \hat{f}(v)$ for any input interval v. Put differently, if an interval represents the known uncertainty for a fixed number, any sequence of interval operations will produce an interval that contains what would be the result of applying the same sequence of operations to the exact number. The principal drawbacks of interval arithmetic are speed (iv arithmetic is typically at least two times slower than mp arithmetic) and that it sometimes provides far too pessimistic bounds.

Note: The support for interval arithmetic in mpmath is still experimental, and many functions do not yet properly support intervals. Please use this feature with caution.

Intervals can be created from single numbers (treated as zero-width intervals) or pairs of endpoint numbers. Strings are treated as exact decimal numbers. Note that a Python float like 0.1 generally does not represent the same number as its literal; use '0.1' instead:

```
>>> from mpmath import iv
>>> iv.dps = 15; iv.pretty = False
>>> iv.mpf(3)
mpi('3.0', '3.0')
>>> print iv.mpf(3)
[3.0, 3.0]
>>> iv.pretty = True
>>> iv.mpf([2,3])
[2.0, 3.0]
>>> iv.mpf(0.1)    # probably not intended
[0.10000000000000000555, 0.10000000000000000555]
>>> iv.mpf('0.1')    # good, gives a containing interval
[0.09999999999999991673, 0.10000000000000000555]
>>> iv.mpf(['0.1', '0.2'])
[0.09999999999999991673, 0.20000000000000000111]
```

The fact that '0.1' results in an interval of nonzero width indicates that 1/10 cannot be represented using binary floating-point numbers at this precision level (in fact, it cannot be represented exactly at any precision).

Intervals may be infinite or half-infinite:

```
>>> print 1 / iv.mpf([2, 'inf'])
[0.0, 0.5]
```

The equality testing operators == and != check whether their operands are identical as intervals; that is, have the same endpoints. The ordering operators < <= > >= permit inequality testing using triple-valued logic: a guaranteed inequality returns True or False while an indeterminate inequality returns None:

```
>>> iv.mpf([1,2]) == iv.mpf([1,2])
True
>>> iv.mpf([1,2]) != iv.mpf([1,2])
False
>>> iv.mpf([1,2]) <= 2
True
>>> iv.mpf([1,2]) > 0
True
>>> iv.mpf([1,2]) < 1
False
>>> iv.mpf([1,2]) < 2    # returns None
>>> iv.mpf([2,2]) < 2
False
>>> iv.mpf([1,2]) <= iv.mpf([2,3])
True
>>> iv.mpf([1,2]) < iv.mpf([2,3])  # returns None
>>> iv.mpf([1,2]) < iv.mpf([-1,0])
False
```

The in operator tests whether a number or interval is contained in another interval:

```
>>> iv.mpf([0,2]) in iv.mpf([0,10])
True
>>> 3 in iv.mpf(['-inf', 0])
False
```

Intervals have the properties .a, .b (endpoints), .mid, and .delta (width):

```
>>> x = iv.mpf([2, 5])
>>> x.a
[2.0, 2.0]
>>> x.b
[5.0, 5.0]
>>> x.mid
[3.5, 3.5]
>>> x.delta
[3.0, 3.0]
```

Some transcendental functions are supported:

```
>>> iv.dps = 15
>>> mp.dps = 15
>>> iv.mpf([0.5,1.5]) ** iv.mpf([0.5, 1.5])
[0.35355339059327373086, 1.837117307087383633]
>>> iv.exp(0)
[1.0, 1.0]
>>> iv.exp(['-inf','inf'])
[0.0, +inf]
>>>
>>> iv.exp(['-inf',0])
[0.0, 1.0]
```

```
>>> iv.exp([0,'inf'])
[1.0, +inf]
>>> iv.exp([0,1])
[1.0, 2.7182818284590455349]
>>>
>>> iv.log(1)
[0.0, 0.0]
>>> iv.log([0,1])
[-inf, 0.0]
>>> iv.log([0,'inf'])
[-inf, +inf]
>>> iv.log(2)
[0.69314718055994528623, 0.69314718055994539725]
>>>
>>> iv.sin([100,'inf'])
[-1.0, 1.0]
>>> iv.cos(['-0.1','0.1'])
[0.99500416527802570954, 1.0]
```

Interval arithmetic is useful for proving inequalities involving irrational numbers. Naive use of `mp` arithmetic may result in wrong conclusions, such as the following:

```
>>> mp.dps = 25
>>> x = mp.exp(mp.pi*mp.sqrt(163))
>>> y = mp.mpf(640320**3+744)
>>> print x
262537412640768744.0000001
>>> print y
262537412640768744.0
>>> x > y
True
```

But the correct result is $e^{\pi\sqrt{163}} < 262537412640768744$, as can be seen by increasing the precision:

```
>>> mp.dps = 50
>>> print mp.exp(mp.pi*mp.sqrt(163))
262537412640768743.99999999999925007259719818568888
```

With interval arithmetic, the comparison returns `None` until the precision is large enough for $x - y$ to have a definite sign:

```
>>> iv.dps = 15
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
>>> iv.dps = 30
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
>>> iv.dps = 60
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
False
>>> iv.dps = 15
```

2.1.4 Fast low-precision arithmetic (`fp`)

Although mpmath is generally designed for arbitrary-precision arithmetic, many of the high-level algorithms work perfectly well with ordinary Python `float` and `complex` numbers, which use hardware double precision (on most systems, this corresponds to 53 bits of precision). Whereas the global functions (which are methods of the `mp` object) always convert inputs to mpmath numbers, the `fp` object instead converts them to `float` or `complex`, and in some cases employs basic functions optimized for double precision. When large amounts of function evaluations (numerical

integration, plotting, etc) are required, and when `fp` arithmetic provides sufficient accuracy, this can give a significant speedup over `mp` arithmetic.

To take advantage of this feature, simply use the `fp` prefix, i.e. write `fp.func` instead of `func` or `mp.func`:

```
>>> u = fp.erfc(2.5)
>>> print u
0.000406952017445
>>> type(u)
<type 'float'>
>>> mp.dps = 15
>>> print mp.erfc(2.5)
0.000406952017444959
>>> fp.matrix([[1,2],[3,4]]) ** 2
matrix(
[['7.0', '10.0'],
 ['15.0', '22.0']])
>>>
>>> type(_[0,0])
<type 'float'>
>>> print fp.quad(fp.sin, [0, fp.pi])    # numerical integration
2.0
```

The `fp` context wraps Python's `math` and `cmath` modules for elementary functions. It supports both real and complex numbers and automatically generates complex results for real inputs (`math` raises an exception):

```
>>> fp.sqrt(5)
2.23606797749979
>>> fp.sqrt(-5)
2.23606797749979j
>>> fp.sin(10)
-0.5440211108893698
>>> fp.power(-1, 0.25)
(0.7071067811865476+0.7071067811865475j)
>>> (-1) ** 0.25
Traceback (most recent call last):
  ...
ValueError: negative number cannot be raised to a fractional power
```

The `prec` and `dps` attributes can be changed (for interface compatibility with the `mp` context) but this has no effect:

```
>>> fp.prec
53
>>> fp.dps
15
>>> fp.prec = 80
>>> fp.prec
53
>>> fp.dps
15
```

Due to intermediate rounding and cancellation errors, results computed with `fp` arithmetic may be much less accurate than those computed with `mp` using an equivalent precision (`mp.prec = 53`), since the latter often uses increased internal precision. The accuracy is highly problem-dependent: for some functions, `fp` almost always gives 14-15 correct digits; for others, results can be accurate to only 2-3 digits or even completely wrong. The recommended use for `fp` is therefore to speed up large-scale computations where accuracy can be verified in advance on a subset of the input set, or where results can be verified afterwards.

2.2 Utility functions

This page lists functions that perform basic operations on numbers or aid general programming.

2.2.1 Conversion and printing

mpmathify() / convert()

mpmath.**mpmathify**(*x*, *strings=True*)

Converts *x* to an `mpf` or `mpc`. If *x* is of type `mpf`, `mpc`, `int`, `float`, `complex`, the conversion will be performed losslessly.

If *x* is a string, the result will be rounded to the present working precision. Strings representing fractions or complex numbers are permitted.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> mpmathify(3.5)
mpf('3.5')
>>> mpmathify('2.1')
mpf('2.1000000000000001')
>>> mpmathify('3/4')
mpf('0.75')
>>> mpmathify('2+3j')
mpc(real='2.0', imag='3.0')
```

nstr()

mpmath.**nstr**(*x*, *n=6*, ***kwargs*)

Convert an `mpf` or `mpc` to a decimal string literal with *n* significant digits. The small default value for *n* is chosen to make this function useful for printing collections of numbers (lists, matrices, etc).

If *x* is a list or tuple, *nstr()* is applied recursively to each element. For unrecognized classes, *nstr()* simply returns `str(x)`.

The companion function *nprint()* prints the result instead of returning it.

The keyword arguments *strip_zeros*, *min_fixed*, *max_fixed* and *show_zero_exponent* are forwarded to `to_str()`.

The number will be printed in fixed-point format if the position of the leading digit is strictly between min_fixed (default = min(-dps/3,-5)) and max_fixed (default = dps).

To force fixed-point format always, set min_fixed = -inf, max_fixed = +inf. To force floating-point format, set min_fixed >= max_fixed.

```
>>> from mpmath import *
>>> nstr([+pi, ldexp(1,-500)])
'[3.14159, 3.05494e-151]'
>>> nprint([+pi, ldexp(1,-500)])
[3.14159, 3.05494e-151]
>>> nstr(mpf("5e-10"), 5)
'5.0e-10'
>>> nstr(mpf("5e-10"), 5, strip_zeros=False)
'5.0000e-10'
>>> nstr(mpf("5e-10"), 5, strip_zeros=False, min_fixed=-11)
```

```
    '0.00000000050000'
    >>> nstr(mpf(0), 5, show_zero_exponent=True)
    '0.0e+0'
```

nprint()

mpmath.**nprint**(*x*, *n=6*, ***kwargs*)
Equivalent to `print(nstr(x, n))`.

2.2.2 Arithmetic operations

See also *mpmath.sqrt()*, *mpmath.exp()* etc., listed in Powers and logarithms

fadd()

mpmath.**fadd**(*ctx*, *x*, *y*, ***kwargs*)
Adds the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

The default precision is the working precision of the context. You can specify a custom precision in bits by passing the *prec* keyword argument, or by providing an equivalent decimal precision with the *dps* keyword argument. If the precision is set to `+inf`, or if the flag *exact=True* is passed, an exact addition with no rounding is performed.

When the precision is finite, the optional *rounding* keyword argument specifies the direction of rounding. Valid options are `'n'` for nearest (default), `'f'` for floor, `'c'` for ceiling, `'d'` for down, `'u'` for up.

Examples

Using *fadd()* with precision and rounding control:

```
    >>> from mpmath import *
    >>> mp.dps = 15; mp.pretty = False
    >>> fadd(2, 1e-20)
    mpf('2.0')
    >>> fadd(2, 1e-20, rounding='u')
    mpf('2.0000000000000004')
    >>> nprint(fadd(2, 1e-20, prec=100), 25)
    2.00000000000000000001
    >>> nprint(fadd(2, 1e-20, dps=15), 25)
    2.0
    >>> nprint(fadd(2, 1e-20, dps=25), 25)
    2.00000000000000000001
    >>> nprint(fadd(2, 1e-20, exact=True), 25)
    2.00000000000000000001
```

Exact addition avoids cancellation errors, enforcing familiar laws of numbers such as $x + y - x = y$, which don't hold in floating-point arithmetic with finite precision:

```
    >>> x, y = mpf(2), mpf('1e-1000')
    >>> print(x + y - x)
    0.0
    >>> print(fadd(x, y, prec=inf) - x)
    1.0e-1000
    >>> print(fadd(x, y, exact=True) - x)
    1.0e-1000
```

Exact addition can be inefficient and may be impossible to perform with large magnitude differences:

```
>>> fadd(1, '1e-100000000000000000000', prec=inf)
Traceback (most recent call last):
  ...
OverflowError: the exact result does not fit in memory
```

fsub()

mpmath.**fsub**(*ctx*, *x*, *y*, ***kwargs*)

Subtracts the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of *fadd()* for a detailed description of how to specify precision and rounding.

Examples

Using *fsub()* with precision and rounding control:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fsub(2, 1e-20)
mpf('2.0')
>>> fsub(2, 1e-20, rounding='d')
mpf('1.9999999999999998')
>>> nprint(fsub(2, 1e-20, prec=100), 25)
1.9999999999999999999
>>> nprint(fsub(2, 1e-20, dps=15), 25)
2.0
>>> nprint(fsub(2, 1e-20, dps=25), 25)
1.9999999999999999999
>>> nprint(fsub(2, 1e-20, exact=True), 25)
1.9999999999999999999
```

Exact subtraction avoids cancellation errors, enforcing familiar laws of numbers such as $x - y + y = x$, which don't hold in floating-point arithmetic with finite precision:

```
>>> x, y = mpf(2), mpf('1e1000')
>>> print(x - y + y)
0.0
>>> print(fsub(x, y, prec=inf) + y)
2.0
>>> print(fsub(x, y, exact=True) + y)
2.0
```

Exact addition can be inefficient and may be impossible to perform with large magnitude differences:

```
>>> fsub(1, '1e-100000000000000000000', prec=inf)
Traceback (most recent call last):
  ...
OverflowError: the exact result does not fit in memory
```

fneg()

mpmath.**fneg**(*ctx*, *x*, ***kwargs*)

Negates the number *x*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of *fadd()* for a detailed description of how to specify precision and rounding.

Examples

An mpmath number is returned:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fneg(2.5)
mpf('-2.5')
>>> fneg(-5+2j)
mpc(real='5.0', imag='-2.0')
```

Precise control over rounding is possible:

```
>>> x = fadd(2, 1e-100, exact=True)
>>> fneg(x)
mpf('-2.0')
>>> fneg(x, rounding='f')
mpf('-2.0000000000000004')
```

Negating with and without roundoff:

```
>>> n = 200000000000000000000001
>>> print(int(-mpf(n)))
-200000000000000016777216
>>> print(int(fneg(n)))
-200000000000000016777216
>>> print(int(fneg(n, prec=log(n,2)+1)))
-200000000000000000000001
>>> print(int(fneg(n, dps=log(n,10)+1)))
-200000000000000000000001
>>> print(int(fneg(n, prec=inf)))
-200000000000000000000001
>>> print(int(fneg(n, dps=inf)))
-200000000000000000000001
>>> print(int(fneg(n, exact=True)))
-200000000000000000000001
```

fmul()

mpmath.**fmul**(*ctx*, *x*, *y*, ***kwargs*)

Multiplies the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of *fadd()* for a detailed description of how to specify precision and rounding.

Examples

The result is an mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fmul(2, 5.0)
mpf('10.0')
>>> fmul(0.5j, 0.5)
mpc(real='0.0', imag='0.25')
```

Avoiding roundoff:

```
>>> x, y = 10**10+1, 10**15+1
>>> print(x*y)
```

```
100000000010000010000000001
>>> print(mpf(x) * mpf(y))
1.0000000001e+25
>>> print(int(mpf(x) * mpf(y)))
100000000010000011026399232
>>> print(int(fmul(x, y)))
100000000010000011026399232
>>> print(int(fmul(x, y, dps=25)))
100000000010000010000000001
>>> print(int(fmul(x, y, exact=True)))
100000000010000010000000001
```

Exact multiplication with complex numbers can be inefficient and may be impossible to perform with large magnitude differences between real and imaginary parts:

```
>>> x = 1+2j
>>> y = mpc(2, '1e-10000000000000000000')
>>> fmul(x, y)
mpc(real='2.0', imag='4.0')
>>> fmul(x, y, rounding='u')
mpc(real='2.0', imag='4.0000000000000009')
>>> fmul(x, y, exact=True)
Traceback (most recent call last):
  ...
OverflowError: the exact result does not fit in memory
```

fdiv()

mpmath.**fdiv**(*ctx*, *x*, *y*, ***kwargs*)

Divides the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of *fadd()* for a detailed description of how to specify precision and rounding.

Examples

The result is an mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fdiv(3, 2)
mpf('1.5')
>>> fdiv(2, 3)
mpf('0.66666666666666663')
>>> fdiv(2+4j, 0.5)
mpc(real='4.0', imag='8.0')
```

The rounding direction and precision can be controlled:

```
>>> fdiv(2, 3, dps=3)    # Should be accurate to at least 3 digits
mpf('0.6666259765625')
>>> fdiv(2, 3, rounding='d')
mpf('0.66666666666666663')
>>> fdiv(2, 3, prec=60)
mpf('0.66666666666666667')
>>> fdiv(2, 3, rounding='u')
mpf('0.66666666666666674')
```

Checking the error of a division by performing it at higher precision:

```
>>> fdiv(2, 3) - fdiv(2, 3, prec=100)
mpf('-3.7007434154172148e-17')
```

Unlike *fadd()*, *fmul()*, etc., exact division is not allowed since the quotient of two floating-point numbers generally does not have an exact floating-point representation. (In the future this might be changed to allow the case where the division is actually exact.)

```
>>> fdiv(2, 3, exact=True)
Traceback (most recent call last):
  ...
ValueError: division is not an exact operation
```

fmod()

mpmath.**fmod**(*x*, *y*)

Converts x and y to mpmath numbers and returns $x \mod y$. For mpmath numbers, this is equivalent to `x % y`.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> fmod(100, pi)
2.61062773871641
```

You can use *fmod()* to compute fractional parts of numbers:

```
>>> fmod(10.25, 1)
0.25
```

fsum()

mpmath.**fsum**(*terms*, *absolute=False*, *squared=False*)

Calculates a sum containing a finite number of terms (for infinite series, see *nsum()*). The terms will be converted to mpmath numbers. For len(terms) > 2, this function is generally faster and produces more accurate results than the builtin Python function `sum()`.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fsum([1, 2, 0.5, 7])
mpf('10.5')
```

With squared=True each term is squared, and with absolute=True the absolute value of each term is used.

fprod()

mpmath.**fprod**(*factors*)

Calculates a product containing a finite number of factors (for infinite products, see *nprod()*). The factors will be converted to mpmath numbers.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fprod([1, 2, 0.5, 7])
mpf('7.0')
```

fdot()

mpmath.**fdot**(*A*, *B=None*, *conjugate=False*)

Computes the dot product of the iterables A and B,

$$\sum_{k=0} A_k B_k.$$

Alternatively, *fdot()* accepts a single iterable of pairs. In other words, `fdot(A,B)` and `fdot(zip(A,B))` are equivalent. The elements are automatically converted to mpmath numbers.

With `conjugate=True`, the elements in the second vector will be conjugated:

$$\sum_{k=0} A_k \overline{B_k}$$

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> A = [2, 1.5, 3]
>>> B = [1, -1, 2]
>>> fdot(A, B)
mpf('6.5')
>>> list(zip(A, B))
[(2, 1), (1.5, -1), (3, 2)]
>>> fdot(_)
mpf('6.5')
>>> A = [2, 1.5, 3j]
>>> B = [1+j, 3, -1-j]
>>> fdot(A, B)
mpc(real='9.5', imag='-1.0')
>>> fdot(A, B, conjugate=True)
mpc(real='3.5', imag='-5.0')
```

2.2.3 Complex components

fabs()

mpmath.**fabs**(*x*)

Returns the absolute value of x, $|x|$. Unlike `abs()`, *fabs()* converts non-mpmath numbers (such as `int`) into mpmath numbers:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fabs(3)
mpf('3.0')
>>> fabs(-3)
mpf('3.0')
>>> fabs(3+4j)
mpf('5.0')
```

sign()

mpmath.**sign**(*x*)

Returns the sign of x, defined as $\operatorname{sign}(x) = x/|x|$ (with the special case $\operatorname{sign}(0) = 0$):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> sign(10)
mpf('1.0')
>>> sign(-10)
mpf('-1.0')
>>> sign(0)
mpf('0.0')
```

Note that the sign function is also defined for complex numbers, for which it gives the projection onto the unit circle:

```
>>> mp.dps = 15; mp.pretty = True
>>> sign(1+j)
(0.707106781186547 + 0.707106781186547j)
```

re()

mpmath.**re**(x)

Returns the real part of x, $\Re(x)$. Unlike `x.real`, *re()* converts x to a mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> re(3)
mpf('3.0')
>>> re(-1+4j)
mpf('-1.0')
```

im()

mpmath.**im**(x)

Returns the imaginary part of x, $\Im(x)$. Unlike `x.imag`, *im()* converts x to a mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> im(3)
mpf('0.0')
>>> im(-1+4j)
mpf('4.0')
```

arg()

mpmath.**arg**(x)

Computes the complex argument (phase) of x, defined as the signed angle between the positive real axis and x in the complex plane:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> arg(3)
0.0
>>> arg(3+3j)
0.785398163397448
>>> arg(3j)
1.5707963267949
```

```
>>> arg(-3)
3.14159265358979
>>> arg(-3j)
-1.5707963267949
```

The angle is defined to satisfy $-\pi < \arg(x) \leq \pi$ and with the sign convention that a nonnegative imaginary part results in a nonnegative argument.

The value returned by *arg()* is an `mpf` instance.

conj()

mpmath.**conj**(x)
Returns the complex conjugate of x, $\overline{x}$. Unlike `x.conjugate()`, *im()* converts x to a mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> conj(3)
mpf('3.0')
>>> conj(-1+4j)
mpc(real='-1.0', imag='-4.0')
```

polar()

mpmath.**polar**(x)
Returns the polar representation of the complex number z as a pair (r, ϕ) such that $z = re^{i\phi}$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> polar(-2)
(2.0, 3.14159265358979)
>>> polar(3-4j)
(5.0, -0.927295218001612)
```

rect()

mpmath.**rect**(x)
Returns the complex number represented by polar coordinates (r, ϕ):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> chop(rect(2, pi))
-2.0
>>> rect(sqrt(2), -pi/4)
(1.0 - 1.0j)
```

2.2.4 Integer and fractional parts

floor()

mpmath.**floor**(x)
Computes the floor of x, $\lfloor x \rfloor$, defined as the largest integer less than or equal to x:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> floor(3.5)
mpf('3.0')
```

Note: *floor()*, *ceil()* and *nint()* return a floating-point number, not a Python `int`. If $\lfloor x \rfloor$ is too large to be represented exactly at the present working precision, the result will be rounded, not necessarily in the direction implied by the mathematical definition of the function.

To avoid rounding, use *prec=0*:

```
>>> mp.dps = 15
>>> print(int(floor(10**30+1)))
1000000000000000019884624838656
>>> print(int(floor(10**30+1, prec=0)))
1000000000000000000000000000001
```

The floor function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> floor(3.25+4.75j)
mpc(real='3.0', imag='4.0')
```

ceil()

mpmath.**ceil**(x)

Computes the ceiling of x, $\lceil x \rceil$, defined as the smallest integer greater than or equal to x:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> ceil(3.5)
mpf('4.0')
```

The ceiling function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> ceil(3.25+4.75j)
mpc(real='4.0', imag='5.0')
```

See notes about rounding for *floor()*.

nint()

mpmath.**nint**(x)

Evaluates the nearest integer function, $\mathrm{nint}(x)$. This gives the nearest integer to x; on a tie, it gives the nearest even integer:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> nint(3.2)
mpf('3.0')
>>> nint(3.8)
mpf('4.0')
>>> nint(3.5)
mpf('4.0')
>>> nint(4.5)
mpf('4.0')
```

The nearest integer function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> nint(3.25+4.75j)
mpc(real='3.0', imag='5.0')
```

See notes about rounding for *floor()*.

frac()

mpmath.**frac**(*x*)

Gives the fractional part of x, defined as $\mathrm{frac}(x) = x - \lfloor x \rfloor$ (see *floor()*). In effect, this computes x modulo 1, or $x + n$ where $n \in \mathbb{Z}$ is such that $x + n \in [0, 1)$:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> frac(1.25)
mpf('0.25')
>>> frac(3)
mpf('0.0')
>>> frac(-1.25)
mpf('0.75')
```

For a complex number, the fractional part function applies to the real and imaginary parts separately:

```
>>> frac(2.25+3.75j)
mpc(real='0.25', imag='0.75')
```

Plotted, the fractional part function gives a sawtooth wave. The Fourier series coefficients have a simple form:

```
>>> mp.dps = 15
>>> nprint(fourier(lambda x: frac(x)-0.5, [0,1], 4))
([0.0, 0.0, 0.0, 0.0, 0.0], [0.0, -0.31831, -0.159155, -0.106103, -0.0795775])
>>> nprint([-1/(pi*k) for k in range(1,5)])
[-0.31831, -0.159155, -0.106103, -0.0795775]
```

Note: The fractional part is sometimes defined as a symmetric function, i.e. returning $-\mathrm{frac}(-x)$ if $x < 0$. This convention is used, for instance, by Mathematica's `FractionalPart`.

2.2.5 Tolerances and approximate comparisons

chop()

mpmath.**chop**(*x*, *tol=None*)

Chops off small real or imaginary parts, or converts numbers close to zero to exact zeros. The input can be a single number or an iterable:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> chop(5+1e-10j, tol=1e-9)
mpf('5.0')
>>> nprint(chop([1.0, 1e-20, 3+1e-18j, -4, 2]))
[1.0, 0.0, 3.0, -4.0, 2.0]
```

The tolerance defaults to `100*eps`.

almosteq()

mpmath.**almosteq**(*s*, *t*, *rel_eps=None*, *abs_eps=None*)

Determine whether the difference between s and t is smaller than a given epsilon, either relatively or absolutely.

Both a maximum relative difference and a maximum difference ('epsilons') may be specified. The absolute difference is defined as $|s-t|$ and the relative difference is defined as $|s-t|/\max(|s|,|t|)$.

If only one epsilon is given, both are set to the same value. If none is given, both epsilons are set to 2^{-p+m} where p is the current working precision and m is a small integer. The default setting typically allows *almosteq()* to be used to check for mathematical equality in the presence of small rounding errors.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15
>>> almosteq(3.141592653589793, 3.141592653589790)
True
>>> almosteq(3.141592653589793, 3.141592653589700)
False
>>> almosteq(3.141592653589793, 3.141592653589700, 1e-10)
True
>>> almosteq(1e-20, 2e-20)
True
>>> almosteq(1e-20, 2e-20, rel_eps=0, abs_eps=0)
False
```

2.2.6 Properties of numbers

isinf()

mpmath.**isinf**(*x*)

Return *True* if the absolute value of *x* is infinite; otherwise return *False*:

```
>>> from mpmath import *
>>> isinf(inf)
True
>>> isinf(-inf)
True
>>> isinf(3)
False
>>> isinf(3+4j)
False
>>> isinf(mpc(3,inf))
True
>>> isinf(mpc(inf,3))
True
```

isnan()

mpmath.**isnan**(*x*)

Return *True* if *x* is a NaN (not-a-number), or for a complex number, whether either the real or complex part is NaN; otherwise return *False*:

```
>>> from mpmath import *
>>> isnan(3.14)
```

```
    False
    >>> isnan(nan)
    True
    >>> isnan(mpc(3.14,2.72))
    False
    >>> isnan(mpc(3.14,nan))
    True
```

isnormal()

mpmath.**isnormal**(*x*)

Determine whether *x* is "normal" in the sense of floating-point representation; that is, return *False* if *x* is zero, an infinity or NaN; otherwise return *True*. By extension, a complex number *x* is considered "normal" if its magnitude is normal:

```
    >>> from mpmath import *
    >>> isnormal(3)
    True
    >>> isnormal(0)
    False
    >>> isnormal(inf); isnormal(-inf); isnormal(nan)
    False
    False
    False
    >>> isnormal(0+0j)
    False
    >>> isnormal(0+3j)
    True
    >>> isnormal(mpc(2,nan))
    False
```

isfinite()

mpmath.**isfinite**(*x*)

Return *True* if *x* is a finite number, i.e. neither an infinity or a NaN.

```
    >>> from mpmath import *
    >>> isfinite(inf)
    False
    >>> isfinite(-inf)
    False
    >>> isfinite(3)
    True
    >>> isfinite(nan)
    False
    >>> isfinite(3+4j)
    True
    >>> isfinite(mpc(3,inf))
    False
    >>> isfinite(mpc(nan,3))
    False
```

isint()

mpmath.**isint**(*x*, *gaussian=False*)

Return *True* if *x* is integer-valued; otherwise return *False*:

```
>>> from mpmath import *
>>> isint(3)
True
>>> isint(mpf(3))
True
>>> isint(3.2)
False
>>> isint(inf)
False
```

Optionally, Gaussian integers can be checked for:

```
>>> isint(3+0j)
True
>>> isint(3+2j)
False
>>> isint(3+2j, gaussian=True)
True
```

ldexp()

mpmath.**ldexp**(*x*, *n*)

Computes $x2^n$ efficiently. No rounding is performed. The argument x must be a real floating-point number (or possible to convert into one) and n must be a Python `int`.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> ldexp(1, 10)
mpf('1024.0')
>>> ldexp(1, -3)
mpf('0.125')
```

frexp()

mpmath.**frexp**(*x*, *n*)

Given a real number x, returns (y, n) with $y \in [0.5, 1)$, n a Python integer, and such that $x = y2^n$. No rounding is performed.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> frexp(7.5)
(mpf('0.9375'), 3)
```

mag()

mpmath.**mag**(*x*)

Quick logarithmic magnitude estimate of a number. Returns an integer or infinity m such that $|x| <= 2^m$. It is not guaranteed that m is an optimal bound, but it will never be too large by more than 2 (and probably not more than 1).

Examples

```
>>> from mpmath import *
>>> mp.pretty = True
>>> mag(10), mag(10.0), mag(mpf(10)), int(ceil(log(10,2)))
(4, 4, 4, 4)
>>> mag(10j), mag(10+10j)
(4, 5)
>>> mag(0.01), int(ceil(log(0.01,2)))
(-6, -6)
>>> mag(0), mag(inf), mag(-inf), mag(nan)
(-inf, +inf, +inf, nan)
```

nint_distance()

mpmath.**nint_distance**(*x*)

Return (n, d) where n is the nearest integer to x and d is an estimate of $\log_2(|x-n|)$. If $d < 0$, $-d$ gives the precision (measured in bits) lost to cancellation when computing $x - n$.

```
>>> from mpmath import *
>>> n, d = nint_distance(5)
>>> print(n); print(d)
5
-inf
>>> n, d = nint_distance(mpf(5))
>>> print(n); print(d)
5
-inf
>>> n, d = nint_distance(mpf(5.00000001))
>>> print(n); print(d)
5
-26
>>> n, d = nint_distance(mpf(4.99999999))
>>> print(n); print(d)
5
-26
>>> n, d = nint_distance(mpc(5,10))
>>> print(n); print(d)
5
4
>>> n, d = nint_distance(mpc(5,0.000001))
>>> print(n); print(d)
5
-19
```

2.2.7 Number generation

fraction()

mpmath.**fraction**(*p*, *q*)

Given Python integers (p, q), returns a lazy `mpf` representing the fraction p/q. The value is updated with the precision.

```
>>> from mpmath import *
>>> mp.dps = 15
>>> a = fraction(1,100)
```

```
>>> b = mpf(1)/100
>>> print(a); print(b)
0.01
0.01
>>> mp.dps = 30
>>> print(a); print(b)      # a will be accurate
0.01
0.010000000000000000002081668171172
>>> mp.dps = 15
```

rand()

mpmath.**rand**()

Returns an `mpf` with value chosen randomly from $[0, 1)$. The number of randomly generated bits in the mantissa is equal to the working precision.

arange()

mpmath.**arange**(**args*)

This is a generalized version of Python's `range()` function that accepts fractional endpoints and step sizes and returns a list of `mpf` instances. Like `range()`, *`arange()`* can be called with 1, 2 or 3 arguments:

`arange(b)` $[0, 1, 2, \ldots, x]$

`arange(a, b)` $[a, a+1, a+2, \ldots, x]$

`arange(a, b, h)` $[a, a+h, a+h, \ldots, x]$

where $b-1 \le x < b$ (in the third case, $b-h \le x < b$).

Like Python's `range()`, the endpoint is not included. To produce ranges where the endpoint is included, *`linspace()`* is more convenient.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> arange(4)
[mpf('0.0'), mpf('1.0'), mpf('2.0'), mpf('3.0')]
>>> arange(1, 2, 0.25)
[mpf('1.0'), mpf('1.25'), mpf('1.5'), mpf('1.75')]
>>> arange(1, -1, -0.75)
[mpf('1.0'), mpf('0.25'), mpf('-0.5')]
```

linspace()

mpmath.**linspace**(**args*, ***kwargs*)

`linspace(a, b, n)` returns a list of n evenly spaced samples from a to b. The syntax `linspace(mpi(a,b), n)` is also valid.

This function is often more convenient than *`arange()`* for partitioning an interval into subintervals, since the endpoint is included:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> linspace(1, 4, 4)
[mpf('1.0'), mpf('2.0'), mpf('3.0'), mpf('4.0')]
```

You may also provide the keyword argument `endpoint=False`:

```
>>> linspace(1, 4, 4, endpoint=False)
[mpf('1.0'), mpf('1.75'), mpf('2.5'), mpf('3.25')]
```

2.2.8 Precision management

autoprec()

mpmath.**autoprec**(*ctx*, *f*, *maxprec=None*, *catch=()*, *verbose=False*)

Return a wrapped copy of *f* that repeatedly evaluates *f* with increasing precision until the result converges to the full precision used at the point of the call.

This heuristically protects against rounding errors, at the cost of roughly a 2x slowdown compared to manually setting the optimal precision. This method can, however, easily be fooled if the results from *f* depend "discontinuously" on the precision, for instance if catastrophic cancellation can occur. Therefore, *autoprec()* should be used judiciously.

Examples

Many functions are sensitive to perturbations of the input arguments. If the arguments are decimal numbers, they may have to be converted to binary at a much higher precision. If the amount of required extra precision is unknown, *autoprec()* is convenient:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> mp.pretty = True
>>> besselj(5, 125 * 10**28)     # Exact input
-8.03284785591801e-17
>>> besselj(5, '1.25e30')    # Bad
7.12954868316652e-16
>>> autoprec(besselj)(5, '1.25e30')    # Good
-8.03284785591801e-17
```

The following fails to converge because $\sin(\pi) = 0$ whereas all finite-precision approximations of π give nonzero values:

```
>>> autoprec(sin)(pi)
Traceback (most recent call last):
  ...
NoConvergence: autoprec: prec increased to 2910 without convergence
```

As the following example shows, *autoprec()* can protect against cancellation, but is fooled by too severe cancellation:

```
>>> x = 1e-10
>>> exp(x)-1; expm1(x); autoprec(lambda t: exp(t)-1)(x)
1.00000008274037e-10
1.00000000005e-10
1.00000000005e-10
>>> x = 1e-50
>>> exp(x)-1; expm1(x); autoprec(lambda t: exp(t)-1)(x)
0.0
1.0e-50
0.0
```

With *catch*, an exception or list of exceptions to intercept may be specified. The raised exception is interpreted as signaling insufficient precision. This permits, for example, evaluating a function where a too low precision results in a division by zero:

```
>>> f = lambda x: 1/(exp(x)-1)
>>> f(1e-30)
Traceback (most recent call last):
  ...
ZeroDivisionError
>>> autoprec(f, catch=ZeroDivisionError)(1e-30)
1.0e+30
```

workprec()

mpmath.**workprec**(*ctx*, *n*, *normalize_output=False*)

The block

> **with workprec(n):** <code>

sets the precision to n bits, executes <code>, and then restores the precision.

workprec(n)(f) returns a decorated version of the function f that sets the precision to n bits before execution, and restores the precision afterwards. With normalize_output=True, it rounds the return value to the parent precision.

workdps()

mpmath.**workdps**(*ctx*, *n*, *normalize_output=False*)

This function is analogous to workprec (see documentation) but changes the decimal precision instead of the number of bits.

extraprec()

mpmath.**extraprec**(*ctx*, *n*, *normalize_output=False*)

The block

> **with extraprec(n):** <code>

increases the precision n bits, executes <code>, and then restores the precision.

extraprec(n)(f) returns a decorated version of the function f that increases the working precision by n bits before execution, and restores the parent precision afterwards. With normalize_output=True, it rounds the return value to the parent precision.

extradps()

mpmath.**extradps**(*ctx*, *n*, *normalize_output=False*)

This function is analogous to extraprec (see documentation) but changes the decimal precision instead of the number of bits.

2.2.9 Performance and debugging

memoize()

mpmath.**memoize**(*ctx*, *f*)

Return a wrapped copy of *f* that caches computed values, i.e. a memoized copy of *f*. Values are only reused if the cached precision is equal to or higher than the working precision:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = memoize(maxcalls(sin, 1))
>>> f(2)
0.909297426825682
>>> f(2)
0.909297426825682
>>> mp.dps = 25
>>> f(2)
Traceback (most recent call last):
  ...
NoConvergence: maxcalls: function evaluated 1 times
```

maxcalls()

mpmath.**maxcalls**(*ctx*, *f*, *N*)

Return a wrapped copy of *f* that raises `NoConvergence` when *f* has been called more than *N* times:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> f = maxcalls(sin, 10)
>>> print(sum(f(n) for n in range(10)))
1.95520948210738
>>> f(10)
Traceback (most recent call last):
  ...
NoConvergence: maxcalls: function evaluated 10 times
```

monitor()

mpmath.**monitor**(*f*, *input='print'*, *output='print'*)

Returns a wrapped copy of *f* that monitors evaluation by calling *input* with every input (*args*, *kwargs*) passed to *f* and *output* with every value returned from *f*. The default action (specify using the special string value `'print'`) is to print inputs and outputs to stdout, along with the total evaluation count:

```
>>> from mpmath import *
>>> mp.dps = 5; mp.pretty = False
>>> diff(monitor(exp), 1)   # diff will eval f(x-h) and f(x+h)
in  0 (mpf('0.99999999906867742538452148'),) {}
out 0 mpf('2.7182818259274480055282064')
in  1 (mpf('1.0000000009313225746154785'),) {}
out 1 mpf('2.7182818309906424675501024')
mpf('2.7182808')
```

To disable either the input or the output handler, you may pass *None* as argument.

Custom input and output handlers may be used e.g. to store results for later analysis:

```
>>> mp.dps = 15
>>> input = []
>>> output = []
>>> findroot(monitor(sin, input.append, output.append), 3.0)
mpf('3.1415926535897932')
>>> len(input)  # Count number of evaluations
9
>>> print(input[3]); print(output[3])
```

```
    ((mpf('3.1415076583334066'),), {})
    8.49952562843408e-5
    >>> print(input[4]); print(output[4])
    ((mpf('3.1415928201669122'),), {})
    -1.66577118985331e-7
```

timing()

mpmath.**timing**(*f*, **args*, ***kwargs*)

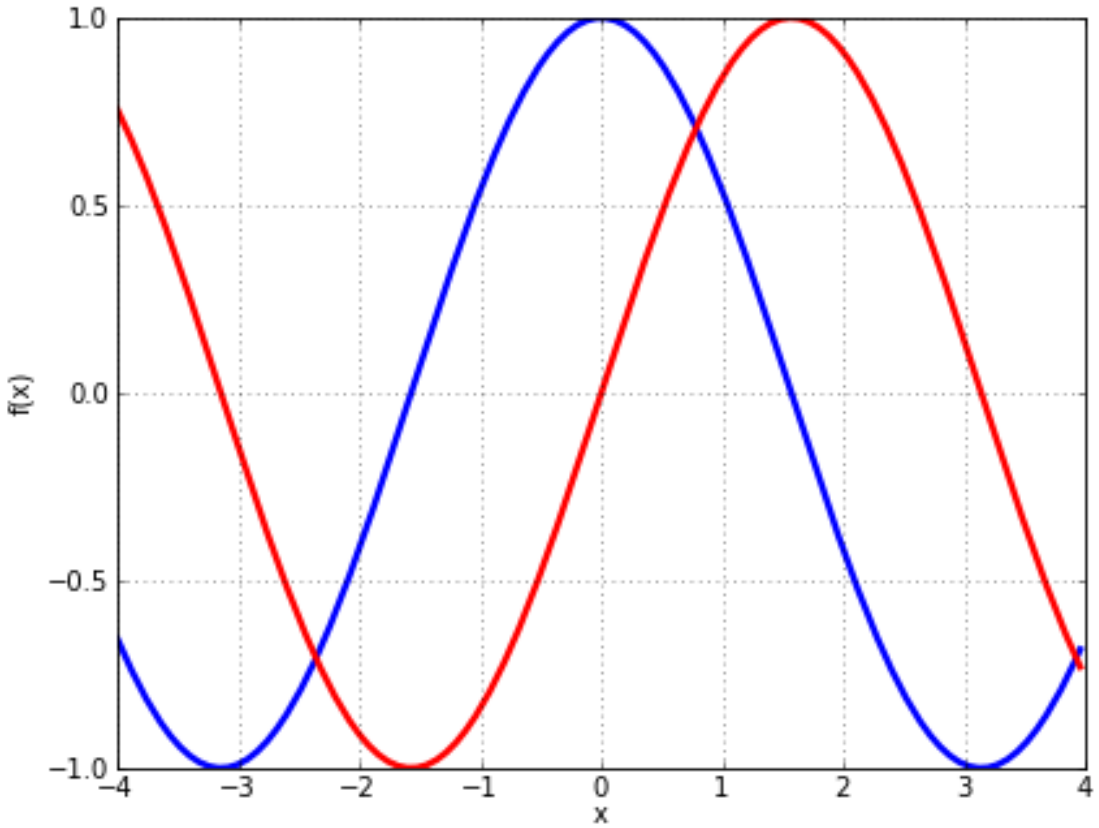
Returns time elapsed for evaluating `f()`. Optionally arguments may be passed to time the execution of `f(*args, **kwargs)`.

If the first call is very quick, `f` is called repeatedly and the best time is returned.

2.3 Plotting

If matplotlib is available, the functions `plot` and `cplot` in mpmath can be used to plot functions respectively as x-y graphs and in the complex plane. Also, `splot` can be used to produce 3D surface plots.

2.3.1 Function curve plots



Output of `plot([cos, sin], [-4, 4])`

mpmath.**plot**(*ctx, f, xlim=[-5, 5], ylim=None, points=200, file=None, dpi=None, singularities=[], axes=None*)

Shows a simple 2D plot of a function $f(x)$ or list of functions $[f_0(x), f_1(x), \ldots, f_n(x)]$ over a given interval specified by *xlim*. Some examples:

```
plot(lambda x: exp(x)*li(x), [1, 4])
plot([cos, sin], [-4, 4])
plot([fresnels, fresnelc], [-4, 4])
plot([sqrt, cbrt], [-4, 4])
plot(lambda t: zeta(0.5+t*j), [-20, 20])
plot([floor, ceil, abs, sign], [-5, 5])
```

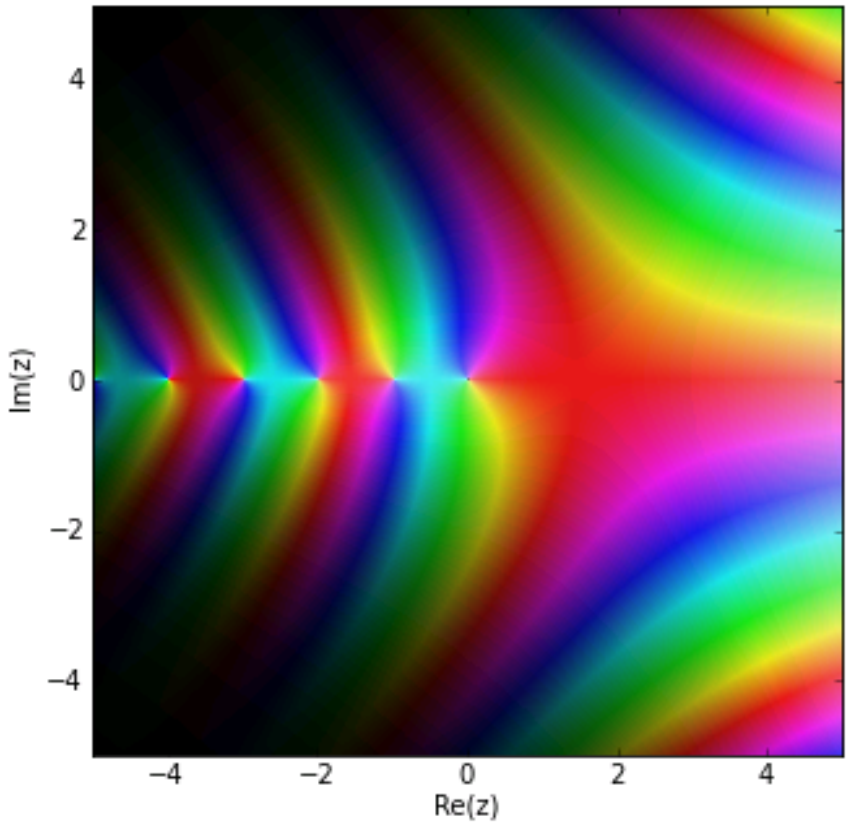
Points where the function raises a numerical exception or returns an infinite value are removed from the graph. Singularities can also be excluded explicitly as follows (useful for removing erroneous vertical lines):

```
plot(cot, ylim=[-5, 5])   # bad
plot(cot, ylim=[-5, 5], singularities=[-pi, 0, pi])  # good
```

For parts where the function assumes complex values, the real part is plotted with dashes and the imaginary part is plotted with dots.

Note: This function requires matplotlib (pylab).

2.3.2 Complex function plots



Output of `fp.cplot(fp.gamma, points=100000)`

mpmath.**cplot**(*ctx, f, re=[-5, 5], im=[-5, 5], points=2000, color=None, verbose=False, file=None, dpi=None, axes=None*)

Plots the given complex-valued function *f* over a rectangular part of the complex plane specified by the pairs of intervals *re* and *im*. For example:

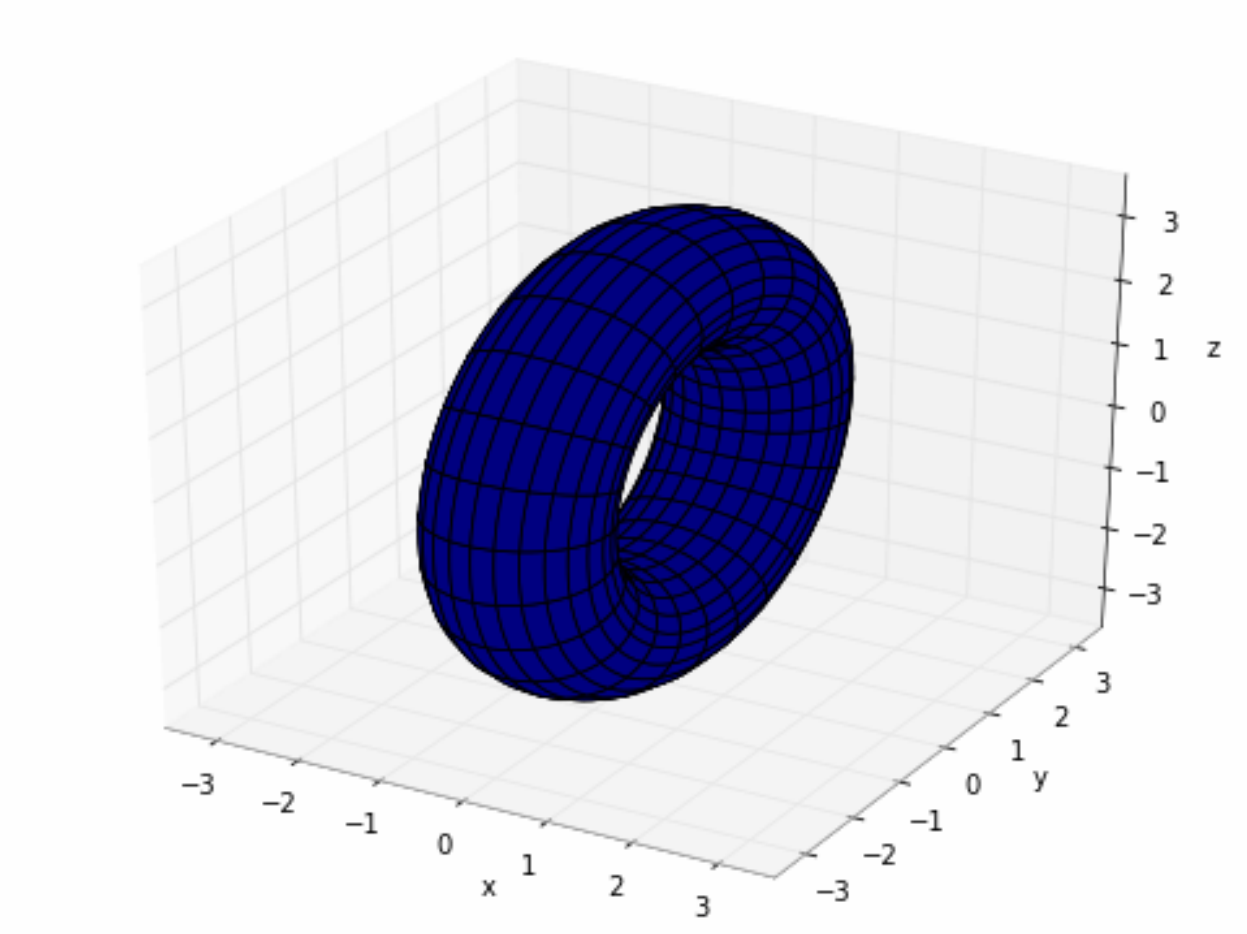
```
cplot(lambda z: z, [-2, 2], [-10, 10])
cplot(exp)
cplot(zeta, [0, 1], [0, 50])
```

By default, the complex argument (phase) is shown as color (hue) and the magnitude is show as brightness. You can also supply a custom color function (*color*). This function should take a complex number as input and return an RGB 3-tuple containing floats in the range 0.0-1.0.

To obtain a sharp image, the number of points may need to be increased to 100,000 or thereabout. Since evaluating the function that many times is likely to be slow, the 'verbose' option is useful to display progress.

Note: This function requires matplotlib (pylab).

2.3.3 3D surface plots



Output of `splot` for the donut example.

`mpmath.`**`splot`**(*ctx, f, u=[-5, 5], v=[-5, 5], points=100, keep_aspect=True, wireframe=False, file=None, dpi=None, axes=None*)

Plots the surface defined by f.

If f returns a single component, then this plots the surface defined by $z = f(x, y)$ over the rectangular domain with $x = u$ and $y = v$.

If f returns three components, then this plots the parametric surface $x, y, z = f(u, v)$ over the pairs of intervals u and v.

For example, to plot a simple function:

```
>>> from mpmath import *
>>> f = lambda x, y: sin(x+y)*cos(y)
>>> splot(f, [-pi,pi], [-pi,pi])
```

Plotting a donut:

```
>>> r, R = 1, 2.5
>>> f = lambda u, v: [r*cos(u), (R+r*sin(u))*cos(v), (R+r*sin(u))*sin(v)]
>>> splot(f, [0, 2*pi], [0, 2*pi])
```

Note: This function requires matplotlib (pylab) 0.98.5.3 or higher.

CHAPTER 3

Advanced mathematics

On top of its support for arbitrary-precision arithmetic, mpmath provides extensive support for transcendental functions, evaluation of sums, integrals, limits, roots, and so on.

3.1 Mathematical functions

Mpmath implements the standard functions from Python's `math` and `cmath` modules, for both real and complex numbers and with arbitrary precision. Many other functions are also available in mpmath, including commonly-used variants of standard functions (such as the alternative trigonometric functions sec, csc, cot), but also a large number of "special functions" such as the gamma function, the Riemann zeta function, error functions, Bessel functions, etc.

3.1.1 Mathematical constants

Mpmath supports arbitrary-precision computation of various common (and less common) mathematical constants. These constants are implemented as lazy objects that can evaluate to any precision. Whenever the objects are used as function arguments or as operands in arithmetic operations, they automagically evaluate to the current working precision. A lazy number can be converted to a regular `mpf` using the unary + operator, or by calling it as a function:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.2831853071795862')
>>> +pi
mpf('3.1415926535897931')
>>> pi()
mpf('3.1415926535897931')
>>> mp.dps = 40
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.2831853071795864769252867665590057683943388')
>>> +pi
mpf('3.1415926535897932384626433832795028841971694')
>>> pi()
mpf('3.1415926535897932384626433832795028841971694')
```

Exact constants

The predefined objects `j` (imaginary unit), `inf` (positive infinity) and `nan` (not-a-number) are shortcuts to `mpc` and `mpf` instances with these fixed values.

Pi (`pi`)

`mp.pi` = **<pi: 3.14159~>**

Degree (`degree`)

`mp.degree` = **<1 deg = pi / 180: 0.0174533~>**

Base of the natural logarithm (`e`)

`mp.e` = **<e = exp(1): 2.71828~>**

Golden ratio (`phi`)

`mp.phi` = **<Golden ratio phi: 1.61803~>**

Euler's constant (`euler`)

`mp.euler` = **<Euler's constant: 0.577216~>**

Catalan's constant (`catalan`)

`mp.catalan` = **<Catalan's constant: 0.915966~>**

Apery's constant (`apery`)

`mp.apery` = **<Apery's constant: 1.20206~>**

Khinchin's constant (`khinchin`)

`mp.khinchin` = **<Khinchin's constant: 2.68545~>**

Glaisher's constant (`glaisher`)

`mp.glaisher` = **<Glaisher's constant: 1.28243~>**

Mertens constant (`mertens`)

`mp.mertens` = **<Mertens' constant: 0.261497~>**

Twin prime constant (twinprime)

mp.**twinprime = <Twin prime constant: 0.660162~>**

3.1.2 Powers and logarithms

Nth roots

sqrt()

mpmath.**sqrt**(*x*, ***kwargs*)

sqrt(x) gives the principal square root of x, $\sqrt{x}$. For positive real numbers, the principal root is simply the positive square root. For arbitrary complex numbers, the principal square root is defined to satisfy $\sqrt{x} = \exp(\log(x)/2)$. The function thus has a branch cut along the negative half real axis.

For all mpmath numbers x, calling sqrt(x) is equivalent to performing x**0.5.

Examples

Basic examples and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sqrt(10)
3.16227766016838
>>> sqrt(100)
10.0
>>> sqrt(-4)
(0.0 + 2.0j)
>>> sqrt(1+1j)
(1.09868411346781 + 0.455089860562227j)
>>> sqrt(inf)
+inf
```

Square root evaluation is fast at huge precision:

```
>>> mp.dps = 50000
>>> a = sqrt(3)
>>> str(a)[-10:]
'9329332814'
```

mpmath.iv.sqrt() supports interval arguments:

```
>>> iv.dps = 15; iv.pretty = True
>>> iv.sqrt([16,100])
[4.0, 10.0]
>>> iv.sqrt(2)
[1.4142135623730949234, 1.4142135623730951455]
>>> iv.sqrt(2) ** 2
[1.9999999999999995559, 2.0000000000000004441]
```

hypot()

mpmath.**hypot**(*x*, *y*)

Computes the Euclidean norm of the vector (x, y), equal to $\sqrt{x^2 + y^2}$. Both x and y must be real.

cbrt()

mpmath.**cbrt**(*x*, ***kwargs*)

cbrt(x) computes the cube root of x, $x^{1/3}$. This function is faster and more accurate than raising to a floating-point fraction:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> 125**(mpf(1)/3)
mpf('4.9999999999999991')
>>> cbrt(125)
mpf('5.0')
```

Every nonzero complex number has three cube roots. This function returns the cube root defined by $\exp(\log(x)/3)$ where the principal branch of the natural logarithm is used. Note that this does not give a real cube root for negative real numbers:

```
>>> mp.pretty = True
>>> cbrt(-1)
(0.5 + 0.866025403784439j)
```

root()

mpmath.**root**(*z*, *n*, *k=0*)

root(z, n, k=0) computes an n-th root of z, i.e. returns a number r that (up to possible approximation error) satisfies $r^n = z$. (nthroot is available as an alias for root.)

Every complex number $z \neq 0$ has n distinct n-th roots, which are equidistant points on a circle with radius $|z|^{1/n}$, centered around the origin. A specific root may be selected using the optional index k. The roots are indexed counterclockwise, starting with $k = 0$ for the root closest to the positive real half-axis.

The $k = 0$ root is the so-called principal n-th root, often denoted by $\sqrt[n]{z}$ or $z^{1/n}$, and also given by $\exp(\log(z)/n)$. If z is a positive real number, the principal root is just the unique positive n-th root of z. Under some circumstances, non-principal real roots exist: for positive real z, n even, there is a negative root given by $k = n/2$; for negative real z, n odd, there is a negative root given by $k = (n-1)/2$.

To obtain all roots with a simple expression, use [root(z,n,k) for k in range(n)].

An important special case, root(1, n, k) returns the k-th n-th root of unity, $\zeta_k = e^{2\pi i k/n}$. Alternatively, *unitroots()* provides a slightly more convenient way to obtain the roots of unity, including the option to compute only the primitive roots of unity.

Both k and n should be integers; k outside of range(n) will be reduced modulo n. If n is negative, $x^{-1/n} = 1/x^{1/n}$ (or the equivalent reciprocal for a non-principal root with $k \neq 0$) is computed.

root() is implemented to use Newton's method for small n. At high precision, this makes $x^{1/n}$ not much more expensive than the regular exponentiation, x^n. For very large n, nthroot() falls back to use the exponential function.

Examples

nthroot()/*root()* is faster and more accurate than raising to a floating-point fraction:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> 16807 ** (mpf(1)/5)
mpf('7.0000000000000009')
>>> root(16807, 5)
mpf('7.0')
```

```
>>> nthroot(16807, 5)    # Alias
mpf('7.0')
```

A high-precision root:

```
>>> mp.dps = 50; mp.pretty = True
>>> nthroot(10, 5)
1.584893192461113485202101373391507013269442133825
>>> nthroot(10, 5) ** 5
10.0
```

Computing principal and non-principal square and cube roots:

```
>>> mp.dps = 15
>>> root(10, 2)
3.16227766016838
>>> root(10, 2, 1)
-3.16227766016838
>>> root(-10, 3)
(1.07721734501594 + 1.86579517236206j)
>>> root(-10, 3, 1)
-2.15443469003188
>>> root(-10, 3, 2)
(1.07721734501594 - 1.86579517236206j)
```

All the 7th roots of a complex number:

```
>>> for r in [root(3+4j, 7, k) for k in range(7)]:
...     print("%s %s" % (r, r**7))
...
(1.24747270589553 + 0.166227124177353j)  (3.0 + 4.0j)
(0.647824911301003 + 1.07895435170559j)  (3.0 + 4.0j)
(-0.439648254723098 + 1.17920694574172j)  (3.0 + 4.0j)
(-1.19605731775069 + 0.391492658196305j)  (3.0 + 4.0j)
(-1.05181082538903 - 0.691023585965793j)  (3.0 + 4.0j)
(-0.115529328478668 - 1.25318497558335j)  (3.0 + 4.0j)
(0.907748109144957 - 0.871672518271819j)  (3.0 + 4.0j)
```

Cube roots of unity:

```
>>> for k in range(3): print(root(1, 3, k))
...
1.0
(-0.5 + 0.866025403784439j)
(-0.5 - 0.866025403784439j)
```

Some exact high order roots:

```
>>> root(75**210, 105)
5625.0
>>> root(1, 128, 96)
(0.0 - 1.0j)
>>> root(4**128, 128, 96)
(0.0 - 4.0j)
```

unitroots()

mpmath.**unitroots**(*n*, *primitive=False*)

`unitroots(n)` returns $\zeta_0, \zeta_1, \ldots, \zeta_{n-1}$, all the distinct n-th roots of unity, as a list. If the option *primi-*

tive=True is passed, only the primitive roots are returned.

Every n-th root of unity satisfies $(\zeta_k)^n = 1$. There are n distinct roots for each n (ζ_k and ζ_j are the same when $k = j \pmod n$), which form a regular polygon with vertices on the unit circle. They are ordered counterclockwise with increasing k, starting with $\zeta_0 = 1$.

Examples

The roots of unity up to $n = 4$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(unitroots(1))
[1.0]
>>> nprint(unitroots(2))
[1.0, -1.0]
>>> nprint(unitroots(3))
[1.0, (-0.5 + 0.866025j), (-0.5 - 0.866025j)]
>>> nprint(unitroots(4))
[1.0, (0.0 + 1.0j), -1.0, (0.0 - 1.0j)]
```

Roots of unity form a geometric series that sums to 0:

```
>>> mp.dps = 50
>>> chop(fsum(unitroots(25)))
0.0
```

Primitive roots up to $n = 4$:

```
>>> mp.dps = 15
>>> nprint(unitroots(1, primitive=True))
[1.0]
>>> nprint(unitroots(2, primitive=True))
[-1.0]
>>> nprint(unitroots(3, primitive=True))
[(-0.5 + 0.866025j), (-0.5 - 0.866025j)]
>>> nprint(unitroots(4, primitive=True))
[(0.0 + 1.0j), (0.0 - 1.0j)]
```

There are only four primitive 12th roots:

```
>>> nprint(unitroots(12, primitive=True))
[(0.866025 + 0.5j), (-0.866025 + 0.5j), (-0.866025 - 0.5j), (0.866025 - 0.5j)]
```

The n-th roots of unity form a group, the cyclic group of order n. Any primitive root r is a generator for this group, meaning that $r^0, r^1, \ldots, r^{n-1}$ gives the whole set of unit roots (in some permuted order):

```
>>> for r in unitroots(6): print(r)
...
1.0
(0.5 + 0.866025403784439j)
(-0.5 + 0.866025403784439j)
-1.0
(-0.5 - 0.866025403784439j)
(0.5 - 0.866025403784439j)
>>> r = unitroots(6, primitive=True)[1]
>>> for k in range(6): print(chop(r**k))
...
1.0
(0.5 - 0.866025403784439j)
(-0.5 - 0.866025403784439j)
```

```
-1.0
(-0.5 + 0.866025403784438j)
(0.5 + 0.866025403784438j)
```

The number of primitive roots equals the Euler totient function $\phi(n)$:

```
>>> [len(unitroots(n, primitive=True)) for n in range(1,20)]
[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]
```

Exponentiation

exp()

mpmath.**exp**(*x*, ***kwargs*)

Computes the exponential function,

$$\exp(x) = e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

For complex numbers, the exponential function also satisfies

$$\exp(x+yi) = e^x(\cos y + i \sin y).$$

Basic examples

Some values of the exponential function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> exp(0)
1.0
>>> exp(1)
2.718281828459045235360287
>>> exp(-1)
0.3678794411714423215955238
>>> exp(inf)
+inf
>>> exp(-inf)
0.0
```

Arguments can be arbitrarily large:

```
>>> exp(10000)
8.806818225662921587261496e+4342
>>> exp(-10000)
1.135483865314736098540939e-4343
```

Evaluation is supported for interval arguments via `mpmath.iv.exp()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.exp([-inf,0])
[0.0, 1.0]
>>> iv.exp([0,1])
[1.0, 2.7182818284590452353602874956]
```

The exponential function can be evaluated efficiently to arbitrary precision:

```
>>> mp.dps = 10000
>>> exp(pi)
23.140692632779269005729...8984304016040616
```

Functional properties

Numerical verification of Euler's identity for the complex exponential function:

```
>>> mp.dps = 15
>>> exp(j*pi)+1
(0.0 + 1.22464679914735e-16j)
>>> chop(exp(j*pi)+1)
0.0
```

This recovers the coefficients (reciprocal factorials) in the Maclaurin series expansion of exp:

```
>>> nprint(taylor(exp, 0, 5))
[1.0, 1.0, 0.5, 0.166667, 0.0416667, 0.00833333]
```

The exponential function is its own derivative and antiderivative:

```
>>> exp(pi)
23.1406926327793
>>> diff(exp, pi)
23.1406926327793
>>> quad(exp, [-inf, pi])
23.1406926327793
```

The exponential function can be evaluated using various methods, including direct summation of the series, limits, and solving the defining differential equation:

```
>>> nsum(lambda k: pi**k/fac(k), [0,inf])
23.1406926327793
>>> limit(lambda k: (1+pi/k)**k, inf)
23.1406926327793
>>> odefun(lambda t, x: x, 0, 1)(pi)
23.1406926327793
```

power()

mpmath.**power**(*x*, *y*)

Converts x and y to mpmath numbers and evaluates $x^y = \exp(y \log(x))$:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> power(2, 0.5)
1.41421356237309504880168872421
```

This shows the leading few digits of a large Mersenne prime (performing the exact calculation `2**43112609-1` and displaying the result in Python would be very slow):

```
>>> power(2, 43112609)-1
3.16470269330255923143453723949e+12978188
```

expj()

mpmath.**expj**(*x*, ***kwargs*)

Convenience function for computing e^{ix}:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> expj(0)
(1.0 + 0.0j)
>>> expj(-1)
(0.5403023058681397174009366 - 0.8414709848078965066525023j)
>>> expj(j)
(0.3678794411714423215955238 + 0.0j)
>>> expj(1+j)
(0.1987661103464129406288032 + 0.3095598756531121984439128j)
```

expjpi()

mpmath.**expjpi**(*x*, ***kwargs*)

Convenience function for computing $e^{i\pi x}$. Evaluation is accurate near zeros (see also `cospi()`, `sinpi()`):

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> expjpi(0)
(1.0 + 0.0j)
>>> expjpi(1)
(-1.0 + 0.0j)
>>> expjpi(0.5)
(0.0 + 1.0j)
>>> expjpi(-1)
(-1.0 + 0.0j)
>>> expjpi(j)
(0.04321391826377224977441774 + 0.0j)
>>> expjpi(1+j)
(-0.04321391826377224977441774 + 0.0j)
```

expm1()

mpmath.**expm1**(*x*)

Computes $e^x - 1$, accurately for small x.

Unlike the expression `exp(x) - 1`, `expm1(x)` does not suffer from potentially catastrophic cancellation:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> exp(1e-10)-1; print(expm1(1e-10))
1.00000008274037e-10
1.00000000005e-10
>>> exp(1e-20)-1; print(expm1(1e-20))
0.0
1.0e-20
>>> 1/(exp(1e-20)-1)
Traceback (most recent call last):
  ...
ZeroDivisionError
>>> 1/expm1(1e-20)
1.0e+20
```

Evaluation works for extremely tiny values:

```
>>> expm1(0)
0.0
>>> expm1('1e-10000000')
1.0e-10000000
```

powm1()

mpmath.**powm1**(*x*, *y*)

Computes $x^y - 1$, accurately when x^y is very close to 1.

This avoids potentially catastrophic cancellation:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> power(0.99999995, 1e-10) - 1
0.0
>>> powm1(0.99999995, 1e-10)
-5.00000012791934e-18
```

Powers exactly equal to 1, and only those powers, yield 0 exactly:

```
>>> powm1(-j, 4)
(0.0 + 0.0j)
>>> powm1(3, 0)
0.0
>>> powm1(fadd(-1, 1e-100, exact=True), 4)
-4.0e-100
```

Evaluation works for extremely tiny y:

```
>>> powm1(2, '1e-100000')
6.93147180559945e-100001
>>> powm1(j, '1e-1000')
(-1.23370055013617e-2000 + 1.5707963267949e-1000j)
```

Logarithms

log()

mpmath.**log**(*x*, *b=None*)

Computes the base-b logarithm of x, $\log_b(x)$. If b is unspecified, *log()* computes the natural (base e) logarithm and is equivalent to *ln()*. In general, the base b logarithm is defined in terms of the natural logarithm as $\log_b(x) = \ln(x)/\ln(b)$.

By convention, we take $\log(0) = -\infty$.

The natural logarithm is real if $x > 0$ and complex if $x < 0$ or if x is complex. The principal branch of the complex logarithm is used, meaning that $\Im(\ln(x)) = -\pi < \arg(x) \le \pi$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> log(1)
0.0
```

```
>>> log(2)
0.693147180559945
>>> log(1000,10)
3.0
>>> log(4, 16)
0.5
>>> log(j)
(0.0 + 1.5707963267949j)
>>> log(-1)
(0.0 + 3.14159265358979j)
>>> log(0)
-inf
>>> log(inf)
+inf
```

The natural logarithm is the antiderivative of $1/x$:

```
>>> quad(lambda x: 1/x, [1, 5])
1.6094379124341
>>> log(5)
1.6094379124341
>>> diff(log, 10)
0.1
```

The Taylor series expansion of the natural logarithm around $x = 1$ has coefficients $(-1)^{n+1}/n$:

```
>>> nprint(taylor(log, 1, 7))
[0.0, 1.0, -0.5, 0.333333, -0.25, 0.2, -0.166667, 0.142857]
```

log() supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> log(pi)
1.1447298858494001741434273513530587116472948129153
>>> log(pi, pi**3)
0.33333333333333333333333333333333333333333333333333
>>> mp.dps = 25
>>> log(3+4j)
(1.6094379124341003746007593 + 0.92729521800161223242851253j)
```

ln()

mpmath.**ln**(*x*, ***kwargs*)

Computes the base-b logarithm of x, $\log_b(x)$. If b is unspecified, *log()* computes the natural (base e) logarithm and is equivalent to *ln()*. In general, the base b logarithm is defined in terms of the natural logarithm as $\log_b(x) = \ln(x)/\ln(b)$.

By convention, we take $\log(0) = -\infty$.

The natural logarithm is real if $x > 0$ and complex if $x < 0$ or if x is complex. The principal branch of the complex logarithm is used, meaning that $\Im(\ln(x)) = -\pi < \arg(x) \le \pi$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> log(1)
```

```
0.0
>>> log(2)
0.693147180559945
>>> log(1000,10)
3.0
>>> log(4, 16)
0.5
>>> log(j)
(0.0 + 1.5707963267949j)
>>> log(-1)
(0.0 + 3.14159265358979j)
>>> log(0)
-inf
>>> log(inf)
+inf
```

The natural logarithm is the antiderivative of $1/x$:

```
>>> quad(lambda x: 1/x, [1, 5])
1.6094379124341
>>> log(5)
1.6094379124341
>>> diff(log, 10)
0.1
```

The Taylor series expansion of the natural logarithm around $x = 1$ has coefficients $(-1)^{n+1}/n$:

```
>>> nprint(taylor(log, 1, 7))
[0.0, 1.0, -0.5, 0.333333, -0.25, 0.2, -0.166667, 0.142857]
```

log() supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> log(pi)
1.1447298858494001741434273513530587116472948129153
>>> log(pi, pi**3)
0.33333333333333333333333333333333333333333333333333
>>> mp.dps = 25
>>> log(3+4j)
(1.609437912434100374600759 + 0.9272952180016122324285125j)
```

log10()

mpmath.**log10**(*x*)

Computes the base-10 logarithm of x, $\log_{10}(x)$. `log10(x)` is equivalent to `log(x, 10)`.

Lambert W function

lambertw()

mpmath.**lambertw**(*z*, *k=0*)

The Lambert W function $W(z)$ is defined as the inverse function of $w \exp(w)$. In other words, the value of $W(z)$ is such that $z = W(z) \exp(W(z))$ for any complex number z.

The Lambert W function is a multivalued function with infinitely many branches $W_k(z)$, indexed by $k \in \mathbb{Z}$. Each branch gives a different solution w of the equation $z = w \exp(w)$. All branches are supported by

lambertw():

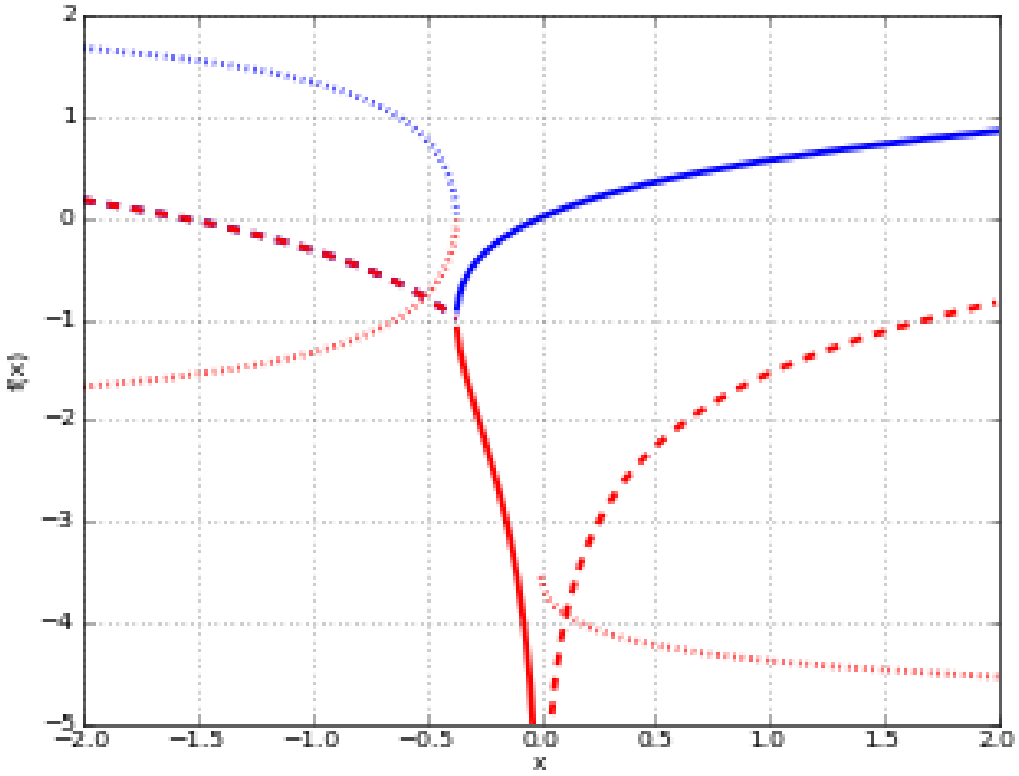
- `lambertw(z)` gives the principal solution (branch 0)
- `lambertw(z, k)` gives the solution on branch k

The Lambert W function has two partially real branches: the principal branch ($k = 0$) is real for real $z > -1/e$, and the $k = -1$ branch is real for $-1/e < z < 0$. All branches except $k = 0$ have a logarithmic singularity at $z = 0$.

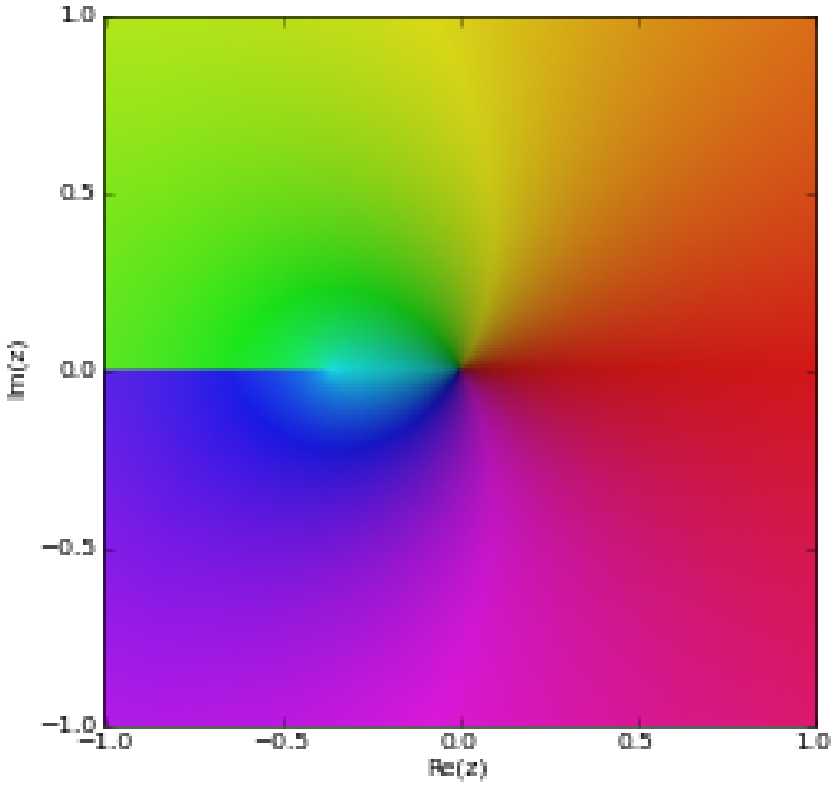
The definition, implementation and choice of branches is based on *[Corless]*.

Plots

```
# Branches 0 and -1 of the Lambert W function
plot([lambertw, lambda x: lambertw(x,-1)], [-2,2], [-5,2], points=2000)
```




```
# Principal branch of the Lambert W function W(z)
cplot(lambertw, [-1,1], [-1,1], points=50000)
```



Basic examples

The Lambert W function is the inverse of $w \exp(w)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> w = lambertw(1)
>>> w
0.5671432904097838729999687
>>> w*exp(w)
1.0
```

Any branch gives a valid inverse:

```
>>> w = lambertw(1, k=3)
>>> w
(-2.853581755409037807206819 + 17.11353553941214591260783j)
>>> w = lambertw(1, k=25)
>>> w
(-5.047020464221569709378686 + 155.4763860949415867162066j)
>>> chop(w*exp(w))
1.0
```

Applications to equation-solving

The Lambert W function may be used to solve various kinds of equations, such as finding the value of the infinite power tower $z^{z^{z^{\cdots}}}$:

```
>>> def tower(z, n):
...     if n == 0:
...         return z
...     return z ** tower(z, n-1)
...
>>> tower(mpf(0.5), 100)
0.64118574450498598448620005
>>> -lambertw(-log(0.5))/log(0.5)
0.64118574450498598448620005
```

Properties

The Lambert W function grows roughly like the natural logarithm for large arguments:

```
>>> lambertw(1000); log(1000)
5.2496028524015962271260056
6.9077552789821370520539743
>>> lambertw(10**100); log(10**100)
224.84310644511850153937310
230.25850929940456840179914
```

The principal branch of the Lambert W function has a rational Taylor series expansion around $z = 0$:

```
>>> nprint(taylor(lambertw, 0, 6), 10)
[0.0, 1.0, -1.0, 1.5, -2.666666667, 5.208333333, -10.8]
```

Some special values and limits are:

```
>>> lambertw(0)
0.0
>>> lambertw(1)
0.56714329040978387299996866
>>> lambertw(e)
1.0
>>> lambertw(inf)
+inf
>>> lambertw(0, k=-1)
-inf
>>> lambertw(0, k=3)
-inf
>>> lambertw(inf, k=2)
(+inf + 12.566370614359172953850573j)
>>> lambertw(inf, k=3)
(+inf + 18.849555921538759430775860j)
>>> lambertw(-inf, k=3)
(+inf + 21.991148575128552669238503j)
```

The $k = 0$ and $k = -1$ branches join at $z = -1/e$ where $W(z) = -1$ for both branches. Since $-1/e$ can only be represented approximately with binary floating-point numbers, evaluating the Lambert W function at this point only gives -1 approximately:

```
>>> lambertw(-1/e, 0)
-0.99999999999983713302282510
>>> lambertw(-1/e, -1)
-1.00000000000016286697717490
```

If $-1/e$ happens to round in the negative direction, there might be a small imaginary part:

```
>>> mp.dps = 15
>>> lambertw(-1/e)
```

```
    (-1.0 + 8.22007971483662e-9j)
    >>> lambertw(-1/e+eps)
    -0.999999966242188
```

References

1. *[Corless]*

Arithmetic-geometric mean

agm()

mpmath.**agm**(*a*, *b=1*)

agm(a, b) computes the arithmetic-geometric mean of a and b, defined as the limit of the following iteration:

$$a_0 = a$$
$$b_0 = b$$
$$a_{n+1} = \frac{a_n + b_n}{2}$$
$$b_{n+1} = \sqrt{a_n b_n}$$

This function can be called with a single argument, computing $\mathrm{agm}(a,1) = \mathrm{agm}(1,a)$.

Examples

It is a well-known theorem that the geometric mean of two distinct positive numbers is less than the arithmetic mean. It follows that the arithmetic-geometric mean lies between the two means:

```
    >>> from mpmath import *
    >>> mp.dps = 15; mp.pretty = True
    >>> a = mpf(3)
    >>> b = mpf(4)
    >>> sqrt(a*b)
    3.46410161513775
    >>> agm(a,b)
    3.48202767635957
    >>> (a+b)/2
    3.5
```

The arithmetic-geometric mean is scale-invariant:

```
    >>> agm(10*e, 10*pi)
    29.261085515723
    >>> 10*agm(e, pi)
    29.261085515723
```

As an order-of-magnitude estimate, $\mathrm{agm}(1,x) \approx x$ for large x:

```
    >>> agm(10**10)
    643448704.760133
    >>> agm(10**50)
    1.34814309345871e+48
```

For tiny x, $\mathrm{agm}(1,x) \approx -\pi/(2\log(x/4))$:

```
    >>> agm('0.01')
    0.262166887202249
    >>> -pi/2/log('0.0025')
    0.262172347753122
```

The arithmetic-geometric mean can also be computed for complex numbers:

```
>>> agm(3, 2+j)
(2.51055133276184 + 0.547394054060638j)
```

The AGM iteration converges very quickly (each step doubles the number of correct digits), so *agm()* supports efficient high-precision evaluation:

```
>>> mp.dps = 10000
>>> a = agm(1,2)
>>> str(a)[-10:]
'1679581912'
```

Mathematical relations

The arithmetic-geometric mean may be used to evaluate the following two parametric definite integrals:

$$I_1 = \int_0^\infty \frac{1}{\sqrt{(x^2+a^2)(x^2+b^2)}}\,dx$$

$$I_2 = \int_0^{\pi/2} \frac{1}{\sqrt{a^2\cos^2(x) + b^2\sin^2(x)}}\,dx$$

We have:

```
>>> mp.dps = 15
>>> a = 3
>>> b = 4
>>> f1 = lambda x: ((x**2+a**2)*(x**2+b**2))**-0.5
>>> f2 = lambda x: ((a*cos(x))**2 + (b*sin(x))**2)**-0.5
>>> quad(f1, [0, inf])
0.451115405388492
>>> quad(f2, [0, pi/2])
0.451115405388492
>>> pi/(2*agm(a,b))
0.451115405388492
```

A formula for $\Gamma(1/4)$:

```
>>> gamma(0.25)
3.62560990822191
>>> sqrt(2*sqrt(2*pi**3)/agm(1,sqrt(2)))
3.62560990822191
```

Possible issues

The branch cut chosen for complex a and b is somewhat arbitrary.

3.1.3 Trigonometric functions

Except where otherwise noted, the trigonometric functions take a radian angle as input and the inverse trigonometric functions return radian angles.

The ordinary trigonometric functions are single-valued functions defined everywhere in the complex plane (except at the poles of tan, sec, csc, and cot). They are defined generally via the exponential function, e.g.

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}.$$

The inverse trigonometric functions are multivalued, thus requiring branch cuts, and are generally real-valued only on a part of the real line. Definitions and branch cuts are given in the documentation of each function. The branch cut conventions used by mpmath are essentially the same as those found in most standard mathematical software, such as Mathematica and Python's own `cmath` libary (as of Python 2.6; earlier Python versions implement some functions erroneously).

Degree-radian conversion

degrees()

mpmath.**degrees**(x)
Converts the radian angle x to a degree angle:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> degrees(pi/3)
60.0
```

radians()

mpmath.**radians**(x)
Converts the degree angle x to radians:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> radians(60)
1.0471975511966
```

Trigonometric functions

cos()

mpmath.**cos**(x, **kwargs*)
Computes the cosine of x, $\cos(x)$.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> cos(pi/3)
0.5
>>> cos(100000001)
-0.98028501132447133531332 43
>>> cos(2+3j)
(-4.189625690968807230132555 - 9.109227893755336597979197j)
>>> cos(inf)
nan
>>> nprint(chop(taylor(cos, 0, 6)))
[1.0, 0.0, -0.5, 0.0, 0.0416667, 0.0, -0.00138889]
```

Intervals are supported via `mpmath.iv.cos()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.cos([0,1])
[0.54030230586813971740093660 2301, 1.0]
```

```
>>> iv.cos([0,2])
[-0.41614683654714238699756823214, 1.0]
```

sin()

mpmath.**sin**(*x*, ***kwargs*)

Computes the sine of x, $\sin(x)$.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sin(pi/3)
0.86602540378443864676372317
>>> sin(100000001)
0.19758870555794968911438743
>>> sin(2+3j)
(9.154499146911429573467 3 - 4.1689069599665643507548 3j)
>>> sin(inf)
nan
>>> nprint(chop(taylor(sin, 0, 6)))
[0.0, 1.0, 0.0, -0.166667, 0.0, 0.00833333, 0.0]
```

Intervals are supported via `mpmath.iv.sin()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.sin([0,1])
[0.0, 0.84147098480789650665250233121]
>>> iv.sin([0,2])
[0.0, 1.0]
```

tan()

mpmath.**tan**(*x*, ***kwargs*)

Computes the tangent of x, $\tan(x) = \frac{\sin(x)}{\cos(x)}$. The tangent function is singular at $x = (n+1/2)\pi$, but `tan(x)` always returns a finite result since $(n+1/2)\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> tan(pi/3)
1.7320508075688772935274463
>>> tan(100000001)
-0.20156250814498645330910 8
>>> tan(2+3j)
(-0.00376402564150424829275122 1 + 1.00323862735360980144635 j)
>>> tan(inf)
nan
>>> nprint(chop(taylor(tan, 0, 6)))
[0.0, 1.0, 0.0, 0.333333, 0.0, 0.133333, 0.0]
```

Intervals are supported via `mpmath.iv.tan()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.tan([0,1])
[0.0, 1.5574077246549022305069748 4]
>>> iv.tan([0,2])   # Interval includes a singularity
[-inf, +inf]
```

sec()

mpmath.**sec**(*x*)

Computes the secant of x, $\sec(x) = \frac{1}{\cos(x)}$. The secant function is singular at $x = (n + 1/2)\pi$, but `sec(x)` always returns a finite result since $(n + 1/2)\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sec(pi/3)
2.0
>>> sec(10000001)
-1.184723164360392819100265
>>> sec(2+3j)
(-0.041674964411144427004834991 + 0.090611137196237596529661 2j)
>>> sec(inf)
nan
>>> nprint(chop(taylor(sec, 0, 6)))
[1.0, 0.0, 0.5, 0.0, 0.208333, 0.0, 0.0847222]
```

Intervals are supported via `mpmath.iv.sec()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.sec([0,1])
[1.0, 1.8508157176809256179117532 6276]
>>> iv.sec([0,2])  # Interval includes a singularity
[-inf, +inf]
```

csc()

mpmath.**csc**(*x*)

Computes the cosecant of x, $\csc(x) = \frac{1}{\sin(x)}$. This cosecant function is singular at $x = n\pi$, but with the exception of the point $x = 0$, `csc(x)` returns a finite result since $n\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> csc(pi/3)
1.154700538379251529018298
>>> csc(10000001)
-1.864910497503629858938891
>>> csc(2+3j)
(0.090473209753207439805790 48 + 0.041200986288574126463009 81j)
>>> csc(inf)
nan
```

Intervals are supported via `mpmath.iv.csc()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.csc([0,1])  # Interval includes a singularity
[1.1883951057781212162615994 3988, +inf]
>>> iv.csc([0,2])
[1.0, +inf]
```

cot()

mpmath.**cot**(*x*)

Computes the cotangent of x, $\cot(x) = \frac{1}{\tan(x)} = \frac{\cos(x)}{\sin(x)}$. This cotangent function is singular at $x = n\pi$, but with the exception of the point $x = 0$, `cot(x)` returns a finite result since $n\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> cot(pi/3)
0.57735026918962576450914878
>>> cot(10000001)
1.5741318762096256560035620
>>> cot(2+3j)
(-0.0037397103763369566601174090 - 0.99675779656935831046096880j)
>>> cot(inf)
nan
```

Intervals are supported via `mpmath.iv.cot()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.cot([0,1])  # Interval includes a singularity
[0.64209261593433070300641997486, +inf]
>>> iv.cot([1,2])
[-inf, +inf]
```

Trigonometric functions with modified argument

cospi()

mpmath.**cospi**(*x*, ***kwargs*)

Computes $\cos(\pi x)$, more accurately than the expression `cos(pi*x)`:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> cospi(10**10), cos(pi*(10**10))
(1.0, 0.999999999997493)
>>> cospi(10**10+0.5), cos(pi*(10**10+0.5))
(0.0, 1.59960492420134e-6)
```

sinpi()

mpmath.**sinpi**(*x*, ***kwargs*)

Computes $\sin(\pi x)$, more accurately than the expression `sin(pi*x)`:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sinpi(10**10), sin(pi*(10**10))
(0.0, -2.23936276195592e-6)
>>> sinpi(10**10+0.5), sin(pi*(10**10+0.5))
(1.0, 0.999999999998721)
```

Inverse trigonometric functions

acos()

mpmath.**acos**(*x*, ***kwargs*)

Computes the inverse cosine or arccosine of x, $\cos^{-1}(x)$. Since $-1 \le \cos(x) \le 1$ for real x, the inverse cosine is real-valued only for $-1 \le x \le 1$. On this interval, *acos()* is defined to be a monotonically decreasing function assuming values between $+\pi$ and 0.

Basic values are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> acos(-1)
3.1415926535897932384626433
>>> acos(0)
1.5707963267948966192313217
>>> acos(1)
0.0
>>> nprint(chop(taylor(acos, 0, 6)))
[1.5708, -1.0, 0.0, -0.166667, 0.0, -0.075, 0.0]
```

acos() is defined so as to be a proper inverse function of $\cos(\theta)$ for $0 \le \theta < \pi$. We have $\cos(\cos^{-1}(x)) = x$ for all x, but $\cos^{-1}(\cos(x)) = x$ only for $0 \le \Re[x] < \pi$:

```
>>> for x in [1, 10, -1, 2+3j, 10+3j]:
...     print("%s %s" % (cos(acos(x)), acos(cos(x))))
...
1.0 1.0
(10.0 + 0.0j) 2.5663706143591729538505735
-1.0 1.0
(2.0 + 3.0j) (2.0 + 3.0j)
(10.0 + 3.0j) (2.5663706143591729538505735 - 3.0j)
```

The inverse cosine has two branch points: $x = \pm 1$. *acos()* places the branch cuts along the line segments $(-\infty, -1)$ and $(+1, +\infty)$. In general,

$$\cos^{-1}(x) = \frac{\pi}{2} + i \log\left(ix + \sqrt{1 - x^2}\right)$$

where the principal-branch log and square root are implied.

asin()

mpmath.**asin**(*x*, ***kwargs*)

Computes the inverse sine or arcsine of x, $\sin^{-1}(x)$. Since $-1 \le \sin(x) \le 1$ for real x, the inverse sine is real-valued only for $-1 \le x \le 1$. On this interval, it is defined to be a monotonically increasing function assuming values between $-\pi/2$ and $\pi/2$.

Basic values are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> asin(-1)
-1.5707963267948966192313217
>>> asin(0)
0.0
>>> asin(1)
```

```
1.5707963267948966192313222
>>> nprint(chop(taylor(asin, 0, 6)))
[0.0, 1.0, 0.0, 0.166667, 0.0, 0.075, 0.0]
```

asin() is defined so as to be a proper inverse function of $\sin(\theta)$ for $-\pi/2 < \theta < \pi/2$. We have $\sin(\sin^{-1}(x)) = x$ for all x, but $\sin^{-1}(\sin(x)) = x$ only for $-\pi/2 < \Re[x] < \pi/2$:

```
>>> for x in [1, 10, -1, 1+3j, -2+3j]:
...     print("%s %s" % (chop(sin(asin(x))), asin(sin(x))))
...
1.0 1.0
10.0 -0.57522203923062028461206982
-1.0 -1.0
(1.0 + 3.0j) (1.0 + 3.0j)
(-2.0 + 3.0j) (-1.1415926535897932384626433 - 3.0j)
```

The inverse sine has two branch points: $x = \pm 1$. *asin()* places the branch cuts along the line segments $(-\infty, -1)$ and $(+1, +\infty)$. In general,

$$\sin^{-1}(x) = -i \log\left(ix + \sqrt{1 - x^2}\right)$$

where the principal-branch log and square root are implied.

atan()

mpmath.**atan**(*x*, ***kwargs*)

Computes the inverse tangent or arctangent of x, $\tan^{-1}(x)$. This is a real-valued function for all real x, with range $(-\pi/2, \pi/2)$.

Basic values are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> atan(-inf)
-1.5707963267948966192313222
>>> atan(-1)
-0.78539816339744830961566091
>>> atan(0)
0.0
>>> atan(1)
0.78539816339744830961566091
>>> atan(inf)
1.5707963267948966192313222
>>> nprint(chop(taylor(atan, 0, 6)))
[0.0, 1.0, 0.0, -0.333333, 0.0, 0.2, 0.0]
```

The inverse tangent is often used to compute angles. However, the atan2 function is often better for this as it preserves sign (see *atan2()*).

atan() is defined so as to be a proper inverse function of $\tan(\theta)$ for $-\pi/2 < \theta < \pi/2$. We have $\tan(\tan^{-1}(x)) = x$ for all x, but $\tan^{-1}(\tan(x)) = x$ only for $-\pi/2 < \Re[x] < \pi/2$:

```
>>> mp.dps = 25
>>> for x in [1, 10, -1, 1+3j, -2+3j]:
...     print("%s %s" % (tan(atan(x)), atan(tan(x))))
...
1.0 1.0
10.0 0.57522203923062028461206982
```

```
-1.0 -1.0
(1.0 + 3.0j) (1.0000000000000000000000001 + 3.0j)
(-2.0 + 3.0j) (1.1415926535897932384626644 + 3.0j)
```

The inverse tangent has two branch points: $x = \pm i$. `atan()` places the branch cuts along the line segments $(-i\infty, -i)$ and $(+i, +i\infty)$. In general,

$$\tan^{-1}(x) = \frac{i}{2}\left(\log(1 - ix) - \log(1 + ix)\right)$$

where the principal-branch log is implied.

atan2()

mpmath.**atan2**(y, x)

Computes the two-argument arctangent, $\mathrm{atan2}(y, x)$, giving the signed angle between the positive x-axis and the point (x, y) in the 2D plane. This function is defined for real x and y only.

The two-argument arctangent essentially computes $\mathrm{atan}(y/x)$, but accounts for the signs of both x and y to give the angle for the correct quadrant. The following examples illustrate the difference:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> atan2(1,1), atan(1/1.)
(0.785398163397448, 0.785398163397448)
>>> atan2(1,-1), atan(1/-1.)
(2.35619449019234, -0.785398163397448)
>>> atan2(-1,1), atan(-1/1.)
(-0.785398163397448, -0.785398163397448)
>>> atan2(-1,-1), atan(-1/-1.)
(-2.35619449019234, 0.785398163397448)
```

The angle convention is the same as that used for the complex argument; see `arg()`.

asec()

mpmath.**asec**(x)

Computes the inverse secant of x, $\mathrm{sec}^{-1}(x) = \cos^{-1}(1/x)$.

acsc()

mpmath.**acsc**(x)

Computes the inverse cosecant of x, $\mathrm{csc}^{-1}(x) = \sin^{-1}(1/x)$.

acot()

mpmath.**acot**(x)

Computes the inverse cotangent of x, $\mathrm{cot}^{-1}(x) = \tan^{-1}(1/x)$.

Sinc function

sinc()

mpmath.**sinc**(*x*)

`sinc(x)` computes the unnormalized sinc function, defined as

$$\operatorname{sinc}(x) = \begin{cases} \sin(x)/x, & \text{if } x \neq 0 \\ 1, & \text{if } x = 0. \end{cases}$$

See *sincpi()* for the normalized sinc function.

Simple values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sinc(0)
1.0
>>> sinc(1)
0.841470984807897
>>> sinc(inf)
0.0
```

The integral of the sinc function is the sine integral Si:

```
>>> quad(sinc, [0, 1])
0.946083070367183
>>> si(1)
0.946083070367183
```

sincpi()

mpmath.**sincpi**(*x*)

`sincpi(x)` computes the normalized sinc function, defined as

$$\operatorname{sinc}_\pi(x) = \begin{cases} \sin(\pi x)/(\pi x), & \text{if } x \neq 0 \\ 1, & \text{if } x = 0. \end{cases}$$

Equivalently, we have $\operatorname{sinc}_\pi(x) = \operatorname{sinc}(\pi x)$.

The normalization entails that the function integrates to unity over the entire real line:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> quadosc(sincpi, [-inf, inf], period=2.0)
1.0
```

Like, *sinpi()*, *sincpi()* is evaluated accurately at its roots:

```
>>> sincpi(10)
0.0
```

3.1.4 Hyperbolic functions

Hyperbolic functions

cosh()

mpmath.**cosh**(*x*, ***kwargs*)

Computes the hyperbolic cosine of x, $\cosh(x) = (e^x + e^{-x})/2$. Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> cosh(0)
1.0
>>> cosh(1)
1.543080634815243778477906
>>> cosh(-inf), cosh(+inf)
(+inf, +inf)
```

The hyperbolic cosine is an even, convex function with a global minimum at $x = 0$, having a Maclaurin series that starts:

```
>>> nprint(chop(taylor(cosh, 0, 5)))
[1.0, 0.0, 0.5, 0.0, 0.0416667, 0.0]
```

Generalized to complex numbers, the hyperbolic cosine is equivalent to a cosine with the argument rotated in the imaginary direction, or $\cosh x = \cos ix$:

```
>>> cosh(2+3j)
(-3.724545504915322565473971 + 0.511822569987384608834463 8j)
>>> cos(3-2j)
(-3.724545504915322565473971 + 0.511822569987384608834463 8j)
```

sinh()

mpmath.**sinh**(*x*, ***kwargs*)

Computes the hyperbolic sine of x, $\sinh(x) = (e^x - e^{-x})/2$. Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sinh(0)
0.0
>>> sinh(1)
1.175201193643801456882382
>>> sinh(-inf), sinh(+inf)
(-inf, +inf)
```

The hyperbolic sine is an odd function, with a Maclaurin series that starts:

```
>>> nprint(chop(taylor(sinh, 0, 5)))
[0.0, 1.0, 0.0, 0.166667, 0.0, 0.00833333]
```

Generalized to complex numbers, the hyperbolic sine is essentially a sine with a rotation i applied to the argument; more precisely, $\sinh x = -i \sin ix$:

```
>>> sinh(2+3j)
(-3.590564589985779952012565 + 0.530921086248519805267040 1j)
```

```
>>> j*sin(3-2j)
(-3.5905645899857799520125 65 + 0.530921086248519805267040 1j)
```

tanh()

mpmath.**tanh**(*x*, ***kwargs*)

Computes the hyperbolic tangent of x, $\tanh(x) = \sinh(x)/\cosh(x)$. Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> tanh(0)
0.0
>>> tanh(1)
0.7615941559557648881194583
>>> tanh(-inf), tanh(inf)
(-1.0, 1.0)
```

The hyperbolic tangent is an odd, sigmoidal function, similar to the inverse tangent and error function. Its Maclaurin series is:

```
>>> nprint(chop(taylor(tanh, 0, 5)))
[0.0, 1.0, 0.0, -0.333333, 0.0, 0.133333]
```

Generalized to complex numbers, the hyperbolic tangent is essentially a tangent with a rotation i applied to the argument; more precisely, $\tanh x = -i \tan ix$:

```
>>> tanh(2+3j)
(0.965385879022133124278480 3 - 0.009884375038322493720314034j)
>>> j*tan(3-2j)
(0.965385879022133124278480 3 - 0.009884375038322493720314034j)
```

sech()

mpmath.**sech**(*x*)

Computes the hyperbolic secant of x, $\mathrm{sech}(x) = \frac{1}{\cosh(x)}$.

csch()

mpmath.**csch**(*x*)

Computes the hyperbolic cosecant of x, $\mathrm{csch}(x) = \frac{1}{\sinh(x)}$.

coth()

mpmath.**coth**(*x*)

Computes the hyperbolic cotangent of x, $\mathrm{coth}(x) = \frac{\cosh(x)}{\sinh(x)}$.

Inverse hyperbolic functions

acosh()

mpmath.**acosh**(*x*, ***kwargs*)
Computes the inverse hyperbolic cosine of x, $\cosh^{-1}(x) = \log(x + \sqrt{x+1}\sqrt{x-1})$.

asinh()

mpmath.**asinh**(*x*, ***kwargs*)
Computes the inverse hyperbolic sine of x, $\sinh^{-1}(x) = \log(x + \sqrt{1+x^2})$.

atanh()

mpmath.**atanh**(*x*, ***kwargs*)
Computes the inverse hyperbolic tangent of x, $\tanh^{-1}(x) = \frac{1}{2}(\log(1+x) - \log(1-x))$.

asech()

mpmath.**asech**(*x*)
Computes the inverse hyperbolic secant of x, $\mathrm{sech}^{-1}(x) = \cosh^{-1}(1/x)$.

acsch()

mpmath.**acsch**(*x*)
Computes the inverse hyperbolic cosecant of x, $\mathrm{csch}^{-1}(x) = \sinh^{-1}(1/x)$.

acoth()

mpmath.**acoth**(*x*)
Computes the inverse hyperbolic cotangent of x, $\coth^{-1}(x) = \tanh^{-1}(1/x)$.

3.1.5 Factorials and gamma functions

Factorials and factorial-like sums and products are basic tools of combinatorics and number theory. Much like the exponential function is fundamental to differential equations and analysis in general, the factorial function (and its extension to complex numbers, the gamma function) is fundamental to difference equations and functional equations.

A large selection of factorial-like functions is implemented in mpmath. All functions support complex arguments, and arguments may be arbitrarily large. Results are numerical approximations, so to compute *exact* values a high enough precision must be set manually:

```
>>> mp.dps = 15; mp.pretty = True
>>> fac(100)
9.33262154439442e+157
>>> print int(_)    # most digits are wrong
933262154439441509656467047959538825784009703731840988310128895405822272385704312
950661130892883272778258496640065242705545359762897193828521818658959597240320
>>> mp.dps = 160
```

```
>>> fac(100)
933262154439441526816992388562667004907159682643816214685929638952175999932299150
608941463976156518286253697920827223758251185210916864000000000000000000000000.0
```

The gamma and polygamma functions are closely related to Zeta functions, L-series and polylogarithms. See also q-functions for q-analogs of factorial-like functions.

Factorials

factorial()/fac()

mpmath.**factorial**(*x*, ***kwargs*)

Computes the factorial, $x!$. For integers $n \geq 0$, we have $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ and more generally the factorial is defined for real or complex x by $x! = \Gamma(x+1)$.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(6):
...     print("%s %s" % (k, fac(k)))
...
0 1.0
1 1.0
2 2.0
3 6.0
4 24.0
5 120.0
>>> fac(inf)
+inf
>>> fac(0.5), sqrt(pi)/2
(0.886226925452758, 0.886226925452758)
```

For large positive x, $x!$ can be approximated by Stirling's formula:

```
>>> x = 10**10
>>> fac(x)
2.32579620567308e+95657055186
>>> sqrt(2*pi*x)*(x/e)**x
2.32579597597705e+95657055186
```

`fac()` supports evaluation for astronomically large values:

```
>>> fac(10**30)
6.22311232304258e+2956570551809674817234887108109 8
```

Reciprocal factorials appear in the Taylor series of the exponential function (among many other contexts):

```
>>> nsum(lambda k: 1/fac(k), [0, inf]), exp(1)
(2.71828182845905, 2.71828182845905)
>>> nsum(lambda k: pi**k/fac(k), [0, inf]), exp(pi)
(23.1406926327793, 23.1406926327793)
```

fac2()

mpmath.**fac2**(*x*)

Computes the double factorial $x!!$, defined for integers $x > 0$ by

$$x!! = \begin{cases} 1 \cdot 3 \cdots (x-2) \cdot x & x \text{ odd} \\ 2 \cdot 4 \cdots (x-2) \cdot x & x \text{ even} \end{cases}$$

and more generally by [1]

$$x!! = 2^{x/2} \left(\frac{\pi}{2}\right)^{(\cos(\pi x)-1)/4} \Gamma\left(\frac{x}{2}+1\right).$$

Examples

The integer sequence of double factorials begins:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint([fac2(n) for n in range(10)])
[1.0, 1.0, 2.0, 3.0, 8.0, 15.0, 48.0, 105.0, 384.0, 945.0]
```

For large x, double factorials follow a Stirling-like asymptotic approximation:

```
>>> x = mpf(10000)
>>> fac2(x)
5.97272691416282e+17830
>>> sqrt(pi)*x**((x+1)/2)*exp(-x/2)
5.97262736954392e+17830
```

The recurrence formula $x!! = x(x-2)!!$ can be reversed to define the double factorial of negative odd integers (but not negative even integers):

```
>>> fac2(-1), fac2(-3), fac2(-5), fac2(-7)
(1.0, -1.0, 0.333333333333333, -0.0666666666666667)
>>> fac2(-2)
Traceback (most recent call last):
  ...
ValueError: gamma function pole
```

With the exception of the poles at negative even integers, *fac2()* supports evaluation for arbitrary complex arguments. The recurrence formula is valid generally:

```
>>> fac2(pi+2j)
(-1.3697207890154e-12 + 3.93665300979176e-12j)
>>> (pi+2j)*fac2(pi-2+2j)
(-1.3697207890154e-12 + 3.93665300979176e-12j)
```

Double factorials should not be confused with nested factorials, which are immensely larger:

```
>>> fac(fac(20))
5.13805976125208e+43675043585825292774
>>> fac2(20)
3715891200.0
```

Double factorials appear, among other things, in series expansions of Gaussian functions and the error function. Infinite series include:

```
>>> nsum(lambda k: 1/fac2(k), [0, inf])
3.05940740534258
>>> sqrt(e)*(1+sqrt(pi/2)*erf(sqrt(2)/2))
3.05940740534258
>>> nsum(lambda k: 2**k/fac2(2*k-1), [1, inf])
4.06015693855741
>>> e * erf(1) * sqrt(pi)
4.06015693855741
```

A beautiful Ramanujan sum:

```
>>> nsum(lambda k: (-1)**k*(fac2(2*k-1)/fac2(2*k))**3, [0,inf])
0.90917279454693
>>> (gamma('9/8')/gamma('5/4')/gamma('7/8'))**2
0.90917279454693
```

References

1. http://functions.wolfram.com/GammaBetaErf/Factorial2/27/01/0002/
2. http://mathworld.wolfram.com/DoubleFactorial.html

Binomial coefficients

binomial()

mpmath.**binomial**(*n*, *k*)

Computes the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

The binomial coefficient gives the number of ways that k items can be chosen from a set of n items. More generally, the binomial coefficient is a well-defined function of arbitrary real or complex n and k, via the gamma function.

Examples

Generate Pascal's triangle:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint([binomial(n,k) for k in range(n+1)])
...
[1.0]
[1.0, 1.0]
[1.0, 2.0, 1.0]
[1.0, 3.0, 3.0, 1.0]
[1.0, 4.0, 6.0, 4.0, 1.0]
```

There is 1 way to select 0 items from the empty set, and 0 ways to select 1 item from the empty set:

```
>>> binomial(0, 0)
1.0
>>> binomial(0, 1)
0.0
```

binomial() supports large arguments:

```
>>> binomial(10**20, 10**20-5)
8.33333333333333e+97
>>> binomial(10**20, 10**10)
2.60784095465201e+104342944813
```

Nonintegral binomial coefficients find use in series expansions:

```
>>> nprint(taylor(lambda x: (1+x)**0.25, 0, 4))
[1.0, 0.25, -0.09375, 0.0546875, -0.0375977]
>>> nprint([binomial(0.25, k) for k in range(5)])
[1.0, 0.25, -0.09375, 0.0546875, -0.0375977]
```

An integral representation:

```
>>> n, k = 5, 3
>>> f = lambda t: exp(-j*k*t)*(1+exp(j*t))**n
>>> chop(quad(f, [-pi,pi])/(2*pi))
10.0
>>> binomial(n,k)
10.0
```

Gamma function

gamma()

mpmath.**gamma**(*x*, ***kwargs*)

Computes the gamma function, $\Gamma(x)$. The gamma function is a shifted version of the ordinary factorial, satisfying $\Gamma(n) = (n-1)!$ for integers $n > 0$. More generally, it is defined by

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

for any real or complex x with $\Re(x) > 0$ and for $\Re(x) < 0$ by analytic continuation.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(1, 6):
...     print("%s %s" % (k, gamma(k)))
...
1 1.0
2 1.0
3 2.0
4 6.0
5 24.0
>>> gamma(inf)
+inf
>>> gamma(0)
Traceback (most recent call last):
  ...
ValueError: gamma function pole
```

The gamma function of a half-integer is a rational multiple of $\sqrt{\pi}$:

```
>>> gamma(0.5), sqrt(pi)
(1.77245385090552, 1.77245385090552)
>>> gamma(1.5), sqrt(pi)/2
(0.886226925452758, 0.886226925452758)
```

We can check the integral definition:

```
>>> gamma(3.5)
3.32335097044784
>>> quad(lambda t: t**2.5*exp(-t), [0,inf])
3.32335097044784
```

gamma() supports arbitrary-precision evaluation and complex arguments:

```
>>> mp.dps = 50
>>> gamma(sqrt(3))
0.91510229697308632046045539308226554038315280564184
>>> mp.dps = 25
>>> gamma(2j)
(0.00990244008092749098595506 - 0.07595200133501806872408048j)
```

Arguments can also be large. Note that the gamma function grows very quickly:

```
>>> mp.dps = 15
>>> gamma(10**20)
1.9328495143101e+1956570551809674817225
```

rgamma()

mpmath.**rgamma**(*x*, ***kwargs*)

Computes the reciprocal of the gamma function, $1/\Gamma(z)$. This function evaluates to zero at the poles of the gamma function, $z = 0, -1, -2, \ldots$.

Examples

Basic examples:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> rgamma(1)
1.0
>>> rgamma(4)
0.16666666666666666666666667
>>> rgamma(0); rgamma(-1)
0.0
0.0
>>> rgamma(1000)
2.485168143266784862783596e-2565
>>> rgamma(inf)
0.0
```

A definite integral that can be evaluated in terms of elementary integrals:

```
>>> quad(rgamma, [0,inf])
2.807770242028519365221501
>>> e + quad(lambda t: exp(-t)/(pi**2+log(t)**2), [0,inf])
2.807770242028519365221501
```

gammaprod()

mpmath.**gammaprod**(*a*, *b*)

Given iterables a and b, `gammaprod(a, b)` computes the product / quotient of gamma functions:

$$\frac{\Gamma(a_0)\Gamma(a_1)\cdots\Gamma(a_p)}{\Gamma(b_0)\Gamma(b_1)\cdots\Gamma(b_q)}$$

Unlike direct calls to *gamma()*, *gammaprod()* considers the entire product as a limit and evaluates this limit properly if any of the numerator or denominator arguments are nonpositive integers such that poles of the gamma function are encountered. That is, *gammaprod()* evaluates

$$\lim_{\epsilon\to 0} \frac{\Gamma(a_0+\epsilon)\Gamma(a_1+\epsilon)\cdots\Gamma(a_p+\epsilon)}{\Gamma(b_0+\epsilon)\Gamma(b_1+\epsilon)\cdots\Gamma(b_q+\epsilon)}$$

In particular:

- If there are equally many poles in the numerator and the denominator, the limit is a rational number times the remaining, regular part of the product.
- If there are more poles in the numerator, *gammaprod()* returns `+inf`.
- If there are more poles in the denominator, *gammaprod()* returns 0.

Examples

The reciprocal gamma function $1/\Gamma(x)$ evaluated at $x = 0$:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> gammaprod([], [0])
0.0
```

A limit:

```
>>> gammaprod([-4], [-3])
-0.25
>>> limit(lambda x: gamma(x-1)/gamma(x), -3, direction=1)
-0.25
>>> limit(lambda x: gamma(x-1)/gamma(x), -3, direction=-1)
-0.25
```

loggamma()

mpmath.**loggamma**(*x*)

Computes the principal branch of the log-gamma function, $\ln\Gamma(z)$. Unlike $\ln(\Gamma(z))$, which has infinitely many complex branch cuts, the principal log-gamma function only has a single branch cut along the negative half-axis. The principal branch continuously matches the asymptotic Stirling expansion

$$\ln\Gamma(z) \sim \frac{\ln(2\pi)}{2} + \left(z-\frac{1}{2}\right)\ln(z) - z + O(z^{-1}).$$

The real parts of both functions agree, but their imaginary parts generally differ by $2n\pi$ for some $n \in \mathbb{Z}$. They coincide for $z \in \mathbb{R}, z > 0$.

Computationally, it is advantageous to use *loggamma()* instead of *gamma()* for extremely large arguments.

Examples

Comparing with $\ln(\Gamma(z))$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> loggamma('13.2'); log(gamma('13.2'))
20.494004194566036784983940
20.494004194566036784983940
>>> loggamma(3+4j)
(-1.7566267846037841105306040 + 4.7426644380346579281948890j)
>>> log(gamma(3+4j))
(-1.7566267846037841105306040 - 1.5405208691449285487303970j)
>>> log(gamma(3+4j)) + 2*pi*j
(-1.7566267846037841105306040 + 4.7426644380346579281948890j)
```

Note the imaginary parts for negative arguments:

```
>>> loggamma(-0.5); loggamma(-1.5); loggamma(-2.5)
(1.2655121234846453964889460 - 3.1415926535897932384626430j)
(0.8600470153764810145109327 - 6.2831853071795864769252870j)
(-0.056243716497674050672594530 - 9.4247779607693797153879300j)
```

Some special values:

```
>>> loggamma(1); loggamma(2)
0.0
0.0
>>> loggamma(3); +ln2
0.69314718055994530941723210
0.69314718055994530941723210
>>> loggamma(3.5); log(15*sqrt(pi)/8)
1.2009736023470742248160220
1.2009736023470742248160220
>>> loggamma(inf)
+inf
```

Huge arguments are permitted:

```
>>> loggamma('1e30')
6.8077552789821370520539740e+31
>>> loggamma('1e300')
6.8977552789821370520539740e+302
>>> loggamma('1e3000')
6.9067552789821370520539740e+3003
>>> loggamma('1e10000000000000000000')
2.3025850929940456840079910e+10000000000000000020
>>> loggamma('1e30j')
(-1.5707963267948966192313220e+30 + 6.8077552789821370520539740e+31j)
>>> loggamma('1e300j')
(-1.5707963267948966192313220e+300 + 6.8977552789821370520539740e+302j)
>>> loggamma('1e3000j')
(-1.5707963267948966192313220e+3000 + 6.9067552789821370520539740e+3003j)
```

The log-gamma function can be integrated analytically on any interval of unit length:

```
>>> z = 0
>>> quad(loggamma, [z,z+1]); log(2*pi)/2
0.91893853320467274178032973
0.91893853320467274178032973
>>> z = 3+4j
>>> quad(loggamma, [z,z+1]); (log(z)-1)*z + log(2*pi)/2
(-0.961928601499475064131442109 + 5.219637303741238195688575j)
(-0.961928601499475064131442109 + 5.219637303741238195688575j)
```

The derivatives of the log-gamma function are given by the polygamma function (*psi()*):

```
>>> diff(loggamma, -4+3j); psi(0, -4+3j)
(1.6884935312229713936071 53 + 2.5548989113568069788927 48j)
(1.6884935312229713936071 53 + 2.5548989113568069788927 48j)
>>> diff(loggamma, -4+3j, 2); psi(1, -4+3j)
(-0.153941482921988237156 1038 - 0.102048519743026771974 6479j)
(-0.153941482921988237156 1038 - 0.102048519743026771974 6479j)
```

The log-gamma function satisfies an additive form of the recurrence relation for the ordinary gamma function:

```
>>> z = 2+3j
>>> loggamma(z); loggamma(z+1) - log(z)
(-2.092851753092733495641 89 + 2.302396543466867626153 708j)
(-2.092851753092733495641 89 + 2.302396543466867626153 708j)
```

Rising and falling factorials

rf()

mpmath.**rf**(*x*, *n*)

Computes the rising factorial or Pochhammer symbol,

$$x^{(n)} = x(x+1)\cdots(x+n-1) = \frac{\Gamma(x+n)}{\Gamma(x)}$$

where the rightmost expression is valid for nonintegral n.

Examples

For integral n, the rising factorial is a polynomial:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(taylor(lambda x: rf(x,n), 0, n))
...
[1.0]
[0.0, 1.0]
[0.0, 1.0, 1.0]
[0.0, 2.0, 3.0, 1.0]
[0.0, 6.0, 11.0, 6.0, 1.0]
```

Evaluation is supported for arbitrary arguments:

```
>>> rf(2+3j, 5.5)
(-7202.03920483347 - 3777.58810701527j)
```

ff()

mpmath.**ff**(*x*, *n*)

Computes the falling factorial,

$$(x)_n = x(x-1)\cdots(x-n+1) = \frac{\Gamma(x+1)}{\Gamma(x-n+1)}$$

where the rightmost expression is valid for nonintegral n.

Examples

For integral n, the falling factorial is a polynomial:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(taylor(lambda x: ff(x,n), 0, n))
...
[1.0]
[0.0, 1.0]
[0.0, -1.0, 1.0]
[0.0, 2.0, -3.0, 1.0]
[0.0, -6.0, 11.0, -6.0, 1.0]
```

Evaluation is supported for arbitrary arguments:

```
>>> ff(2+3j, 5.5)
(-720.41085888203 + 316.101124983878j)
```

Beta function

beta()

mpmath.**beta**(x, y)

Computes the beta function, $B(x,y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$. The beta function is also commonly defined by the integral representation

$$B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$$

Examples

For integer and half-integer arguments where all three gamma functions are finite, the beta function becomes either rational number or a rational multiple of π:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> beta(5, 2)
0.0333333333333333
>>> beta(1.5, 2)
0.266666666666667
>>> 16*beta(2.5, 1.5)
3.14159265358979
```

Where appropriate, *beta()* evaluates limits. A pole of the beta function is taken to result in `+inf`:

```
>>> beta(-0.5, 0.5)
0.0
>>> beta(-3, 3)
-0.333333333333333
>>> beta(-2, 3)
+inf
>>> beta(inf, 1)
0.0
>>> beta(inf, 0)
nan
```

beta() supports complex numbers and arbitrary precision evaluation:

```
>>> beta(1, 2+j)
(0.4 - 0.2j)
>>> mp.dps = 25
>>> beta(j,0.5)
(1.079424249270925780135675 - 1.410032405664160838288752j)
>>> mp.dps = 50
>>> beta(pi, e)
0.037890298781212201348153837138927165984170287886464
```

Various integrals can be computed by means of the beta function:

```
>>> mp.dps = 15
>>> quad(lambda t: t**2.5*(1-t)**2, [0, 1])
0.0230880230880231
>>> beta(3.5, 3)
0.0230880230880231
>>> quad(lambda t: sin(t)**4 * sqrt(cos(t)), [0, pi/2])
0.319504062596158
>>> beta(2.5, 0.75)/2
0.319504062596158
```

betainc()

mpmath.**betainc**(*a*, *b*, *x1=0*, *x2=1*, *regularized=False*)

`betainc(a, b, x1=0, x2=1, regularized=False)` gives the generalized incomplete beta function,

$$I_{x_1}^{x_2}(a,b) = \int_{x_1}^{x_2} t^{a-1}(1-t)^{b-1}dt.$$

When $x_1 = 0, x_2 = 1$, this reduces to the ordinary (complete) beta function $B(a,b)$; see *beta()*.

With the keyword argument `regularized=True`, *betainc()* computes the regularized incomplete beta function $I_{x_1}^{x_2}(a,b)/B(a,b)$. This is the cumulative distribution of the beta distribution with parameters a, b.

Examples

Verifying that *betainc()* computes the integral in the definition:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> x,y,a,b = 3, 4, 0, 6
>>> betainc(x, y, a, b)
-4010.4
>>> quad(lambda t: t**(x-1) * (1-t)**(y-1), [a, b])
-4010.4
```

The arguments may be arbitrary complex numbers:

```
>>> betainc(0.75, 1-4j, 0, 2+3j)
(0.2241657956955709603655887 + 0.3619619242700451992411724j)
```

With regularization:

```
>>> betainc(1, 2, 0, 0.25, regularized=True)
0.4375
>>> betainc(pi, e, 0, 1, regularized=True)   # Complete
1.0
```

The beta integral satisfies some simple argument transformation symmetries:

```
>>> mp.dps = 15
>>> betainc(2,3,4,5), -betainc(2,3,5,4), betainc(3,2,1-5,1-4)
(56.0833333333333, 56.0833333333333, 56.0833333333333)
```

The beta integral can often be evaluated analytically. For integer and rational arguments, the incomplete beta function typically reduces to a simple algebraic-logarithmic expression:

```
>>> mp.dps = 25
>>> identify(chop(betainc(0, 0, 3, 4)))
'-(log((9/8)))'
>>> identify(betainc(2, 3, 4, 5))
'(673/12)'
>>> identify(betainc(1.5, 1, 1, 2))
'((-12+sqrt(1152))/18)'
```

Super- and hyperfactorials

superfac()

mpmath.**superfac**(*z*)

Computes the superfactorial, defined as the product of consecutive factorials

$$\mathrm{sf}(n) = \prod_{k=1}^{n} k!$$

For general complex z, $\mathrm{sf}(z)$ is defined in terms of the Barnes G-function (see *barnesg()*).

Examples

The first few superfactorials are (OEIS A000178):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(10):
...     print("%s %s" % (n, superfac(n)))
...
0 1.0
1 1.0
2 2.0
3 12.0
4 288.0
5 34560.0
6 24883200.0
7 125411328000.0
8 5.05658474496e+15
9 1.83493347225108e+21
```

Superfactorials grow very rapidly:

```
>>> superfac(1000)
3.24570818422368e+1177245
>>> superfac(10**10)
2.61398543581249e+467427913956904067453
```

Evaluation is supported for arbitrary arguments:

```
>>> mp.dps = 25
>>> superfac(pi)
17.200515501212979852853 33
>>> superfac(2+3j)
(-0.00591548563319978962746 6468 + 0.00815644946460404494873 8263j)
>>> diff(superfac, 1)
0.26450720340160702056 73056
```

References

1. http://oeis.org/A000178

hyperfac()

mpmath.**hyperfac**(*z*)

Computes the hyperfactorial, defined for integers as the product

$$H(n) = \prod_{k=1}^{n} k^k.$$

The hyperfactorial satisfies the recurrence formula $H(z) = z^z H(z-1)$. It can be defined more generally in terms of the Barnes G-function (see *barnesg()*) and the gamma function by the formula

$$H(z) = \frac{\Gamma(z+1)^z}{G(z)}.$$

The extension to complex numbers can also be done via the integral representation

$$H(z) = (2\pi)^{-z/2} \exp\left[\binom{z+1}{2} + \int_0^z \log(t!)\,dt\right].$$

Examples

The rapidly-growing sequence of hyperfactorials begins (OEIS A002109):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(10):
...     print("%s %s" % (n, hyperfac(n)))
...
0 1.0
1 1.0
2 4.0
3 108.0
4 27648.0
5 86400000.0
6 4031078400000.0
7 3.3197663987712e+18
8 5.56964379417266e+25
9 2.15779412229419e+34
```

Some even larger hyperfactorials are:

```
>>> hyperfac(1000)
5.46458120882585e+1392926
>>> hyperfac(10**10)
4.60408207642219e+489142638002418704309
```

The hyperfactorial can be evaluated for arbitrary arguments:

```
>>> hyperfac(0.5)
0.880449235173423
>>> diff(hyperfac, 1)
0.581061466795327
>>> hyperfac(pi)
205.211134637462
>>> hyperfac(-10+1j)
(3.01144471378225e+46 - 2.45285242480185e+46j)
```

The recurrence property of the hyperfactorial holds generally:

```
>>> z = 3-4*j
>>> hyperfac(z)
(-4.49795891462086e-7 - 6.33262283196162e-7j)
>>> z**z * hyperfac(z-1)
(-4.49795891462086e-7 - 6.33262283196162e-7j)
>>> z = mpf(-0.6)
>>> chop(z**z * hyperfac(z-1))
1.28170142849352
>>> hyperfac(z)
1.28170142849352
```

The hyperfactorial may also be computed using the integral definition:

```
>>> z = 2.5
>>> hyperfac(z)
15.9842119922237
>>> (2*pi)**(-z/2)*exp(binomial(z+1,2) +
...     quad(lambda t: loggamma(t+1), [0, z]))
15.9842119922237
```

hyperfac() supports arbitrary-precision evaluation:

```
>>> mp.dps = 50
>>> hyperfac(10)
215779412229418562091680268288000000000000000.0
>>> hyperfac(1/sqrt(2))
0.89404818005227001975423476035729076375705084390942
```

References

1. http://oeis.org/A002109

2. http://mathworld.wolfram.com/Hyperfactorial.html

barnesg()

mpmath.**barnesg**(*z*)

Evaluates the Barnes G-function, which generalizes the superfactorial (*superfac()*) and by extension also the hyperfactorial (*hyperfac()*) to the complex numbers in an analogous way to how the gamma function generalizes the ordinary factorial.

The Barnes G-function may be defined in terms of a Weierstrass product:

$$G(z+1) = (2\pi)^{z/2} e^{-[z(z+1)+\gamma z^2]/2} \prod_{n=1}^{\infty} \left[\left(1+\frac{z}{n}\right)^n e^{-z+z^2/(2n)}\right]$$

For positive integers n, we have have relation to superfactorials $G(n) = \mathrm{sf}(n-2) = 0! \cdot 1! \cdots (n-2)!$.

Examples

Some elementary values and limits of the Barnes G-function:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> barnesg(1), barnesg(2), barnesg(3)
(1.0, 1.0, 1.0)
>>> barnesg(4)
2.0
>>> barnesg(5)
12.0
>>> barnesg(6)
288.0
>>> barnesg(7)
34560.0
>>> barnesg(8)
24883200.0
>>> barnesg(inf)
+inf
>>> barnesg(0), barnesg(-1), barnesg(-2)
(0.0, 0.0, 0.0)
```

Closed-form values are known for some rational arguments:

```
>>> barnesg('1/2')
0.603244281209446
>>> sqrt(exp(0.25+log(2)/12)/sqrt(pi)/glaisher**3)
0.603244281209446
>>> barnesg('1/4')
0.29375596533861
>>> nthroot(exp('3/8')/exp(catalan/pi)/
...      gamma(0.25)**3/sqrt(glaisher)**9, 4)
0.29375596533861
```

The Barnes G-function satisfies the functional equation $G(z+1) = \Gamma(z)G(z)$:

```
>>> z = pi
>>> barnesg(z+1)
2.39292119327948
>>> gamma(z)*barnesg(z)
2.39292119327948
```

The asymptotic growth rate of the Barnes G-function is related to the Glaisher-Kinkelin constant:

```
>>> limit(lambda n: barnesg(n+1)/(n**(n**2/2-mpf(1)/12)*
...     (2*pi)**(n/2)*exp(-3*n**2/4)), inf)
0.847536694177301
>>> exp('1/12')/glaisher
0.847536694177301
```

The Barnes G-function can be differentiated in closed form:

```
>>> z = 3
>>> diff(barnesg, z)
0.264507203401607
>>> barnesg(z)*((z-1)*psi(0,z)-z+(log(2*pi)+1)/2)
0.264507203401607
```

Evaluation is supported for arbitrary arguments and at arbitrary precision:

```
>>> barnesg(6.5)
2548.7457695685
>>> barnesg(-pi)
0.00535976768353037
>>> barnesg(3+4j)
(-0.000676375932234244 - 4.42236140124728e-5j)
>>> mp.dps = 50
>>> barnesg(1/sqrt(2))
0.81305501090451340843586085064413533788206204124732
>>> q = barnesg(10j)
>>> q.real
0.00000000002185236084035655724154303672479981237199585055223 4
>>> q.imag
-0.000000000007003533532006230484902065421554583905321004145 7588
>>> mp.dps = 15
>>> barnesg(100)
3.10361006263698e+6626
>>> barnesg(-101)
0.0
>>> barnesg(-10.5)
5.94463017605008e+25
>>> barnesg(-10000.5)
-6.14322868174828e+167480422
>>> barnesg(1000j)
(5.21133054865546e-1173597 + 4.27461836811016e-1173597j)
>>> barnesg(-1000+1000j)
(2.43114569750291e+1026623 + 2.24851410674842e+1026623j)
```

References

1. Whittaker & Watson, *A Course of Modern Analysis*, Cambridge University Press, 4th edition (1927), p.264
2. http://en.wikipedia.org/wiki/Barnes_G-function
3. http://mathworld.wolfram.com/BarnesG-Function.html

Polygamma functions and harmonic numbers

psi()/digamma()

mpmath.**psi**(*m*, *z*)

Gives the polygamma function of order m of z, $\psi^{(m)}(z)$. Special cases are known as the *digamma function* ($\psi^{(0)}(z)$), the *trigamma function* ($\psi^{(1)}(z)$), etc. The polygamma functions are defined as the logarithmic derivatives of the gamma function:

$$\psi^{(m)}(z) = \left(\frac{d}{dz}\right)^{m+1} \log \Gamma(z)$$

In particular, $\psi^{(0)}(z) = \Gamma'(z)/\Gamma(z)$. In the present implementation of `psi()`, the order m must be a nonnegative integer, while the argument z may be an arbitrary complex number (with exception for the polygamma function's poles at $z = 0, -1, -2, \ldots$).

Examples

For various rational arguments, the polygamma function reduces to a combination of standard mathematical constants:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> psi(0, 1), -euler
(-0.57721566490153286060065121, -0.57721566490153286060065121)
>>> psi(1, '1/4'), pi**2+8*catalan
(17.197329154507110739271320, 17.197329154507110739271320)
>>> psi(2, '1/2'), -14*apery
(-16.828796644234319995596330, -16.828796644234319995596330)
```

The polygamma functions are derivatives of each other:

```
>>> diff(lambda x: psi(3, x), pi), psi(4, pi)
(-0.11057493125788627345269520, -0.11057493125788627345269520)
>>> quad(lambda x: psi(4, x), [2, 3]), psi(3,3)-psi(3,2)
(-0.375, -0.375)
```

The digamma function diverges logarithmically as $z \to \infty$, while higher orders tend to zero:

```
>>> psi(0,inf), psi(1,inf), psi(2,inf)
(+inf, 0.0, 0.0)
```

Evaluation for a complex argument:

```
>>> psi(2, -1-2j)
(0.039024354053649526548384450 + 0.15743252404130299546853660j)
```

Evaluation is supported for large orders m and/or large arguments z:

```
>>> psi(3, 10**100)
2.0e-300
>>> psi(250, 10**30+10**20*j)
(-1.2931425043636426872048650e-7010 + 3.2328562609091073915131080e-7018j)
```

Application to infinite series

Any infinite series where the summand is a rational function of the index k can be evaluated in closed form in terms of polygamma functions of the roots and poles of the summand:

```
>>> a = sqrt(2)
>>> b = sqrt(3)
>>> nsum(lambda k: 1/((k+a)**2*(k+b)), [0, inf])
0.40496689275178570619175310
>>> (psi(0,a)-psi(0,b)-a*psi(1,a)+b*psi(1,a))/(a-b)**2
0.40496689275178570619175310
```

This follows from the series representation ($m > 0$)

$$\psi^{(m)}(z) = (-1)^{m+1} m! \sum_{k=0}^{\infty} \frac{1}{(z+k)^{m+1}}.$$

Since the roots of a polynomial may be complex, it is sometimes necessary to use the complex polygamma function to evaluate an entirely real-valued sum:

```
>>> nsum(lambda k: 1/(k**2-2*k+3), [0, inf])
1.6943614339070612561546650
>>> nprint(polyroots([1,-2,3]))
[(1.0 - 1.41421j), (1.0 + 1.41421j)]
>>> r1 = 1-sqrt(2)*j
>>> r2 = r1.conjugate()
>>> (psi(0,-r2)-psi(0,-r1))/(r1-r2)
(1.6943614339070612561546650 + 0.0j)
```

mpmath.**digamma**(*z*)
Shortcut for `psi(0,z)`.

harmonic()

mpmath.**harmonic**(*z*)
If n is an integer, `harmonic(n)` gives a floating-point approximation of the n-th harmonic number $H(n)$, defined as

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The first few harmonic numbers are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(8):
...     print("%s %s" % (n, harmonic(n)))
...
0 0.0
1 1.0
2 1.5
3 1.83333333333333
4 2.08333333333333
5 2.28333333333333
6 2.45
7 2.59285714285714
```

The infinite harmonic series $1 + 1/2 + 1/3 + \ldots$ diverges:

```
>>> harmonic(inf)
+inf
```

harmonic() is evaluated using the digamma function rather than by summing the harmonic series term by term. It can therefore be computed quickly for arbitrarily large n, and even for nonintegral arguments:

```
>>> harmonic(10**100)
230.835724964306
>>> harmonic(0.5)
0.613705638880109
>>> harmonic(3+4j)
(2.24757548223494 + 0.850502209186044j)
```

harmonic() supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> harmonic(11)
3.0198773448773448773448773448773448773448773448773
>>> harmonic(pi)
1.8727388590273302654363491032336134987519132374152
```

The harmonic series diverges, but at a glacial pace. It is possible to calculate the exact number of terms required before the sum exceeds a given amount, say 100:

```
>>> mp.dps = 50
>>> v = 10**findroot(lambda x: harmonic(10**x) - 100, 10)
>>> v
15092688622113788323693563264538101449859496.864101
>>> v = int(ceil(v))
```

```
>>> print(v)
150926886221137883236935632645381014498595 97
>>> harmonic(v-1)
99.999999999999999999999999999999999999999942747
>>> harmonic(v)
100.000000000000000000000000000000000000000000009
```

3.1.6 Exponential integrals and error functions

Exponential integrals give closed-form solutions to a large class of commonly occurring transcendental integrals that cannot be evaluated using elementary functions. Integrals of this type include those with an integrand of the form $t^a e^t$ or e^{-x^2}, the latter giving rise to the Gaussian (or normal) probability distribution.

The most general function in this section is the incomplete gamma function, to which all others can be reduced. The incomplete gamma function, in turn, can be expressed using hypergeometric functions (see Hypergeometric functions).

Incomplete gamma functions

gammainc()

mpmath.**gammainc**(*z*, *a=0*, *b=inf*, *regularized=False*)

`gammainc(z, a=0, b=inf)` computes the (generalized) incomplete gamma function with integration limits $[a, b]$:

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

The generalized incomplete gamma function reduces to the following special cases when one or both endpoints are fixed:

- $\Gamma(z,0,\infty)$ is the standard ("complete") gamma function, $\Gamma(z)$ (available directly as the mpmath function *gamma()*)
- $\Gamma(z,a,\infty)$ is the "upper" incomplete gamma function, $\Gamma(z,a)$
- $\Gamma(z,0,b)$ is the "lower" incomplete gamma function, $\gamma(z,b)$.

Of course, we have $\Gamma(z,0,x) + \Gamma(z,x,\infty) = \Gamma(z)$ for all z and x.

Note however that some authors reverse the order of the arguments when defining the lower and upper incomplete gamma function, so one should be careful to get the correct definition.

If also given the keyword argument `regularized=True`, *gammainc()* computes the "regularized" incomplete gamma function

$$P(z,a,b) = \frac{\Gamma(z,a,b)}{\Gamma(z)}.$$

Examples

We can compare with numerical quadrature to verify that *gammainc()* computes the integral in the definition:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> gammainc(2+3j, 4, 10)
(0.00977212668627705160602312 - 0.0770637306312989892451977j)
>>> quad(lambda t: t**(2+3j-1) * exp(-t), [4, 10])
(0.00977212668627705160602312 - 0.0770637306312989892451977j)
```

Argument symmetries follow directly from the integral definition:

```
>>> gammainc(3, 4, 5) + gammainc(3, 5, 4)
0.0
>>> gammainc(3,0,2) + gammainc(3,2,4); gammainc(3,0,4)
1.523793388892911312363331
1.523793388892911312363331
>>> findroot(lambda z: gammainc(2,z,3), 1)
3.0
```

Evaluation for arbitrarily large arguments:

```
>>> gammainc(10, 100)
4.083660630910611272288592e-26
>>> gammainc(10, 10000000000000000)
5.2904024499011747529724 86e-4342944819032375
>>> gammainc(3+4j, 1000000+1000000j)
(-1.257913707524362408877881e-434284 + 2.556691003883483531962095e-434284j)
```

Evaluation of a generalized incomplete gamma function automatically chooses the representation that gives a more accurate result, depending on which parameter is larger:

```
>>> gammainc(10000000, 3) - gammainc(10000000, 2)    # Bad
0.0
>>> gammainc(10000000, 2, 3)    # Good
1.755146243738946045873491e+4771204
>>> gammainc(2, 0, 100000001) - gammainc(2, 0, 100000000)    # Bad
0.0
>>> gammainc(2, 100000000, 100000001)    # Good
4.078258353474186729184421e-43429441
```

The incomplete gamma functions satisfy simple recurrence relations:

```
>>> mp.dps = 25
>>> z, a = mpf(3.5), mpf(2)
>>> gammainc(z+1, a); z*gammainc(z,a) + a**z*exp(-a)
10.601302969335334592673290
10.601302969335334592673290
>>> gammainc(z+1,0,a); z*gammainc(z,0,a) - a**z*exp(-a)
1.030425427232114336470932
1.030425427232114336470932
```

Evaluation at integers and poles:

```
>>> gammainc(-3, -4, -5)
(-0.22145770489677985662341 92 + 0.0j)
>>> gammainc(-3, 0, 5)
+inf
```

If z is an integer, the recurrence reduces the incomplete gamma function to $P(a)\exp(-a) + Q(b)\exp(-b)$ where P and Q are polynomials:

```
>>> gammainc(1, 2); exp(-2)
0.13533528323661269189399950
0.13533528323661269189399950
>>> mp.dps = 50
>>> identify(gammainc(6, 1, 2), ['exp(-1)', 'exp(-2)'])
'(326*exp(-1) + (-872)*exp(-2))'
```

The incomplete gamma functions reduce to functions such as the exponential integral Ei and the error function for special arguments:

```
>>> mp.dps = 25
>>> gammainc(0, 4); -ei(-4)
0.00377935240984890647887486
0.00377935240984890647887486
>>> gammainc(0.5, 0, 2); sqrt(pi)*erf(sqrt(2))
1.691806732945198336509541
1.691806732945198336509541
```

Exponential integrals

ei()

mpmath.**ei**(*x*, ***kwargs*)

Computes the exponential integral or Ei-function, $\mathrm{Ei}(x)$. The exponential integral is defined as

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t}\, dt.$$

When the integration range includes $t = 0$, the exponential integral is interpreted as providing the Cauchy principal value.

For real x, the Ei-function behaves roughly like $\mathrm{Ei}(x) \approx \exp(x) + \log(|x|)$.

The Ei-function is related to the more general family of exponential integral functions denoted by E_n, which are available as *expint()*.

Basic examples

Some basic values and limits are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> ei(0)
-inf
>>> ei(1)
1.89511781635594
>>> ei(inf)
+inf
>>> ei(-inf)
0.0
```

For $x < 0$, the defining integral can be evaluated numerically as a reference:

```
>>> ei(-4)
-0.00377935240984891
>>> quad(lambda t: exp(t)/t, [-inf, -4])
-0.00377935240984891
```

ei() supports complex arguments and arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> ei(pi)
10.928374389331410348638445906907535171566338835056
>>> mp.dps = 25
>>> ei(3+4j)
(-4.1540916516426898225353 59 + 4.2944186200243574769855 35j)
```

Related functions

The exponential integral is closely related to the logarithmic integral. See *li()* for additional information.

The exponential integral is related to the hyperbolic and trigonometric integrals (see *chi()*, *shi()*, *ci()*, *si()*) similarly to how the ordinary exponential function is related to the hyperbolic and trigonometric functions:

```
>>> mp.dps = 15
>>> ei(3)
9.93383257062542
>>> chi(3) + shi(3)
9.93383257062542
>>> chop(ci(3j) - j*si(3j) - pi*j/2)
9.93383257062542
```

Beware that logarithmic corrections, as in the last example above, are required to obtain the correct branch in general. For details, see [1].

The exponential integral is also a special case of the hypergeometric function ${}_2F_2$:

```
>>> z = 0.6
>>> z*hyper([1,1],[2,2],z) + (ln(z)-ln(1/z))/2 + euler
0.769881289937359
>>> ei(z)
0.769881289937359
```

References

1.Relations between Ei and other functions: http://functions.wolfram.com/GammaBetaErf/ExpIntegralEi/27/01/

2.Abramowitz & Stegun, section 5: http://people.math.sfu.ca/~cbm/aands/page_228.htm

3.Asymptotic expansion for Ei: http://mathworld.wolfram.com/En-Function.html

e1()

mpmath.**e1**(*x*, ***kwargs*)

Computes the exponential integral $\mathrm{E}_1(z)$, given by

$$\mathrm{E}_1(z) = \int_z^\infty \frac{e^{-t}}{t} dt.$$

This is equivalent to *expint()* with $n = 1$.

Examples

Two ways to evaluate this function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> e1(6.25)
0.0002704758872637179088496194
>>> expint(1,6.25)
0.0002704758872637179088496194
```

The E1-function is essentially the same as the Ei-function (*ei()*) with negated argument, except for an imaginary branch cut term:

```
>>> e1(2.5)
0.02491491787026973549562801
>>> -ei(-2.5)
0.02491491787026973549562801
```

```
>>> e1(-2.5)
(-7.07376589457860071192355 2 - 3.14159265358979323846243j)
>>> -ei(2.5)
-7.073765894578600711923552
```

expint()

mpmath.**expint**(**args*)

`expint(n,z)()` gives the generalized exponential integral or En-function,

$$\mathrm{E}_n(z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt,$$

where n and z may both be complex numbers. The case with $n = 1$ is also given by *e1()*.

Examples

Evaluation at real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> expint(1, 6.25)
0.00027047588724637179088496194
>>> expint(-3, 2+3j)
(0.00299658467335472929656159 + 0.06100816202125885450319632j)
>>> expint(2+3j, 4-5j)
(0.001803529474663565056945248 - 0.002235061547756185403349091j)
```

At negative integer values of n, $E_n(z)$ reduces to a rational-exponential function:

```
>>> f = lambda n, z: fac(n)*sum(z**k/fac(k-1) for k in range(1,n+2))/\
...     exp(z)/z**(n+2)
>>> n = 3
>>> z = 1/pi
>>> expint(-n,z)
584.2604820613019908668219
>>> f(n,z)
584.2604820613019908668219
>>> n = 5
>>> expint(-n,z)
115366.5762594725451811138
>>> f(n,z)
115366.5762594725451811138
```

Logarithmic integral

li()

mpmath.**li**(*x*, ***kwargs*)

Computes the logarithmic integral or li-function $\mathrm{li}(x)$, defined by

$$\mathrm{li}(x) = \int_0^x \frac{1}{\log t} dt$$

The logarithmic integral has a singularity at $x = 1$.

Alternatively, `li(x, offset=True)` computes the offset logarithmic integral (used in number theory)

$$\mathrm{Li}(x) = \int_2^x \frac{1}{\log t}\,dt.$$

These two functions are related via the simple identity $\mathrm{Li}(x) = \mathrm{li}(x) - \mathrm{li}(2)$.

The logarithmic integral should also not be confused with the polylogarithm (also denoted by Li), which is implemented as *polylog()*.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> li(0)
0.0
>>> li(1)
-inf
>>> li(1)
-inf
>>> li(2)
1.04516378011749278484458888919
>>> findroot(li, 2)
1.45136923488338105028396848589
>>> li(inf)
+inf
>>> li(2, offset=True)
0.0
>>> li(1, offset=True)
-inf
>>> li(0, offset=True)
-1.04516378011749278484458888919
>>> li(10, offset=True)
5.12043572466980515267839286347
```

The logarithmic integral can be evaluated for arbitrary complex arguments:

```
>>> mp.dps = 20
>>> li(3+4j)
(3.1343755504645775265 + 2.6769247817778742392j)
```

The logarithmic integral is related to the exponential integral:

```
>>> ei(log(3))
2.1635885946671919729
>>> li(3)
2.1635885946671919729
```

The logarithmic integral grows like $O(x/\log(x))$:

```
>>> mp.dps = 15
>>> x = 10**100
>>> x/log(x)
4.34294481903252e+97
>>> li(x)
4.3619719871407e+97
```

The prime number theorem states that the number of primes less than x is asymptotic to $\mathrm{Li}(x)$ (equivalently $\mathrm{li}(x)$). For example, it is known that there are exactly 1,925,320,391,606,803,968,923 prime numbers less than 10^{23} [1]. The logarithmic integral provides a very accurate estimate:

```
>>> li(10**23, offset=True)
1.92532039161405e+21
```

A definite integral is:

```
>>> quad(li, [0, 1])
-0.693147180559945
>>> -ln(2)
-0.693147180559945
```

References

1. http://mathworld.wolfram.com/PrimeCountingFunction.html

2. http://mathworld.wolfram.com/LogarithmicIntegral.html

Trigonometric integrals

ci()

mpmath.**ci**(*x*, ***kwargs*)

Computes the cosine integral,

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

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ci(0)
-inf
>>> ci(1)
0.3374039229009681346626462
>>> ci(pi)
0.07366791204642548599010096
>>> ci(inf)
0.0
>>> ci(-inf)
(0.0 + 3.141592653589793238462643j)
>>> ci(2+3j)
(1.408292501520849518759125 - 2.983617742029605093121118j)
```

The cosine integral behaves roughly like the sinc function (see *sinc()*) for large real x:

```
>>> ci(10**10)
-4.87506025174822653785729e-11
>>> sinc(10**10)
-4.87506025087510691527794e-11
>>> chop(limit(ci, inf))
0.0
```

It has infinitely many roots on the positive real axis:

```
>>> findroot(ci, 1)
0.6165054856207162337971104
>>> findroot(ci, 2)
3.384180422551186426397851
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> ci(10**6*(1+j))
(4.4494105876110357249843 76e+434287 + 9.7574487429001352641705 9e+434287j)
```

We can evaluate the defining integral as a reference:

```
>>> mp.dps = 15
>>> -quadosc(lambda t: cos(t)/t, [5, inf], omega=1)
-0.190029749656644
>>> ci(5)
-0.190029749656644
```

Some infinite series can be evaluated using the cosine integral:

```
>>> nsum(lambda k: (-1)**k/(fac(2*k)*(2*k)), [1,inf])
-0.239811742000565
>>> ci(1) - euler
-0.239811742000565
```

si()

mpmath.**si**(*x*, ***kwargs*)

Computes the sine integral,

$$\mathrm{Si}(x) = \int_0^x \frac{\sin t}{t}\, dt.$$

The sine integral is thus the antiderivative of the sinc function (see *sinc()*).

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> si(0)
0.0
>>> si(1)
0.9460830703671830149413533
>>> si(-1)
-0.9460830703671830149413533
>>> si(pi)
1.851937051982466170361053
>>> si(inf)
1.570796326794896619231322
>>> si(-inf)
-1.570796326794896619231322
>>> si(2+3j)
(4.547513889562289219853204 + 1.399196580646054789459839j)
```

The sine integral approaches $\pi/2$ for large real x:

```
>>> si(10**10)
1.570796326707584656968511
>>> pi/2
1.570796326794896619231322
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> si(10**6*(1+j))
(-9.75744874290013526417059e+434287 + 4.44941058761103572498437 6e+434287j)
```

We can evaluate the defining integral as a reference:

```
>>> mp.dps = 15
>>> quad(sinc, [0, 5])
1.54993124494467
>>> si(5)
1.54993124494467
```

Some infinite series can be evaluated using the sine integral:

```
>>> nsum(lambda k: (-1)**k/(fac(2*k+1)*(2*k+1)), [0,inf])
0.946083070367183
>>> si(1)
0.946083070367183
```

Hyperbolic integrals

chi()

mpmath.**chi**(*x*, ***kwargs*)

Computes the hyperbolic cosine integral, defined in analogy with the cosine integral (see *ci()*) as

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

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> chi(0)
-inf
>>> chi(1)
0.8378669409802082408946786
>>> chi(inf)
+inf
>>> findroot(chi, 0.5)
0.5238225713898644064509583
>>> chi(2+3j)
(-0.16836286832772046624293 21 + 2.6251158804513250021516 88j)
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> chi(10**6*(1+j))
(4.44941058761103572498437 6e+434287 - 9.75744874290013526417059e+434287j)
```

shi()

mpmath.**shi**(*x*, ***kwargs*)

Computes the hyperbolic sine integral, defined in analogy with the sine integral (see *si()*) as

$$\mathrm{Shi}(x) = \int_0^x \frac{\sinh t}{t}\,dt.$$

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> shi(0)
0.0
>>> shi(1)
1.057250875375728514571842
>>> shi(-1)
-1.057250875375728514571842
>>> shi(inf)
+inf
>>> shi(2+3j)
(-0.19318907627191982916780 95 + 2.64543255536236962481852 5j)
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> shi(10**6*(1+j))
(4.44941058761103572498437 6e+434287 - 9.75744874290013526417059e+434287j)
```

Error functions

erf()

mpmath.**erf**(*x*, ***kwargs*)

Computes the error function, $\mathrm{erf}(x)$. The error function is the normalized antiderivative of the Gaussian function $\exp(-t^2)$. More precisely,

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt$$

Basic examples

Simple values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> erf(0)
0.0
>>> erf(1)
0.842700792949715
>>> erf(-1)
-0.842700792949715
>>> erf(inf)
1.0
>>> erf(-inf)
-1.0
```

For large real x, $\mathrm{erf}(x)$ approaches 1 very rapidly:

```
>>> erf(3)
0.999977909503001
>>> erf(5)
0.999999999998463
```

The error function is an odd function:

```
>>> nprint(chop(taylor(erf, 0, 5)))
[0.0, 1.12838, 0.0, -0.376126, 0.0, 0.112838]
```

erf() implements arbitrary-precision evaluation and supports complex numbers:

```
>>> mp.dps = 50
>>> erf(0.5)
0.52049987781304653768274665389196452873645157575796
>>> mp.dps = 25
>>> erf(1+j)
(1.316151281697947644880271 + 0.1904534692378346862841089j)
```

Evaluation is supported for large arguments:

```
>>> mp.dps = 25
>>> erf('1e1000')
1.0
>>> erf('-1e1000')
-1.0
>>> erf('1e-1000')
1.128379167095512573896159e-1000
>>> erf('1e7j')
(0.0 + 8.593897639029319267398803e+43429448190317j)
>>> erf('1e7+1e7j')
(0.9999999858172446172631323 + 3.728805278735270407053139e-8j)
```

Related functions

See also *erfc()*, which is more accurate for large x, and *erfi()* which gives the antiderivative of $\exp(t^2)$.

The Fresnel integrals *fresnels()* and *fresnelc()* are also related to the error function.

erfc()

mpmath.**erfc**(*x*, ***kwargs*)

Computes the complementary error function, $\operatorname{erfc}(x) = 1-\operatorname{erf}(x)$. This function avoids cancellation that occurs when naively computing the complementary error function as `1-erf(x)`:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> 1 - erf(10)
0.0
>>> erfc(10)
2.08848758376254e-45
```

erfc() works accurately even for ludicrously large arguments:

```
>>> erfc(10**10)
4.3504398860243e-43429448190325182776
```

Complex arguments are supported:

```
>>> erfc(500+50j)
(1.19739830969552e-107492 + 1.46072418957528e-107491j)
```

erfi()

mpmath.**erfi**(*x*)

Computes the imaginary error function, $\operatorname{erfi}(x)$. The imaginary error function is defined in analogy with the

error function, but with a positive sign in the integrand:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(t^2)\, dt$$

Whereas the error function rapidly converges to 1 as x grows, the imaginary error function rapidly diverges to infinity. The functions are related as $\operatorname{erfi}(x) = -i \operatorname{erf}(ix)$ for all complex numbers x.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> erfi(0)
0.0
>>> erfi(1)
1.65042575879754
>>> erfi(-1)
-1.65042575879754
>>> erfi(inf)
+inf
>>> erfi(-inf)
-inf
```

Note the symmetry between erf and erfi:

```
>>> erfi(3j)
(0.0 + 0.999977909503001j)
>>> erf(3)
0.999977909503001
>>> erf(1+2j)
(-0.536643565778565 - 5.04914370344703j)
>>> erfi(2+1j)
(-5.04914370344703 - 0.536643565778565j)
```

Large arguments are supported:

```
>>> erfi(1000)
1.71130938718796e+434291
>>> erfi(10**10)
7.3167287567024e+43429448190325182754
>>> erfi(-10**10)
-7.3167287567024e+43429448190325182754
>>> erfi(1000-500j)
(2.49895233563961e+325717 + 2.6846779342253e+325717j)
>>> erfi(100000j)
(0.0 + 1.0j)
>>> erfi(-100000j)
(0.0 - 1.0j)
```

erfinv()

mpmath.**erfinv**(*x*)

Computes the inverse error function, satisfying

$$\operatorname{erf}(\operatorname{erfinv}(x)) = \operatorname{erfinv}(\operatorname{erf}(x)) = x.$$

This function is defined only for $-1 \le x \le 1$.

Examples

Special values include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> erfinv(0)
0.0
>>> erfinv(1)
+inf
>>> erfinv(-1)
-inf
```

The domain is limited to the standard interval:

```
>>> erfinv(2)
Traceback (most recent call last):
  ...
ValueError: erfinv(x) is defined only for -1 <= x <= 1
```

It is simple to check that *erfinv()* computes inverse values of *erf()* as promised:

```
>>> erf(erfinv(0.75))
0.75
>>> erf(erfinv(-0.995))
-0.995
```

erfinv() supports arbitrary-precision evaluation:

```
>>> mp.dps = 50
>>> x = erf(2)
>>> x
0.99532226501895273416206925636725292861089179704006
>>> erfinv(x)
2.0
```

A definite integral involving the inverse error function:

```
>>> mp.dps = 15
>>> quad(erfinv, [0, 1])
0.564189583547756
>>> 1/sqrt(pi)
0.564189583547756
```

The inverse error function can be used to generate random numbers with a Gaussian distribution (although this is a relatively inefficient algorithm):

```
>>> nprint([erfinv(2*rand()-1) for n in range(6)])
[-0.586747, 1.10233, -0.376796, 0.926037, -0.708142, -0.732012]
```

The normal distribution

npdf()

mpmath.**npdf**(*x*, *mu=0*, *sigma=1*)

`npdf(x, mu=0, sigma=1)` evaluates the probability density function of a normal distribution with mean value μ and variance σ^2.

Elementary properties of the probability distribution can be verified using numerical integration:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> quad(npdf, [-inf, inf])
1.0
>>> quad(lambda x: npdf(x, 3), [3, inf])
0.5
>>> quad(lambda x: npdf(x, 3, 2), [3, inf])
0.5
```

See also *ncdf()*, which gives the cumulative distribution.

ncdf()

mpmath.**ncdf**(*x*, *mu=0*, *sigma=1*)

`ncdf(x, mu=0, sigma=1)` evaluates the cumulative distribution function of a normal distribution with mean value μ and variance σ^2.

See also *npdf()*, which gives the probability density.

Elementary properties include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> ncdf(pi, mu=pi)
0.5
>>> ncdf(-inf)
0.0
>>> ncdf(+inf)
1.0
```

The cumulative distribution is the integral of the density function having identical mu and sigma:

```
>>> mp.dps = 15
>>> diff(ncdf, 2)
0.053990966513188
>>> npdf(2)
0.053990966513188
>>> diff(lambda x: ncdf(x, 1, 0.5), 0)
0.107981933026376
>>> npdf(0, 1, 0.5)
0.107981933026376
```

Fresnel integrals

fresnels()

mpmath.**fresnels**(*x*)

Computes the Fresnel sine integral

$$S(x) = \int_0^x \sin\left(\frac{\pi t^2}{2}\right)\,dt$$

Note that some sources define this function without the normalization factor $\pi/2$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> fresnels(0)
0.0
>>> fresnels(inf)
0.5
>>> fresnels(-inf)
-0.5
>>> fresnels(1)
0.43825914739035476607675 67
>>> fresnels(1+2j)
(36.725464883991438428387 88 + 15.587751104404587327482 79j)
```

Comparing with the definition:

```
>>> fresnels(3)
0.49631299896737503609761 23
>>> quad(lambda t: sin(pi*t**2/2), [0,3])
0.49631299896737503609761 23
```

fresnelc()

mpmath.**fresnelc**(*x*)

Computes the Fresnel cosine integral

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right) dt$$

Note that some sources define this function without the normalization factor $\pi/2$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> fresnelc(0)
0.0
>>> fresnelc(inf)
0.5
>>> fresnelc(-inf)
-0.5
>>> fresnelc(1)
0.77989340037682282947420 64
>>> fresnelc(1+2j)
(16.087871374125480417294 89 - 36.225687992881650215787 58j)
```

Comparing with the definition:

```
>>> fresnelc(3)
0.60572078929768562955616 11
>>> quad(lambda t: cos(pi*t**2/2), [0,3])
0.60572078929768562955616 11
```

3.1.7 Bessel functions and related functions

The functions in this section arise as solutions to various differential equations in physics, typically describing wavelike oscillatory behavior or a combination of oscillation and exponential decay or growth. Mathematically, they are special

cases of the confluent hypergeometric functions ${}_0F_1$, ${}_1F_1$ and ${}_1F_2$ (see Hypergeometric functions).

Bessel functions

besselj()

mpmath.**besselj**(*n*, *x*, *derivative=0*)

besselj(n, x, derivative=0) gives the Bessel function of the first kind $J_n(x)$. Bessel functions of the first kind are defined as solutions of the differential equation

$$x^2y'' + xy' + (x^2 - n^2)y = 0$$

which appears, among other things, when solving the radial part of Laplace's equation in cylindrical coordinates. This equation has two solutions for given n, where the J_n-function is the solution that is nonsingular at $x = 0$. For positive integer n, $J_n(x)$ behaves roughly like a sine (odd n) or cosine (even n) multiplied by a magnitude factor that decays slowly as $x \to \pm\infty$.

Generally, J_n is a special case of the hypergeometric function ${}_0F_1$:

$$J_n(x) = \frac{x^n}{2^n\Gamma(n+1)}\,{}_0F_1\left(n+1, -\frac{x^2}{4}\right)$$

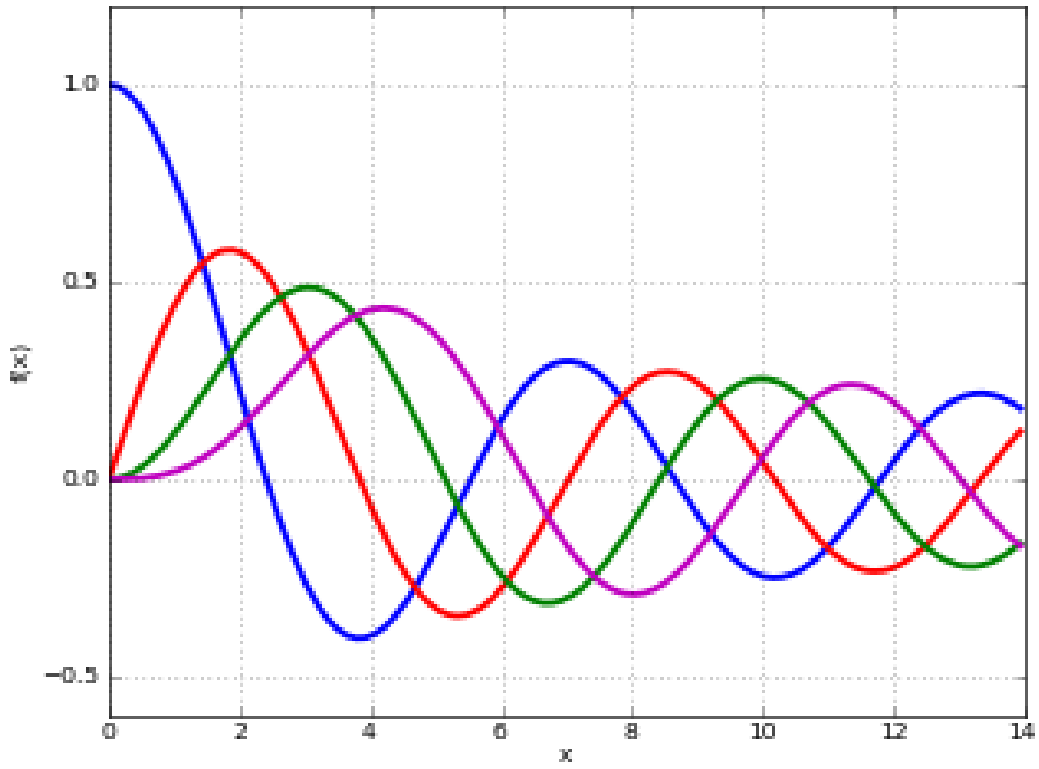
With *derivative* = $m \neq 0$, the m-th derivative

$$\frac{d^m}{dx^m}J_n(x)$$

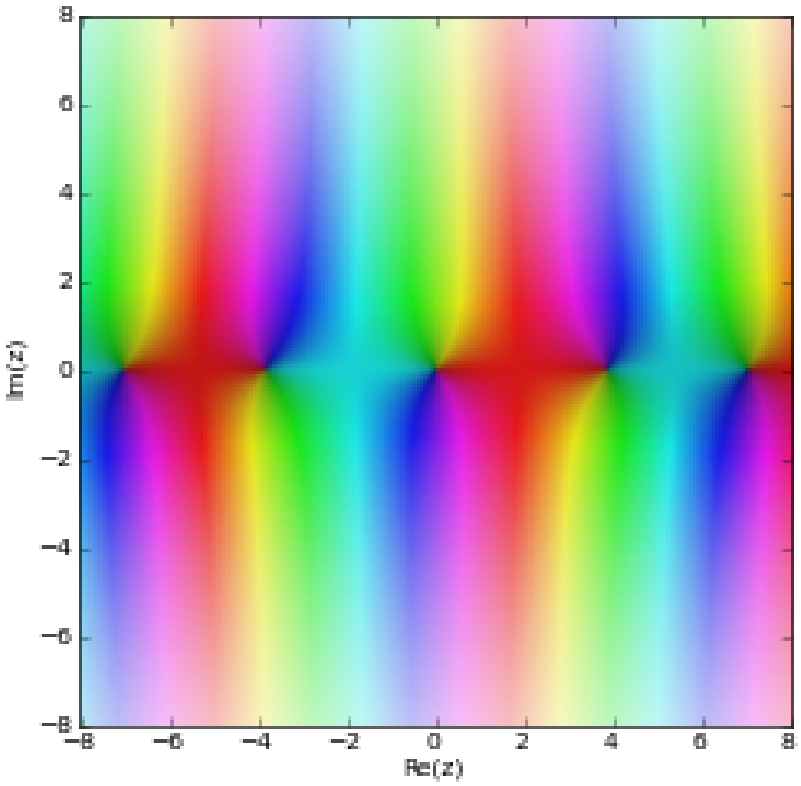
is computed.

Plots

```
# Bessel function J_n(x) on the real line for n=0,1,2,3
j0 = lambda x: besselj(0,x)
j1 = lambda x: besselj(1,x)
j2 = lambda x: besselj(2,x)
j3 = lambda x: besselj(3,x)
plot([j0,j1,j2,j3],[0,14])
```




```
    # Bessel function J_n(z) in the complex plane
    cplot(lambda z: besselj(1,z), [-8,8], [-8,8], points=50000)
```



Examples

Evaluation is supported for arbitrary arguments, and at arbitrary precision:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> besselj(2, 1000)
-0.024777229528606
>>> besselj(4, 0.75)
0.000801070086542314
>>> besselj(2, 1000j)
(-2.48071721019185e+432 + 6.41567059811949e-437j)
>>> mp.dps = 25
>>> besselj(0.75j, 3+4j)
(-2.778118364828153309919653 - 1.586360388901862158553 3j)
>>> mp.dps = 50
>>> besselj(1, pi)
0.28461534317975275734531059968613140570981118184947
```

Arguments may be large:

```
>>> mp.dps = 25
>>> besselj(0, 10000)
-0.0070961603533888014772651 64
>>> besselj(0, 10**10)
0.000002175591750246891726859055
>>> besselj(2, 10**100)
7.337048736538615712436929e-51
>>> besselj(2, 10**5*j)
```

```
(-3.5407254119709488601737 35e+43426 + 4.4949812409615803110051e-43433j)
```

The Bessel functions of the first kind satisfy simple symmetries around $x = 0$:

```
>>> mp.dps = 15
>>> nprint([besselj(n,0) for n in range(5)])
[1.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint([besselj(n,pi) for n in range(5)])
[-0.304242, 0.284615, 0.485434, 0.333458, 0.151425]
>>> nprint([besselj(n,-pi) for n in range(5)])
[-0.304242, -0.284615, 0.485434, -0.333458, 0.151425]
```

Roots of Bessel functions are often used:

```
>>> nprint([findroot(j0, k) for k in [2, 5, 8, 11, 14]])
[2.40483, 5.52008, 8.65373, 11.7915, 14.9309]
>>> nprint([findroot(j1, k) for k in [3, 7, 10, 13, 16]])
[3.83171, 7.01559, 10.1735, 13.3237, 16.4706]
```

The roots are not periodic, but the distance between successive roots asymptotically approaches 2π. Bessel functions of the first kind have the following normalization:

```
>>> quadosc(j0, [0, inf], period=2*pi)
1.0
>>> quadosc(j1, [0, inf], period=2*pi)
1.0
```

For $n = 1/2$ or $n = -1/2$, the Bessel function reduces to a trigonometric function:

```
>>> x = 10
>>> besselj(0.5, x), sqrt(2/(pi*x))*sin(x)
(-0.13726373575505, -0.13726373575505)
>>> besselj(-0.5, x), sqrt(2/(pi*x))*cos(x)
(-0.211708866331398, -0.211708866331398)
```

Derivatives of any order can be computed (negative orders correspond to integration):

```
>>> mp.dps = 25
>>> besselj(0, 7.5, 1)
-0.1352484275797055051822405
>>> diff(lambda x: besselj(0,x), 7.5)
-0.1352484275797055051822405
>>> besselj(0, 7.5, 10)
-0.1377811164763244890135677
>>> diff(lambda x: besselj(0,x), 7.5, 10)
-0.1377811164763244890135677
>>> besselj(0,7.5,-1) - besselj(0,3.5,-1)
-0.1241343240399987693521378
>>> quad(j0, [3.5, 7.5])
-0.1241343240399987693521378
```

Differentiation with a noninteger order gives the fractional derivative in the sense of the Riemann-Liouville differintegral, as computed by *differint()*:

```
>>> mp.dps = 15
>>> besselj(1, 3.5, 0.75)
-0.385977722939384
>>> differint(lambda x: besselj(1, x), 3.5, 0.75)
-0.385977722939384
```

mpmath.**j0**(*x*)
Computes the Bessel function $J_0(x)$. See *besselj()*.

mpmath.**j1**(*x*)
Computes the Bessel function $J_1(x)$. See *besselj()*.

bessely()

mpmath.**bessely**(*n*, *x*, *derivative=0*)
`bessely(n, x, derivative=0)` gives the Bessel function of the second kind,

$$Y_n(x) = \frac{J_n(x)\cos(\pi n) - J_{-n}(x)}{\sin(\pi n)}.$$

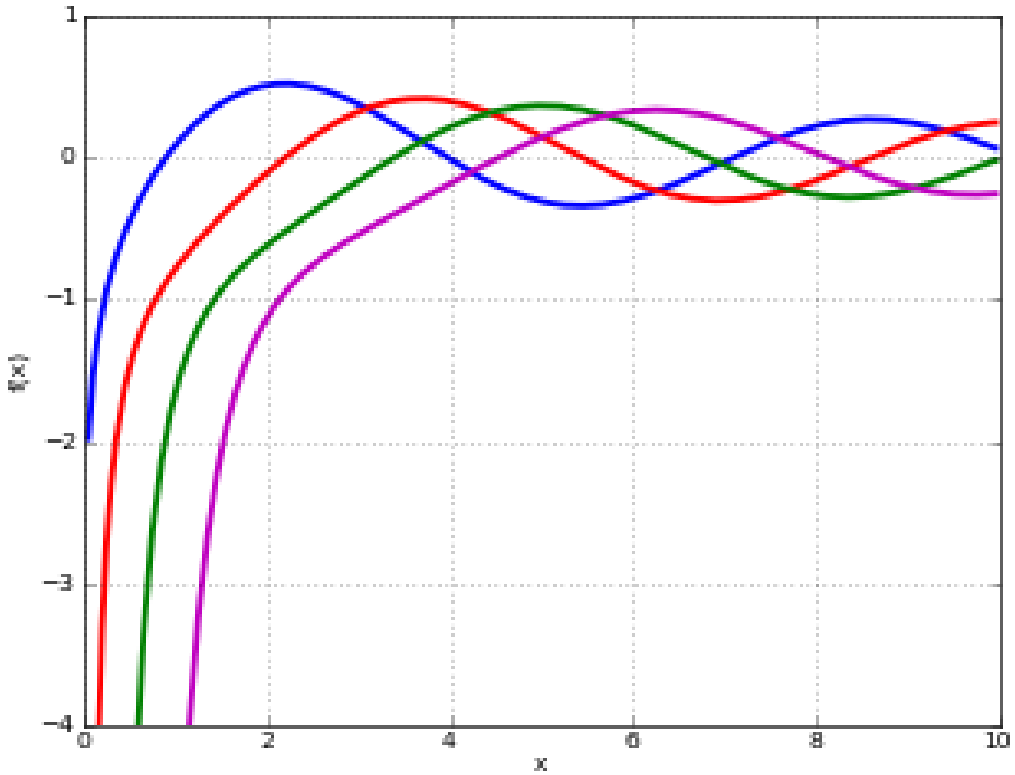
For n an integer, this formula should be understood as a limit. With *derivative* = $m \neq 0$, the m-th derivative

$$\frac{d^m}{dx^m} Y_n(x)$$

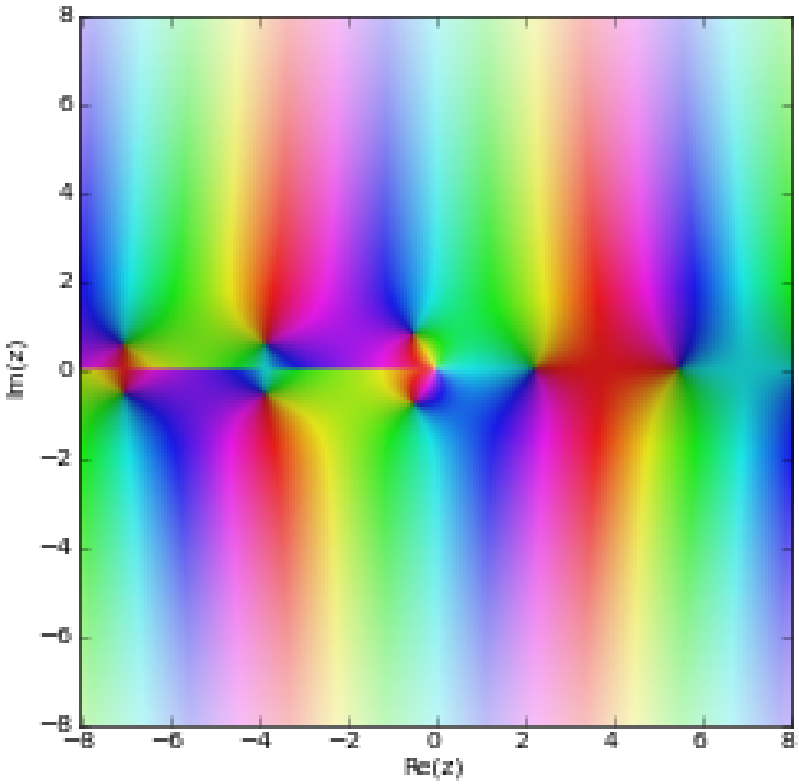
is computed.

Plots

```
# Bessel function of 2nd kind Y_n(x) on the real line for n=0,1,2,3
y0 = lambda x: bessely(0,x)
y1 = lambda x: bessely(1,x)
y2 = lambda x: bessely(2,x)
y3 = lambda x: bessely(3,x)
plot([y0,y1,y2,y3],[0,10],[-4,1])
```



```
    # Bessel function of 2nd kind Y_n(z) in the complex plane
    cplot(lambda z: bessely(1,z), [-8,8], [-8,8], points=50000)
```



Examples

Some values of $Y_n(x)$:

```
    >>> from mpmath import *
    >>> mp.dps = 25; mp.pretty = True
    >>> bessely(0,0), bessely(1,0), bessely(2,0)
    (-inf, -inf, -inf)
    >>> bessely(1, pi)
    0.3588729167767189594679827
    >>> bessely(0.5, 3+4j)
    (9.242861436961450520325216 - 3.085042824915332562522402j)
```

Arguments may be large:

```
    >>> bessely(0, 10000)
    0.00364780555898660588668872
    >>> bessely(2.5, 10**50)
    -4.8952500412050989295774e-26
    >>> bessely(2.5, -10**50)
    (0.0 + 4.8952500412050989295774e-26j)
```

Derivatives and antiderivatives of any order can be computed:

```
    >>> bessely(2, 3.5, 1)
    0.38426188204226600660089231
```

```
>>> diff(lambda x: bessely(2, x), 3.5)
0.38426188204226600660089231
>>> bessely(0.5, 3.5, 1)
-0.20665983041567643379004l7
>>> diff(lambda x: bessely(0.5, x), 3.5)
-0.20665983041567643379004l7
>>> diff(lambda x: bessely(2, x), 0.5, 10)
-208173867409.5547350101511
>>> bessely(2, 0.5, 10)
-208173867409.5547350101511
>>> bessely(2, 100.5, 100)
0.026684875473013723348490l3
>>> quad(lambda x: bessely(2,x), [1,3])
-1.3770468590931819692l3262
>>> bessely(2,3,-1) - bessely(2,1,-1)
-1.3770468590931819692l3262
```

besseli()

mpmath.**besseli**(*n*, *x*, *derivative=0*)

`besseli(n, x, derivative=0)` gives the modified Bessel function of the first kind,

$$I_n(x) = i^{-n} J_n(ix).$$

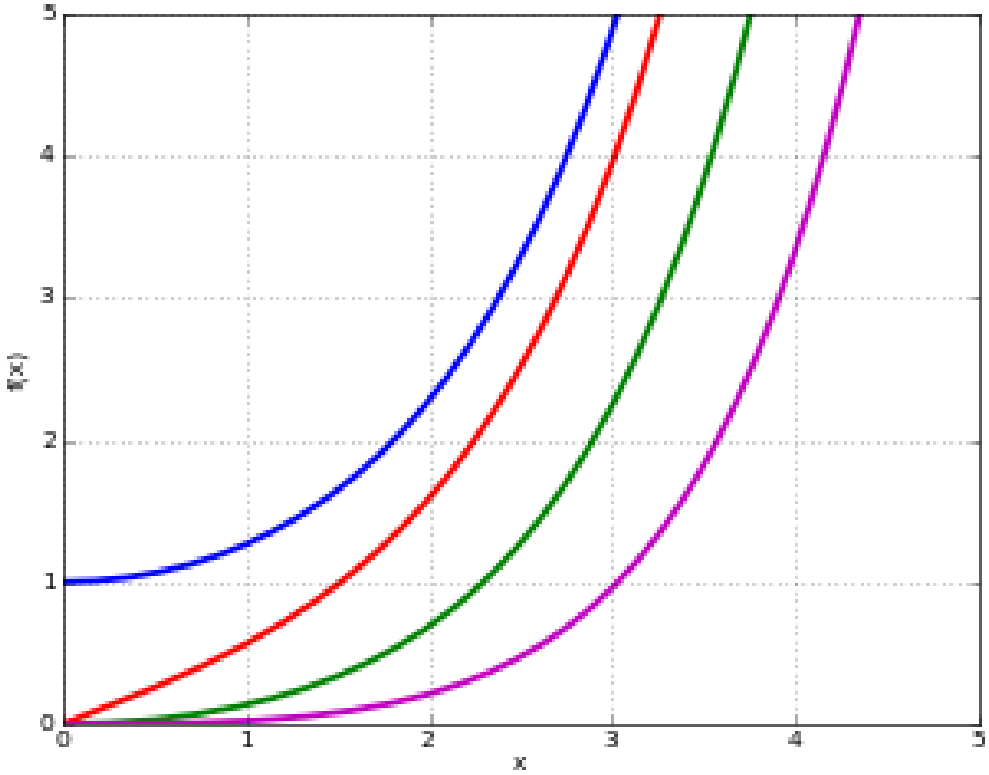
With *derivative* = $m \neq 0$, the m-th derivative

$$\frac{d^m}{dx^m} I_n(x)$$

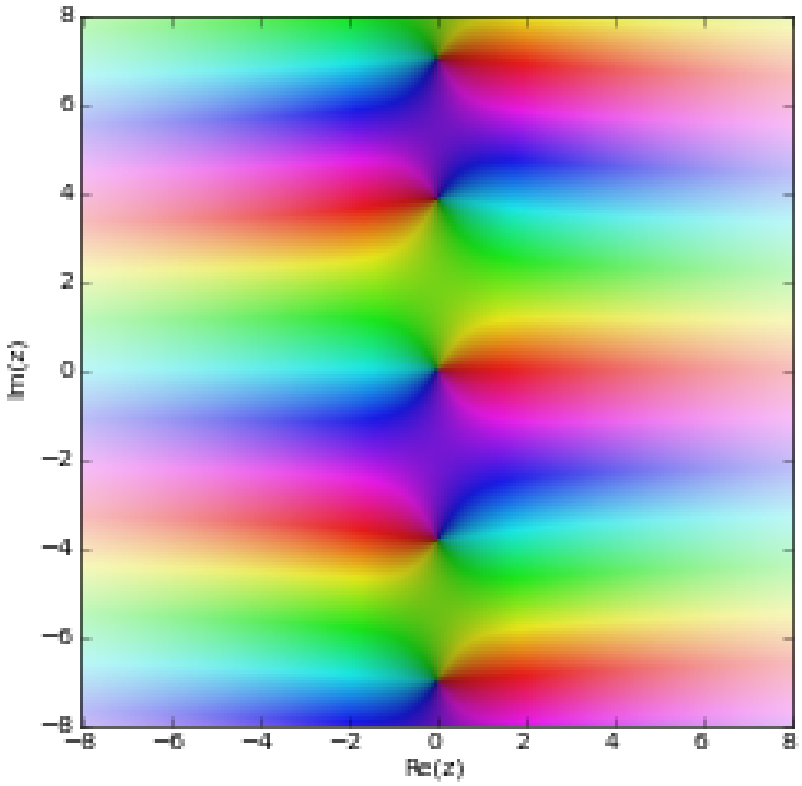
is computed.

Plots

```
# Modified Bessel function I_n(x) on the real line for n=0,1,2,3
i0 = lambda x: besseli(0,x)
i1 = lambda x: besseli(1,x)
i2 = lambda x: besseli(2,x)
i3 = lambda x: besseli(3,x)
plot([i0,i1,i2,i3],[0,5],[0,5])
```




```
    # Modified Bessel function I_n(z) in the complex plane
    cplot(lambda z: besseli(1,z), [-8,8], [-8,8], points=50000)
```



Examples

Some values of $I_n(x)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besseli(0,0)
1.0
>>> besseli(1,0)
0.0
>>> besseli(0,1)
1.266065877752008335598245
>>> besseli(3.5, 2+3j)
(-0.2904369752642538144289025 - 0.4469098397654815837307006j)
```

Arguments may be large:

```
>>> besseli(2, 1000)
2.480717210191852440616782e+432
>>> besseli(2, 10**10)
4.299602851624027900335391e+4342944813
>>> besseli(2, 6000+10000j)
(-2.114650753239580827144204e+2603 + 4.385040221241629041351886e+2602j)
```

For integers n, the following integral representation holds:

```
>>> mp.dps = 15
>>> n = 3
>>> x = 2.3
```

```
>>> quad(lambda t: exp(x*cos(t))*cos(n*t), [0,pi])/pi
0.349223221159309
>>> besseli(n,x)
0.349223221159309
```

Derivatives and antiderivatives of any order can be computed:

```
>>> mp.dps = 25
>>> besseli(2, 7.5, 1)
195.82290389313990625658830
>>> diff(lambda x: besseli(2,x), 7.5)
195.82290389313990625658830
>>> besseli(2, 7.5, 10)
153.3296508971734525525176
>>> diff(lambda x: besseli(2,x), 7.5, 10)
153.3296508971734525525176
>>> besseli(2,7.5,-1) - besseli(2,3.5,-1)
202.5043900051930141956876
>>> quad(lambda x: besseli(2,x), [3.5, 7.5])
202.5043900051930141956876
```

besselk()

mpmath.**besselk**(*n*, *x*)

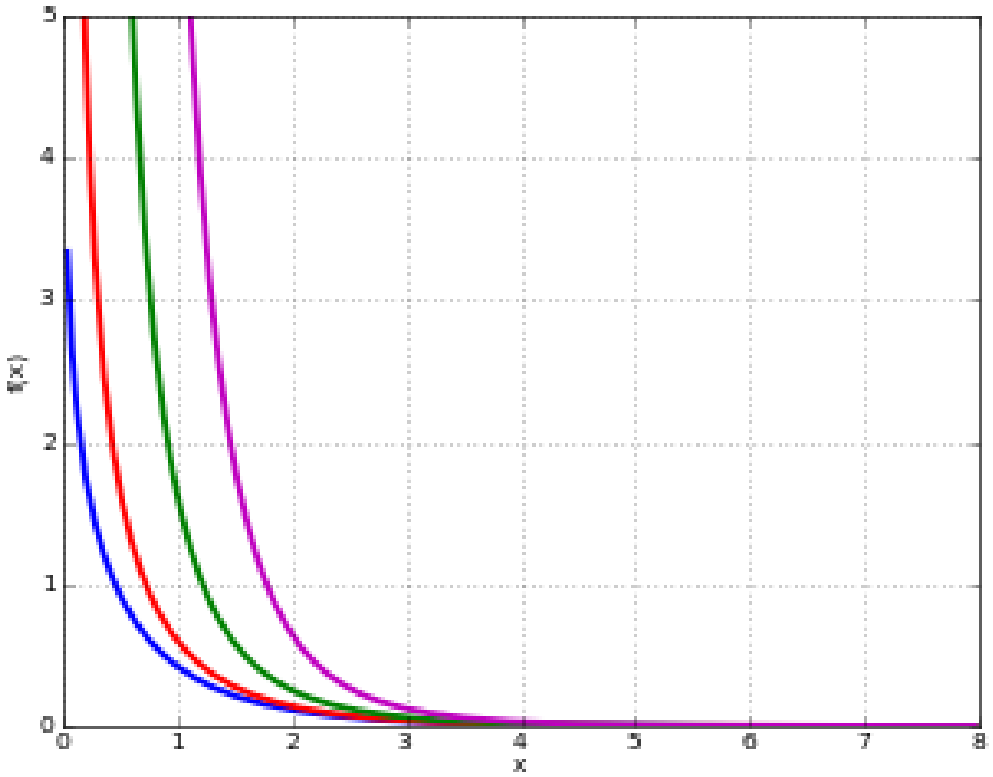
`besselk(n, x)` gives the modified Bessel function of the second kind,

$$K_n(x) = \frac{\pi}{2} \frac{I_{-n}(x) - I_n(x)}{\sin(\pi n)}$$

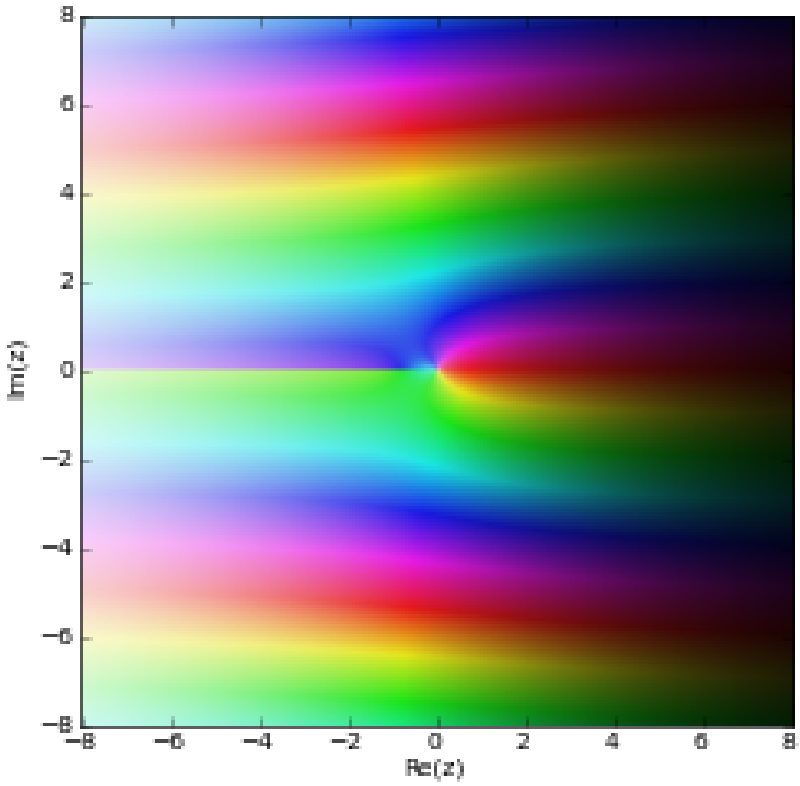
For n an integer, this formula should be understood as a limit.

Plots

```
# Modified Bessel function of 2nd kind K_n(x) on the real line for n=0,1,2,3
k0 = lambda x: besselk(0,x)
k1 = lambda x: besselk(1,x)
k2 = lambda x: besselk(2,x)
k3 = lambda x: besselk(3,x)
plot([k0,k1,k2,k3],[0,8],[0,5])
```




```
    # Modified Bessel function of 2nd kind K_n(z) in the complex plane
    cplot(lambda z: besselk(1,z), [-8,8], [-8,8], points=50000)
```



Examples

Evaluation is supported for arbitrary complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besselk(0,1)
0.4210244382407083333356274
>>> besselk(0, -1)
(0.4210244382407083333356274 - 3.977463260506422637256 61j)
>>> besselk(3.5, 2+3j)
(-0.02090732889633760668464128 + 0.2464022641351420167819697j)
>>> besselk(2+3j, 0.5)
(0.9615816021726349402626083 + 0.1918250181801757416908224j)
```

Arguments may be large:

```
>>> besselk(0, 100)
4.656628229175902018939005e-45
>>> besselk(1, 10**6)
4.131967049321725588398296e-434298
>>> besselk(1, 10**6*j)
(0.001140348428252385844876706 - 0.000520001720168115290900096 1j)
>>> besselk(4.5, fmul(10**50, j, exact=True))
(1.561034538142413947789221e-26 + 1.243554598118700063281496e-25j)
```

The point $x = 0$ is a singularity (logarithmic if $n = 0$):

```
>>> besselk(0,0)
+inf
>>> besselk(1,0)
+inf
>>> for n in range(-4, 5):
...     print(besselk(n, '1e-1000'))
...
4.8e+4001
8.0e+3000
2.0e+2000
1.0e+1000
2302.701024509704096466802
1.0e+1000
2.0e+2000
8.0e+3000
4.8e+4001
```

Bessel function zeros

besseljzero()

mpmath.**besseljzero**(*v*, *m*, *derivative=0*)

For a real order $\nu \geq 0$ and a positive integer m, returns $j_{\nu,m}$, the m-th positive zero of the Bessel function of the first kind $J_\nu(z)$ (see *besselj()*). Alternatively, with *derivative=1*, gives the first nonnegative simple zero $j'_{\nu,m}$ of $J'_\nu(z)$.

The indexing convention is that used by Abramowitz & Stegun and the DLMF. Note the special case $j'_{0,1} = 0$, while all other zeros are positive. In effect, only simple zeros are counted (all zeros of Bessel functions are simple except possibly $z = 0$) and $j_{\nu,m}$ becomes a monotonic function of both ν and m.

The zeros are interlaced according to the inequalities

$$j'_{\nu,k} < j_{\nu,k} < j'_{\nu,k+1}$$
$$j_{\nu,1} < j_{\nu+1,2} < j_{\nu,2} < j_{\nu+1,2} < j_{\nu,3} < \cdots$$

Examples

Initial zeros of the Bessel functions $J_0(z), J_1(z), J_2(z)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besseljzero(0,1); besseljzero(0,2); besseljzero(0,3)
2.404825557695772768621632
5.520078110286310649596604
8.653727912911012216954199
>>> besseljzero(1,1); besseljzero(1,2); besseljzero(1,3)
3.831705970207512315614436
7.015586669815618753537 05
10.173468135062722077185 71
>>> besseljzero(2,1); besseljzero(2,2); besseljzero(2,3)
5.135622301840682556301402
8.417244140399864857783614
11.619841172149059427094 15
```

Initial zeros of $J'_0(z), J'_1(z), J'_2(z)$:

```
0.0
3.831705970207512315614436
7.015586669815618753537 05
>>> besseljzero(1,1,1); besseljzero(1,2,1); besseljzero(1,3,1)
1.841183781340659302643 63
5.331442773525032636884016
8.536316366346285834358961
>>> besseljzero(2,1,1); besseljzero(2,2,1); besseljzero(2,3,1)
3.054236928227140322755932
6.706133194158459146634394
9.969467823087595793179143
```

Zeros with large index:

```
>>> besseljzero(0,100); besseljzero(0,1000); besseljzero(0,10000)
313.3742660775278447196902
3140.807295225078628895545
31415.14114171350798533666
>>> besseljzero(5,100); besseljzero(5,1000); besseljzero(5,10000)
321.1893195676003157339222
3148.657306813047523500494
31422.9947255486291798943
>>> besseljzero(0,100,1); besseljzero(0,1000,1); besseljzero(0,10000,1)
311.8018681873704508125112
3139.236339643802482833973
31413.57032947022399485808
```

Zeros of functions with large order:

```
>>> besseljzero(50,1)
57.11689916011917411936228
>>> besseljzero(50,2)
62.80769876483536093435393
>>> besseljzero(50,100)
388.6936600656058834640981
>>> besseljzero(50,1,1)
52.99764038731665010944037
>>> besseljzero(50,2,1)
60.02631933279942589882363
>>> besseljzero(50,100,1)
387.1083151608726181086283
```

Zeros of functions with fractional order:

```
>>> besseljzero(0.5,1); besseljzero(1.5,1); besseljzero(2.25,4)
3.14159265358979323846264 3
4.49340945790906417530788 1
15.15657692957458622921634
```

Both $J_\nu(z)$ and $J'_\nu(z)$ can be expressed as infinite products over their zeros:

```
>>> v,z = 2, mpf(1)
>>> (z/2)**v/gamma(v+1) * \
...     nprod(lambda k: 1-(z/besseljzero(v,k))**2, [1,inf])
...
0.11490348493190048046964 69
>>> besselj(v,z)
0.11490348493190048046964 69
>>> (z/2)**(v-1)/2/gamma(v) * \
...     nprod(lambda k: 1-(z/besseljzero(v,k,1))**2, [1,inf])
```

```
    ...
    0.21024361588113255502038840
    >>> besselj(v,z,1)
    0.21024361588113255502038840
```

besselyzero()

mpmath.**besselyzero**(*v*, *m*, *derivative=0*)

For a real order $\nu \geq 0$ and a positive integer m, returns $y_{\nu,m}$, the m-th positive zero of the Bessel function of the second kind $Y_\nu(z)$ (see `bessely()`). Alternatively, with *derivative=1*, gives the first positive zero $y'_{\nu,m}$ of $Y'_\nu(z)$.

The zeros are interlaced according to the inequalities

$$y_{\nu,k} < y'_{\nu,k} < y_{\nu,k+1}$$

$$y_{\nu,1} < y_{\nu+1,2} < y_{\nu,2} < y_{\nu+1,2} < y_{\nu,3} < \cdots$$

Examples

Initial zeros of the Bessel functions $Y_0(z), Y_1(z), Y_2(z)$:

```
    >>> from mpmath import *
    >>> mp.dps = 25; mp.pretty = True
    >>> besselyzero(0,1); besselyzero(0,2); besselyzero(0,3)
    0.8935769662791675215848871
    3.9576784193148578683756770
    7.0860510603017726976233625
    >>> besselyzero(1,1); besselyzero(1,2); besselyzero(1,3)
    2.1971413260310170351490340
    5.4296810407941351327720050
    8.5960058683311689264296060
    >>> besselyzero(2,1); besselyzero(2,2); besselyzero(2,3)
    3.3842417671495934727014260
    6.7938075132682675382911670
    10.023477979360037978505390
```

Initial zeros of $Y'_0(z), Y'_1(z), Y'_2(z)$:

```
    >>> besselyzero(0,1,1); besselyzero(0,2,1); besselyzero(0,3,1)
    2.1971413260310170351490340
    5.4296810407941351327720050
    8.5960058683311689264296060
    >>> besselyzero(1,1,1); besselyzero(1,2,1); besselyzero(1,3,1)
    3.6830228565851776989896710
    6.9414999536541756557519440
    10.123404655436613079787750
    >>> besselyzero(2,1,1); besselyzero(2,2,1); besselyzero(2,3,1)
    5.0025829314460639452001760
    8.3507247014130795263497140
    11.574195465217646546242650
```

Zeros with large index:

```
    >>> besselyzero(0,100); besselyzero(0,1000); besselyzero(0,10000)
    311.80347176018715493334190
    3139.2364989181980067940260
    31413.570345386912052291880
    >>> besselyzero(5,100); besselyzero(5,1000); besselyzero(5,10000)
```

```
    319.61833385627821562350 62
    3147.086508524556404473186
    31421.42392920214673402828
    >>> besselyzero(0,100,1); besselyzero(0,1000,1); besselyzero(0,10000,1)
    313.372670542635934505044 9
    3140.807136030340213610065
    31415.141125797615782201 75
```

Zeros of functions with large order:

```
    >>> besselyzero(50,1)
    53.50285882040036394680237
    >>> besselyzero(50,2)
    60.112444427740581146860 22
    >>> besselyzero(50,100)
    387.1096509824943957706835
    >>> besselyzero(50,1,1)
    56.962904275167513200636 05
    >>> besselyzero(50,2,1)
    62.748881669459339440366 23
    >>> besselyzero(50,100,1)
    388.6923300548309258355475
```

Zeros of functions with fractional order:

```
    >>> besselyzero(0.5,1); besselyzero(1.5,1); besselyzero(2.25,4)
    1.570796326794896619231322
    2.798386045783887136720249
    13.567212087707351233760 18
```

Hankel functions

hankel1()

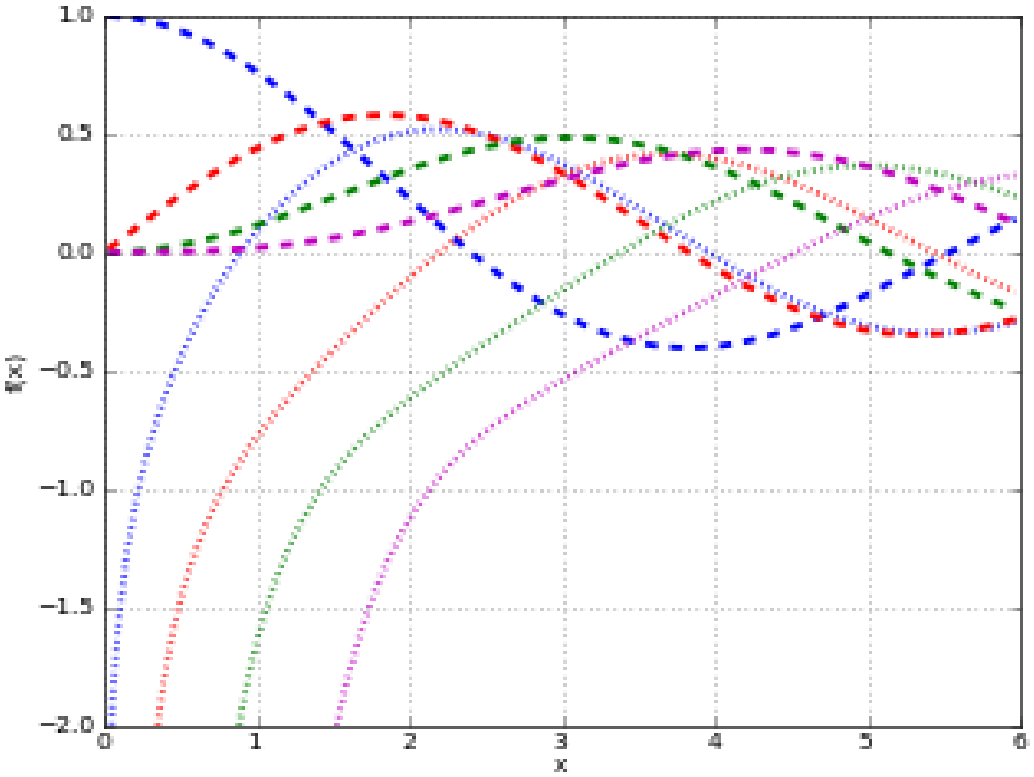
mpmath.**hankel1**(n, x)

hankel1(n,x) computes the Hankel function of the first kind, which is the complex combination of Bessel functions given by

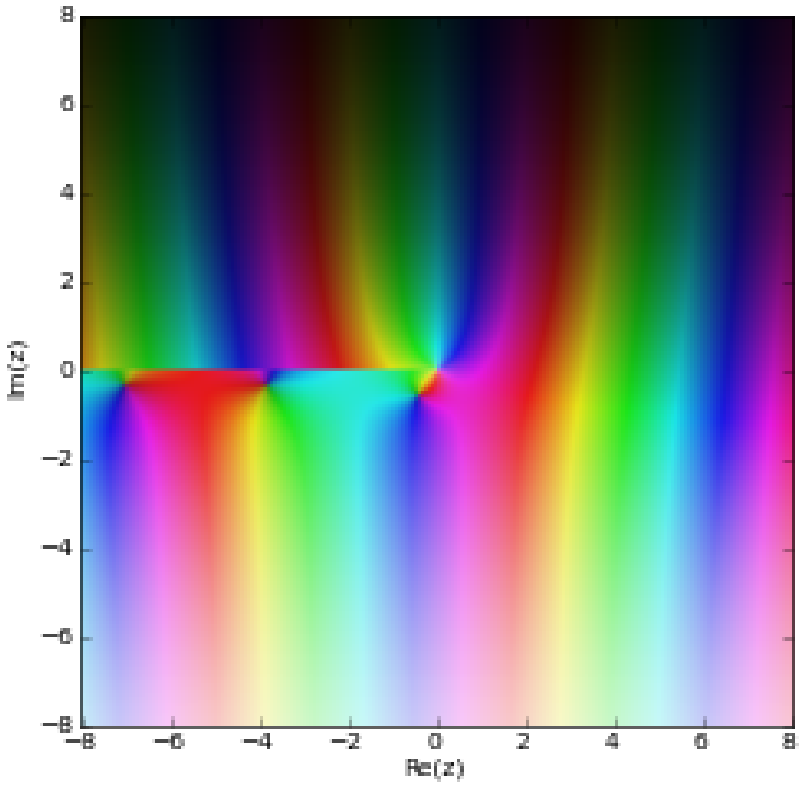
$$H_n^{(1)}(x) = J_n(x) + iY_n(x).$$

Plots

```
    # Hankel function H1_n(x) on the real line for n=0,1,2,3
    h0 = lambda x: hankel1(0,x)
    h1 = lambda x: hankel1(1,x)
    h2 = lambda x: hankel1(2,x)
    h3 = lambda x: hankel1(3,x)
    plot([h0,h1,h2,h3],[0,6],[-2,1])
```




```
    # Hankel function H1_n(z) in the complex plane
    cplot(lambda z: hankel1(1,z), [-8,8], [-8,8], points=50000)
```



Examples

The Hankel function is generally complex-valued:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hankel1(2, pi)
(0.4854339326315091097054957 - 0.099007139290278787734903j)
>>> hankel1(3.5, pi)
(0.2340002029630507922628888 - 0.6419643823412927142424049j)
```

hankel2()

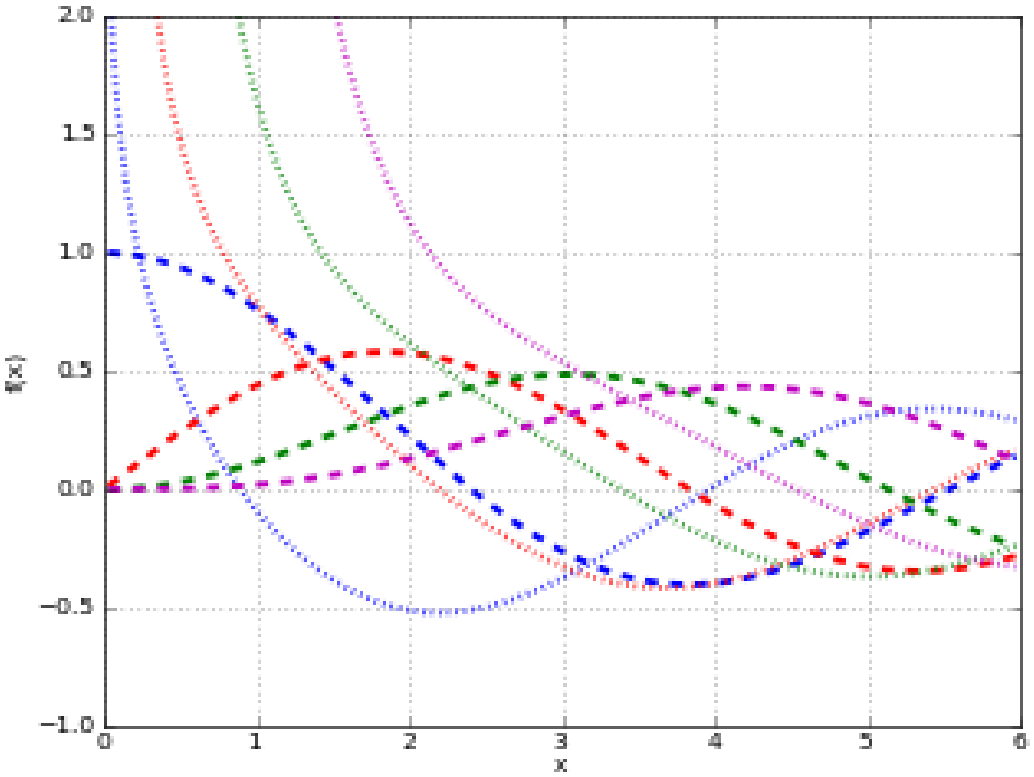
mpmath.**hankel2**(*n*, *x*)

`hankel2(n,x)` computes the Hankel function of the second kind, which is the complex combination of Bessel functions given by

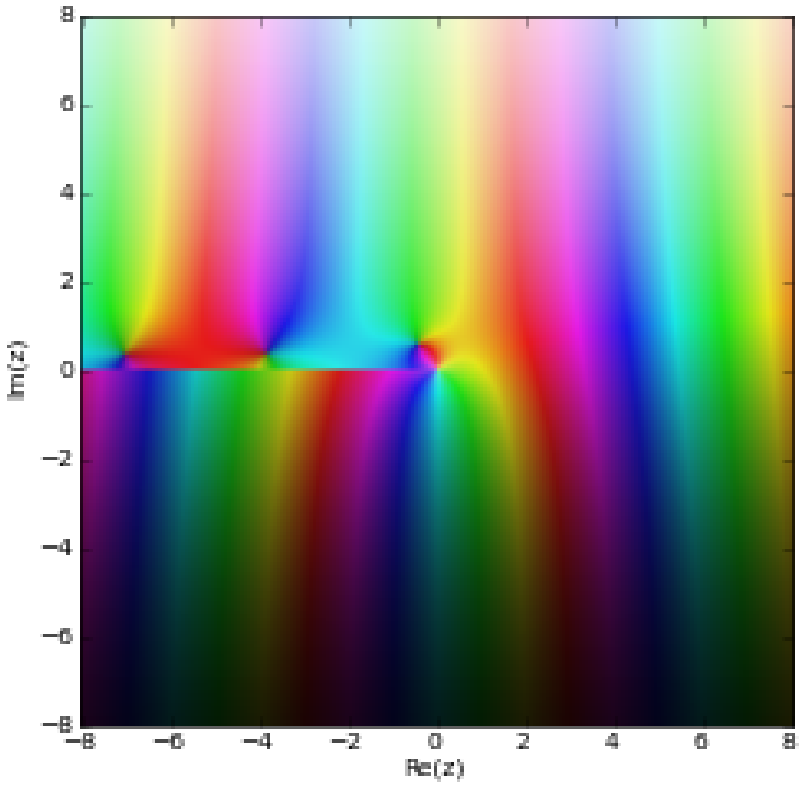
$$H_n^{(2)}(x) = J_n(x) - iY_n(x).$$

Plots

```
# Hankel function H2_n(x) on the real line for n=0,1,2,3
h0 = lambda x: hankel2(0,x)
h1 = lambda x: hankel2(1,x)
h2 = lambda x: hankel2(2,x)
h3 = lambda x: hankel2(3,x)
plot([h0,h1,h2,h3],[0,6],[-1,2])
```




```
    # Hankel function H2_n(z) in the complex plane
    cplot(lambda z: hankel2(1,z), [-8,8], [-8,8], points=50000)
```



Examples

The Hankel function is generally complex-valued:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hankel2(2, pi)
(0.4854339326315091097054957 + 0.09990071392902787877349 03j)
>>> hankel2(3.5, pi)
(0.2340002029630507922628888 + 0.6419643823412927142424049j)
```

Kelvin functions

ber()

mpmath.**ber**(*ctx*, *n*, *z*, ***kwargs*)

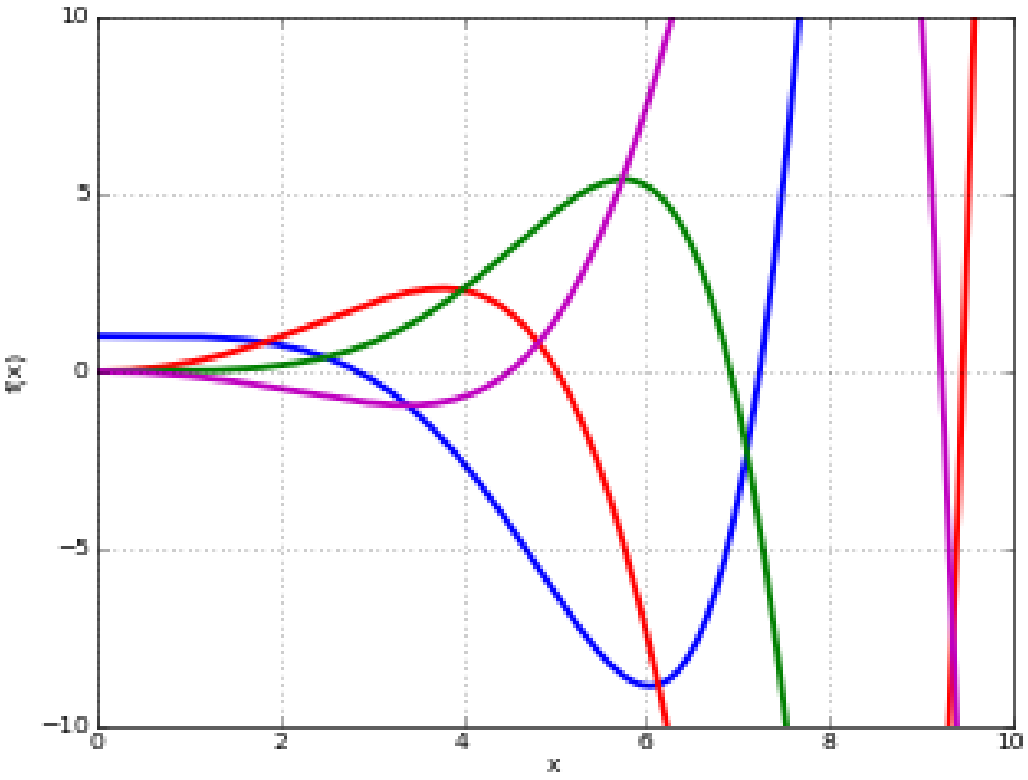
Computes the Kelvin function ber, which for real arguments gives the real part of the Bessel J function of a rotated argument

$$J_n\left(xe^{3\pi i/4}\right) = \mathrm{ber}_n(x) + i\,\mathrm{bei}_n(x).$$

The imaginary part is given by *bei()*.

Plots

```
# Kelvin functions ber_n(x) and bei_n(x) on the real line for n=0,2
f0 = lambda x: ber(0,x)
f1 = lambda x: bei(0,x)
f2 = lambda x: ber(2,x)
f3 = lambda x: bei(2,x)
plot([f0,f1,f2,f3],[0,10],[-10,10])
```



Examples

Verifying the defining relation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> n, x = 2, 3.5
>>> ber(n,x)
1.442338852571888752631129
>>> bei(n,x)
-0.948359035324558320217678
>>> besselj(n, x*root(1,8,3))
(1.442338852571888752631129 - 0.948359035324558320217678j)
```

The ber and bei functions are also defined by analytic continuation for complex arguments:

```
>>> ber(1+j, 2+3j)
(4.675445984756614424069563 - 15.84901771719130765656316j)
>>> bei(1+j, 2+3j)
(15.838866791937076993643 98 + 4.684053288183046528703611j)
```

bei()

mpmath.**bei**(*ctx*, *n*, *z*, ***kwargs*)

Computes the Kelvin function bei, which for real arguments gives the imaginary part of the Bessel J function of a rotated argument. See *ber()*.

ker()

mpmath.**ker**(*ctx*, *n*, *z*, ***kwargs*)

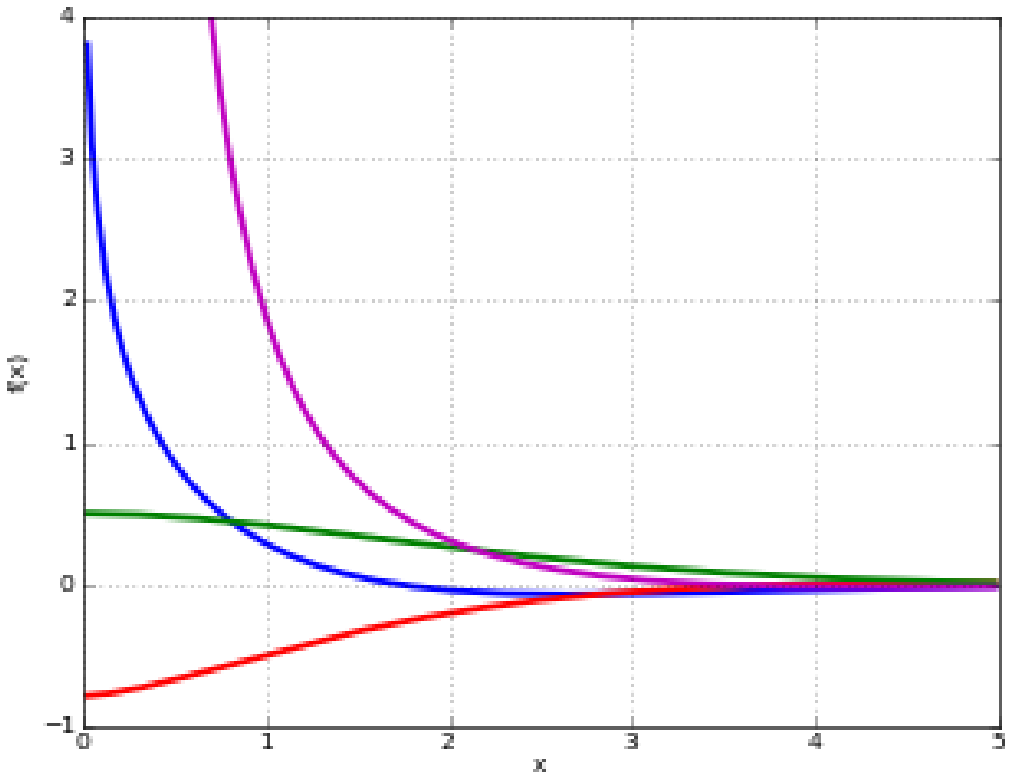
Computes the Kelvin function ker, which for real arguments gives the real part of the (rescaled) Bessel K function of a rotated argument

$$e^{-\pi i/2} K_n\left(x e^{3\pi i/4}\right) = \mathrm{ker}_n(x) + i\mathrm{kei}_n(x).$$

The imaginary part is given by *kei()*.

Plots

```
# Kelvin functions ker_n(x) and kei_n(x) on the real line for n=0,2
f0 = lambda x: ker(0,x)
f1 = lambda x: kei(0,x)
f2 = lambda x: ker(2,x)
f3 = lambda x: kei(2,x)
plot([f0,f1,f2,f3],[0,5],[-1,4])
```



Examples

Verifying the defining relation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> n, x = 2, 4.5
>>> ker(n,x)
0.025428952019063696402498 01
>>> kei(n,x)
-0.020749604672228232370553 51
>>> exp(-n*pi*j/2) * besselk(n, x*root(1,8,1))
(0.025428952019063696402498 01 - 0.020749604672228232370553 51j)
```

The ker and kei functions are also defined by analytic continuation for complex arguments:

```
>>> ker(1+j, 3+4j)
(1.586084268115490421090533 - 2.939717517906339193598719j)
>>> kei(1+j, 3+4j)
(-2.940403256319453402690132 - 1.585621643835618941044855j)
```

kei()

mpmath.**kei**(*ctx*, *n*, *z*, ***kwargs*)

Computes the Kelvin function kei, which for real arguments gives the imaginary part of the (rescaled) Bessel K function of a rotated argument. See *ker()*.

Struve functions

struveh()

mpmath.**struveh**(*ctx*, *n*, *z*, ***kwargs*)

Gives the Struve function

$$\mathbf{H}_n(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma(k+\frac{3}{2})\Gamma(k+n+\frac{3}{2})} \left(\frac{z}{2}\right)^{2k+n+1}$$

which is a solution to the Struve differential equation

$$z^2 f''(z) + z f'(z) + (z^2 - n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!}.$$

Examples

Evaluation for arbitrary real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> struveh(0, 3.5)
0.360820773377829502497779 7
>>> struveh(-1, 10)
-0.255212719726956768034732
>>> struveh(1, -100.5)
0.581956681679736228750224 6
>>> struveh(2.5, 10000000000000)
3153915652525200060.308937
>>> struveh(2.5, -10000000000000)
(0.0 - 3153915652525200060.308937j)
```

```
>>> struveh(1+j, 1000000+4000000j)
(-3.066421087689197632388731e+1737173 - 1.596619701076529803290973e+1737173j)
```

A Struve function of half-integer order is elementary; for example:

```
>>> z = 3
>>> struveh(0.5, 3)
0.91670768675641381786715 95
>>> sqrt(2/(pi*z))*(1-cos(z))
0.91670768675641381786715 95
```

Numerically verifying the differential equation:

```
>>> z = mpf(4.5)
>>> n = 3
>>> f = lambda z: struveh(n,z)
>>> lhs = z**2*diff(f,z,2) + z*diff(f,z) + (z**2-n**2)*f(z)
>>> rhs = 2*z**(n+1)/fac2(2*n-1)/pi
>>> lhs
17.403593027098754966327 44
>>> rhs
17.403593027098754966327 44
```

struvel()

mpmath.**struvel**(*ctx*, *n*, *z*, ***kwargs*)

Gives the modified Struve function

$$\mathbf{L}_n(z) = -ie^{-n\pi i/2}\mathbf{H}_n(iz)$$

which solves to the modified Struve differential equation

$$z^2 f''(z) + z f'(z) - (z^2 + n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!}.$$

Examples

Evaluation for arbitrary real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> struvel(0, 3.5)
7.180846515103737996249972
>>> struvel(-1, 10)
2670.994904980850550721511
>>> struvel(1, -100.5)
1.757089288053346261497686e+42
>>> struvel(2.5, 100000000000000)
4.160893281017115450519948e+4342944819025
>>> struvel(2.5, -100000000000000)
(0.0 - 4.160893281017115450519948e+4342944819025j)
>>> struvel(1+j, 700j)
(-0.172115004948007945124607 6 + 0.124077095312683109346405 5j)
>>> struvel(1+j, 1000000+4000000j)
(-2.973341637511505389128708e+434290 - 5.164633059729968297147448e+434290j)
```

Numerically verifying the differential equation:

```
>>> z = mpf(3.5)
>>> n = 3
>>> f = lambda z: struvel(n,z)
>>> lhs = z**2*diff(f,z,2) + z*diff(f,z) - (z**2+n**2)*f(z)
>>> rhs = 2*z**(n+1)/fac2(2*n-1)/pi
>>> lhs
6.3688503060606783530181 65
>>> rhs
6.3688503060606783530181 65
```

Anger-Weber functions

angerj()

mpmath.**angerj**(*ctx*, *v*, *z*, ***kwargs*)

Gives the Anger function

$$\mathbf{J}_\nu(z) = \frac{1}{\pi}\int_0^\pi \cos(\nu t - z\sin t)dt$$

which is an entire function of both the parameter ν and the argument z. It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z}f'(z) + \left(1 - \frac{\nu^2}{z^2}\right)f(z) = \frac{(z-\nu)}{\pi z^2}\sin(\pi\nu).$$

Examples

Evaluation for real and complex parameter and argument:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> angerj(2,3)
0.486091260585891076907831 1
>>> angerj(-3+4j, 2+5j)
(-5033.358320403384472395612 + 585.801189247614511855 1756j)
>>> angerj(3.25, 1e6j)
(4.6307436397158933465707 43e+434290 - 1.1179604098875059068484 56e+434291j)
>>> angerj(-1.5, 1e6)
0.000279571974707387939308 7011
```

The Anger function coincides with the Bessel J-function when ν is an integer:

```
>>> angerj(1,3); besselj(1,3)
0.339058958525936458925514 6
0.339058958525936458925514 6
>>> angerj(1.5,3); besselj(1.5,3)
0.408896984869108085932884 7
0.477718215087091771551501 5
```

Verifying the differential equation:

```
>>> v,z = mpf(2.25), 0.75
>>> f = lambda z: angerj(v,z)
>>> diff(f,z,2) + diff(f,z)/z + (1-(v/z)**2)*f(z)
-0.60021087743807071303679 95
>>> (z-v)/(pi*z**2) * sinpi(v)
-0.60021087743807071303679 95
```

Verifying the integral representation:

```
>>> angerj(v,z)
0.11453807599193331809005 01
>>> quad(lambda t: cos(v*t-z*sin(t))/pi, [0,pi])
0.11453807599193331809005 01
```

References

1. *[DLMF]* section 11.10: Anger-Weber Functions

webere()

mpmath.**webere**(*ctx*, *v*, *z*, ***kwargs*)

Gives the Weber function

$$\mathbf{E}_\nu(z) = \frac{1}{\pi}\int_0^\pi \sin(\nu t - z\sin t)dt$$

which is an entire function of both the parameter ν and the argument z. It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z}f'(z) + \left(1 - \frac{\nu^2}{z^2}\right)f(z) = -\frac{1}{\pi z^2}(z + \nu + (z-\nu)\cos(\pi\nu)).$$

Examples

Evaluation for real and complex parameter and argument:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> webere(2,3)
-0.10576689730990184256626 46
>>> webere(-3+4j, 2+5j)
(-585.8081418209852019290498 - 5033.3144888999269215972 03j)
>>> webere(3.25, 1e6j)
(-1.1179604098875059068484 56e+434291 - 4.6307436397158933465707 43e+434290j)
>>> webere(3.25, 1e6)
-0.0000281251826589431560491 4453
```

Up to addition of a rational function of z, the Weber function coincides with the Struve H-function when ν is an integer:

```
>>> webere(1,3); 2/pi-struveh(1,3)
-0.3834897968188690177372881
-0.3834897968188690177372881
>>> webere(5,3); 26/(35*pi)-struveh(5,3)
0.2009680659308154011878075
0.2009680659308154011878075
```

Verifying the differential equation:

```
>>> v,z = mpf(2.25), 0.75
>>> f = lambda z: webere(v,z)
>>> diff(f,z,2) + diff(f,z)/z + (1-(v/z)**2)*f(z)
-1.0974418488754795351646 27
>>> -(z+v+(z-v)*cospi(v))/(pi*z**2)
-1.0974418488754795351646 27
```

Verifying the integral representation:

```
>>> webere(v,z)
0.14865073515342837444854 21
>>> quad(lambda t: sin(v*t-z*sin(t))/pi, [0,pi])
0.14865073515342837444854 21
```

References

1.*[DLMF]* section 11.10: Anger-Weber Functions

Lommel functions

lommels1()

mpmath.**lommels1**(*ctx*, *u*, *v*, *z*, ***kwargs*)

Gives the Lommel function $s_{\mu,\nu}$ or $s^{(1)}_{\mu,\nu}$

$$s_{\mu,\nu}(z) = \frac{z^{\mu+1}}{(\mu-\nu+1)(\mu+\nu+1)}\,{}_1F_2\left(1; \frac{\mu-\nu+3}{2}, \frac{\mu+\nu+3}{2}; -\frac{z^2}{4}\right)$$

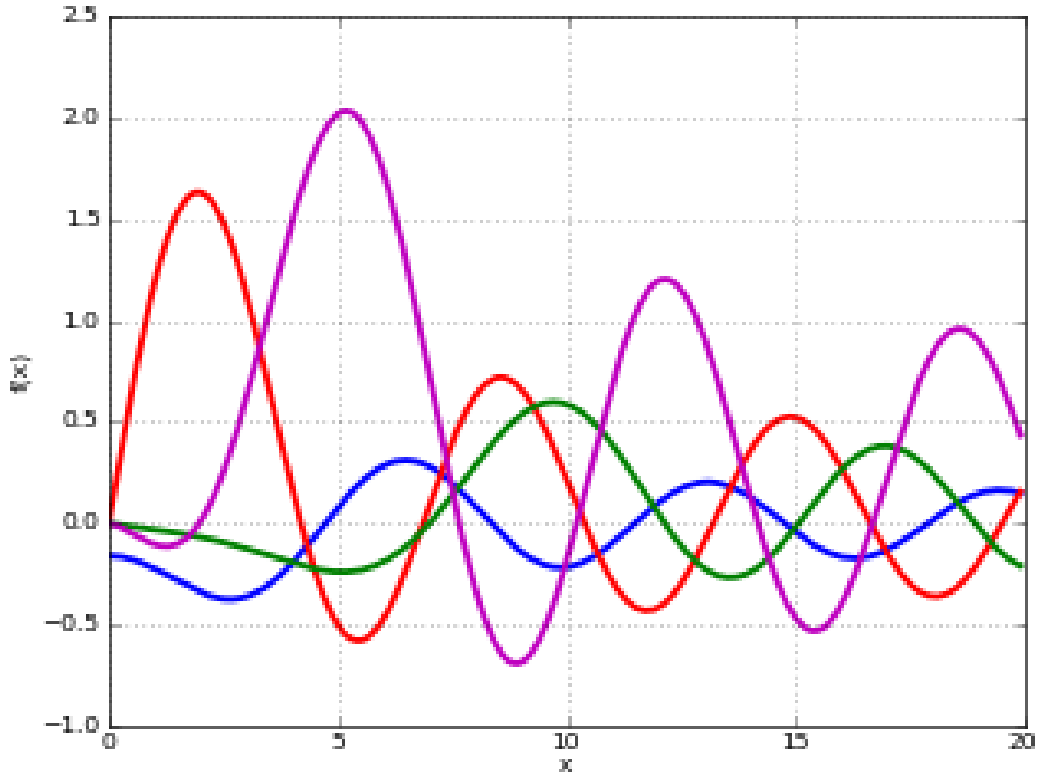
which solves the inhomogeneous Bessel equation

$$z^2 f''(z) + z f'(z) + (z^2-\nu^2) f(z) = z^{\mu+1}.$$

A second solution is given by *lommels2()*.

Plots

```
# Lommel function s_(u,v)(x) on the real line for a few different u,v
f1 = lambda x: lommels1(-1,2.5,x)
f2 = lambda x: lommels1(0,0.5,x)
f3 = lambda x: lommels1(0,6,x)
f4 = lambda x: lommels1(0.5,3,x)
plot([f1,f2,f3,f4], [0,20])
```



Examples

An integral representation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> u,v,z = 0.25, 0.125, mpf(0.75)
>>> lommels1(u,v,z)
0.42762438775651503729991 26
>>> (bessely(v,z)*quad(lambda t: t**u*besselj(v,t), [0,z]) - \
...  besselj(v,z)*quad(lambda t: t**u*bessely(v,t), [0,z]))*(pi/2)
0.42762438775651503729991 26
```

A special value:

```
>>> lommels1(v,v,z)
0.54612213677460480549325 53
>>> gamma(v+0.5)*sqrt(pi)*power(2,v-1)*struveh(v,z)
0.54612213677460480549325 53
```

Verifying the differential equation:

```
>>> f = lambda z: lommels1(u,v,z)
>>> z**2*diff(f,z,2) + z*diff(f,z) + (z**2-v**2)*f(z)
0.69795364432657469920591 41
>>> z**(u+1)
0.69795364432657469920591 41
```

References

1.*[GradshteynRyzhik]*

2.*[Weisstein]* http://mathworld.wolfram.com/LommelFunction.html

lommels2()

mpmath.**lommels2**(*ctx*, *u*, *v*, *z*, ***kwargs*)

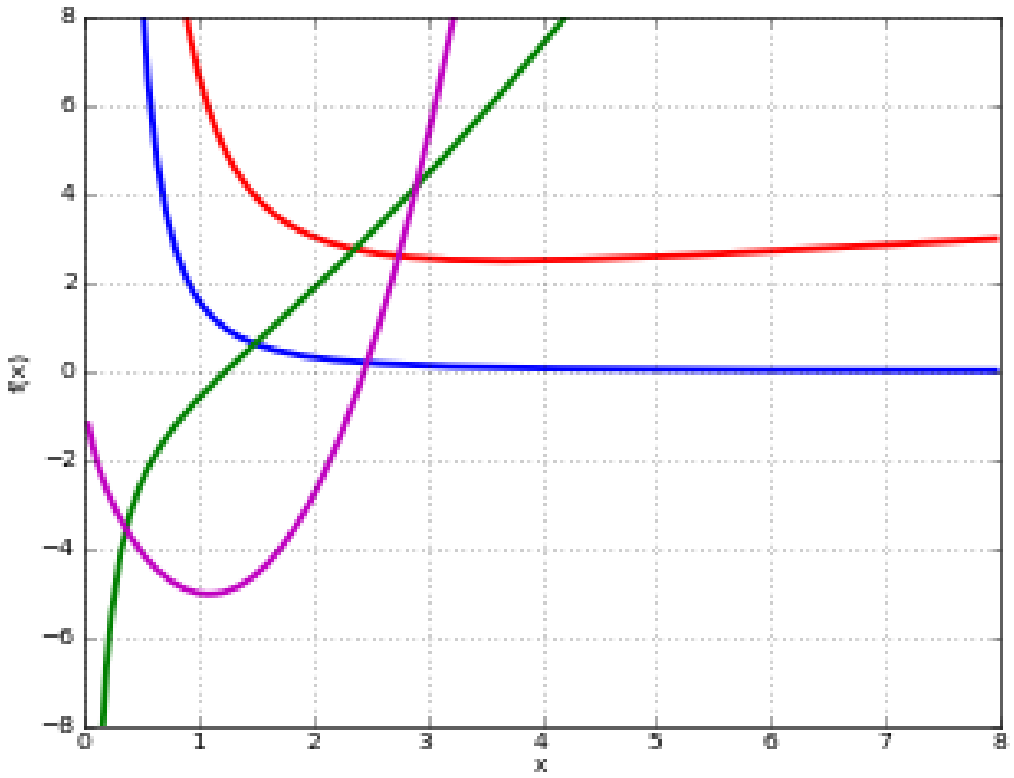
Gives the second Lommel function $S_{\mu,\nu}$ or $s^{(2)}_{\mu,\nu}$

$$S_{\mu,\nu}(z) = s_{\mu,\nu}(z) + 2^{\mu-1}\Gamma\left(\tfrac{1}{2}(\mu-\nu+1)\right)\Gamma\left(\tfrac{1}{2}(\mu+\nu+1)\right)\times \left[\sin(\tfrac{1}{2}(\mu-\nu)\pi)J_\nu(z) - \cos(\tfrac{1}{2}(\mu-\nu)\pi)Y_\nu(z)\right]$$

which solves the same differential equation as *lommels1()*.

Plots

```
# Lommel function S_(u,v)(x) on the real line for a few different u,v
f1 = lambda x: lommels2(-1,2.5,x)
f2 = lambda x: lommels2(1.5,2,x)
f3 = lambda x: lommels2(2.5,1,x)
f4 = lambda x: lommels2(3.5,-0.5,x)
plot([f1,f2,f3,f4], [0,8], [-8,8])
```



Examples

For large $|z|$, $S_{\mu,\nu} \sim z^{\mu-1}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> lommels2(10,2,30000)
1.968299831601008419949804e+40
>>> power(30000,9)
1.9683e+40
```

A special value:

```
>>> u,v,z = 0.5, 0.125, mpf(0.75)
>>> lommels2(v,v,z)
0.95896831996246720999697652
>>> (struveh(v,z)-bessely(v,z))*power(2,v-1)*sqrt(pi)*gamma(v+0.5)
0.95896831996246720999697652
```

Verifying the differential equation:

```
>>> f = lambda z: lommels2(u,v,z)
>>> z**2*diff(f,z,2) + z*diff(f,z) + (z**2-v**2)*f(z)
0.64951905283832898507279243
>>> z**(u+1)
0.64951905283832898507279243
```

References

1. *[GradshteynRyzhik]*

2. *[Weisstein]* http://mathworld.wolfram.com/LommelFunction.html

Airy and Scorer functions

airyai()

mpmath.**airyai**(*z*, *derivative=0*, ***kwargs*)

Computes the Airy function $\mathrm{Ai}(z)$, which is the solution of the Airy differential equation $f''(z) - zf(z) = 0$ with initial conditions

$$\mathrm{Ai}(0) = \frac{1}{3^{2/3}\Gamma\left(\frac{2}{3}\right)}$$
$$\mathrm{Ai}'(0) = -\frac{1}{3^{1/3}\Gamma\left(\frac{1}{3}\right)}.$$

Other common ways of defining the Ai-function include integrals such as

$$\mathrm{Ai}(x) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt \qquad x \in \mathbb{R}$$
$$\mathrm{Ai}(z) = \frac{\sqrt{3}}{2\pi}\int_0^\infty \exp\left(-\frac{t^3}{3} - \frac{z^3}{3t^3}\right) dt.$$

The Ai-function is an entire function with a turning point, behaving roughly like a slowly decaying sine wave for $z < 0$ and like a rapidly decreasing exponential for $z > 0$. A second solution of the Airy differential equation is given by $\mathrm{Bi}(z)$ (see *airybi()*).

Optionally, with *derivative=alpha*, airyai() can compute 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

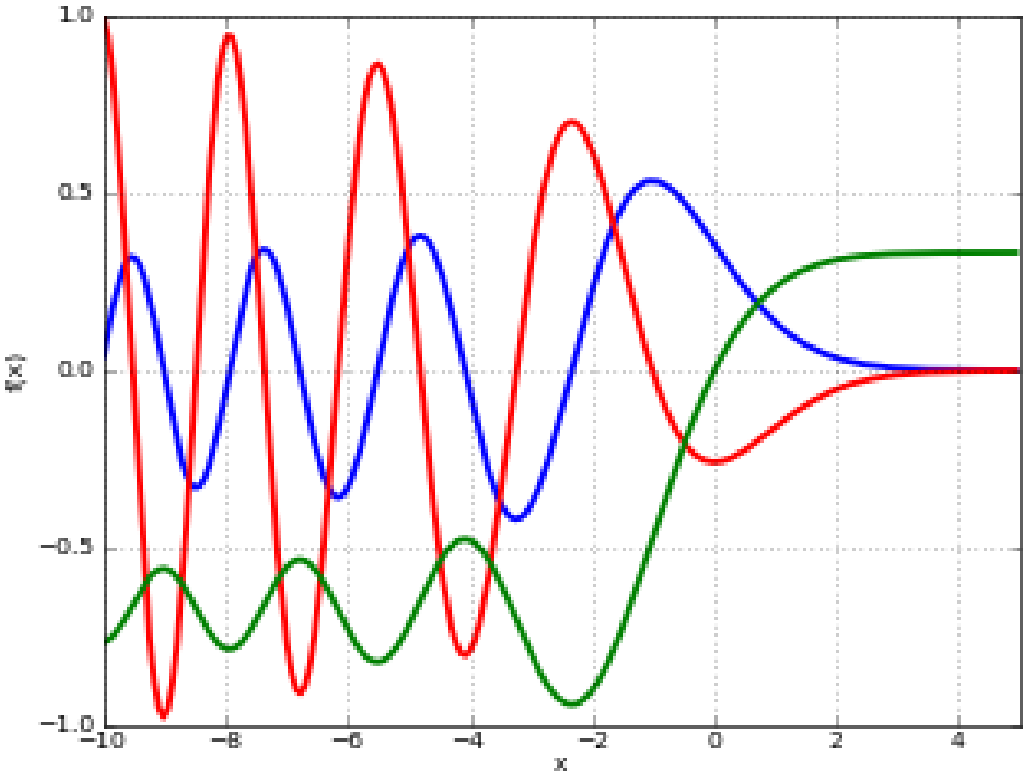
n-fold iterated integral

$$f_0(z) = \mathrm{Ai}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt.$$

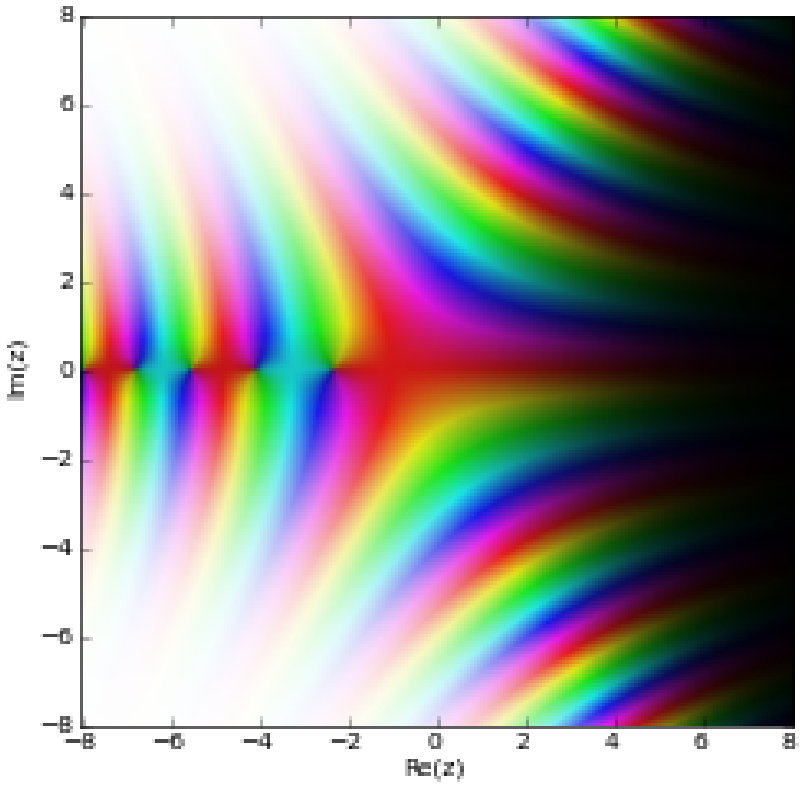
The Ai-function has infinitely many zeros, all located along the negative half of the real axis. They can be computed with *airyaizero()*.

Plots

```
# Airy function Ai(x), Ai'(x) and int_0^x Ai(t) dt on the real line
f = airyai
f_diff = lambda z: airyai(z, derivative=1)
f_int = lambda z: airyai(z, derivative=-1)
plot([f, f_diff, f_int], [-10,5])
```




```
# Airy function Ai(z) in the complex plane
cplot(airyai, [-8,8], [-8,8], points=50000)
```



Basic examples

Limits and values include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airyai(0); 1/(power(3,'2/3')*gamma('2/3'))
0.35502805388781723926006318
0.35502805388781723926006318
>>> airyai(1)
0.13529241631288141552414741
>>> airyai(-1)
0.53556088329235211879951660
>>> airyai(inf); airyai(-inf)
0.0
0.0
```

Evaluation is supported for large magnitudes of the argument:

```
>>> airyai(-100)
0.17675339323955287809083110
>>> airyai(100)
2.6344821520881844895505530e-291
>>> airyai(50+50j)
(-5.3179019570745604099817e-68 - 1.1635880037770970974872010e-67j)
>>> airyai(-50+50j)
(1.0412425373631676325872450e+158 + 3.3475255449236003218382810e+157j)
```

Huge arguments are also fine:

```
>>> airyai(10**10)
1.1622359782987417799536 93e-289529654602171
>>> airyai(-10**10)
0.00017362064481528185105 10181
>>> w = airyai(10**10*(1+j))
>>> w.real
5.7115086837213555283225 67e-186339621747698
>>> w.imag
1.8672455069623125778481 66e-186339621747697
```

The first root of the Ai-function is:

```
>>> findroot(airyai, -2)
-2.3381074104597670384891 97
>>> airyaizero(1)
-2.3381074104597670384891 97
```

Properties and relations

Verifying the Airy differential equation:

```
>>> for z in [-3.4, 0, 2.5, 1+2j]:
...     chop(airyai(z,2) - z*airyai(z))
...
0.0
0.0
0.0
0.0
```

The first few terms of the Taylor series expansion around $z = 0$ (every third term is zero):

```
>>> nprint(taylor(airyai, 0, 5))
[0.355028, -0.258819, 0.0, 0.0591713, -0.0215683, 0.0]
```

The Airy functions satisfy the Wronskian relation $\mathrm{Ai}(z)\,\mathrm{Bi}'(z) - \mathrm{Ai}'(z)\,\mathrm{Bi}(z) = 1/\pi$:

```
>>> z = -0.5
>>> airyai(z)*airybi(z,1) - airyai(z,1)*airybi(z)
0.31830988618379067153776 75
>>> 1/pi
0.31830988618379067153776 75
```

The Airy functions can be expressed in terms of Bessel functions of order $\pm 1/3$. For $\Re[z] \le 0$, we have:

```
>>> z = -3
>>> airyai(z)
-0.37881429367765807434724 39
>>> y = 2*power(-z,'3/2')/3
>>> (sqrt(-z) * (besselj('1/3',y) + besselj('-1/3',y)))/3
-0.37881429367765807434724 39
```

Derivatives and integrals

Derivatives of the Ai-function (directly and using *diff()*):

```
>>> airyai(-3,1); diff(airyai,-3)
0.31458376921659881365078 73
0.31458376921659881365078 73
>>> airyai(-3,2); diff(airyai,-3,2)
1.13644288103297422304173 2
1.13644288103297422304173 2
```

```
>>> airyai(1000,1); diff(airyai,1000)
-2.9431339179103360904597 48e-9156
-2.9431339179103360904597 48e-9156
```

Several derivatives at $z = 0$:

```
>>> airyai(0,0); airyai(0,1); airyai(0,2)
0.355028053887817239260063 2
-0.258819403792806798405183 6
0.0
>>> airyai(0,3); airyai(0,4); airyai(0,5)
0.355028053887817239260063 2
-0.517638807585613596810367 1
0.0
>>> airyai(0,15); airyai(0,16); airyai(0,17)
1292.30211615165475090663
-3188.655054727379756351861
0.0
```

The integral of the Ai-function:

```
>>> airyai(3,-1); quad(airyai, [0,3])
0.32992037600702177250027 01
0.32992037600702177250027 01
>>> airyai(-10,-1); quad(airyai, [0,-10])
-0.765698403134212917425148
-0.765698403134212917425148
```

Integrals of high or fractional order:

```
>>> airyai(-2,0.5); differint(airyai,-2,0.5,0)
(0.0 + 0.24535961013514382738447 25j)
(0.0 + 0.24535961013514382738447 25j)
>>> airyai(-2,-4); differint(airyai,-2,-4,0)
0.29391764416368095803393 65
0.29391764416368095803393 65
>>> airyai(0,-1); airyai(0,-2); airyai(0,-3)
0.0
0.0
0.0
```

Integrals of the Ai-function can be evaluated at limit points:

```
>>> airyai(-1000000,-1); airyai(-inf,-1)
-0.666684372831153997875 1512
-0.666666666666666666666 6667
>>> airyai(10,-1); airyai(+inf,-1)
0.333333333299169015942 7932
0.333333333333333333333 3333
>>> airyai(+inf,-2); airyai(+inf,-3)
+inf
+inf
>>> airyai(-1000000,-2); airyai(-inf,-2)
666666.4078472650651209742
+inf
>>> airyai(-1000000,-3); airyai(-inf,-3)
-333333074513.7520264995733
-inf
```

References

1.*[DLMF]* Chapter 9: Airy and Related Functions

2.*[WolframFunctions]* section: Bessel-Type Functions

airybi()

mpmath.**airybi**(*z*, *derivative=0*, ***kwargs*)

Computes the Airy function $\mathrm{Bi}(z)$, which is the solution of the Airy differential equation $f''(z) - zf(z) = 0$ with initial conditions

$$\mathrm{Bi}(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{2}{3}\right)}$$
$$\mathrm{Bi}'(0) = \frac{3^{1/6}}{\Gamma\left(\frac{1}{3}\right)}.$$

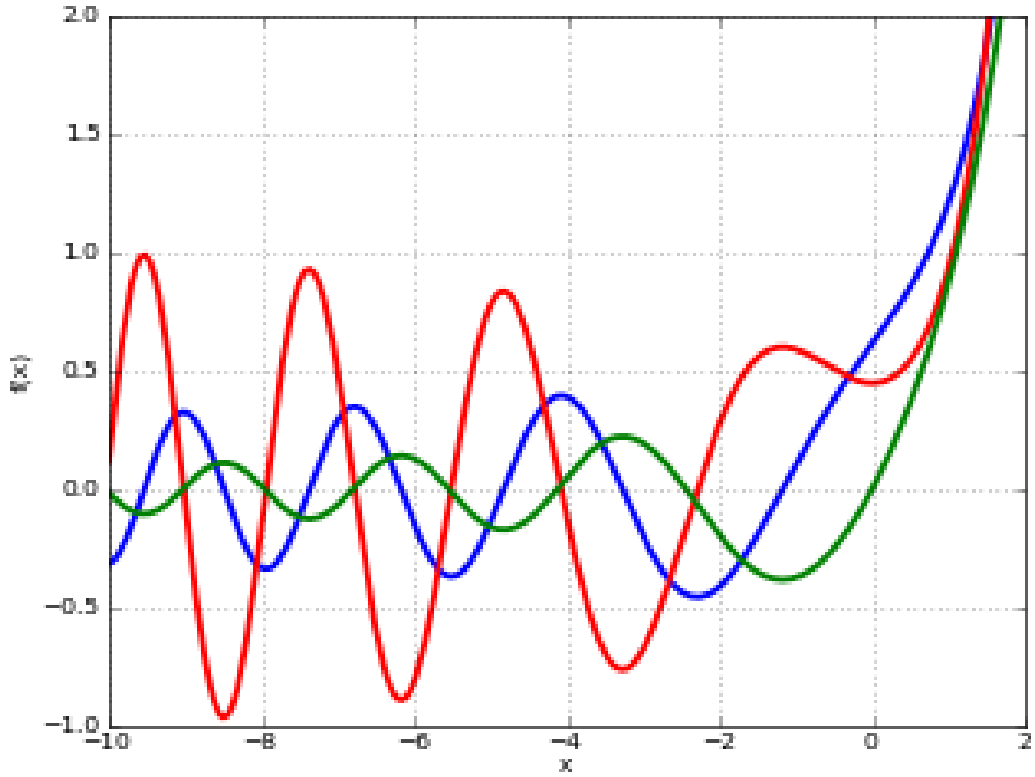
Like the Ai-function (see *airyai()*), the Bi-function is oscillatory for $z < 0$, but it grows rather than decreases for $z > 0$.

Optionally, as for *airyai()*, derivatives, integrals and fractional derivatives can be computed with the *derivative* parameter.

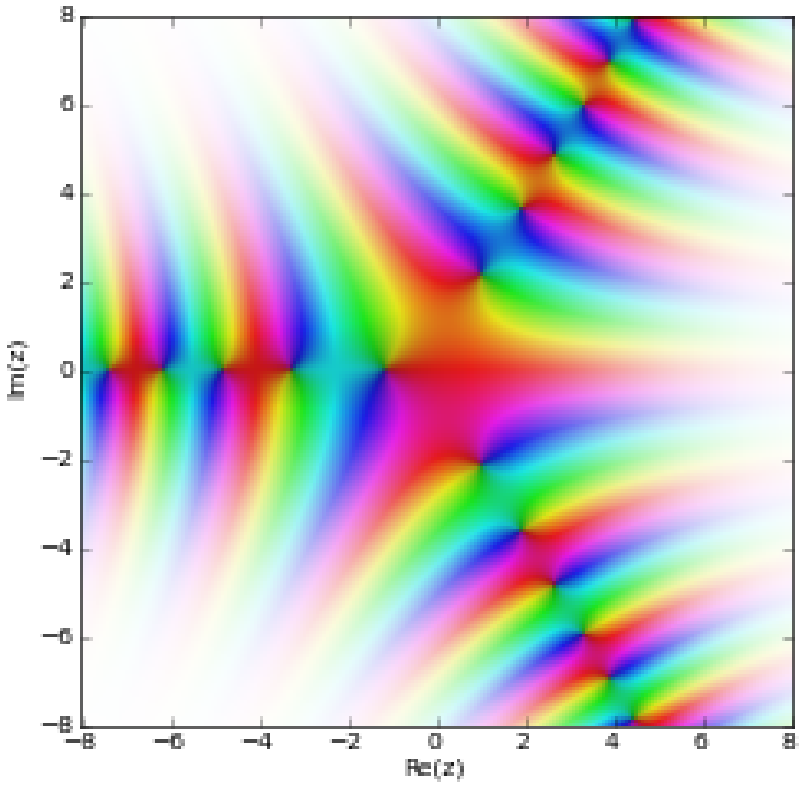
The Bi-function has infinitely many zeros along the negative half-axis, as well as complex zeros, which can all be computed with *airybizero()*.

Plots

```
# Airy function Bi(x), Bi'(x) and int_0^x Bi(t) dt on the real line
f = airybi
f_diff = lambda z: airybi(z, derivative=1)
f_int = lambda z: airybi(z, derivative=-1)
plot([f, f_diff, f_int], [-10,2], [-1,2])
```




```
    # Airy function Bi(z) in the complex plane
    cplot(airybi, [-8,8], [-8,8], points=50000)
```



Basic examples

Limits and values include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airybi(0); 1/(power(3,'1/6')*gamma('2/3'))
0.61492662744600073515092241
0.61492662744600073515092241
>>> airybi(1)
1.2074235949528712594363791
>>> airybi(-1)
0.10399738949694461188869
>>> airybi(inf); airybi(-inf)
+inf
0.0
```

Evaluation is supported for large magnitudes of the argument:

```
>>> airybi(-100)
0.024273887680160131605667471
>>> airybi(100)
6.041223996670201399005265e+288
>>> airybi(50+50j)
(-5.322076267321435669290334e+63 + 1.478450291165243789749427e+65j)
>>> airybi(-50+50j)
(-3.347525544923600321838281e+157 + 1.041242537363167632587245e+158j)
```

Huge arguments:

```
>>> airybi(10**10)
1.369385787943539818688433e+289529654602165
>>> airybi(-10**10)
0.00177565614169293274761097 3
>>> w = airybi(10**10*(1+j))
>>> w.real
-6.559955931096196875845858e+186339621747689
>>> w.imag
-6.822462726981357180929024e+186339621747690
```

The first real root of the Bi-function is:

```
>>> findroot(airybi, -1); airybizero(1)
-1.17371322270912792491998
-1.17371322270912792491998
```

Properties and relations

Verifying the Airy differential equation:

```
>>> for z in [-3.4, 0, 2.5, 1+2j]:
...     chop(airybi(z,2) - z*airybi(z))
...
0.0
0.0
0.0
0.0
```

The first few terms of the Taylor series expansion around $z = 0$ (every third term is zero):

```
>>> nprint(taylor(airybi, 0, 5))
[0.614927, 0.448288, 0.0, 0.102488, 0.0373574, 0.0]
```

The Airy functions can be expressed in terms of Bessel functions of order $\pm 1/3$. For $\Re[z] \leq 0$, we have:

```
>>> z = -3
>>> airybi(z)
-0.19828962637492654322064 49
>>> p = 2*power(-z,'3/2')/3
>>> sqrt(-mpf(z)/3)*(besselj('-1/3',p) - besselj('1/3',p))
-0.19828962637492654322064 49
```

Derivatives and integrals

Derivatives of the Bi-function (directly and using *diff()*):

```
>>> airybi(-3,1); diff(airybi,-3)
-0.675611222685258537668032
-0.675611222685258537668032
>>> airybi(-3,2); diff(airybi,-3,2)
0.594868879124779629661934 6
0.594868879124779629661934 6
>>> airybi(1000,1); diff(airybi,1000)
1.710055114624614989262335e+9156
1.710055114624614989262335e+9156
```

Several derivatives at $z = 0$:

```
>>> airybi(0,0); airybi(0,1); airybi(0,2)
0.614926627446000735150922 4
0.448288357353826357914823 7
```

```
0.0
>>> airybi(0,3); airybi(0,4); airybi(0,5)
0.61492662744600073515092224
0.89657671470765271582964 74
0.0
>>> airybi(0,15); airybi(0,16); airybi(0,17)
2238.332923903442675949357
5522.912562599140729510628
0.0
```

The integral of the Bi-function:

```
>>> airybi(3,-1); quad(airybi, [0,3])
10.062003031306200563166 55
10.062003031306200563166 55
>>> airybi(-10,-1); quad(airybi, [0,-10])
-0.015040424806140020451354 83
-0.015040424806140020451354 83
```

Integrals of high or fractional order:

```
>>> airybi(-2,0.5); differint(airybi, -2, 0.5, 0)
(0.0 + 0.5019859055341699223453257j)
(0.0 + 0.5019859055341699223453257j)
>>> airybi(-2,-4); differint(airybi,-2,-4,0)
0.28093145999224472521390 92
0.28093145999224472521390 92
>>> airybi(0,-1); airybi(0,-2); airybi(0,-3)
0.0
0.0
0.0
```

Integrals of the Bi-function can be evaluated at limit points:

```
>>> airybi(-1000000,-1); airybi(-inf,-1)
0.00000219126112806343404 7966873
0.0
>>> airybi(10,-1); airybi(+inf,-1)
147809803.1074067161675853
+inf
>>> airybi(+inf,-2); airybi(+inf,-3)
+inf
+inf
>>> airybi(-1000000,-2); airybi(-inf,-2)
0.44828837505999084798510 85
0.44828835735382635791482 37
>>> gamma('2/3')*power(3,'2/3')/(2*pi)
0.44828835735382635791482 37
>>> airybi(-100000,-3); airybi(-inf,-3)
-44828.528272069328724931 33
-inf
>>> airybi(-100000,-4); airybi(-inf,-4)
2241411040.437759489540248
+inf
```

airyaizero()

mpmath.**airyaizero**(*k*, *derivative=0*)

Gives the k-th zero of the Airy Ai-function, i.e. the k-th number a_k ordered by magnitude for which $\mathrm{Ai}(a_k) = 0$.

Optionally, with *derivative=1*, the corresponding zero a'_k of the derivative function, i.e. $\mathrm{Ai}'(a'_k) = 0$, is computed.

Examples

Some values of a_k:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airyaizero(1)
-2.338107410459767038489197
>>> airyaizero(2)
-4.087949444130970616636989
>>> airyaizero(3)
-5.520559828095551059129856
>>> airyaizero(1000)
-281.0315196125215528353364
```

Some values of a'_k:

```
>>> airyaizero(1,1)
-1.018792971647471089017325
>>> airyaizero(2,1)
-3.248197582179836537875424
>>> airyaizero(3,1)
-4.820099211178735639400616
>>> airyaizero(1000,1)
-280.9378080358935070607097
```

Verification:

```
>>> chop(airyai(airyaizero(1)))
0.0
>>> chop(airyai(airyaizero(1,1),1))
0.0
```

airybizero()

mpmath.**airybizero**(*k*, *derivative=0*, *complex=0*)

With *complex=False*, gives the k-th real zero of the Airy Bi-function, i.e. the k-th number b_k ordered by magnitude for which $\mathrm{Bi}(b_k) = 0$.

With *complex=True*, gives the k-th complex zero in the upper half plane β_k. Also the conjugate $\overline{\beta_k}$ is a zero.

Optionally, with *derivative=1*, the corresponding zero b'_k or β'_k of the derivative function, i.e. $\mathrm{Bi}'(b'_k) = 0$ or $\mathrm{Bi}'(\beta'_k) = 0$, is computed.

Examples

Some values of b_k:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airybizero(1)
-1.17371322270912792491998
```

```
>>> airybizero(2)
-3.271093302836352715680228
>>> airybizero(3)
-4.830737841662015932667709
>>> airybizero(1000)
-280.937811203415240157883 4
```

Some values of b_k:

```
>>> airybizero(1,1)
-2.2944396826141232466224 59
>>> airybizero(2,1)
-4.0731550890718282155523 69
>>> airybizero(3,1)
-5.5123957296635994962595 93
>>> airybizero(1000,1)
-281.0315164471118527161362
```

Some values of β_k:

```
>>> airybizero(1,complex=True)
(0.97754488673162068594699 27 + 2.1412907060387445757491 39j)
>>> airybizero(2,complex=True)
(1.89677501389533634662721 7 + 3.6272917643589194104404 99j)
>>> airybizero(3,complex=True)
(2.63315773935494659570801 9 + 4.8554681799798449831746 28j)
>>> airybizero(1000,complex=True)
(140.4978560578493018899793 + 243.39077242157921212448 67j)
```

Some values of β'_k:

```
>>> airybizero(1,1,complex=True)
(0.21494707453743056760883 29 + 1.1006001433027978806471 94j)
>>> airybizero(2,1,complex=True)
(1.45816830922350739202821 1 + 2.9122493674584454192350 83j)
>>> airybizero(3,1,complex=True)
(2.27376076301348229979236 2 + 4.2545285492170978621670 15j)
>>> airybizero(1000,1,complex=True)
(140.4509972835270559730423 + 243.30961753985628118962 08j)
```

Verification:

```
>>> chop(airybi(airybizero(1)))
0.0
>>> chop(airybi(airybizero(1,1),1))
0.0
>>> u = airybizero(1,complex=True)
>>> chop(airybi(u))
0.0
>>> chop(airybi(conj(u)))
0.0
```

The complex zeros (in the upper and lower half-planes respectively) asymptotically approach the rays $z = R\exp(\pm i\pi/3)$:

```
>>> arg(airybizero(1,complex=True))
1.1425325102863340223053 64
>>> arg(airybizero(1000,complex=True))
1.0472711147862120615839 17
>>> arg(airybizero(1000000,complex=True))
```

```
    1.0471976247418161833413 55
    >>> pi/3
    1.0471975511965977461542 14
```

scorergi()

mpmath.**scorergi**(*z*, ***kwargs*)

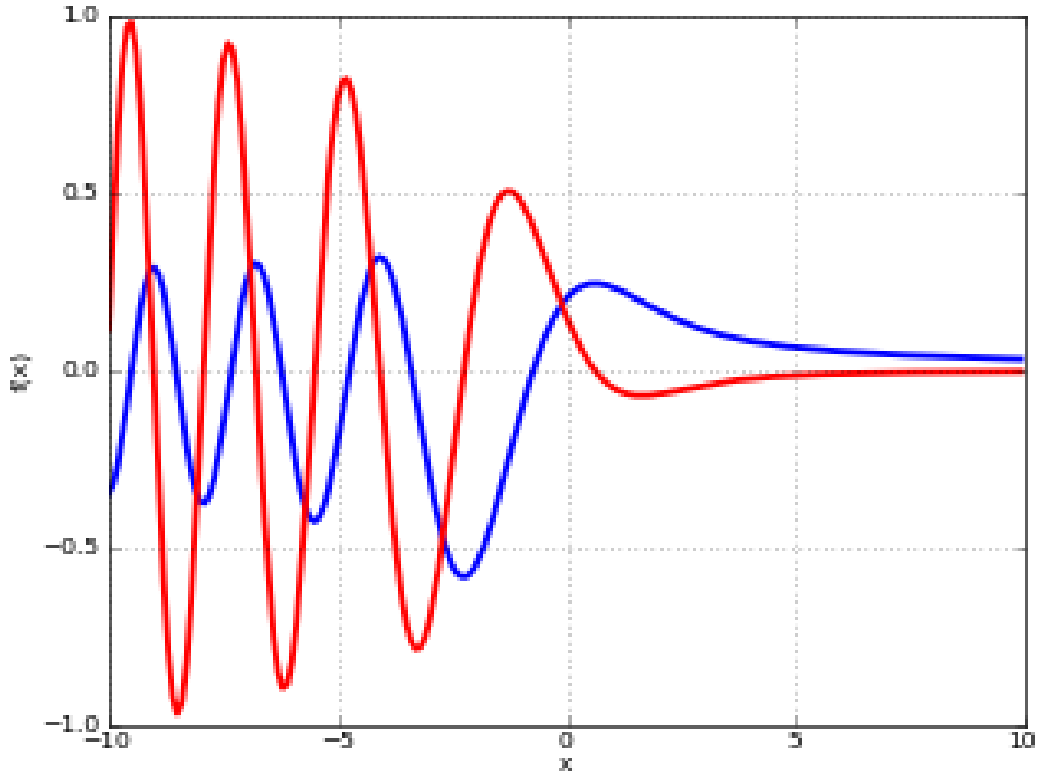
Evaluates the Scorer function

$$\mathrm{Gi}(z) = \mathrm{Ai}(z)\int_0^z \mathrm{Bi}(t)dt + \mathrm{Bi}(z)\int_z^\infty \mathrm{Ai}(t)dt$$

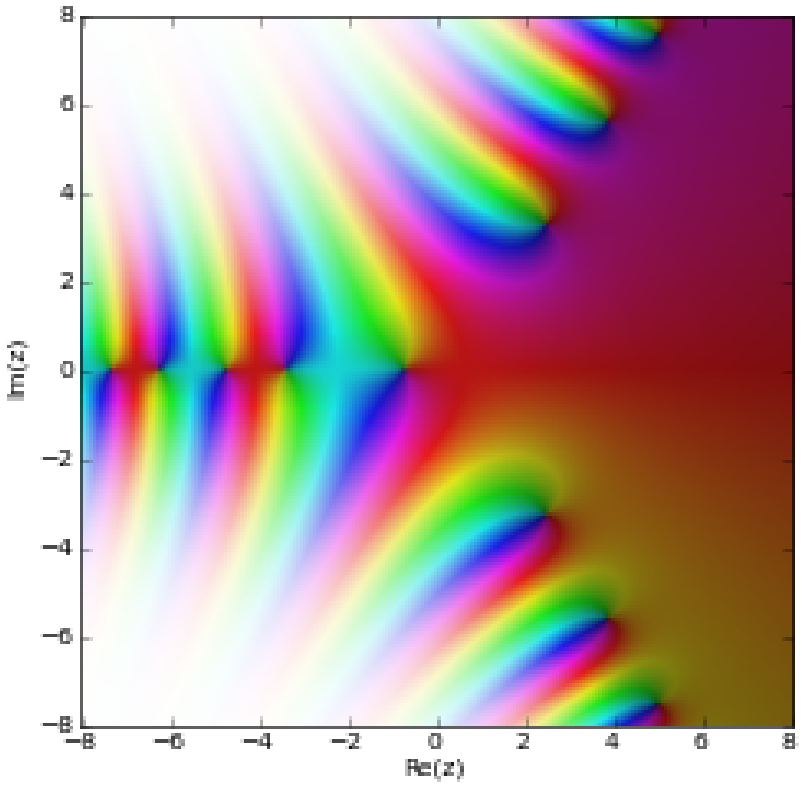
which gives a particular solution to the inhomogeneous Airy differential equation $f''(z) - zf(z) = 1/\pi$. Another particular solution is given by the Scorer Hi-function (*scorerhi()*). The two functions are related as $\mathrm{Gi}(z) + \mathrm{Hi}(z) = \mathrm{Bi}(z)$.

Plots

```
    # Scorer function Gi(x) and Gi'(x) on the real line
    plot([scorergi, diffun(scorergi)], [-10,10])
```




```
    # Scorer function Gi(z) in the complex plane
    cplot(scorergi, [-8,8], [-8,8], points=50000)
```



Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> scorergi(0); 1/(power(3,'7/6')*gamma('2/3'))
0.2049755424820002450503075
0.2049755424820002450503075
>>> diff(scorergi, 0); 1/(power(3,'5/6')*gamma('1/3'))
0.1494294524512754526382746
0.1494294524512754526382746
>>> scorergi(+inf); scorergi(-inf)
0.0
0.0
>>> scorergi(1)
0.2352184398104379375986902
>>> scorergi(-1)
-0.1166722172960152826494198
```

Evaluation for large arguments:

```
>>> scorergi(10)
0.03189600510067958798062034
>>> scorergi(100)
0.003183105228162961476590531
>>> scorergi(1000000)
0.000000318309886183790672174387 3
>>> 1/(pi*1000000)
```

```
0.00000031830988618379067153776752
>>> scorergi(-1000)
-0.08358288400262780392338014
>>> scorergi(-100000)
0.02886866118619660226809581
>>> scorergi(50+10j)
(0.006121410279977857879098 - 0.001224335676457532180747917j)
>>> scorergi(-50-10j)
(5.236047850352252236372551e+29 - 3.082542242337013814822 28e+29j)
>>> scorergi(100000j)
(-8.80665928533623105267902 5e+6474077 + 8.6847313035008355148509 62e+6474077j)
```

Verifying the connection between Gi and Hi:

```
>>> z = 0.25
>>> scorergi(z) + scorerhi(z)
0.72874690393621500786945 43
>>> airybi(z)
0.72874690393621500786945 43
```

Verifying the differential equation:

```
>>> for z in [-3.4, 0, 2.5, 1+2j]:
...     chop(diff(scorergi,z,2) - z*scorergi(z))
...
-0.31830988618379067153776752
-0.31830988618379067153776752
-0.31830988618379067153776752
-0.31830988618379067153776752
```

Verifying the integral representation:

```
>>> z = 0.5
>>> scorergi(z)
0.24472104327655819769105 39
>>> Ai,Bi = airyai,airybi
>>> Bi(z)*(Ai(inf,-1)-Ai(z,-1)) + Ai(z)*(Bi(z,-1)-Bi(0,-1))
0.24472104327655819769105 39
```

References

1. *[DLMF]* section 9.12: Scorer Functions

scorerhi()

mpmath.**scorerhi**(*z*, ***kwargs*)

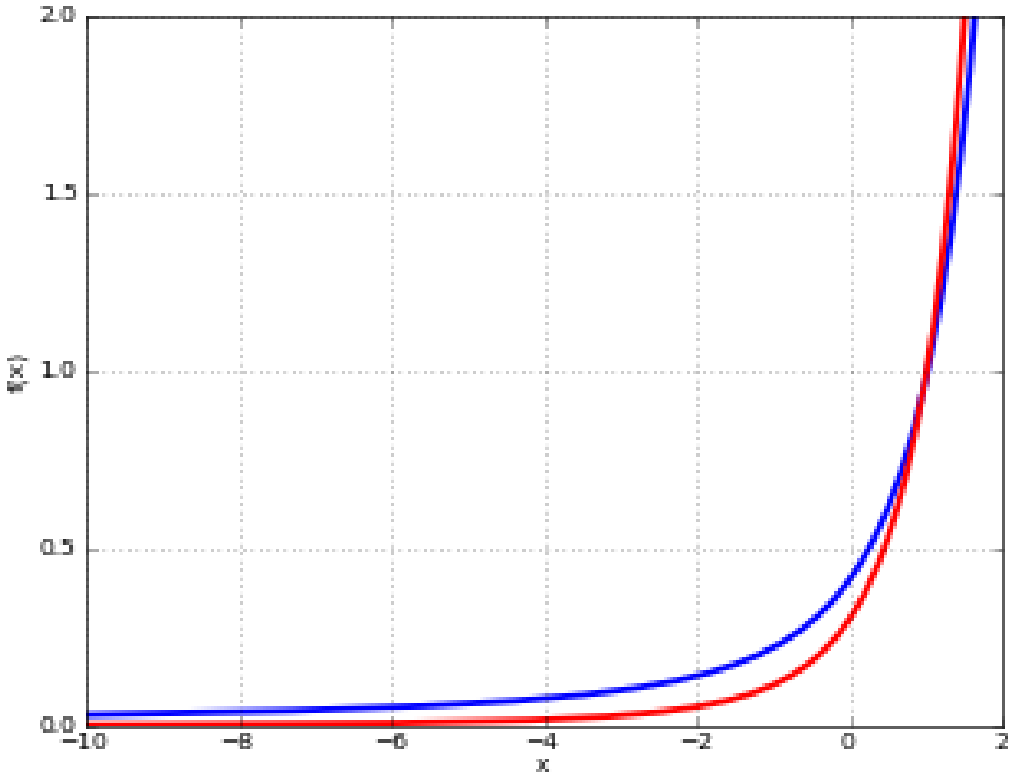
Evaluates the second Scorer function

$$\mathrm{Hi}(z) = \mathrm{Bi}(z) \int_{-\infty}^{z} \mathrm{Ai}(t)dt - \mathrm{Ai}(z) \int_{-\infty}^{z} \mathrm{Bi}(t)dt$$

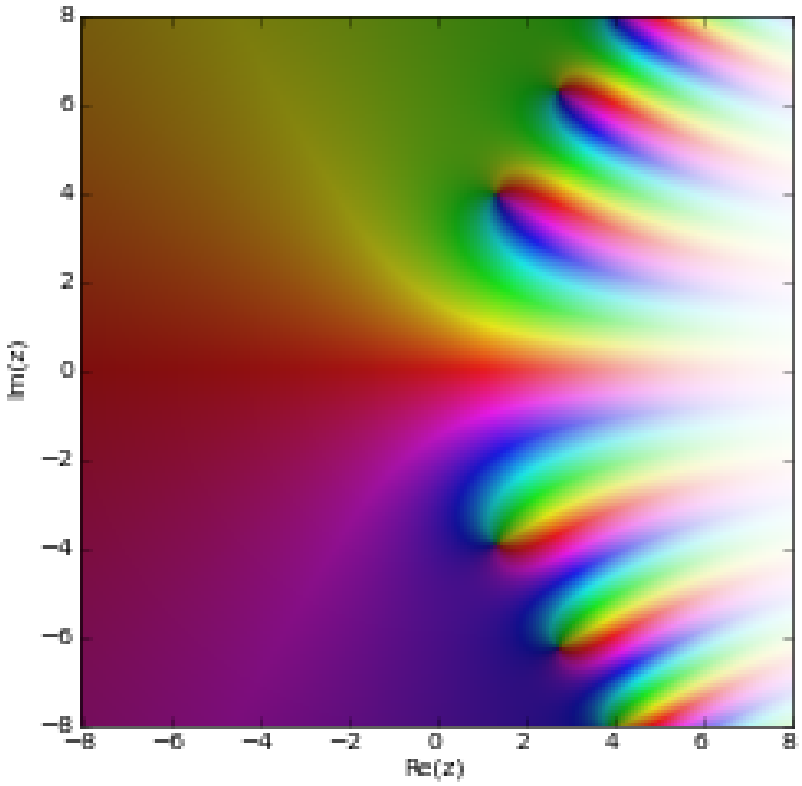
which gives a particular solution to the inhomogeneous Airy differential equation $f''(z) - zf(z) = 1/\pi$. See also *scorergi()*.

Plots

```
# Scorer function Hi(x) and Hi'(x) on the real line
plot([scorerhi, diffun(scorerhi)], [-10,2], [0,2])
```




```
    # Scorer function Hi(z) in the complex plane
    cplot(scorerhi, [-8,8], [-8,8], points=50000)
```



Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> scorerhi(0); 2/(power(3,'7/6')*gamma('2/3'))
0.4099510849640004901006149
0.4099510849640004901006149
>>> diff(scorerhi,0); 2/(power(3,'5/6')*gamma('1/3'))
0.2988589049025509052765491
0.2988589049025509052765491
>>> scorerhi(+inf); scorerhi(-inf)
+inf
0.0
>>> scorerhi(1)
0.9722051551424333218376886
>>> scorerhi(-1)
0.2206696067929598945381098
```

Evaluation for large arguments:

```
>>> scorerhi(10)
455641153.5163291358991077
>>> scorerhi(100)
6.041223996670201399005265e+288
>>> scorerhi(1000000)
7.138269638197858094311122e+289529652
>>> scorerhi(-10)
```

```
0.031768535282502272741501 1
>>> scorerhi(-100)
0.003183092495767499864680483
>>> scorerhi(100j)
(-6.3661977165456721229838572e-9 + 0.003183098861710582761688475j)
>>> scorerhi(50+50j)
(-5.3220762673214356692903346e+63 + 1.4784502911652437897494276e+65j)
>>> scorerhi(-1000-1000j)
(0.00015915494325105027965655384 - 0.00015915494309189533497310 9j)
```

Verifying the differential equation:

```
>>> for z in [-3.4, 0, 2, 1+2j]:
...     chop(diff(scorerhi,z,2) - z*scorerhi(z))
...
0.31830988618379067153776752
0.31830988618379067153776752
0.31830988618379067153776752
0.31830988618379067153776752
```

Verifying the integral representation:

```
>>> z = 0.5
>>> scorerhi(z)
0.60955599982659729560899494
>>> Ai,Bi = airyai,airybi
>>> Bi(z)*(Ai(z,-1)-Ai(-inf,-1)) - Ai(z)*(Bi(z,-1)-Bi(-inf,-1))
0.60955599982659729560899494
```

Coulomb wave functions

coulombf()

mpmath.**coulombf**(*l*, *eta*, *z*)

Calculates the regular Coulomb wave function

$$F_l(\eta, z) = C_l(\eta) z^{l+1} e^{-iz} \, {}_1F_1(l+1-i\eta, 2l+2, 2iz)$$

where the normalization constant $C_l(\eta)$ is as calculated by *coulombc()*. This function solves the differential equation

$$f''(z) + \left(1 - \frac{2\eta}{z} - \frac{l(l+1)}{z^2}\right) f(z) = 0.$$

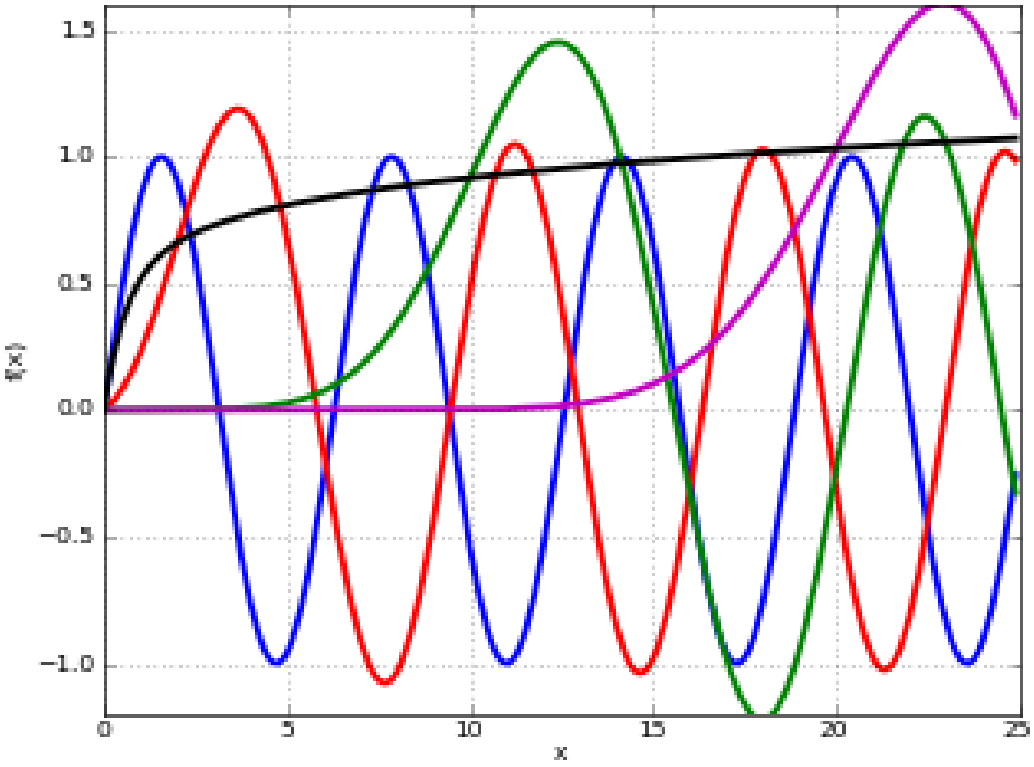
A second linearly independent solution is given by the irregular Coulomb wave function $G_l(\eta, z)$ (see *coulombg()*) and thus the general solution is $f(z) = C_1 F_l(\eta, z) + C_2 G_l(\eta, z)$ for arbitrary constants C_1, C_2. Physically, the Coulomb wave functions give the radial solution to the Schrodinger equation for a point particle in a $1/z$ potential; z is then the radius and l, η are quantum numbers.

The Coulomb wave functions with real parameters are defined in Abramowitz & Stegun, section 14. However, all parameters are permitted to be complex in this implementation (see references).

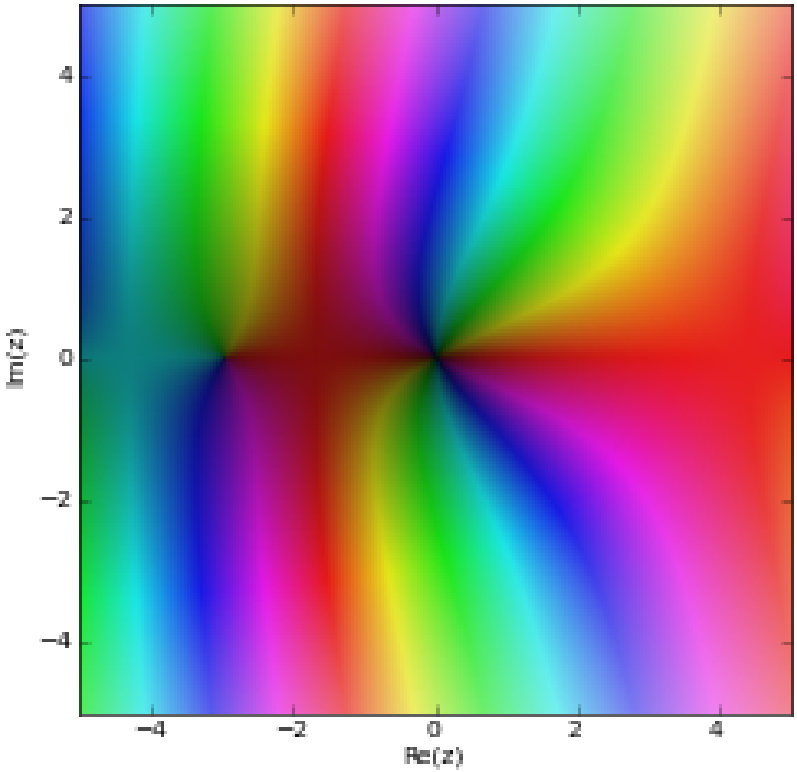
Plots

```
# Regular Coulomb wave functions -- equivalent to figure 14.3 in A&S
F1 = lambda x: coulombf(0,0,x)
F2 = lambda x: coulombf(0,1,x)
F3 = lambda x: coulombf(0,5,x)
```

```
    F4 = lambda x: coulombf(0,10,x)
    F5 = lambda x: coulombf(0,x/2,x)
    plot([F1,F2,F3,F4,F5], [0,25], [-1.2,1.6])
```




```
    # Regular Coulomb wave function in the complex plane
    cplot(lambda z: coulombf(1,1,z), points=50000)
```



Examples

Evaluation is supported for arbitrary magnitudes of z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> coulombf(2, 1.5, 3.5)
0.40809989610887611874266445
>>> coulombf(-2, 1.5, 3.5)
0.71030408494925367475334653
>>> coulombf(2, 1.5, '1e-10')
4.1433249174922564487707690e-33
>>> coulombf(2, 1.5, 1000)
0.44826231403255670507161790
>>> coulombf(2, 1.5, 10**10)
-0.066804196437694360046619
```

Verifying the differential equation:

```
>>> l, eta, z = 2, 3, mpf(2.75)
>>> A, B = 1, 2
>>> f = lambda z: A*coulombf(l,eta,z) + B*coulombg(l,eta,z)
>>> chop(diff(f,z,2) + (1-2*eta/z - l*(l+1)/z**2)*f(z))
0.0
```

A Wronskian relation satisfied by the Coulomb wave functions:

```
>>> l = 2
>>> eta = 1.5
```

```
>>> F = lambda z: coulombf(l,eta,z)
>>> G = lambda z: coulombg(l,eta,z)
>>> for z in [3.5, -1, 2+3j]:
...     chop(diff(F,z)*G(z) - F(z)*diff(G,z))
...
1.0
1.0
1.0
```

Another Wronskian relation:

```
>>> F = coulombf
>>> G = coulombg
>>> for z in [3.5, -1, 2+3j]:
...     chop(F(l-1,eta,z)*G(l,eta,z)-F(l,eta,z)*G(l-1,eta,z) - l/sqrt(l**2+eta**2))
...
0.0
0.0
0.0
```

An integral identity connecting the regular and irregular wave functions:

```
>>> l, eta, z = 4+j, 2-j, 5+2j
>>> coulombf(l,eta,z) + j*coulombg(l,eta,z)
(0.7997977752284033239714479 + 0.9294486669502295512503127j)
>>> g = lambda t: exp(-t)*t**(l-j*eta)*(t+2*j*z)**(l+j*eta)
>>> j*exp(-j*z)*z**(-l)/fac(2*l+1)/coulombc(l,eta)*quad(g, [0,inf])
(0.7997977752284033239714479 + 0.9294486669502295512503127j)
```

Some test case with complex parameters, taken from Michel [2]:

```
>>> mp.dps = 15
>>> coulombf(1+0.1j, 50+50j, 100.156)
(-1.02107292320897e+15 - 2.83675545731519e+15j)
>>> coulombg(1+0.1j, 50+50j, 100.156)
(2.83675545731519e+15 - 1.02107292320897e+15j)
>>> coulombf(1e-5j, 10+1e-5j, 0.1+1e-6j)
(4.30566371247811e-14 - 9.03347835361657e-19j)
>>> coulombg(1e-5j, 10+1e-5j, 0.1+1e-6j)
(778709182061.134 + 18418936.2660553j)
```

The following reproduces a table in Abramowitz & Stegun, at twice the precision:

```
>>> mp.dps = 10
>>> eta = 2; z = 5
>>> for l in [5, 4, 3, 2, 1, 0]:
...     print("%s %s %s" % (l, coulombf(l,eta,z),
...         diff(lambda z: coulombf(l,eta,z), z)))
...
5 0.09079533488 0.1042553261
4 0.2148205331 0.2029591779
3 0.4313159311 0.320534053
2 0.7212774133 0.3952408216
1 0.9935056752 0.3708676452
0 1.143337392 0.2937960375
```

References

1. I.J. Thompson & A.R. Barnett, "Coulomb and Bessel Functions of Complex Arguments and Order", J. Comp. Phys., vol 64, no. 2, June 1986.

2.N. Michel, "Precise Coulomb wave functions for a wide range of complex l, η and z", http://arxiv.org/abs/physics/0702051v1

coulombg()

mpmath.**coulombg**(*l*, *eta*, *z*)

Calculates the irregular Coulomb wave function

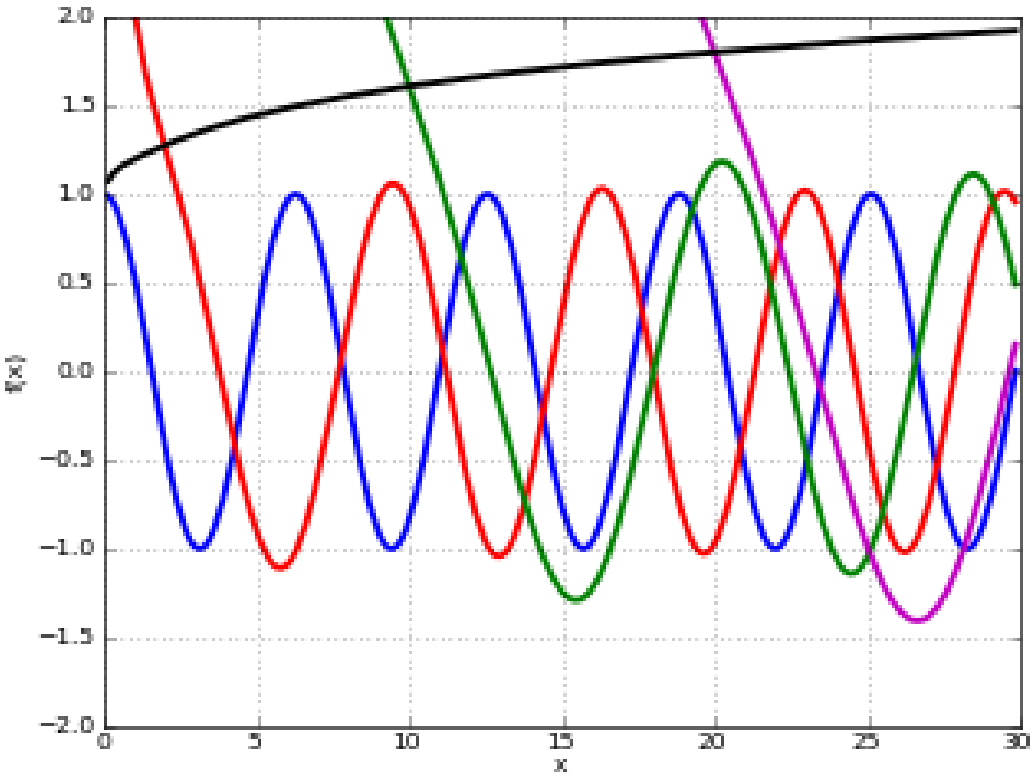
$$G_l(\eta, z) = \frac{F_l(\eta, z)\cos(\chi) - F_{-l-1}(\eta, z)}{\sin(\chi)}$$

where $\chi = \sigma_l - \sigma_{-l-1} - (l + 1/2)\pi$ and $\sigma_l(\eta) = (\ln\Gamma(1 + l + i\eta) - \ln\Gamma(1 + l - i\eta))/(2i)$.

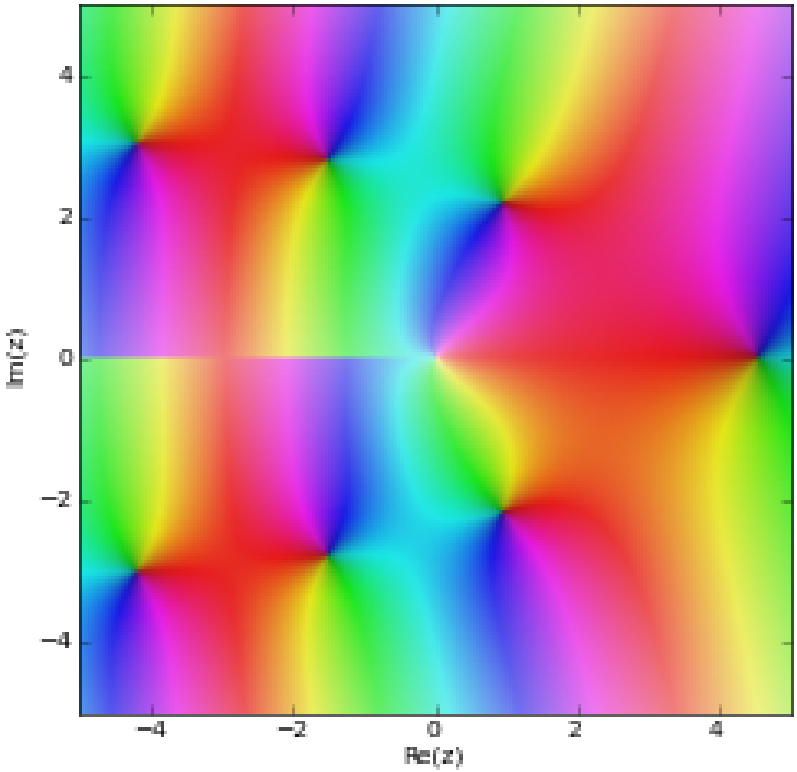
See *coulombf()* for additional information.

Plots

```
# Irregular Coulomb wave functions -- equivalent to figure 14.5 in A&S
F1 = lambda x: coulombg(0,0,x)
F2 = lambda x: coulombg(0,1,x)
F3 = lambda x: coulombg(0,5,x)
F4 = lambda x: coulombg(0,10,x)
F5 = lambda x: coulombg(0,x/2,x)
plot([F1,F2,F3,F4,F5], [0,30], [-2,2])
```




```
# Irregular Coulomb wave function in the complex plane
cplot(lambda z: coulombg(1,1,z), points=50000)
```



Examples

Evaluation is supported for arbitrary magnitudes of z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> coulombg(-2, 1.5, 3.5)
1.380011900612186346255524
>>> coulombg(2, 1.5, 3.5)
1.919153700722748795245926
>>> coulombg(-2, 1.5, '1e-10')
201126715824.7329115106793
>>> coulombg(-2, 1.5, 1000)
0.1802071520691149410425512
>>> coulombg(-2, 1.5, 10**10)
0.6521030200616780709297 94
```

The following reproduces a table in Abramowitz & Stegun, at twice the precision:

```
>>> mp.dps = 10
>>> eta = 2; z = 5
>>> for l in [1, 2, 3, 4, 5]:
...     print("%s %s %s" % (l, coulombg(l,eta,z),
...         -diff(lambda z: coulombg(l,eta,z), z)))
...
1 1.08148276 0.6028279961
2 1.496877075 0.5661803178
3 2.048694714 0.7959909551
4 3.09408669 1.731802374
```

```
5 5.629840456 4.549343289
```

Evaluation close to the singularity at $z = 0$:

```
>>> mp.dps = 15
>>> coulombg(0,10,1)
3088184933.67358
>>> coulombg(0,10,'1e-10')
5554866000719.8
>>> coulombg(0,10,'1e-100')
5554866221524.1
```

Evaluation with a half-integer value for l:

```
>>> coulombg(1.5, 1, 10)
0.852320038297334
```

coulombc()

mpmath.**coulombc**(l, *eta*)

Gives the normalizing Gamow constant for Coulomb wave functions,

$$C_l(\eta) = 2^l \exp\left(-\pi\eta/2 + [\ln\Gamma(1+l+i\eta) + \ln\Gamma(1+l-i\eta)]/2 - \ln\Gamma(2l+2)\right),$$

where the log gamma function with continuous imaginary part away from the negative half axis (see *loggamma()*) is implied.

This function is used internally for the calculation of Coulomb wave functions, and automatically cached to make multiple evaluations with fixed l, η fast.

Confluent U and Whittaker functions

hyperu()

mpmath.**hyperu**(a, b, z)

Gives the Tricomi confluent hypergeometric function U, also known as the Kummer or confluent hypergeometric function of the second kind. This function gives a second linearly independent solution to the confluent hypergeometric differential equation (the first is provided by ${}_1F_1$ – see *hyp1f1()*).

Examples

Evaluation for arbitrary complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyperu(2,3,4)
0.0625
>>> hyperu(0.25, 5, 1000)
0.1779949416140579573763523
>>> hyperu(0.25, 5, -1000)
(0.1256256609322773150118907 - 0.1256256609322773150118907j)
```

The U function may be singular at $z = 0$:

```
>>> hyperu(1.5, 2, 0)
+inf
>>> hyperu(1.5, -2, 0)
0.171943492128840011260367l
```

Verifying the differential equation:

```
>>> a, b = 1.5, 2
>>> f = lambda z: hyperu(a,b,z)
>>> for z in [-10, 3, 3+4j]:
...     chop(z*diff(f,z,2) + (b-z)*diff(f,z) - a*f(z))
...
0.0
0.0
0.0
```

An integral representation:

```
>>> a,b,z = 2, 3.5, 4.25
>>> hyperu(a,b,z)
0.06674960718150520648014567
>>> quad(lambda t: exp(-z*t)*t**(a-1)*(1+t)**(b-a-1),[0,inf]) / gamma(a)
0.06674960718150520648014567
```

[1] http://people.math.sfu.ca/~cbm/aands/page_504.htm

whitm()

mpmath.**whitm**(k, m, z)

Evaluates the Whittaker function $M(k, m, z)$, which gives a solution to the Whittaker differential equation

$$\frac{d^2 f}{dz^2} + \left(-\frac{1}{4} + \frac{k}{z} + \frac{(\frac{1}{4} - m^2)}{z^2}\right) f = 0.$$

A second solution is given by *whitw()*.

The Whittaker functions are defined in Abramowitz & Stegun, section 13.1. They are alternate forms of the confluent hypergeometric functions ${}_1F_1$ and U:

$$M(k, m, z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} {}_1F_1(\tfrac{1}{2} + m - k, 1 + 2m, z)$$
$$W(k, m, z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} U(\tfrac{1}{2} + m - k, 1 + 2m, z).$$

Examples

Evaluation for arbitrary real and complex arguments is supported:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> whitm(1, 1, 1)
0.7302596799460411820509668
>>> whitm(1, 1, -1)
(0.0 - 1.417977827655098025684246j)
>>> whitm(j, j/2, 2+3j)
(3.245477713363581112736478 - 0.822879187542699127327782j)
>>> whitm(2, 3, 100000)
4.303985255686378497193063e+21707
```

Evaluation at zero:

```
>>> whitm(1,-1,0); whitm(1,-0.5,0); whitm(1,0,0)
+inf
nan
0.0
```

We can verify that *whitm()* numerically satisfies the differential equation for arbitrarily chosen values:

```
>>> k = mpf(0.25)
>>> m = mpf(1.5)
>>> f = lambda z: whitm(k,m,z)
>>> for z in [-1, 2.5, 3, 1+2j]:
...     chop(diff(f,z,2) + (-0.25 + k/z + (0.25-m**2)/z**2)*f(z))
...
0.0
0.0
0.0
0.0
```

An integral involving both *whitm()* and *whitw()*, verifying evaluation along the real axis:

```
>>> quad(lambda x: exp(-x)*whitm(3,2,x)*whitw(1,-2,x), [0,inf])
3.43886984257680022520734 1
>>> 128/(21*sqrt(pi))
3.43886984257680022520734 1
```

whitw()

mpmath.**whitw**(k, m, z)

Evaluates the Whittaker function $W(k, m, z)$, which gives a second solution to the Whittaker differential equation. (See *whitm()*.)

Examples

Evaluation for arbitrary real and complex arguments is supported:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> whitw(1, 1, 1)
1.1953206310758115566 1012
>>> whitw(1, 1, -1)
(-0.9424875979222187313924639 - 0.2607738054097702293308689j)
>>> whitw(j, j/2, 2+3j)
(0.1782899315111033879430369 - 0.0160957836040364934016940 6j)
>>> whitw(2, 3, 100000)
1.8877051148895274468912 74e-21705
>>> whitw(-1, -1, 100)
1.9052506928240461624620 58e-24
```

Evaluation at zero:

```
>>> for m in [-1, -0.5, 0, 0.5, 1]:
...     whitw(1, m, 0)
...
+inf
nan
0.0
nan
+inf
```

We can verify that *whitw()* numerically satisfies the differential equation for arbitrarily chosen values:

```
>>> k = mpf(0.25)
>>> m = mpf(1.5)
>>> f = lambda z: whitw(k,m,z)
```

```
>>> for z in [-1, 2.5, 3, 1+2j]:
...     chop(diff(f,z,2) + (-0.25 + k/z + (0.25-m**2)/z**2)*f(z))
...
0.0
0.0
0.0
0.0
```

Parabolic cylinder functions

pcfd()

mpmath.**pcfd**(*n*, *z*, ***kwargs*)

Gives the parabolic cylinder function in Whittaker's notation $D_n(z) = U(-n-1/2, z)$ (see *pcfu()*). It solves the differential equation

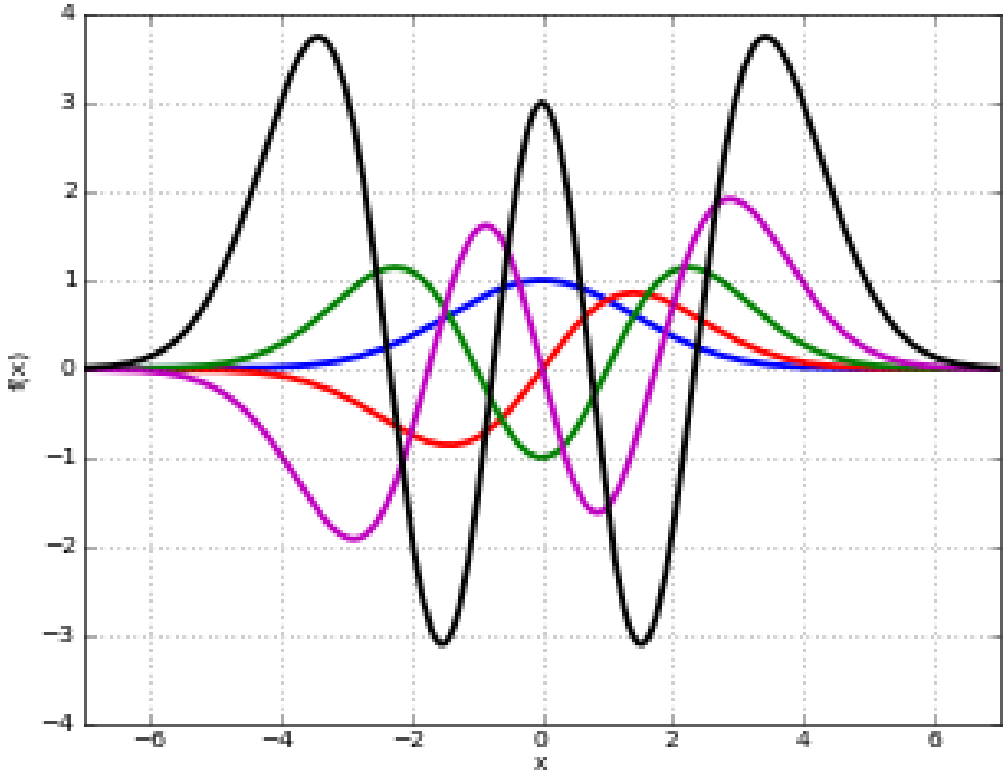
$$y'' + \left(n + \frac{1}{2} - \frac{1}{4}z^2\right) y = 0.$$

and can be represented in terms of Hermite polynomials (see *hermite()*) as

$$D_n(z) = 2^{-n/2} e^{-z^2/4} H_n\left(\frac{z}{\sqrt{2}}\right).$$

Plots

```
# Parabolic cylinder function D_n(x) on the real line for n=0,1,2,3,4
d0 = lambda x: pcfd(0,x)
d1 = lambda x: pcfd(1,x)
d2 = lambda x: pcfd(2,x)
d3 = lambda x: pcfd(3,x)
d4 = lambda x: pcfd(4,x)
plot([d0,d1,d2,d3,d4],[-7,7])
```



Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> pcfd(0,0); pcfd(1,0); pcfd(2,0); pcfd(3,0)
1.0
0.0
-1.0
0.0
>>> pcfd(4,0); pcfd(-3,0)
3.0
0.62665706865775012560394131
>>> pcfd('1/2', 2+3j)
(-5.3633311612329207348490561 - 3.8588778217900107141634871j)
>>> pcfd(2, -10)
1.3749064426314380388715151e-9
```

Verifying the differential equation:

```
>>> n = mpf(2.5)
>>> y = lambda z: pcfd(n,z)
>>> z = 1.75
>>> chop(diff(y,z,2) + (n+0.5-0.25*z**2)*y(z))
0.0
```

Rational Taylor series expansion when n is an integer:

```
>>> taylor(lambda z: pcfd(5,z), 0, 7)
[0.0, 15.0, 0.0, -13.75, 0.0, 3.96875, 0.0, -0.6015625]
```

pcfu()

mpmath.**pcfu**(*a*, *z*, ***kwargs*)

Gives the parabolic cylinder function $U(a,z)$, which may be defined for $\Re(z) > 0$ in terms of the confluent U-function (see *hyperu()*) by

$$U(a,z) = 2^{-\frac{1}{4}-\frac{a}{2}} e^{-\frac{1}{4}z^2} U\left(\frac{a}{2}+\frac{1}{4}, \frac{1}{2}, \frac{1}{2}z^2\right)$$

or, for arbitrary z,

$$e^{-\frac{1}{4}z^2} U(a,z) = U(a,0)\,{}_1F_1\left(-\tfrac{a}{2}+\tfrac{1}{4}; \tfrac{1}{2}; -\tfrac{1}{2}z^2\right) + U'(a,0) z\,{}_1F_1\left(-\tfrac{a}{2}+\tfrac{3}{4}; \tfrac{3}{2}; -\tfrac{1}{2}z^2\right).$$

Examples

Connection to other functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> z = mpf(3)
>>> pcfu(0.5,z)
0.03210358129311151450551963
>>> sqrt(pi/2)*exp(z**2/4)*erfc(z/sqrt(2))
0.03210358129311151450551963
>>> pcfu(0.5,-z)
23.75012332835297233711255
>>> sqrt(pi/2)*exp(z**2/4)*erfc(-z/sqrt(2))
23.75012332835297233711255
>>> pcfu(0.5,-z)
23.75012332835297233711255
>>> sqrt(pi/2)*exp(z**2/4)*erfc(-z/sqrt(2))
23.75012332835297233711255
```

pcfv()

mpmath.**pcfv**(*a*, *z*, ***kwargs*)

Gives the parabolic cylinder function $V(a,z)$, which can be represented in terms of *pcfu()* as

$$V(a,z) = \frac{\Gamma(a+\frac{1}{2})(U(a,-z) - \sin(\pi a) U(a,z)}{\pi}.$$

Examples

Wronskian relation between U and V:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a, z = 2, 3
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z)
0.7978845608028653558798921
>>> sqrt(2/pi)
0.7978845608028653558798921
>>> a, z = 2.5, 3
```

```
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z)
0.797884560802865355879892119
>>> a, z = 0.25, -1
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z)
0.797884560802865355879892119
>>> a, z = 2+1j, 2+3j
>>> chop(pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z))
0.797884560802865355879892119
```

pcfw()

mpmath.**pcfw**(*a*, *z*, ***kwargs*)

Gives the parabolic cylinder function $W(a,z)$ defined in (DLMF 12.14).

Examples

Value at the origin:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a = mpf(0.25)
>>> pcfw(a,0)
0.9722833245718180765617104
>>> power(2,-0.75)*sqrt(abs(gamma(0.25+0.5j*a)/gamma(0.75+0.5j*a)))
0.9722833245718180765617104
>>> diff(pcfw,(a,0),(0,1))
-0.5142533944210078966003624
>>> -power(2,-0.25)*sqrt(abs(gamma(0.75+0.5j*a)/gamma(0.25+0.5j*a)))
-0.5142533944210078966003624
```

3.1.8 Orthogonal polynomials

An orthogonal polynomial sequence is a sequence of polynomials $P_0(x), P_1(x), \ldots$ of degree $0, 1, \ldots$, which are mutually orthogonal in the sense that

$$\int_S P_n(x) P_m(x) w(x) dx = \begin{cases} c_n \neq 0 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

where S is some domain (e.g. an interval $[a,b] \in \mathbb{R}$) and $w(x)$ is a fixed *weight function*. A sequence of orthogonal polynomials is determined completely by w, S, and a normalization convention (e.g. $c_n = 1$). Applications of orthogonal polynomials include function approximation and solution of differential equations.

Orthogonal polynomials are sometimes defined using the differential equations they satisfy (as functions of x) or the recurrence relations they satisfy with respect to the order n. Other ways of defining orthogonal polynomials include differentiation formulas and generating functions. The standard orthogonal polynomials can also be represented as hypergeometric series (see Hypergeometric functions), more specifically using the Gauss hypergeometric function ${}_2F_1$ in most cases. The following functions are generally implemented using hypergeometric functions since this is computationally efficient and easily generalizes.

For more information, see the Wikipedia article on orthogonal polynomials.

Legendre functions

legendre()

mpmath.**legendre**(*n*, *x*)

legendre(n, x) evaluates the Legendre polynomial $P_n(x)$. The Legendre polynomials are given by the formula

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

Alternatively, they can be computed recursively using

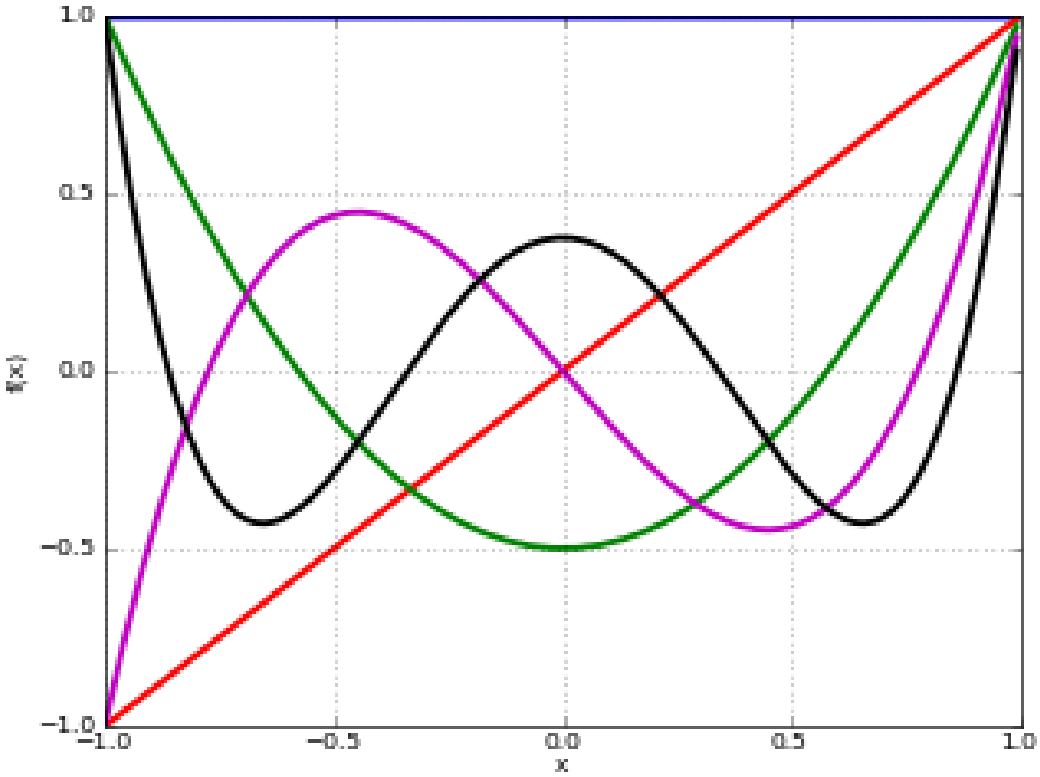
$$P_0(x) = 1$$
$$P_1(x) = x$$
$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x).$$

A third definition is in terms of the hypergeometric function ${}_2F_1$, whereby they can be generalized to arbitrary n:

$$P_n(x) = {}_2F_1\left(-n, n+1, 1, \frac{1-x}{2}\right)$$

Plots

```
# Legendre polynomials P_n(x) on [-1,1] for n=0,1,2,3,4
f0 = lambda x: legendre(0,x)
f1 = lambda x: legendre(1,x)
f2 = lambda x: legendre(2,x)
f3 = lambda x: legendre(3,x)
f4 = lambda x: legendre(4,x)
plot([f0,f1,f2,f3,f4],[-1,1])
```



Basic evaluation

The Legendre polynomials assume fixed values at the points $x = -1$ and $x = 1$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint([legendre(n, 1) for n in range(6)])
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> nprint([legendre(n, -1) for n in range(6)])
[1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
```

The coefficients of Legendre polynomials can be recovered using degree-n Taylor expansion:

```
>>> for n in range(5):
...     nprint(chop(taylor(lambda x: legendre(n, x), 0, n)))
...
[1.0]
[0.0, 1.0]
[-0.5, 0.0, 1.5]
[0.0, -1.5, 0.0, 2.5]
[0.375, 0.0, -3.75, 0.0, 4.375]
```

The roots of Legendre polynomials are located symmetrically on the interval $[-1, 1]$:

```
>>> for n in range(5):
...     nprint(polyroots(taylor(lambda x: legendre(n, x), 0, n)[::-1]))
...
[]
[0.0]
```

```
[-0.57735, 0.57735]
[-0.774597, 0.0, 0.774597]
[-0.861136, -0.339981, 0.339981, 0.861136]
```

An example of an evaluation for arbitrary n:

```
>>> legendre(0.75, 2+4j)
(1.94952805264875 + 2.1071073099422j)
```

Orthogonality

The Legendre polynomials are orthogonal on $[-1, 1]$ with respect to the trivial weight $w(x) = 1$. That is, $P_m(x)P_n(x)$ integrates to zero if $m \neq n$ and to $2/(2n+1)$ if $m = n$:

```
>>> m, n = 3, 4
>>> quad(lambda x: legendre(m,x)*legendre(n,x), [-1, 1])
0.0
>>> m, n = 4, 4
>>> quad(lambda x: legendre(m,x)*legendre(n,x), [-1, 1])
0.222222222222222
```

Differential equation

The Legendre polynomials satisfy the differential equation

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

We can verify this numerically:

```
>>> n = 3.6
>>> x = 0.73
>>> P = legendre
>>> A = diff(lambda t: (1-t**2)*diff(lambda u: P(n,u), t), x)
>>> B = n*(n+1)*P(n,x)
>>> nprint(A+B,1)
9.0e-16
```

legenp()

mpmath.**legenp**(*n*, *m*, *z*, *type=2*)

Calculates the (associated) Legendre function of the first kind of degree *n* and order *m*, $P_n^m(z)$. Taking $m = 0$ gives the ordinary Legendre function of the first kind, $P_n(z)$. The parameters may be complex numbers.

In terms of the Gauss hypergeometric function, the (associated) Legendre function is defined as

$$P_n^m(z) = \frac{1}{\Gamma(1-m)} \frac{(1+z)^{m/2}}{(1-z)^{m/2}} \, {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right).$$

With *type=3* instead of *type=2*, the alternative definition

$$\hat{P}_n^m(z) = \frac{1}{\Gamma(1-m)} \frac{(z+1)^{m/2}}{(z-1)^{m/2}} \, {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right).$$

is used. These functions correspond respectively to `LegendreP[n,m,2,z]` and `LegendreP[n,m,3,z]` in Mathematica.

The general solution of the (associated) Legendre differential equation

$$(1-z^2)f''(z) - 2zf'(z) + \left(n(n+1) - \frac{m^2}{1-z^2}\right) f(z) = 0$$

is given by $C_1 P_n^m(z) + C_2 Q_n^m(z)$ for arbitrary constants C_1, C_2, where $Q_n^m(z)$ is a Legendre function of the second kind as implemented by *legenq()*.

Examples

Evaluation for arbitrary parameters and arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> legenp(2, 0, 10); legendre(2, 10)
149.5
149.5
>>> legenp(-2, 0.5, 2.5)
(1.972260393822275434196053 - 1.972260393822275434196053j)
>>> legenp(2+3j, 1-j, -0.5+4j)
(-3.335677248386698208736542 - 5.663270217461022307645625j)
>>> chop(legenp(3, 2, -1.5, type=2))
28.125
>>> chop(legenp(3, 2, -1.5, type=3))
-28.125
```

Verifying the associated Legendre differential equation:

```
>>> n, m = 2, -0.5
>>> C1, C2 = 1, -3
>>> f = lambda z: C1*legenp(n,m,z) + C2*legenq(n,m,z)
>>> deq = lambda z: (1-z**2)*diff(f,z,2) - 2*z*diff(f,z) + \
...     (n*(n+1)-m**2/(1-z**2))*f(z)
>>> for z in [0, 2, -1.5, 0.5+2j]:
...     chop(deq(mpmathify(z)))
...
0.0
0.0
0.0
0.0
```

legenq()

mpmath.**legenq**(*n*, *m*, *z*, *type=2*)

Calculates the (associated) Legendre function of the second kind of degree *n* and order *m*, $Q_n^m(z)$. Taking $m = 0$ gives the ordinary Legendre function of the second kind, $Q_n(z)$. The parameters may be complex numbers.

The Legendre functions of the second kind give a second set of solutions to the (associated) Legendre differential equation. (See *legenp()*.) Unlike the Legendre functions of the first kind, they are not polynomials of z for integer n, m but rational or logarithmic functions with poles at $z = \pm 1$.

There are various ways to define Legendre functions of the second kind, giving rise to different complex structure. A version can be selected using the *type* keyword argument. The *type=2* and *type=3* functions are given respectively by

$$Q_n^m(z) = \frac{\pi}{2\sin(\pi m)}\left(\cos(\pi m) P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} P_n^{-m}(z)\right)$$

$$\hat{Q}_n^m(z) = \frac{\pi}{2\sin(\pi m)} e^{\pi i m}\left(\hat{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} \hat{P}_n^{-m}(z)\right)$$

where P and $\hat{P}$ are the *type=2* and *type=3* Legendre functions of the first kind. The formulas above should be understood as limits when m is an integer.

These functions correspond to `LegendreQ[n,m,2,z]` (or `LegendreQ[n,m,z]`) and `LegendreQ[n,m,3,z]` in Mathematica. The *type=3* function is essentially the same as the function defined in Abramowitz & Stegun (eq. 8.1.3) but with $(z+1)^{m/2}(z-1)^{m/2}$ instead of $(z^2-1)^{m/2}$, giving slightly different branches.

Examples

Evaluation for arbitrary parameters and arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> legenq(2, 0, 0.5)
-0.8186632680417568557122028
>>> legenq(-1.5, -2, 2.5)
(0.6655964618250228714288277 + 0.3937692045497259717762649j)
>>> legenq(2-j, 3+4j, -6+5j)
(-10001.95256487468541686564 - 6011.691337610097577791134j)
```

Different versions of the function:

```
>>> legenq(2, 1, 0.5)
0.7298060598018049369381857
>>> legenq(2, 1, 1.5)
(-7.902916572420817192300921 + 0.1998650072605976600724502j)
>>> legenq(2, 1, 0.5, type=3)
(2.040524284763495081918338 - 0.7298060598018049369381857j)
>>> chop(legenq(2, 1, 1.5, type=3))
-0.1998650072605976600724502
```

Chebyshev polynomials

chebyt()

mpmath.**chebyt**(n, x)

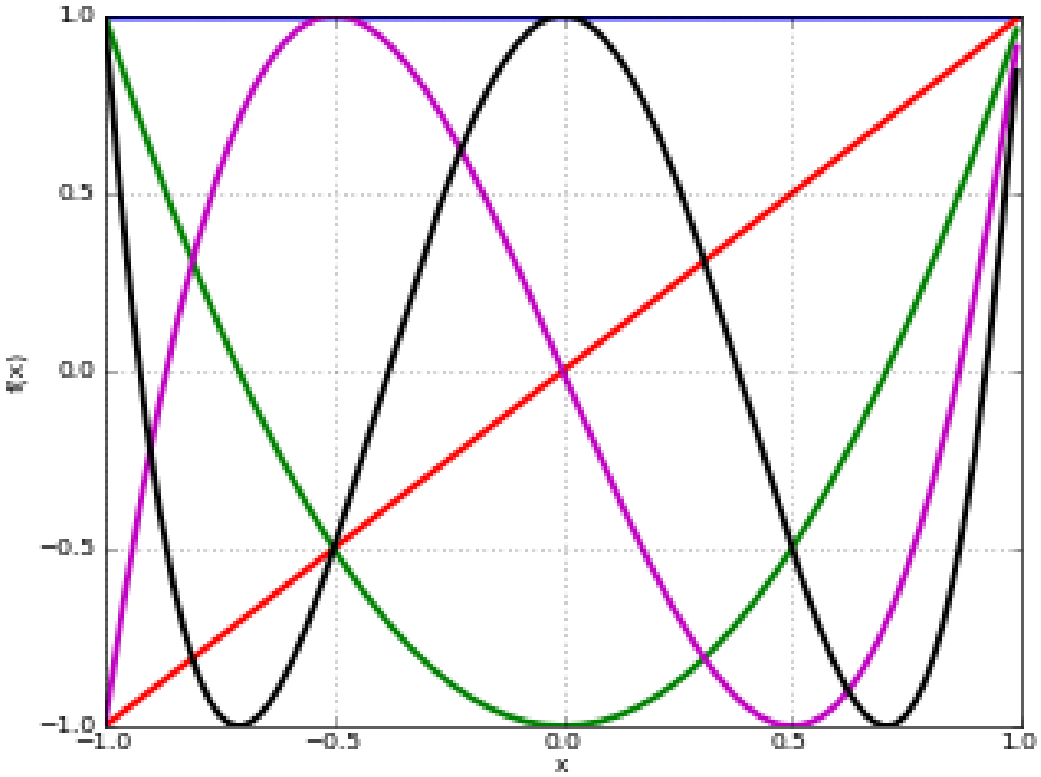
`chebyt(n, x)` evaluates the Chebyshev polynomial of the first kind $T_n(x)$, defined by the identity

$$T_n(\cos x) = \cos(nx).$$

The Chebyshev polynomials of the first kind are a special case of the Jacobi polynomials, and by extension of the hypergeometric function ${}_2F_1$. They can thus also be evaluated for nonintegral n.

Plots

```
# Chebyshev polynomials T_n(x) on [-1,1] for n=0,1,2,3,4
f0 = lambda x: chebyt(0,x)
f1 = lambda x: chebyt(1,x)
f2 = lambda x: chebyt(2,x)
f3 = lambda x: chebyt(3,x)
f4 = lambda x: chebyt(4,x)
plot([f0,f1,f2,f3,f4],[-1,1])
```



Basic evaluation

The coefficients of the n-th polynomial can be recovered using using degree-n Taylor expansion:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(chop(taylor(lambda x: chebyt(n, x), 0, n)))
...
[1.0]
[0.0, 1.0]
[-1.0, 0.0, 2.0]
[0.0, -3.0, 0.0, 4.0]
[1.0, 0.0, -8.0, 0.0, 8.0]
```

Orthogonality

The Chebyshev polynomials of the first kind are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = 1/\sqrt{1-x^2}$:

```
>>> f = lambda x: chebyt(m,x)*chebyt(n,x)/sqrt(1-x**2)
>>> m, n = 3, 4
>>> nprint(quad(f, [-1, 1]),1)
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.57079632596448
```

chebyu()

mpmath.**chebyu**(n, x)

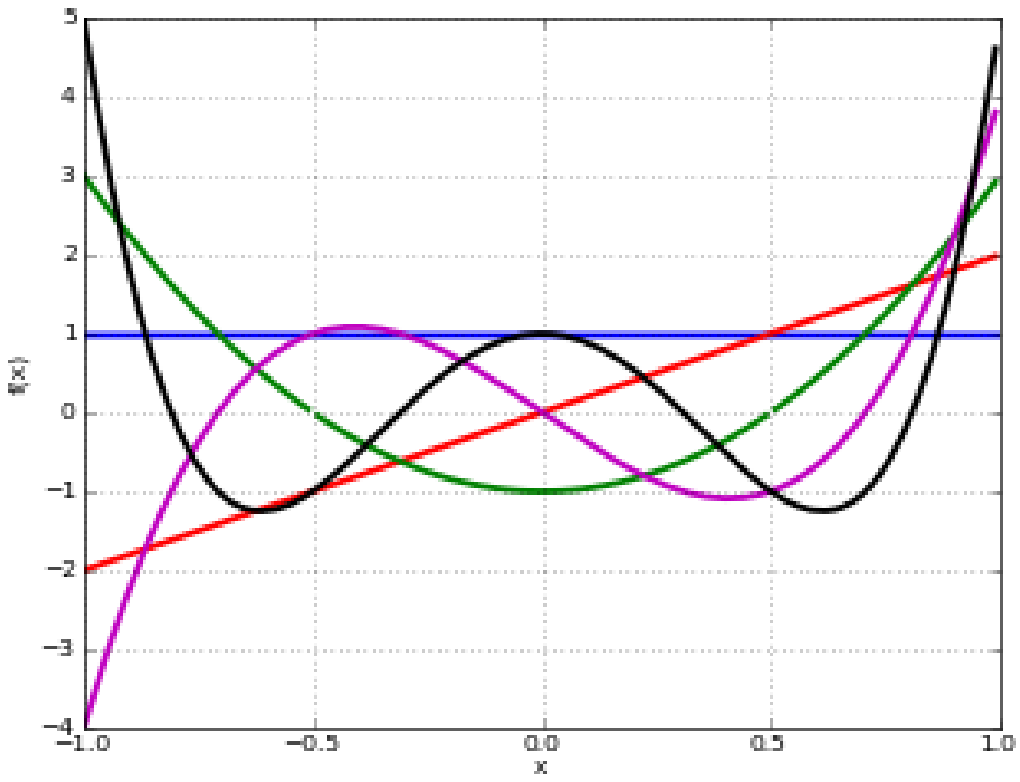
`chebyu(n, x)` evaluates the Chebyshev polynomial of the second kind $U_n(x)$, defined by the identity

$$U_n(\cos x) = \frac{\sin((n+1)x)}{\sin(x)}.$$

The Chebyshev polynomials of the second kind are a special case of the Jacobi polynomials, and by extension of the hypergeometric function ${}_2F_1$. They can thus also be evaluated for nonintegral n.

Plots

```
# Chebyshev polynomials U_n(x) on [-1,1] for n=0,1,2,3,4
f0 = lambda x: chebyu(0,x)
f1 = lambda x: chebyu(1,x)
f2 = lambda x: chebyu(2,x)
f3 = lambda x: chebyu(3,x)
f4 = lambda x: chebyu(4,x)
plot([f0,f1,f2,f3,f4],[-1,1])
```



Basic evaluation

The coefficients of the n-th polynomial can be recovered using using degree-n Taylor expansion:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(chop(taylor(lambda x: chebyu(n, x), 0, n)))
```

```
...
[1.0]
[0.0, 2.0]
[-1.0, 0.0, 4.0]
[0.0, -4.0, 0.0, 8.0]
[1.0, 0.0, -12.0, 0.0, 16.0]
```

Orthogonality

The Chebyshev polynomials of the second kind are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = \sqrt{1-x^2}$:

```
>>> f = lambda x: chebyu(m,x)*chebyu(n,x)*sqrt(1-x**2)
>>> m, n = 3, 4
>>> quad(f, [-1, 1])
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.5707963267949
```

Jacobi polynomials

jacobi()

mpmath.**jacobi**(*n*, *a*, *b*, *z*)

`jacobi(n, a, b, x)` evaluates the Jacobi polynomial $P_n^{(a,b)}(x)$. The Jacobi polynomials are a special case of the hypergeometric function ${}_2F_1$ given by:

$$P_n^{(a,b)}(x) = \binom{n+a}{n} {}_2F_1\left(-n, 1+a+b+n, a+1, \frac{1-x}{2}\right).$$

Note that this definition generalizes to nonintegral values of n. When n is an integer, the hypergeometric series terminates after a finite number of terms, giving a polynomial in x.

Evaluation of Jacobi polynomials

A special evaluation is $P_n^{(a,b)}(1) = \binom{n+a}{n}$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> jacobi(4, 0.5, 0.25, 1)
2.4609375
>>> binomial(4+0.5, 4)
2.4609375
```

A Jacobi polynomial of degree n is equal to its Taylor polynomial of degree n. The explicit coefficients of Jacobi polynomials can therefore be recovered easily using *taylor()*:

```
>>> for n in range(5):
...     nprint(taylor(lambda x: jacobi(n,1,2,x), 0, n))
...
[1.0]
[-0.5, 2.5]
[-0.75, -1.5, 5.25]
[0.5, -3.5, -3.5, 10.5]
[0.625, 2.5, -11.25, -7.5, 20.625]
```

For nonintegral n, the Jacobi "polynomial" is no longer a polynomial:

```
>>> nprint(taylor(lambda x: jacobi(0.5,1,2,x), 0, 4))
[0.309983, 1.84119, -1.26933, 1.26699, -1.34808]
```

Orthogonality

The Jacobi polynomials are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = (1-x)^a(1+x)^b$. That is, $w(x)P_n^{(a,b)}(x)P_m^{(a,b)}(x)$ integrates to zero if $m \neq n$ and to a nonzero number if $m = n$.

The orthogonality is easy to verify using numerical quadrature:

```
>>> P = jacobi
>>> f = lambda x: (1-x)**a * (1+x)**b * P(m,a,b,x) * P(n,a,b,x)
>>> a = 2
>>> b = 3
>>> m, n = 3, 4
>>> chop(quad(f, [-1, 1]), 1)
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.9047619047619
```

Differential equation

The Jacobi polynomials are solutions of the differential equation

$$(1-x^2)y'' + (b-a-(a+b+2)x)y' + n(n+a+b+1)y = 0.$$

We can verify that *jacobi()* approximately satisfies this equation:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> a = 2.5
>>> b = 4
>>> n = 3
>>> y = lambda x: jacobi(n,a,b,x)
>>> x = pi
>>> A0 = n*(n+a+b+1)*y(x)
>>> A1 = (b-a-(a+b+2)*x)*diff(y,x)
>>> A2 = (1-x**2)*diff(y,x,2)
>>> nprint(A2 + A1 + A0, 1)
4.0e-12
```

The difference of order 10^{-12} is as close to zero as it could be at 15-digit working precision, since the terms are large:

```
>>> A0, A1, A2
(26560.2328981879, -21503.7641037294, -5056.46879445852)
```

Gegenbauer polynomials

gegenbauer()

mpmath.**gegenbauer**(*n*, *a*, *z*)

Evaluates the Gegenbauer polynomial, or ultraspherical polynomial,

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

When n is a nonnegative integer, this formula gives a polynomial in z of degree n, but all parameters are permitted to be complex numbers. With $a = 1/2$, the Gegenbauer polynomial reduces to a Legendre polynomial.

Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> gegenbauer(3, 0.5, -10)
-2485.0
>>> gegenbauer(1000, 10, 100)
3.012757178975667428359374e+2322
>>> gegenbauer(2+3j, -0.75, -1000j)
(-5038991.358609026523401901 + 9414549.285447104177860806j)
```

Evaluation at negative integer orders:

```
>>> gegenbauer(-4, 2, 1.75)
-1.0
>>> gegenbauer(-4, 3, 1.75)
0.0
>>> gegenbauer(-4, 2j, 1.75)
0.0
>>> gegenbauer(-7, 0.5, 3)
8989.0
```

The Gegenbauer polynomials solve the differential equation:

```
>>> n, a = 4.5, 1+2j
>>> f = lambda z: gegenbauer(n, a, z)
>>> for z in [0, 0.75, -0.5j]:
...     chop((1-z**2)*diff(f,z,2) - (2*a+1)*z*diff(f,z) + n*(n+2*a)*f(z))
...
0.0
0.0
0.0
```

The Gegenbauer polynomials have generating function $(1-2zt+t^2)^{-a}$:

```
>>> a, z = 2.5, 1
>>> taylor(lambda t: (1-2*z*t+t**2)**(-a), 0, 3)
[1.0, 5.0, 15.0, 35.0]
>>> [gegenbauer(n,a,z) for n in range(4)]
[1.0, 5.0, 15.0, 35.0]
```

The Gegenbauer polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1-z^2)^{a-\frac{1}{2}}$:

```
>>> a, n, m = 2.5, 4, 5
>>> Cn = lambda z: gegenbauer(n, a, z, zeroprec=1000)
>>> Cm = lambda z: gegenbauer(m, a, z, zeroprec=1000)
>>> chop(quad(lambda z: Cn(z)*Cm(z)*(1-z**2)*(a-0.5), [-1, 1]))
0.0
```

Hermite polynomials

hermite()

mpmath.**hermite**(*n*, *z*)

Evaluates the Hermite polynomial $H_n(z)$, which may be defined using the recurrence

$$H_0(z) = 1$$
$$H_1(z) = 2z$$
$$H_{n+1} = 2zH_n(z) - 2nH_{n-1}(z).$$

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight e^{-z^2}. More generally, allowing arbitrary complex values of n, the Hermite function $H_n(z)$ is defined as

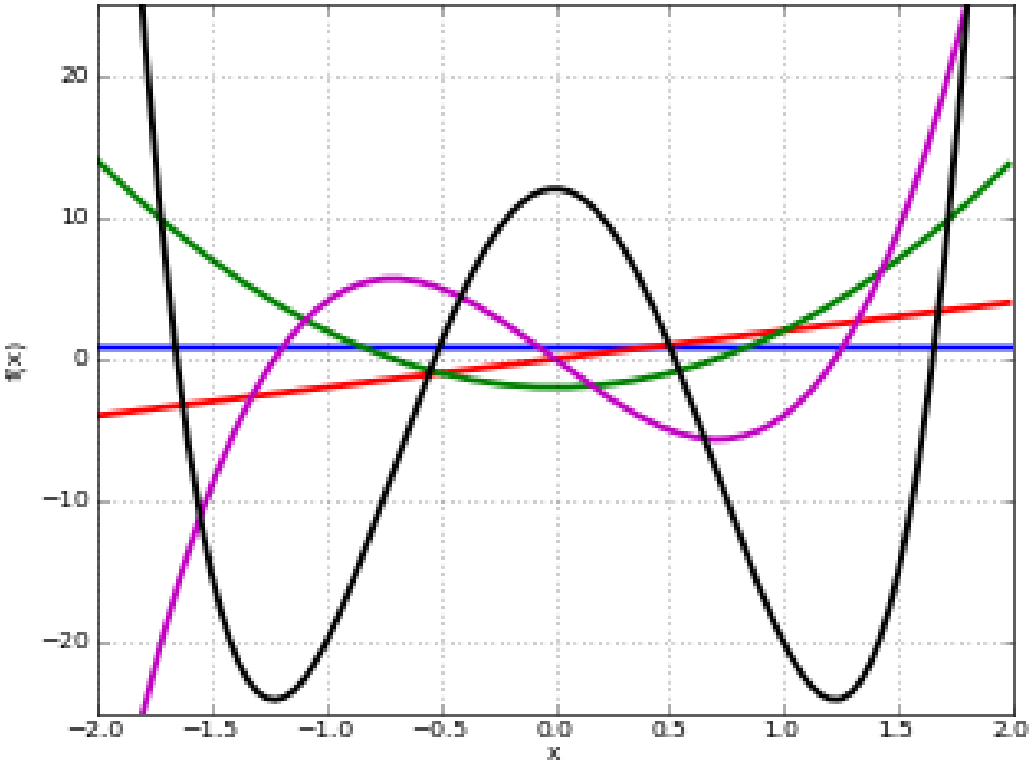
$$H_n(z) = (2z)^n \, {}_2F_0\left(-\frac{n}{2}, \frac{1-n}{2}, -\frac{1}{z^2}\right)$$

for $\Re z > 0$, or generally

$$H_n(z) = 2^n \sqrt{\pi} \left(\frac{1}{\Gamma\left(\frac{1-n}{2}\right)} \, {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma\left(-\frac{n}{2}\right)} \, {}_1F_1\left(\frac{1-n}{2}, \frac{3}{2}, z^2\right) \right).$$

Plots

```
# Hermite polynomials H_n(x) on the real line for n=0,1,2,3,4
f0 = lambda x: hermite(0,x)
f1 = lambda x: hermite(1,x)
f2 = lambda x: hermite(2,x)
f3 = lambda x: hermite(3,x)
f4 = lambda x: hermite(4,x)
plot([f0,f1,f2,f3,f4],[-2,2],[-25,25])
```



Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hermite(0, 10)
1.0
>>> hermite(1, 10); hermite(2, 10)
20.0
398.0
>>> hermite(10000, 2)
4.950440066552087387515653e+19334
>>> hermite(3, -10**8)
-799999999999999988000000000.0
>>> hermite(-3, -10**8)
1.675159751729877682920301e+4342944819032534
>>> hermite(2+3j, -1+2j)
(-0.07652130602993513389421901 - 0.10846624499619145802760076j)
```

Coefficients of the first few Hermite polynomials are:

```
>>> for n in range(7):
...     chop(taylor(lambda z: hermite(n, z), 0, n))
...
[1.0]
[0.0, 2.0]
[-2.0, 0.0, 4.0]
```

```
[0.0, -12.0, 0.0, 8.0]
[12.0, 0.0, -48.0, 0.0, 16.0]
[0.0, 120.0, 0.0, -160.0, 0.0, 32.0]
[-120.0, 0.0, 720.0, 0.0, -480.0, 0.0, 64.0]
```

Values at $z = 0$:

```
>>> for n in range(-5, 9):
...     hermite(n, 0)
...
0.027694591420398687925633 87
0.083333333333333333333333333
0.22155673136318950341227 09
0.5
0.88622692545275801364908 37
1.0
0.0
-2.0
0.0
12.0
0.0
-120.0
0.0
1680.0
```

Hermite functions satisfy the differential equation:

```
>>> n = 4
>>> f = lambda z: hermite(n, z)
>>> z = 1.5
>>> chop(diff(f,z,2) - 2*z*diff(f,z) + 2*n*f(z))
0.0
```

Verifying orthogonality:

```
>>> chop(quad(lambda t: hermite(2,t)*hermite(4,t)*exp(-t**2), [-inf,inf]))
0.0
```

Laguerre polynomials

laguerre()

mpmath.**laguerre**(n, a, z)

Gives the generalized (associated) Laguerre polynomial, defined by

$$L_n^a(z) = \frac{\Gamma(n+b+1)}{\Gamma(b+1)\Gamma(n+1)}\, {}_1F_1(-n, a+1, z).$$

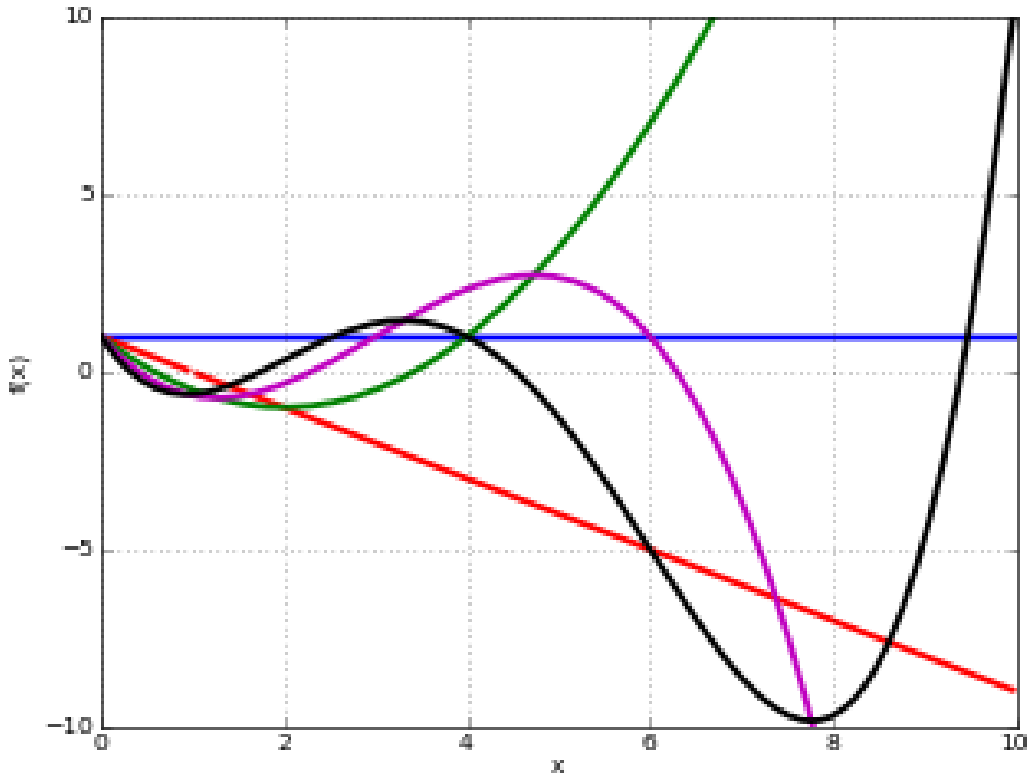
With $a = 0$ and n a nonnegative integer, this reduces to an ordinary Laguerre polynomial, the sequence of which begins $L_0(z) = 1, L_1(z) = 1 - z, L_2(z) = z^2 - 2z + 1, \ldots$.

The Laguerre polynomials are orthogonal with respect to the weight $z^a e^{-z}$ on $[0, \infty)$.

Plots

```
# Hermite polynomials L_n(x) on the real line for n=0,1,2,3,4
f0 = lambda x: laguerre(0,0,x)
f1 = lambda x: laguerre(1,0,x)
```

```
    f2 = lambda x: laguerre(2,0,x)
    f3 = lambda x: laguerre(3,0,x)
    f4 = lambda x: laguerre(4,0,x)
    plot([f0,f1,f2,f3,f4],[0,10],[-10,10])
```



Examples

Evaluation for arbitrary arguments:

```
    >>> from mpmath import *
    >>> mp.dps = 25; mp.pretty = True
    >>> laguerre(5, 0, 0.25)
    0.03726399739583333333333333
    >>> laguerre(1+j, 0.5, 2+3j)
    (4.474921610704496808379097 - 11.020580503720689580692 41j)
    >>> laguerre(2, 0, 10000)
    49980001.0
    >>> laguerre(2.5, 0, 10000)
    -9.327764910194842158583189e+4328
```

The first few Laguerre polynomials, normalized to have integer coefficients:

```
    >>> for n in range(7):
    ...     chop(taylor(lambda z: fac(n)*laguerre(n, 0, z), 0, n))
    ...
    [1.0]
    [1.0, -1.0]
    [2.0, -4.0, 1.0]
    [6.0, -18.0, 9.0, -1.0]
```

```
[24.0, -96.0, 72.0, -16.0, 1.0]
[120.0, -600.0, 600.0, -200.0, 25.0, -1.0]
[720.0, -4320.0, 5400.0, -2400.0, 450.0, -36.0, 1.0]
```

Verifying orthogonality:

```
>>> Lm = lambda t: laguerre(m,a,t)
>>> Ln = lambda t: laguerre(n,a,t)
>>> a, n, m = 2.5, 2, 3
>>> chop(quad(lambda t: exp(-t)*t**a*Lm(t)*Ln(t), [0,inf]))
0.0
```

Spherical harmonics

spherharm()

mpmath.**spherharm**(*l*, *m*, *theta*, *phi*)

Evaluates the spherical harmonic $Y_l^m(\theta, \phi)$,

$$Y_l^m(\theta,\phi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta) e^{im\phi}$$

where P_l^m is an associated Legendre function (see *legenp()*).

Here $\theta \in [0, \pi]$ denotes the polar coordinate (ranging from the north pole to the south pole) and $\phi \in [0, 2\pi]$ denotes the azimuthal coordinate on a sphere. Care should be used since many different conventions for spherical coordinate variables are used.

Usually spherical harmonics are considered for $l \in \mathbb{N}$, $m \in \mathbb{Z}$, $|m| \le l$. More generally, l, m, θ, ϕ are permitted to be complex numbers.

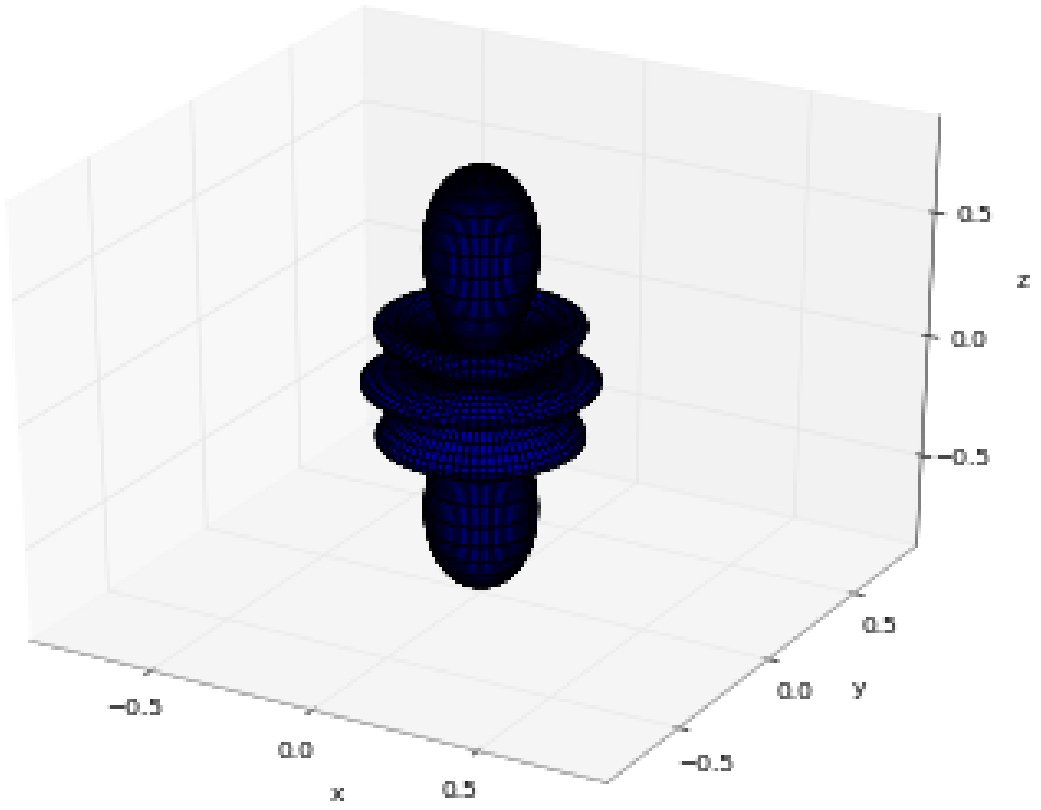
Note: *spherharm()* returns a complex number, even the value is purely real.

Plots

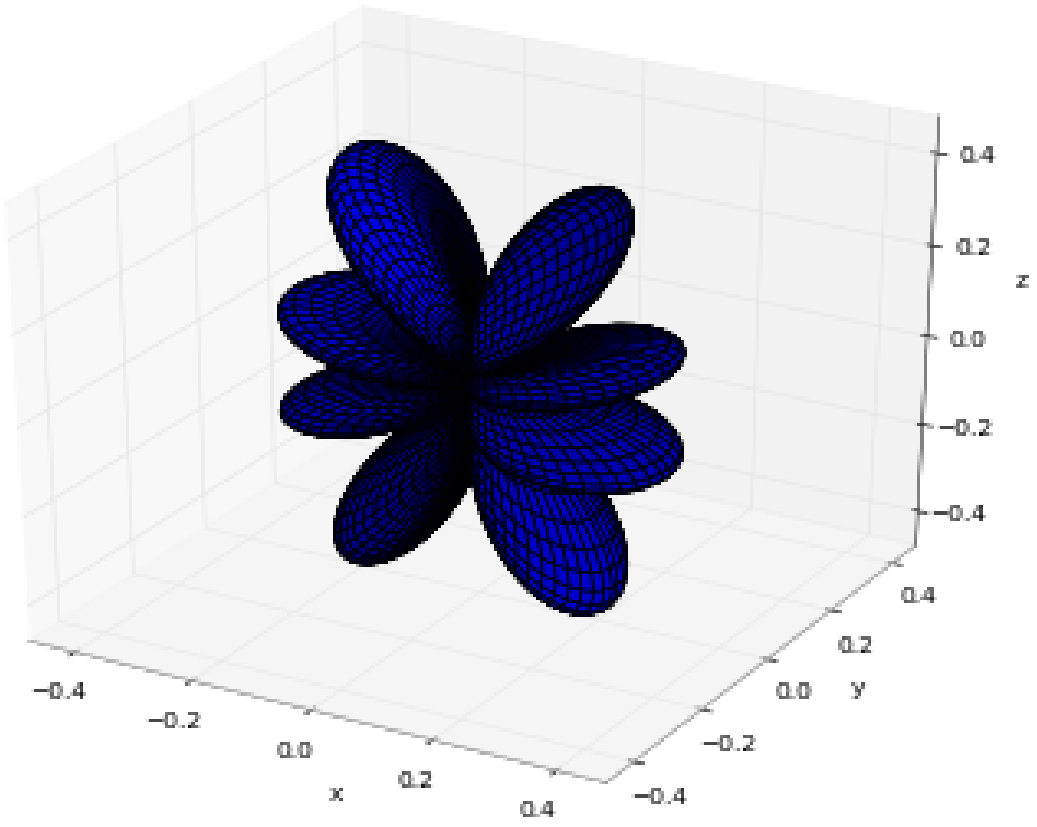
```
# Real part of spherical harmonic Y_(4,0)(theta,phi)
def Y(l,m):
    def g(theta,phi):
        R = abs(fp.re(fp.spherharm(l,m,theta,phi)))
        x = R*fp.cos(phi)*fp.sin(theta)
        y = R*fp.sin(phi)*fp.sin(theta)
        z = R*fp.cos(theta)
        return [x,y,z]
    return g

fp.splot(Y(4,0), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,0), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,1), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,2), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,3), [0,fp.pi], [0,2*fp.pi], points=300)
```

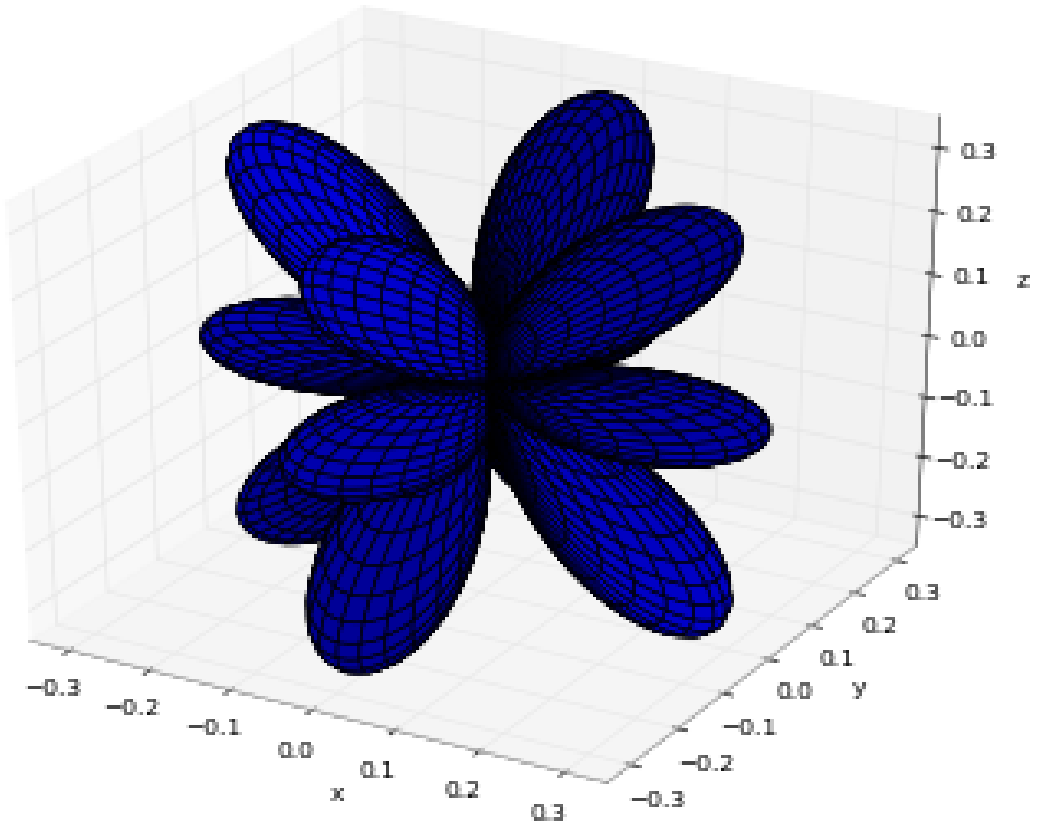
$Y_{4,0}$:



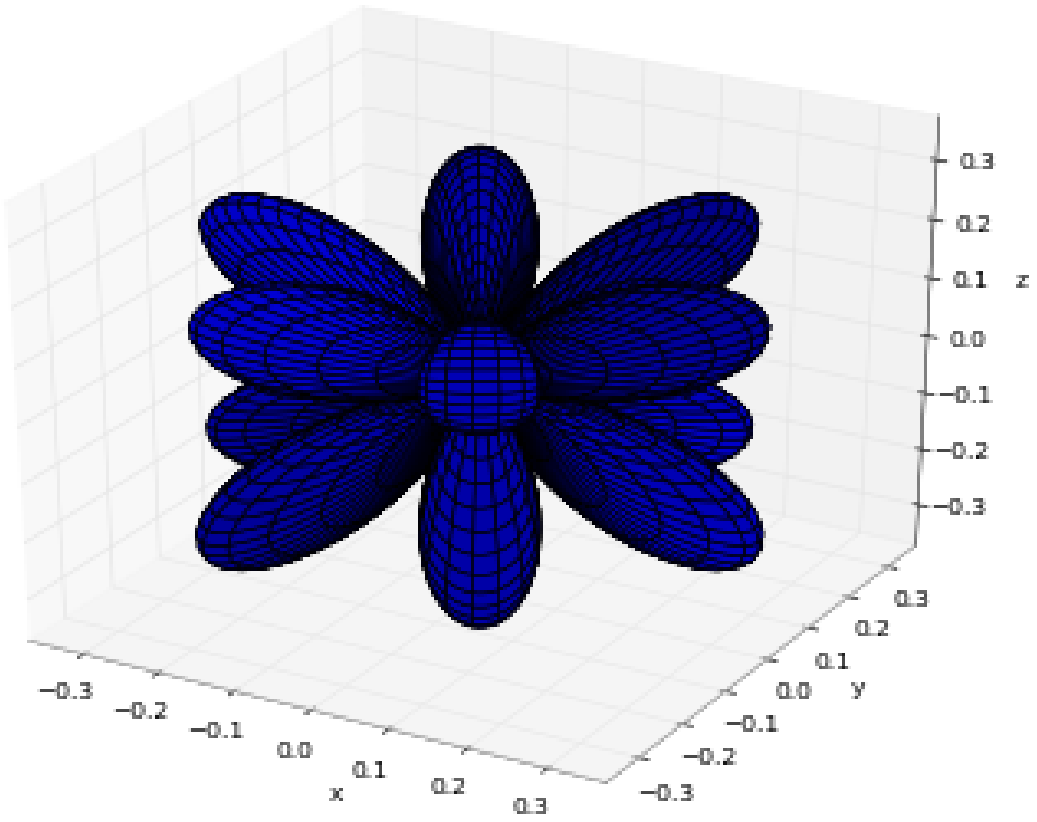
$Y_{4,1}$:



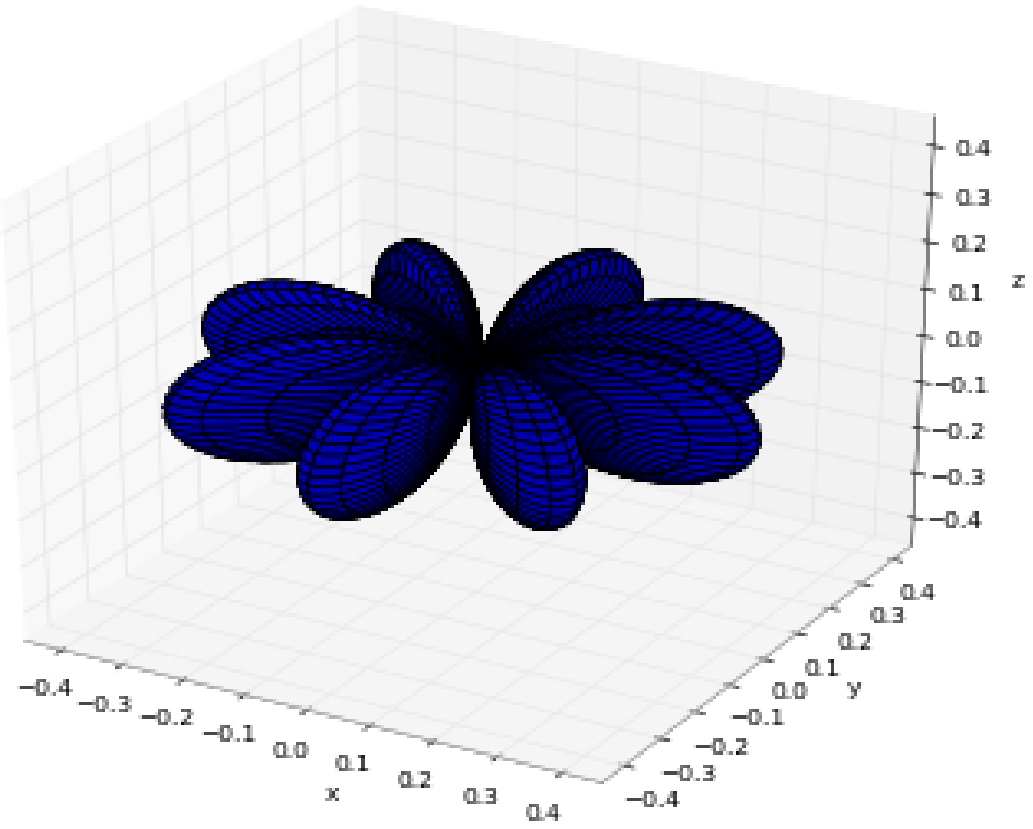
$Y_{4,2}$:



$Y_{4,3}$:



$Y_{4,4}$:



Examples

Some low-order spherical harmonics with reference values:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> theta = pi/4
>>> phi = pi/3
>>> spherharm(0,0,theta,phi); 0.5*sqrt(1/pi)*expj(0)
(0.2820947917738781434740397 + 0.0j)
(0.2820947917738781434740397 + 0.0j)
>>> spherharm(1,-1,theta,phi); 0.5*sqrt(3/(2*pi))*expj(-phi)*sin(theta)
(0.1221506279757299803965962 - 0.2115710938304086076055298j)
(0.1221506279757299803965962 - 0.2115710938304086076055298j)
>>> spherharm(1,0,theta,phi); 0.5*sqrt(3/pi)*cos(theta)*expj(0)
(0.3454941494713354792652446 + 0.0j)
(0.3454941494713354792652446 + 0.0j)
>>> spherharm(1,1,theta,phi); -0.5*sqrt(3/(2*pi))*expj(phi)*sin(theta)
(-0.1221506279757299803965962 - 0.2115710938304086076055298j)
(-0.1221506279757299803965962 - 0.2115710938304086076055298j)
```

With the normalization convention used, the spherical harmonics are orthonormal on the unit sphere:

```
>>> sphere = [0,pi], [0,2*pi]
>>> dS = lambda t,p: fp.sin(t)   # differential element
>>> Y1 = lambda t,p: fp.spherharm(l1,m1,t,p)
>>> Y2 = lambda t,p: fp.conj(fp.spherharm(l2,m2,t,p))
>>> l1 = l2 = 3; m1 = m2 = 2
```

```
>>> print(fp.quad(lambda t,p: Y1(t,p)*Y2(t,p)*dS(t,p), *sphere))
(1+0j)
>>> m2 = 1     # m1 != m2
>>> print(fp.chop(fp.quad(lambda t,p: Y1(t,p)*Y2(t,p)*dS(t,p), *sphere)))
0.0
```

Evaluation is accurate for large orders:

```
>>> spherharm(1000,750,0.5,0.25)
(3.7764457853042528790265 85e-102 - 5.8244127877183479449348 4e-102j)
```

Evaluation works with complex parameter values:

```
>>> spherharm(1+j, 2j, 2+3j, -0.5j)
(64.44922331113759992154992 + 1981.6939198414080896817 43j)
```

3.1.9 Hypergeometric functions

The functions listed in Exponential integrals and error functions, Bessel functions and related functions and Orthogonal polynomials, and many other functions as well, are merely particular instances of the generalized hypergeometric function ${}_pF_q$. The functions listed in the following section enable efficient direct evaluation of the underlying hypergeometric series, as well as linear combinations, limits with respect to parameters, and analytic continuations thereof. Extensions to twodimensional series are also provided. See also the basic or q-analog of the hypergeometric series in q-functions.

For convenience, most of the hypergeometric series of low order are provided as standalone functions. They can equivalently be evaluated using *hyper()*. As will be demonstrated in the respective docstrings, all the `hyp#f#` functions implement analytic continuations and/or asymptotic expansions with respect to the argument z, thereby permitting evaluation for z anywhere in the complex plane. Functions of higher degree can be computed via *hyper()*, but generally only in rapidly convergent instances.

Most hypergeometric and hypergeometric-derived functions accept optional keyword arguments to specify options for `hypercomb()` or `hyper()`. Some useful options are *maxprec*, *maxterms*, *zeroprec*, *accurate_small*, *hmag*, *force_series*, *asymp_tol* and *eliminate*. These options give control over what to do in case of slow convergence, extreme loss of accuracy or evaluation at zeros (these two cases cannot generally be distinguished from each other automatically), and singular parameter combinations.

Common hypergeometric series

hyp0f1()

mpmath.**hyp0f1**(*a*, *z*)

Gives the hypergeometric function ${}_0F_1$, sometimes known as the confluent limit function, defined as

$$ {}_0F_1(a,z) = \sum_{k=0}^{\infty} \frac{1}{(a)_k} \frac{z^k}{k!}. $$

This function satisfies the differential equation $zf''(z) + af'(z) = f(z)$, and is related to the Bessel function of the first kind (see *besselj()*).

`hyp0f1(a,z)` is equivalent to `hyper([],[a],z)`; see documentation for *hyper()* for more information.

Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp0f1(2, 0.25)
1.130318207984970054415392
>>> hyp0f1((1,2), 1234567)
6.27287187546220705604627e+964
>>> hyp0f1(3+4j, 1000000j)
(3.905169561300910030267132e+606 + 3.807708544441684513934213e+606j)
```

Evaluation is supported for arbitrarily large values of z, using asymptotic expansions:

```
>>> hyp0f1(1, 10**50)
2.131705322874965310390701e+8685889638065036553022565
>>> hyp0f1(1, -10**50)
1.115945364792025420300208e-13
```

Verifying the differential equation:

```
>>> a = 2.5
>>> f = lambda z: hyp0f1(a,z)
>>> for z in [0, 10, 3+4j]:
...     chop(z*diff(f,z,2) + a*diff(f,z) - f(z))
...
0.0
0.0
0.0
```

hyp1f1()

mpmath.**hyp1f1**(a, b, z)

Gives the confluent hypergeometric function of the first kind,

$$ {}_1F_1(a,b,z) = \sum_{k=0}^{\infty} \frac{(a)_k}{(b)_k} \frac{z^k}{k!}, $$

also known as Kummer's function and sometimes denoted by $M(a,b,z)$. This function gives one solution to the confluent (Kummer's) differential equation

$$ zf''(z) + (b-z)f'(z) - af(z) = 0. $$

A second solution is given by the U function; see *hyperu()*. Solutions are also given in an alternate form by the Whittaker functions (*whitm()*, *whitw()*).

`hyp1f1(a,b,z)` is equivalent to `hyper([a],[b],z)`; see documentation for *hyper()* for more information.

Examples

Evaluation for real and complex values of the argument z, with fixed parameters $a = 2, b = -1/3$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp1f1(2, (-1,3), 3.25)
-2815.956856924817275640248
>>> hyp1f1(2, (-1,3), -3.25)
-1.145036502407444445553107
>>> hyp1f1(2, (-1,3), 1000)
-8.021799872770764149793693e+441
```

```
>>> hyp1f1(2, (-1,3), -1000)
0.00000313198763300681359453533 1
>>> hyp1f1(2, (-1,3), 100+100j)
(-3.189190365227034385898282e+48 - 1.106169926814270418999315e+49j)
```

Parameters may be complex:

```
>>> hyp1f1(2+3j, -1+j, 10j)
(261.8977905181045142673351 + 160.8930312845682213562172j)
```

Arbitrarily large values of z are supported:

```
>>> hyp1f1(3, 4, 10**20)
3.890569218254486878220752e+43429448190325182745
>>> hyp1f1(3, 4, -10**20)
6.0e-60
>>> hyp1f1(3, 4, 10**20*j)
(-1.935753855797342532571597e-20 - 2.291911213325184901239155e-20j)
```

Verifying the differential equation:

```
>>> a, b = 1.5, 2
>>> f = lambda z: hyp1f1(a,b,z)
>>> for z in [0, -10, 3, 3+4j]:
...     chop(z*diff(f,z,2) + (b-z)*diff(f,z) - a*f(z))
...
0.0
0.0
0.0
0.0
```

An integral representation:

```
>>> a, b = 1.5, 3
>>> z = 1.5
>>> hyp1f1(a,b,z)
2.26938146091995277858744 1
>>> g = lambda t: exp(z*t)*t**(a-1)*(1-t)**(b-a-1)
>>> gammaprod([b],[a,b-a])*quad(g, [0,1])
2.26938146091995277858744 1
```

hyp1f2()

mpmath.**hyp1f2**($a1, b1, b2, z$)

Gives the hypergeometric function ${}_1F_2(a_1, a_2; b_1, b_2; z)$. The call `hyp1f2(a1,b1,b2,z)` is equivalent to `hyper([a1],[b1,b2],z)`.

Evaluation works for complex and arbitrarily large arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a, b, c = 1.5, (-1,3), 2.25
>>> hyp1f2(a, b, c, 10**20)
-1.159388148811981535941434e+8685889639
>>> hyp1f2(a, b, c, -10**20)
-12.6026260789265594579507
>>> hyp1f2(a, b, c, 10**20*j)
(4.237220401382240876065501e+6141851464 - 2.950930337531768015892987e+6141851464j)
```

```
>>> hyp1f2(2+3j, -2j, 0.5j, 10-20j)
(135881.9905586966432662004 - 86681.95885418079535738828j)
```

hyp2f0()

mpmath.**hyp2f0**(a, b, z)

Gives the hypergeometric function ${}_2F_0$, defined formally by the series

$$ {}_2F_0(a,b;;z) = \sum_{n=0}^{\infty} (a)_n (b)_n \frac{z^n}{n!}. $$

This series usually does not converge. For small enough z, it can be viewed as an asymptotic series that may be summed directly with an appropriate truncation. When this is not the case, *hyp2f0()* gives a regularized sum, or equivalently, it uses a representation in terms of the hypergeometric U function [1]. The series also converges when either a or b is a nonpositive integer, as it then terminates into a polynomial after $-a$ or $-b$ terms.

Examples

Evaluation is supported for arbitrary complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp2f0((2,3), 1.25, -100)
0.07095851870980052763312791
>>> hyp2f0((2,3), 1.25, 100)
(-0.03254379032170590665041131 + 0.07269254613282301012735797j)
>>> hyp2f0(-0.75, 1-j, 4j)
(-0.3579987031082732264862155 - 3.052951783922142735255881j)
```

Even with real arguments, the regularized value of 2F0 is often complex-valued, but the imaginary part decreases exponentially as $z \to 0$. In the following example, the first call uses complex evaluation while the second has a small enough z to evaluate using the direct series and thus the returned value is strictly real (this should be taken to indicate that the imaginary part is less than `eps`):

```
>>> mp.dps = 15
>>> hyp2f0(1.5, 0.5, 0.05)
(1.04166637647907 + 8.34584913683906e-8j)
>>> hyp2f0(1.5, 0.5, 0.0005)
1.00037535207621
```

The imaginary part can be retrieved by increasing the working precision:

```
>>> mp.dps = 80
>>> nprint(hyp2f0(1.5, 0.5, 0.009).imag)
1.23828e-46
```

In the polynomial case (the series terminating), 2F0 can evaluate exactly:

```
>>> mp.dps = 15
>>> hyp2f0(-6,-6,2)
291793.0
>>> identify(hyp2f0(-2,1,0.25))
'(5/8)'
```

The coefficients of the polynomials can be recovered using Taylor expansion:

```
>>> nprint(taylor(lambda x: hyp2f0(-3,0.5,x), 0, 10))
[1.0, -1.5, 2.25, -1.875, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint(taylor(lambda x: hyp2f0(-4,0.5,x), 0, 10))
[1.0, -2.0, 4.5, -7.5, 6.5625, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

[1] http://people.math.sfu.ca/~cbm/aands/page_504.htm

hyp2f1()

mpmath.**hyp2f1**(a, b, c, z)

Gives the Gauss hypergeometric function ${}_2F_1$ (often simply referred to as *the* hypergeometric function), defined for $|z| < 1$ as

$$ {}_2F_1(a,b,c,z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!}. $$

and for $|z| \geq 1$ by analytic continuation, with a branch cut on $(1, \infty)$ when necessary.

Special cases of this function include many of the orthogonal polynomials as well as the incomplete beta function and other functions. Properties of the Gauss hypergeometric function are documented comprehensively in many references, for example Abramowitz & Stegun, section 15.

The implementation supports the analytic continuation as well as evaluation close to the unit circle where $|z| \approx 1$. The syntax `hyp2f1(a,b,c,z)` is equivalent to `hyper([a,b],[c],z)`.

Examples

Evaluation with z inside, outside and on the unit circle, for fixed parameters:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp2f1(2, (1,2), 4, 0.75)
1.303703703703703703703704
>>> hyp2f1(2, (1,2), 4, -1.75)
0.7431290566046919177853916
>>> hyp2f1(2, (1,2), 4, 1.75)
(1.418075801749271137026239 - 1.114976146679907015775102j)
>>> hyp2f1(2, (1,2), 4, 1)
1.6
>>> hyp2f1(2, (1,2), 4, -1)
0.8235498012182875315037882
>>> hyp2f1(2, (1,2), 4, j)
(0.9144026291433065674259078 + 0.2050415770437884900574923j)
>>> hyp2f1(2, (1,2), 4, 2+j)
(0.9274013540258103029011549 + 0.7455257875808100868984496j)
>>> hyp2f1(2, (1,2), 4, 0.25j)
(0.9931169055799728251931672 + 0.06154836525312066938147793j)
```

Evaluation with complex parameter values:

```
>>> hyp2f1(1+j, 0.75, 10j, 1+5j)
(0.8834833319713479923389638 + 0.7053886880648105068343509j)
```

Evaluation with $z = 1$:

```
>>> hyp2f1(-2.5, 3.5, 1.5, 1)
0.0
>>> hyp2f1(-2.5, 3, 4, 1)
```

```
0.069264069264069264069264069264
>>> hyp2f1(2, 3, 4, 1)
+inf
```

Evaluation for huge arguments:

```
>>> hyp2f1((-1,3), 1.75, 4, '1e100')
(7.883714220959876246415651e+32 + 1.365499358305579597618785e+33j)
>>> hyp2f1((-1,3), 1.75, 4, '1e1000000')
(7.883714220959876246415651e+333332 + 1.365499358305579597618785e+333333j)
>>> hyp2f1((-1,3), 1.75, 4, '1e1000000j')
(1.365499358305579597618785e+333333 - 7.883714220959876246415651e+333332j)
```

An integral representation:

```
>>> a,b,c,z = -0.5, 1, 2.5, 0.25
>>> g = lambda t: t**(b-1) * (1-t)**(c-b-1) * (1-t*z)**(-a)
>>> gammaprod([c],[b,c-b]) * quad(g, [0,1])
0.94804588143628244788526 18
>>> hyp2f1(a,b,c,z)
0.94804588143628244788526 18
```

Verifying the hypergeometric differential equation:

```
>>> f = lambda z: hyp2f1(a,b,c,z)
>>> chop(z*(1-z)*diff(f,z,2) + (c-(a+b+1)*z)*diff(f,z) - a*b*f(z))
0.0
```

hyp2f2()

mpmath.**hyp2f2**(*a1*, *a2*, *b1*, *b2*, *z*)

Gives the hypergeometric function ${}_2F_2(a_1, a_2; b_1, b_2; z)$. The call `hyp2f2(a1,a2,b1,b2,z)` is equivalent to `hyper([a1,a2],[b1,b2],z)`.

Evaluation works for complex and arbitrarily large arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a, b, c, d = 1.5, (-1,3), 2.25, 4
>>> hyp2f2(a, b, c, d, 10**20)
-5.2757582290079022998238 21e+43429448190325182663
>>> hyp2f2(a, b, c, d, -10**20)
2561445.079983207701073448
>>> hyp2f2(a, b, c, d, 10**20*j)
(2218276.509664121194836667 - 1280722.539991603850462856j)
>>> hyp2f2(2+3j, -2j, 0.5j, 4j, 10-20j)
(80500.683214056669573427 88 - 20346.827529828135409935 02j)
```

hyp2f3()

mpmath.**hyp2f3**(*a1*, *a2*, *b1*, *b2*, *b3*, *z*)

Gives the hypergeometric function ${}_2F_3(a_1, a_2; b_1, b_2, b_3; z)$. The call `hyp2f3(a1,a2,b1,b2,b3,z)` is equivalent to `hyper([a1,a2],[b1,b2,b3],z)`.

Evaluation works for arbitrarily large arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a1,a2,b1,b2,b3 = 1.5, (-1,3), 2.25, 4, (1,5)
>>> hyp2f3(a1,a2,b1,b2,b3,10**20)
-4.169178177065714963568963e+8685889590
>>> hyp2f3(a1,a2,b1,b2,b3,-10**20)
7064472.587757755088178629
>>> hyp2f3(a1,a2,b1,b2,b3,10**20*j)
(-5.163368465314934589818543e+6141851415 + 1.783578125755972803440364e+6141851416j)
>>> hyp2f3(2+3j, -2j, 0.5j, 4j, -1-j, 10-20j)
(-2280.938956687033150740228 + 13620.973366095736591996 32j)
>>> hyp2f3(2+3j, -2j, 0.5j, 4j, -1-j, 10000000-20000000j)
(4.849835186175096516193e+3504 - 3.365981529122220091353633e+3504j)
```

hyp3f2()

mpmath.**hyp3f2**(*a1*, *a2*, *a3*, *b1*, *b2*, *z*)

Gives the generalized hypergeometric function ${}_3F_2$, defined for $|z| < 1$ as

$$ {}_3F_2(a_1,a_2,a_3,b_1,b_2,z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k (a_3)_k}{(b_1)_k (b_2)_k} \frac{z^k}{k!}. $$

and for $|z| \geq 1$ by analytic continuation. The analytic structure of this function is similar to that of ${}_2F_1$, generally with a singularity at $z = 1$ and a branch cut on $(1, \infty)$.

Evaluation is supported inside, on, and outside the circle of convergence $|z| = 1$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp3f2(1,2,3,4,5,0.25)
1.0835331233809342415487 07
>>> hyp3f2(1,2+2j,3,4,5,-10+10j)
(0.15746510660060046329143 61 - 0.03194209021885226400892963j)
>>> hyp3f2(1,2,3,4,5,-10)
0.30711411692087726032664 89
>>> hyp3f2(1,2,3,4,5,10)
(-0.48570453205239470505814 23 - 0.59883114404548884368880 28j)
>>> hyp3f2(0.25,1,1,2,1.5,1)
1.15737099509677204756763 1
>>> (8-pi-2*ln2)/3
1.15737099509677204756763 1
>>> hyp3f2(1+j,0.5j,2,1,-2j,-1)
(1.7451849061502948647595 9 + 0.14547015250566822976140 29j)
>>> hyp3f2(1+j,0.5j,2,1,-2j,sqrt(j))
(0.98298164818342775111380 55 - 0.40590400202769370850811 27j)
>>> hyp3f2(-3,2,1,-5,4,1)
1.41
>>> hyp3f2(-3,2,1,-5,4,2)
2.12
```

Evaluation very close to the unit circle:

```
>>> hyp3f2(1,2,3,4,5,'1.0001')
(1.5648777967432827668722 79 - 3.7682151878743818603197 3e-11j)
>>> hyp3f2(1,2,3,4,5,'1+0.0001j')
(1.5647471530616715732128 31 + 0.0001305757570366084557648482j)
>>> hyp3f2(1,2,3,4,5,'0.9999')
```

```
1.56461664488168613498366 4
>>> hyp3f2(1,2,3,4,5,'-0.9999')
0.78238962534616780601962 07
```

Note: Evaluation for $|z-1|$ small can currently be inaccurate or slow for some parameter combinations.

For various parameter combinations, ${}_3F_2$ admits representation in terms of hypergeometric functions of lower degree, or in terms of simpler functions:

```
>>> for a, b, z in [(1,2,-1), (2,0.5,1)]:
...     hyp2f1(a,b,a+b+0.5,z)**2
...     hyp3f2(2*a,a+b,2*b,a+b+0.5,2*a+2*b,z)
...
0.42461044619664390060863 08
0.42461044619664390060863 08
7.11111111111111111111111
7.11111111111111111111111

>>> z = 2+3j
>>> hyp3f2(0.5,1,1.5,2,2,z)
(0.76214409392433424197291 44 + 0.42491177350580376499157 23j)
>>> 4*(pi-2*ellipe(z))/(pi*z)
(0.76214409392433424197291 44 + 0.42491177350580376499157 23j)
```

Generalized hypergeometric functions

hyper()

mpmath.**hyper**(*a_s*, *b_s*, *z*)

Evaluates the generalized hypergeometric function

$$ {}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n \ldots (a_p)_n}{(b_1)_n (b_2)_n \ldots (b_q)_n} \frac{z^n}{n!} $$

where $(x)_n$ denotes the rising factorial (see *rf()*).

The parameters lists a_s and b_s may contain integers, real numbers, complex numbers, as well as exact fractions given in the form of tuples (p, q). *hyper()* is optimized to handle integers and fractions more efficiently than arbitrary floating-point parameters (since rational parameters are by far the most common).

Examples

Verifying that *hyper()* gives the sum in the definition, by comparison with *nsum()*:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a,b,c,d = 2,3,4,5
>>> x = 0.25
>>> hyper([a,b],[c,d],x)
1.07890394116493487608623 7
>>> fn = lambda n: rf(a,n)*rf(b,n)/rf(c,n)/rf(d,n)*x**n/fac(n)
>>> nsum(fn, [0, inf])
1.07890394116493487608623 7
```

The parameters can be any combination of integers, fractions, floats and complex numbers:

```
>>> a, b, c, d, e = 1, (-1,2), pi, 3+4j, (2,3)
>>> x = 0.2j
>>> hyper([a,b],[c,d,e],x)
(0.992357161643402481083188 7 - 0.00575384873388379742993122j)
>>> b, e = -0.5, mpf(2)/3
>>> fn = lambda n: rf(a,n)*rf(b,n)/rf(c,n)/rf(d,n)/rf(e,n)*x**n/fac(n)
>>> nsum(fn, [0, inf])
(0.992357161643402481083188 7 - 0.00575384873388379742993122j)
```

The ${}_0F_0$ and ${}_1F_0$ series are just elementary functions:

```
>>> a, z = sqrt(2), +pi
>>> hyper([],[],z)
23.14069263277926900572909
>>> exp(z)
23.14069263277926900572909
>>> hyper([a],[],z)
(-0.09069132879922920160334114 + 0.32832243239461620835796 56j)
>>> (1-z)**(-a)
(-0.09069132879922920160334114 + 0.32832243239461620835796 56j)
```

If any a_k coefficient is a nonpositive integer, the series terminates into a finite polynomial:

```
>>> hyper([1,1,1,-3],[2,5],1)
0.790476190476190476190476 2
>>> identify(_)
'(83/105)'
```

If any b_k is a nonpositive integer, the function is undefined (unless the series terminates before the division by zero occurs):

```
>>> hyper([1,1,1,-3],[-2,5],1)
Traceback (most recent call last):
  ...
ZeroDivisionError: pole in hypergeometric series
>>> hyper([1,1,1,-1],[-2,5],1)
1.1
```

Except for polynomial cases, the radius of convergence R of the hypergeometric series is either $R = \infty$ (if $p \leq q$), $R = 1$ (if $p = q+1$), or $R = 0$ (if $p > q+1$).

The analytic continuations of the functions with $p = q+1$, i.e. ${}_2F_1$, ${}_3F_2$, ${}_4F_3$, etc, are all implemented and therefore these functions can be evaluated for $|z| \geq 1$. The shortcuts *hyp2f1()*, *hyp3f2()* are available to handle the most common cases (see their documentation), but functions of higher degree are also supported via *hyper()*:

```
>>> hyper([1,2,3,4], [5,6,7], 1)   # 4F3 at finite-valued branch point
1.141783505526870731311423
>>> hyper([4,5,6,7], [1,2,3], 1)   # 4F3 at pole
+inf
>>> hyper([1,2,3,4,5], [6,7,8,9], 10)    # 5F4
(1.543998916527972259717257 - 0.58763099295804080288163 65j)
>>> hyper([1,2,3,4,5,6], [7,8,9,10,11], 1j)   # 6F5
(0.999656582185357906350246 6 + 0.012972107590563060444 5669j)
```

Near $z = 1$ with noninteger parameters:

```
>>> hyper(['1/3',1,'3/2',2], ['1/5','11/6','41/8'], 1)
2.219433352235586121250027
>>> hyper(['1/3',1,'3/2',2], ['1/5','11/6','5/4'], 1)
```

```
+inf
>>> eps1 = extradps(6)(lambda: 1 - mpf('1e-6'))()
>>> hyper(['1/3',1,'3/2',2], ['1/5','11/6','5/4'], eps1)
2923978034.412973409330956
```

Please note that, as currently implemented, evaluation of ${}_pF_{p-1}$ with $p \geq 3$ may be slow or inaccurate when $|z-1|$ is small, for some parameter values.

When $p > q+1$, `hyper` computes the (iterated) Borel sum of the divergent series. For ${}_2F_0$ the Borel sum has an analytic solution and can be computed efficiently (see *hyp2f0()*). For higher degrees, the functions is evaluated first by attempting to sum it directly as an asymptotic series (this only works for tiny $|z|$), and then by evaluating the Borel regularized sum using numerical integration. Except for special parameter combinations, this can be extremely slow.

```
>>> hyper([1,1], [], 0.5)           # regularization of 2F0
(1.3409654195801465620864 48 + 0.8503366631752726568782447j)
>>> hyper([1,1,1,1], [1], 0.5)      # regularization of 4F1
(1.1082872136894751458306 99 + 0.5327107430640678181200491j)
```

With the following magnitude of argument, the asymptotic series for ${}_3F_1$ gives only a few digits. Using Borel summation, `hyper` can produce a value with full accuracy:

```
>>> mp.dps = 15
>>> hyper([2,0.5,4], [5.25], '0.08', force_series=True)
Traceback (most recent call last):
  ...
NoConvergence: Hypergeometric series converges too slowly. Try increasing maxterms.
>>> hyper([2,0.5,4], [5.25], '0.08', asymp_tol=1e-4)
1.0725535790737
>>> hyper([2,0.5,4], [5.25], '0.08')
(1.07269542893559 + 5.54668863216891e-5j)
>>> hyper([2,0.5,4], [5.25], '-0.08', asymp_tol=1e-4)
0.946344925484879
>>> hyper([2,0.5,4], [5.25], '-0.08')
0.946312503737771
>>> mp.dps = 25
>>> hyper([2,0.5,4], [5.25], '-0.08')
0.9463125037377662296700858
```

Note that with the positive z value, there is a complex part in the correct result, which falls below the tolerance of the asymptotic series.

hypercomb()

mpmath.**hypercomb**(*ctx*, *function*, *params=[]*, *discard_known_zeros=True*, ***kwargs*)

Computes a weighted combination of hypergeometric functions

$$\sum_{r=1}^{N} \left[\prod_{k=1}^{l_r} {w_{r,k}}^{c_{r,k}} \frac{\prod_{k=1}^{m_r} \Gamma(\alpha_{r,k})}{\prod_{k=1}^{n_r} \Gamma(\beta_{r,k})} \, {}_{r}F_{q_r}(a_{r,1}, \ldots, a_{r,p}; b_{r,1}, \ldots, b_{r,q}; z_r) \right].$$

Typically the parameters are linear combinations of a small set of base parameters; *hypercomb()* permits computing a correct value in the case that some of the α, β, b turn out to be nonpositive integers, or if division by zero occurs for some w^c, assuming that there are opposing singularities that cancel out. The limit is computed by evaluating the function with the base parameters perturbed, at a higher working precision.

The first argument should be a function that takes the perturbable base parameters `params` as input and returns N tuples `(w, c, alpha, beta, a, b, z)`, where the coefficients `w`, `c`, gamma factors `alpha`, `beta`, and hypergeometric coefficients `a`, `b` each should be lists of numbers, and `z` should be a single number.

Examples

The following evaluates

$$(a-1)\frac{\Gamma(a-3)}{\Gamma(a-4)}\,{}_1F_1(a,a-1,z) = e^z(a-4)(a+z-1)$$

with $a = 1, z = 3$. There is a zero factor, two gamma function poles, and the 1F1 function is singular; all singularities cancel out to give a finite value:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> hypercomb(lambda a: [([a-1],[1],[a-3],[a-4],[a],[a-1],3)], [1])
-180.769832308689
>>> -9*exp(3)
-180.769832308689
```

Meijer G-function

meijerg()

mpmath.**meijerg**(*a_s*, *b_s*, *z*, *r=1*, ***kwargs*)

Evaluates the Meijer G-function, defined as

$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_n;a_{n+1}\ldots a_p\\ b_1,\ldots,b_m;b_{m+1}\ldots b_q\end{matrix}\;\middle|\;z;r\right) = \frac{1}{2\pi i}\int_L \frac{\prod_{j=1}^m\Gamma(b_j+s)\prod_{j=1}^n\Gamma(1-a_j-s)}{\prod_{j=n+1}^p\Gamma(a_j+s)\prod_{j=m+1}^q\Gamma(1-b_j-s)}z^{-s/r}ds$$

for an appropriate choice of the contour L (see references).

There are p elements a_j. The argument *a_s* should be a pair of lists, the first containing the n elements $a_1,\ldots,a_n$ and the second containing the $p-n$ elements $a_{n+1},\ldots a_p$.

There are q elements b_j. The argument *b_s* should be a pair of lists, the first containing the m elements $b_1,\ldots,b_m$ and the second containing the $q-m$ elements $b_{m+1},\ldots b_q$.

The implicit tuple (m,n,p,q) constitutes the order or degree of the Meijer G-function, and is determined by the lengths of the coefficient vectors. Confusingly, the indices in this tuple appear in a different order from the coefficients, but this notation is standard. The many examples given below should hopefully clear up any potential confusion.

Algorithm

The Meijer G-function is evaluated as a combination of hypergeometric series. There are two versions of the function, which can be selected with the optional *series* argument.

series=1 uses a sum of m ${}_pF_{q-1}$ functions of z

series=2 uses a sum of n ${}_qF_{p-1}$ functions of $1/z$

The default series is chosen based on the degree and $|z|$ in order to be consistent with Mathematica's. This definition of the Meijer G-function has a discontinuity at $|z| = 1$ for some orders, which can be avoided by explicitly specifying a series.

Keyword arguments are forwarded to *hypercomb()*.

Examples

Many standard functions are special cases of the Meijer G-function (possibly rescaled and/or with branch cut corrections). We define some test parameters:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a = mpf(0.75)
>>> b = mpf(1.5)
>>> z = mpf(2.25)
```

The exponential function: $e^z = G^{1,0}_{0,1}\left(\begin{matrix} - \\ 0 \end{matrix} \middle| -z\right)$

```
>>> meijerg([[],[]], [[0],[]], -z)
9.4877358363585257205503691
>>> exp(z)
9.4877358363585257205503691
```

The natural logarithm: $\log(1+z) = G^{1,2}_{2,2}\left(\begin{matrix} 1,1 \\ 1,0 \end{matrix} \middle| -z\right)$

```
>>> meijerg([[1,1],[]], [[1],[0]], z)
1.1786549963416461172190231
>>> log(1+z)
1.1786549963416461172190231
```

A rational function: $\frac{z}{z+1} = G^{1,2}_{2,2}\left(\begin{matrix} 1,1 \\ 1,1 \end{matrix} \middle| z\right)$

```
>>> meijerg([[1,1],[]], [[1],[1]], z)
0.69230769230769230769230769
>>> z/(z+1)
0.69230769230769230769230769
```

The sine and cosine functions:

$\frac{1}{\sqrt{\pi}}\sin(2\sqrt{z}) = G^{1,0}_{0,2}\left(\begin{matrix} - \\ \frac{1}{2}, 0 \end{matrix} \middle| z\right)$

$\frac{1}{\sqrt{\pi}}\cos(2\sqrt{z}) = G^{1,0}_{0,2}\left(\begin{matrix} - \\ 0, \frac{1}{2} \end{matrix} \middle| z\right)$

```
>>> meijerg([[],[]], [[0.5],[0]], (z/2)**2)
0.43898079292186766822964531
>>> sin(z)/sqrt(pi)
0.43898079292186766822964531
>>> meijerg([[],[]], [[0],[0.5]], (z/2)**2)
-0.35440901459962754233317623
>>> cos(z)/sqrt(pi)
-0.35440901459962754233317623
```

Bessel functions:

$J_a(2\sqrt{z}) = G^{1,0}_{0,2}\left(\begin{matrix} - \\ \frac{a}{2}, -\frac{a}{2} \end{matrix} \middle| z\right)$

$Y_a(2\sqrt{z}) = G^{2,0}_{1,3}\left(\begin{matrix} \frac{-a-1}{2} \\ \frac{a}{2}, -\frac{a}{2}, \frac{-a-1}{2} \end{matrix} \middle| z\right)$

$(-z)^{a/2} z^{-a/2} I_a(2\sqrt{z}) = G^{1,0}_{0,2}\left(\begin{matrix} - \\ \frac{a}{2}, -\frac{a}{2} \end{matrix} \middle| -z\right)$

$2K_a(2\sqrt{z}) = G^{2,0}_{0,2}\left(\begin{matrix} - \\ \frac{a}{2}, -\frac{a}{2} \end{matrix} \middle| z\right)$

As the example with the Bessel *I* function shows, a branch factor is required for some arguments when inverting the square root.

```
>>> meijerg([[],[]], [[a/2],[-a/2]], (z/2)**2)
0.50594257895971548585272 64
>>> besselj(a,z)
0.50594257895971548585272 64
>>> meijerg([[],[(-a-1)/2]], [[a/2,-a/2],[(-a-1)/2]], (z/2)**2)
0.18538689500665569414425 59
>>> bessely(a, z)
0.18538689500665569414425 59
>>> meijerg([[],[]], [[a/2],[-a/2]], -(z/2)**2)
(0.86859133224276538757174 76 + 2.09696497446019920055173 8j)
>>> (-z)**(a/2) / z**(a/2) * besseli(a, z)
(0.86859133224276538757174 76 + 2.09696497446019920055173 8j)
>>> 0.5*meijerg([[],[]], [[a/2,-a/2],[]], (z/2)**2)
0.09334163695597828403796071
>>> besselk(a,z)
0.09334163695597828403796071
```

Error functions:

$$\sqrt{\pi} z^{2(a-1)} \mathrm{erfc}(z) = G_{1,2}^{2,0}\left(\begin{matrix} a \\ a-1, a-\frac{1}{2} \end{matrix} \middle| \, z, \frac{1}{2} \right)$$

```
>>> meijerg([[],[a]], [[a-1,a-0.5],[]], z, 0.5)
0.00172839843123091957468712
>>> sqrt(pi) * z**(2*a-2) * erfc(z)
0.00172839843123091957468712
```

A Meijer G-function of higher degree, (1,1,2,3):

```
>>> meijerg([[a],[b]], [[a],[b,a-1]], z)
1.5598446744305021011561 7
>>> sin((b-a)*pi)/pi*(exp(z)-1)*z**(a-1)
1.5598446744305021011561 7
```

A Meijer G-function of still higher degree, (4,1,2,4), that can be expanded as a messy combination of exponential integrals:

```
>>> meijerg([[a],[2*b-a]], [[b,a,b-0.5,-1-a+2*b],[]], z)
0.33236671336585572718980 61
>>> chop(4**(a-b+1)*sqrt(pi)*gamma(2*b-2*a)*z**a*\
...     expint(2*b-2*a, -2*sqrt(-z))*expint(2*b-2*a, 2*sqrt(-z)))
0.33236671336585572718980 61
```

In the following case, different series give different values:

```
>>> chop(meijerg([[1],[0.25]],[[3],[0.5]],-2))
-0.06417628097442437076207337
>>> meijerg([[1],[0.25]],[[3],[0.5]],-2,series=1)
0.14286994261551175118730 47
>>> chop(meijerg([[1],[0.25]],[[3],[0.5]],-2,series=2))
-0.06417628097442437076207337
```

References

1. http://en.wikipedia.org/wiki/Meijer_G-function
2. http://mathworld.wolfram.com/MeijerG-Function.html
3. http://functions.wolfram.com/HypergeometricFunctions/MeijerG/
4. http://functions.wolfram.com/HypergeometricFunctions/MeijerG1/

Bilateral hypergeometric series

bihyper()

mpmath.**bihyper**(*a_s*, *b_s*, *z*, ***kwargs*)

Evaluates the bilateral hypergeometric series

$$ {}_AH_B(a_1,\ldots,a_k;b_1,\ldots,b_B;z) = \sum_{n=-\infty}^{\infty} \frac{(a_1)_n \ldots (a_A)_n}{(b_1)_n \ldots (b_B)_n} z^n $$

where, for direct convergence, $A = B$ and $|z| = 1$, although a regularized sum exists more generally by considering the bilateral series as a sum of two ordinary hypergeometric functions. In order for the series to make sense, none of the parameters may be integers.

Examples

The value of ${}_2H_2$ at $z = 1$ is given by Dougall's formula:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a,b,c,d = 0.5, 1.5, 2.25, 3.25
>>> bihyper([a,b],[c,d],1)
-14.491180262123457861488 47
>>> gammaprod([c,d,1-a,1-b,c+d-a-b-1],[c-a,d-a,c-b,d-b])
-14.491180262123457861488 47
```

The regularized function ${}_1H_0$ can be expressed as the sum of one ${}_2F_0$ function and one ${}_1F_1$ function:

```
>>> a = mpf(0.25)
>>> z = mpf(0.75)
>>> bihyper([a], [], z)
(0.24543933896572738413855 82 + 0.24543933896572738413855 82j)
>>> hyper([a,1],[],z) + (hyper([1],[1-a],-1/z)-1)
(0.24543933896572738413855 82 + 0.24543933896572738413855 82j)
>>> hyper([a,1],[],z) + hyper([1],[2-a],-1/z)/z/(a-1)
(0.24543933896572738413855 82 + 0.24543933896572738413855 82j)
```

References

1. *[Slater]* (chapter 6: "Bilateral Series", pp. 180-189)
2. *[Wikipedia]* http://en.wikipedia.org/wiki/Bilateral_hypergeometric_series

Hypergeometric functions of two variables

hyper2d()

mpmath.**hyper2d**(*a*, *b*, *x*, *y*, ***kwargs*)

Sums the generalized 2D hypergeometric series

$$ \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{P((a),m,n)}{Q((b),m,n)} \frac{x^m y^n}{m!n!} $$

where $(a) = (a_1,\ldots,a_r)$, $(b) = (b_1,\ldots,b_s)$ and where P and Q are products of rising factorials such as $(a_j)_n$ or $(a_j)_{m+n}$. P and Q are specified in the form of dicts, with the m and n dependence as keys and parameter lists as values. The supported rising factorials are given in the following table (note that only a few are supported in Q):

Key	Rising factorial	Q
'm'	$(a_j)_m$	Yes
'n'	$(a_j)_n$	Yes
'm+n'	$(a_j)_{m+n}$	Yes
'm-n'	$(a_j)_{m-n}$	No
'n-m'	$(a_j)_{n-m}$	No
'2m+n'	$(a_j)_{2m+n}$	No
'2m-n'	$(a_j)_{2m-n}$	No
'2n-m'	$(a_j)_{2n-m}$	No

For example, the Appell F1 and F4 functions

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

$$F_4 = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a)_{m+n}(b)_{m+n}}{(c)_m(d)_n} \frac{x^m y^n}{m!n!}$$

can be represented respectively as

```
hyper2d({'m+n':[a], 'm':[b], 'n':[c]}, {'m+n':[d]}, x, y)
```

```
hyper2d({'m+n':[a,b]}, {'m':[c], 'n':[d]}, x, y)
```

More generally, *hyper2d()* can evaluate any of the 34 distinct convergent second-order (generalized Gaussian) hypergeometric series enumerated by Horn, as well as the Kampe de Feriet function.

The series is computed by rewriting it so that the inner series (i.e. the series containing n and y) has the form of an ordinary generalized hypergeometric series and thereby can be evaluated efficiently using *hyper()*. If possible, manually swapping x and y and the corresponding parameters can sometimes give better results.

Examples

Two separable cases: a product of two geometric series, and a product of two Gaussian hypergeometric functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> x, y = mpf(0.25), mpf(0.5)
>>> hyper2d({'m':1,'n':1}, {}, x,y)
2.666666666666666666666667
>>> 1/(1-x)/(1-y)
2.666666666666666666666667
>>> hyper2d({'m':[1,2],'n':[3,4]}, {'m':[5],'n':[6]}, x,y)
4.164358531238938319669856
>>> hyp2f1(1,2,5,x)*hyp2f1(3,4,6,y)
4.164358531238938319669856
```

Some more series that can be done in closed form:

```
>>> hyper2d({'m':1,'n':1},{'m+n':1},x,y)
2.013417124712514809623881
>>> (exp(x)*x-exp(y)*y)/(x-y)
2.013417124712514809623881
```

Six of the 34 Horn functions, G1-G3 and H1-H3:

```
>>> from mpmath import *
>>> mp.dps = 10; mp.pretty = True
>>> x, y = 0.0625, 0.125
>>> a1,a2,b1,b2,c1,c2,d = 1.1,-1.2,-1.3,-1.4,1.5,-1.6,1.7
```

```
>>> hyper2d({'m+n':a1,'n-m':b1,'m-n':b2},{},x,y)  # G1
1.139090746
>>> nsum(lambda m,n: rf(a1,m+n)*rf(b1,n-m)*rf(b2,m-n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
1.139090746
>>> hyper2d({'m':a1,'n':a2,'n-m':b1,'m-n':b2},{},x,y)  # G2
0.9503682696
>>> nsum(lambda m,n: rf(a1,m)*rf(a2,n)*rf(b1,n-m)*rf(b2,m-n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
0.9503682696
>>> hyper2d({'2n-m':a1,'2m-n':a2},{},x,y)  # G3
1.029372029
>>> nsum(lambda m,n: rf(a1,2*n-m)*rf(a2,2*m-n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
1.029372029
>>> hyper2d({'m-n':a1,'m+n':b1,'n':c1},{'m':d},x,y)  # H1
-1.605331256
>>> nsum(lambda m,n: rf(a1,m-n)*rf(b1,m+n)*rf(c1,n)/rf(d,m)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
-1.605331256
>>> hyper2d({'m-n':a1,'m':b1,'n':[c1,c2]},{'m':d},x,y)  # H2
-2.35405404
>>> nsum(lambda m,n: rf(a1,m-n)*rf(b1,m)*rf(c1,n)*rf(c2,n)/rf(d,m)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
-2.35405404
>>> hyper2d({'2m+n':a1,'n':b1},{'m+n':c1},x,y)  # H3
0.974479074
>>> nsum(lambda m,n: rf(a1,2*m+n)*rf(b1,n)/rf(c1,m+n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
0.974479074
```

References

1. *[SrivastavaKarlsson]*
2. *[Weisstein]* http://mathworld.wolfram.com/HornFunction.html
3. *[Weisstein]* http://mathworld.wolfram.com/AppellHypergeometricFunction.html

appellf1()

mpmath.**appellf1**(*a*, *b1*, *b2*, *c*, *x*, *y*, ***kwargs*)

Gives the Appell F1 hypergeometric function of two variables,

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

This series is only generally convergent when $|x| < 1$ and $|y| < 1$, although *appellf1()* can evaluate an analytic continuation with respecto to either variable, and sometimes both.

Examples

Evaluation is supported for real and complex parameters:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf1(1,0,0.5,1,0.5,0.25)
1.1547005383792515290182 98
```

```
>>> appellf1(1,1+j,0.5,1,0.5,0.5j)
(1.13840386035014808517941 5 + 1.5105447410585176211106 5j)
```

For some integer parameters, the F1 series reduces to a polynomial:

```
>>> appellf1(2,-4,-3,1,2,5)
-816.0
>>> appellf1(-5,1,2,1,4,5)
-20528.0
```

The analytic continuation with respect to either x or y, and sometimes with respect to both, can be evaluated:

```
>>> appellf1(2,3,4,5,100,0.5)
(0.000623104271416532927973866 2 + 0.000000576914927714842577449985 7j)
>>> appellf1('1.1', '0.3', '0.2+2j', '0.4', '0.2', 1.5+3j)
(-0.178260456689395489712870 2 + 0.00247240710454621611716149 9j)
>>> appellf1(1,2,3,4,10,12)
-0.071229938300667763749293 3
```

For certain arguments, F1 reduces to an ordinary hypergeometric function:

```
>>> appellf1(1,2,3,5,0.5,0.25)
1.54790227030268401933555 5
>>> 4*hyp2f1(1,2,5,'1/3')/3
1.54790227030268401933555 5
>>> appellf1(1,2,3,4,0,1.5)
(-1.71720250616893750274023 8 - 2.79252680319092732307790 5j)
>>> hyp2f1(1,3,4,1.5)
(-1.71720250616893750274023 8 - 2.79252680319092732307790 5j)
```

The F1 function satisfies a system of partial differential equations:

```
>>> a,b1,b2,c,x,y = map(mpf, [1,0.5,0.25,1.125,0.25,-0.25])
>>> F = lambda x,y: appellf1(a,b1,b2,c,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) +
...      y*(1-x)*diff(F,(x,y),(1,1)) +
...      (c-(a+b1+1)*x)*diff(F,(x,y),(1,0)) -
...      b1*y*diff(F,(x,y),(0,1)) -
...      a*b1*F(x,y))
0.0
>>>
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) +
...      x*(1-y)*diff(F,(x,y),(1,1)) +
...      (c-(a+b2+1)*y)*diff(F,(x,y),(0,1)) -
...      b2*x*diff(F,(x,y),(1,0)) -
...      a*b2*F(x,y))
0.0
```

The Appell F1 function allows for closed-form evaluation of various integrals, such as any integral of the form $\int x^r(x+a)^p(x+b)^q dx$:

```
>>> def integral(a,b,p,q,r,x1,x2):
...     a,b,p,q,r,x1,x2 = map(mpmathify, [a,b,p,q,r,x1,x2])
...     f = lambda x: x**r * (x+a)**p * (x+b)**q
...     def F(x):
...         v = x**(r+1)/(r+1) * (a+x)**p * (b+x)**q
...         v *= (1+x/a)**(-p)
...         v *= (1+x/b)**(-q)
...         v *= appellf1(r+1,-p,-q,2+r,-x/a,-x/b)
...         return v
```

```
...     print("Num. quad: %s" % quad(f, [x1,x2]))
...     print("Appell F1: %s" % (F(x2)-F(x1)))
...
>>> integral('1/5','4/3','-2','3','1/2',0,1)
Num. quad: 9.0733353587857762065769810
Appell F1: 9.0733353587857762065769810
>>> integral('3/2','4/3','-2','3','1/2',0,1)
Num. quad: 1.0928291719996264543446780
Appell F1: 1.0928291719996264543446780
>>> integral('3/2','4/3','-2','3','1/2',12,25)
Num. quad: 1106.323225040235116498270
Appell F1: 1106.323225040235116498270
```

Also incomplete elliptic integrals fall into this category [1]:

```
>>> def E(z, m):
...     if (pi/2).ae(z):
...         return ellipe(m)
...     return 2*round(re(z)/pi)*ellipe(m) + mpf(-1)**round(re(z)/pi)*\
...         sin(z)*appellf1(0.5,0.5,-0.5,1.5,sin(z)**2,m*sin(z)**2)
...
>>> z, m = 1, 0.5
>>> E(z,m); quad(lambda t: sqrt(1-m*sin(t)**2), [0,pi/4,3*pi/4,z])
0.9273298836244400669659042
0.9273298836244400669659042
>>> z, m = 3, 2
>>> E(z,m); quad(lambda t: sqrt(1-m*sin(t)**2), [0,pi/4,3*pi/4,z])
(1.0574957523372342297158360 + 1.1981402347355922074399220j)
(1.0574957523372342297158360 + 1.1981402347355922074399220j)
```

References

1. *[WolframFunctions]* http://functions.wolfram.com/EllipticIntegrals/EllipticE2/26/01/
2. *[SrivastavaKarlsson]*
3. *[CabralRosetti]*
4. *[Vidunas]*
5. *[Slater]*

appellf2()

mpmath.**appellf2**(*a*, *b1*, *b2*, *c1*, *c2*, *x*, *y*, ***kwargs*)

Gives the Appell F2 hypergeometric function of two variables

$$F_2(a,b_1,b_2,c_1,c_2,x,y) = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a)_{m+n}(b_1)_m(b_2)_n}{(c_1)_m(c_2)_n}\frac{x^m y^n}{m!n!}.$$

The series is generally absolutely convergent for $|x| + |y| < 1$.

Examples

Evaluation for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf2(1,2,3,4,5,0.25,0.125)
```

```
1.257417193533135344785602
>>> appellf2(1,-3,-4,2,3,2,3)
-42.8
>>> appellf2(0.5,0.25,-0.25,2,3,0.25j,0.25)
(0.9880539519421899867041719 + 0.014976161650311026614769 78j)
>>> chop(appellf2(1,1+j,1-j,3j,-3j,0.25,0.25))
1.20131121928741133795519 2
>>> appellf2(1,1,1,4,6,0.125,16)
(-0.094555322502747442821251 52 - 0.76472822530462078367692 97j)
```

A transformation formula:

```
>>> a,b1,b2,c1,c2,x,y = map(mpf, [1,2,0.5,0.25,1.625,-0.125,0.125])
>>> appellf2(a,b1,b2,c1,c2,x,y)
0.22992117178411807833096 88
>>> (1-x)**(-a)*appellf2(a,c1-b1,b2,c1,c2,x/(x-1),y/(1-x))
0.22992117178411807833096 88
```

A system of partial differential equations satisfied by F2:

```
>>> a,b1,b2,c1,c2,x,y = map(mpf, [1,0.5,0.25,1.125,1.5,0.0625,-0.0625])
>>> F = lambda x,y: appellf2(a,b1,b2,c1,c2,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) -
...      x*y*diff(F,(x,y),(1,1)) +
...      (c1-(a+b1+1)*x)*diff(F,(x,y),(1,0)) -
...      b1*y*diff(F,(x,y),(0,1)) -
...      a*b1*F(x,y))
0.0
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) -
...      x*y*diff(F,(x,y),(1,1)) +
...      (c2-(a+b2+1)*y)*diff(F,(x,y),(0,1)) -
...      b2*x*diff(F,(x,y),(1,0)) -
...      a*b2*F(x,y))
0.0
```

References

See references for *appellf1()*.

appellf3()

mpmath.**appellf3**(*a1*, *a2*, *b1*, *b2*, *c*, *x*, *y*, ***kwargs*)

Gives the Appell F3 hypergeometric function of two variables

$$F_3(a_1,a_2,b_1,b_2,c,x,y) = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a_1)_m (a_2)_n (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m! n!}.$$

The series is generally absolutely convergent for $|x| < 1, |y| < 1$.

Examples

Evaluation for various parameters and variables:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf3(1,2,3,4,5,0.5,0.25)
2.22155777810743893815870 5
>>> appellf3(1,2,3,4,5,6,0); hyp2f1(1,3,5,6)
(-0.51895545890898612845373 89 - 0.14544410433286079807697 42j)
```

```
(-0.51895545890898612845373889 - 0.14544410433286079807697422j)
>>> appellf3(1,-2,-3,1,1,4,6)
-17.4
>>> appellf3(1,2,-3,1,1,4,6)
(17.787613677367735664182 5 + 19.54768762233649126154534j)
>>> appellf3(1,2,-3,1,1,6,4)
(85.020541750679294029536 45 + 148.4402528821177305173599j)
>>> chop(appellf3(1+j,2,1-j,2,3,0.25,0.25))
1.7199921695452002866960 07
```

Many transformations and evaluations for special combinations of the parameters are possible, e.g.:

```
>>> a,b,c,x,y = map(mpf, [0.5,0.25,0.125,0.125,-0.125])
>>> appellf3(a,c-a,b,c-b,c,x,y)
1.0934323408960871074443 63
>>> (1-y)**(a+b-c)*hyp2f1(a,b,c,x+y-x*y)
1.0934323408960871074443 63
>>> x**2*appellf3(1,1,1,1,3,x,-x)
0.0156864627744538539094 5083
>>> polylog(2,x**2)
0.0156864627744538539094 5083
>>> a1,a2,b1,b2,c,x = map(mpf, [0.5,0.25,0.125,0.5,4.25,0.125])
>>> appellf3(a1,a2,b1,b2,c,x,1)
1.0394736170911140096947
>>> gammaprod([c,c-a2-b2],[c-a2,c-b2])*hyp3f2(a1,b1,c-a2-b2,c-a2,c-b2,x)
1.0394736170911140096947
```

The Appell F3 function satisfies a pair of partial differential equations:

```
>>> a1,a2,b1,b2,c,x,y = map(mpf, [0.5,0.25,0.125,0.5,0.625,0.0625,-0.0625])
>>> F = lambda x,y: appellf3(a1,a2,b1,b2,c,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) +
...      y*diff(F,(x,y),(1,1)) +
...     (c-(a1+b1+1)*x)*diff(F,(x,y),(1,0)) -
...     a1*b1*F(x,y))
0.0
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) +
...     x*diff(F,(x,y),(1,1)) +
...     (c-(a2+b2+1)*y)*diff(F,(x,y),(0,1)) -
...     a2*b2*F(x,y))
0.0
```

References

See references for *appellf1()*.

appellf4()

mpmath.**appellf4**(*a*, *b*, *c1*, *c2*, *x*, *y*, ***kwargs*)

Gives the Appell F4 hypergeometric function of two variables

$$F_4(a,b,c_1,c_2,x,y) = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a)_{m+n}(b)_{m+n}}{(c_1)_m(c_2)_n}\frac{x^m y^n}{m!n!}.$$

The series is generally absolutely convergent for $\sqrt{|x|}+\sqrt{|y|}<1$.

Examples

Evaluation for various parameters and arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf4(1,1,2,2,0.25,0.125)
1.2861820690797183135466 08
>>> appellf4(-2,-3,4,5,4,5)
34.8
>>> appellf4(5,4,2,3,0.25j,-0.125j)
(-0.25859672154378466421633 52 + 2.43610223355358271181874 3j)
```

Reduction to ${}_2F_1$ in a special case:

```
>>> a,b,c,x,y = map(mpf, [0.5,0.25,0.125,0.125,-0.125])
>>> appellf4(a,b,c,a+b-c+1,x*(1-y),y*(1-x))
1.12914348846685086824836 4
>>> hyp2f1(a,b,c,x)*hyp2f1(a,b,a+b-c+1,y)
1.12914348846685086824836 4
```

A system of partial differential equations satisfied by F4:

```
>>> a,b,c1,c2,x,y = map(mpf, [1,0.5,0.25,1.125,0.0625,-0.0625])
>>> F = lambda x,y: appellf4(a,b,c1,c2,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) -
...       y**2*diff(F,(x,y),(0,2)) -
...       2*x*y*diff(F,(x,y),(1,1)) +
...       (c1-(a+b+1)*x)*diff(F,(x,y),(1,0)) -
...       ((a+b+1)*y)*diff(F,(x,y),(0,1)) -
...       a*b*F(x,y))
0.0
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) -
...       x**2*diff(F,(x,y),(2,0)) -
...       2*x*y*diff(F,(x,y),(1,1)) +
...       (c2-(a+b+1)*y)*diff(F,(x,y),(0,1)) -
...       ((a+b+1)*x)*diff(F,(x,y),(1,0)) -
...       a*b*F(x,y))
0.0
```

References

See references for *appellf1()*.

3.1.10 Elliptic functions

Elliptic functions historically comprise the elliptic integrals and their inverses, and originate from the problem of computing the arc length of an ellipse. From a more modern point of view, an elliptic function is defined as a doubly periodic function, i.e. a function which satisfies

$$f(z + 2\omega_1) = f(z + 2\omega_2) = f(z)$$

for some half-periods ω_1, ω_2 with $\mathrm{Im}[\omega_1/\omega_2] > 0$. The canonical elliptic functions are the Jacobi elliptic functions. More broadly, this section includes quasi-doubly periodic functions (such as the Jacobi theta functions) and other functions useful in the study of elliptic functions.

Many different conventions for the arguments of elliptic functions are in use. It is even standard to use different parameterizations for different functions in the same text or software (and mpmath is no exception). The usual parameters are the elliptic nome q, which usually must satisfy $|q| < 1$; the elliptic parameter m (an arbitrary complex number); the elliptic modulus k (an arbitrary complex number); and the half-period ratio τ, which usually must satisfy $\mathrm{Im}[\tau] > 0$. These quantities can be expressed in terms of each other using the following relations:

$$m = k^2$$

$$\tau = -i\frac{K(1-m)}{K(m)}$$

$$q = e^{i\pi\tau}$$

$$k = \frac{\vartheta_2^4(q)}{\vartheta_3^4(q)}$$

In addition, an alternative definition is used for the nome in number theory, which we here denote by q-bar:

$$\bar{q} = q^2 = e^{2i\pi\tau}$$

For convenience, mpmath provides functions to convert between the various parameters (*qfrom()*, *mfrom()*, *kfrom()*, *taufrom()*, *qbarfrom()*).

References

1. *[AbramowitzStegun]*
2. *[WhittakerWatson]*

Elliptic arguments

qfrom()

mpmath.**qfrom**(***kwargs*)

Returns the elliptic nome q, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qfrom(q=0.25)
0.25
>>> qfrom(m=mfrom(q=0.25))
0.25
>>> qfrom(k=kfrom(q=0.25))
0.25
>>> qfrom(tau=taufrom(q=0.25))
(0.25 + 0.0j)
>>> qfrom(qbar=qbarfrom(q=0.25))
0.25
```

qbarfrom()

mpmath.**qbarfrom**(***kwargs*)

Returns the number-theoretic nome $\bar{q}$, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qbarfrom(qbar=0.25)
0.25
>>> qbarfrom(q=qfrom(qbar=0.25))
0.25
>>> qbarfrom(m=extraprec(20)(mfrom)(qbar=0.25))   # ill-conditioned
0.25
>>> qbarfrom(k=extraprec(20)(kfrom)(qbar=0.25))   # ill-conditioned
```

```
0.25
>>> qbarfrom(tau=taufrom(qbar=0.25))
(0.25 + 0.0j)
```

mfrom()

mpmath.**mfrom**(**kwargs*)

Returns the elliptic parameter m, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> mfrom(m=0.25)
0.25
>>> mfrom(q=qfrom(m=0.25))
0.25
>>> mfrom(k=kfrom(m=0.25))
0.25
>>> mfrom(tau=taufrom(m=0.25))
(0.25 + 0.0j)
>>> mfrom(qbar=qbarfrom(m=0.25))
0.25
```

As $q \to 1$ and $q \to -1$, m rapidly approaches 1 and $-\infty$ respectively:

```
>>> mfrom(q=0.75)
0.9999999999999798332943533
>>> mfrom(q=-0.75)
-49586681013729.32611558353
>>> mfrom(q=1)
1.0
>>> mfrom(q=-1)
-inf
```

The inverse nome as a function of q has an integer Taylor series expansion:

```
>>> taylor(lambda q: mfrom(q), 0, 7)
[0.0, 16.0, -128.0, 704.0, -3072.0, 11488.0, -38400.0, 117632.0]
```

kfrom()

mpmath.**kfrom**(**kwargs*)

Returns the elliptic modulus k, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> kfrom(k=0.25)
0.25
>>> kfrom(m=mfrom(k=0.25))
0.25
>>> kfrom(q=qfrom(k=0.25))
0.25
>>> kfrom(tau=taufrom(k=0.25))
(0.25 + 0.0j)
>>> kfrom(qbar=qbarfrom(k=0.25))
0.25
```

As $q \to 1$ and $q \to -1$, k rapidly approaches 1 and $i\infty$ respectively:

```
>>> kfrom(q=0.75)
0.99999999999998899166471767
>>> kfrom(q=-0.75)
(0.0 + 7041781.096692038332790615j)
>>> kfrom(q=1)
1
>>> kfrom(q=-1)
(0.0 + +infj)
```

taufrom()

mpmath.**taufrom**(***kwargs*)

Returns the elliptic half-period ratio τ, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> taufrom(tau=0.5j)
(0.0 + 0.5j)
>>> taufrom(q=qfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(m=mfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(k=kfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(qbar=qbarfrom(tau=0.5j))
(0.0 + 0.5j)
```

Legendre elliptic integrals

ellipk()

mpmath.**ellipk**(*m*, ***kwargs*)

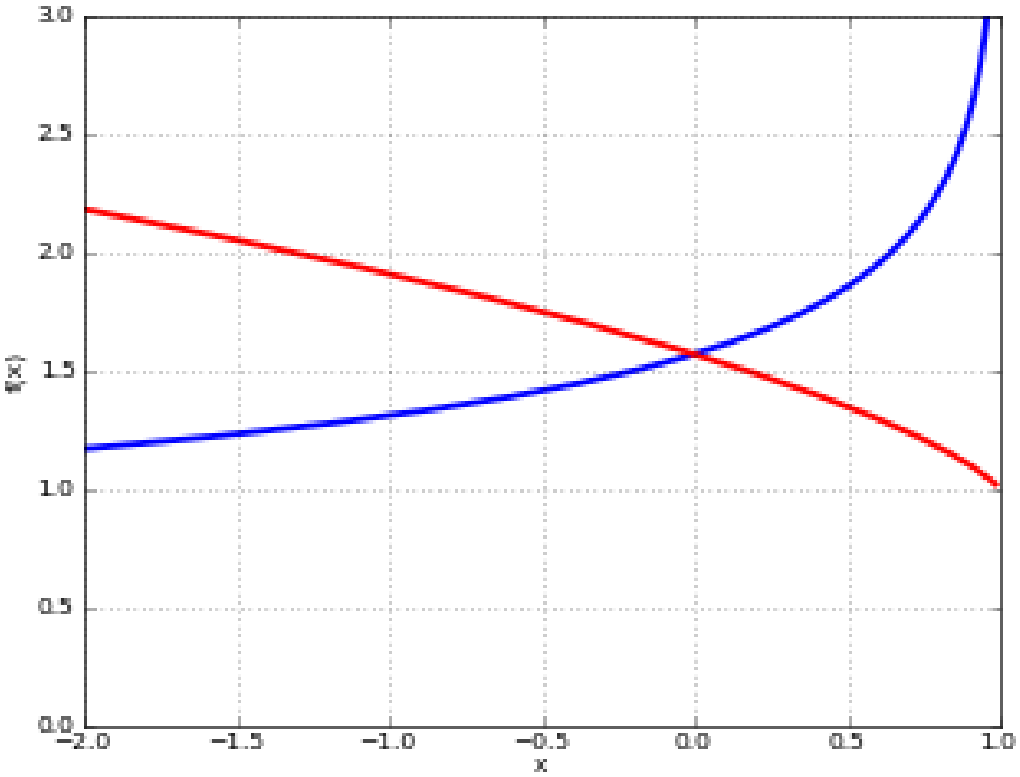
Evaluates the complete elliptic integral of the first kind, $K(m)$, defined by

$$K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \frac{\pi}{2} {}_2F_1\left(\frac{1}{2}, \frac{1}{2}, 1, m\right).$$

Note that the argument is the parameter $m = k^2$, not the modulus k which is sometimes used.

Plots

```
# Complete elliptic integrals K(m) and E(m)
plot([ellipk, ellipe], [-2,1], [0,3], points=600)
```



Examples

Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipk(0)
1.570796326794896619231322
>>> ellipk(inf)
(0.0 + 0.0j)
>>> ellipk(-inf)
0.0
>>> ellipk(1)
+inf
>>> ellipk(-1)
1.311028777146059905232 42
>>> ellipk(2)
(1.311028777146059905232 42 - 1.311028777146059905232 42j)
```

Verifying the defining integral and hypergeometric representation:

```
>>> ellipk(0.5)
1.854074677301371918433 85
>>> quad(lambda t: (1-0.5*sin(t)**2)**-0.5, [0, pi/2])
1.854074677301371918433 85
>>> pi/2*hyp2f1(0.5,0.5,1,0.5)
1.854074677301371918433 85
```

Evaluation is supported for arbitrary complex m:

```
>>> ellipk(3+4j)
(0.91119556380496500865621781 + 0.63133428324134524388845091j)
```

A definite integral:

```
>>> quad(ellipk, [0, 1])
2.0
```

ellipf()

mpmath.**ellipf**(*phi*, *m*)

Evaluates the Legendre incomplete elliptic integral of the first kind

$$F(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

or equivalently

$$F(\phi, m) = \int_0^{\sin \phi} \frac{dt}{\left(\sqrt{1 - t^2}\right)\left(\sqrt{1 - mt^2}\right)}.$$

The function reduces to a complete elliptic integral of the first kind (see *ellipk()*) when $\phi = \frac{\pi}{2}$; that is,

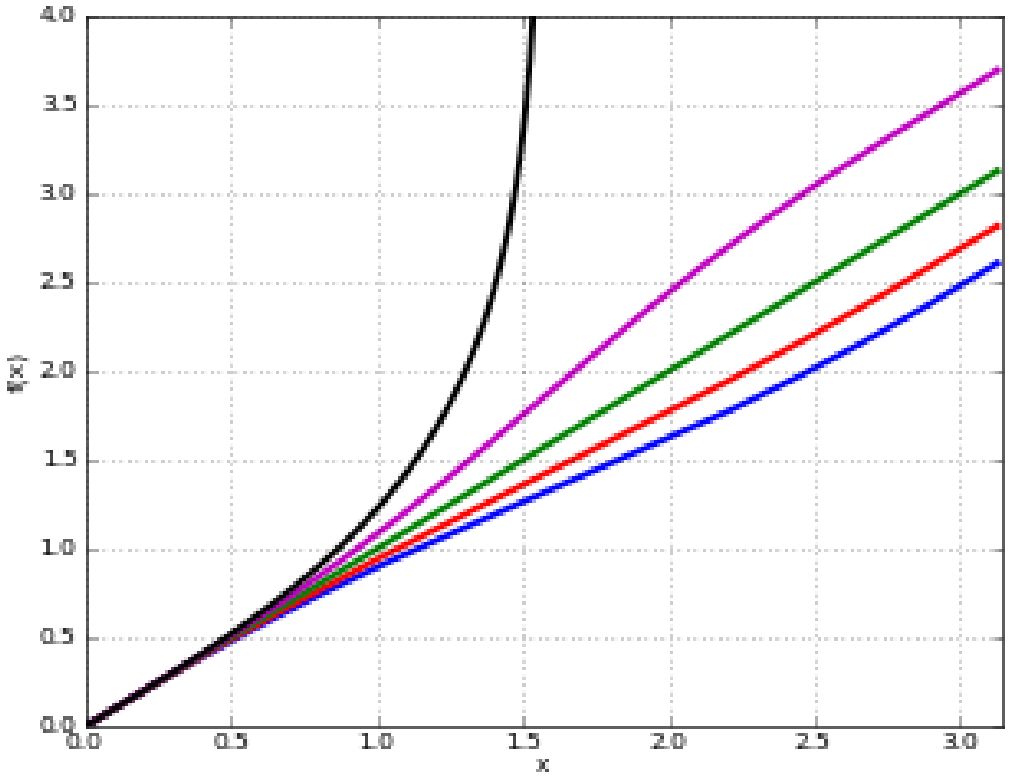
$$F\left(\frac{\pi}{2}, m\right) = K(m).$$

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside $-\pi/2 \le \Re(\phi) \le \pi/2$, the function extends quasi-periodically as

$$F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}.$$

Plots

```
# Elliptic integral F(z,m) for some different m
f1 = lambda z: ellipf(z,-1)
f2 = lambda z: ellipf(z,-0.5)
f3 = lambda z: ellipf(z,0)
f4 = lambda z: ellipf(z,0.5)
f5 = lambda z: ellipf(z,1)
plot([f1,f2,f3,f4,f5], [0,pi], [0,4])
```



Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipf(0,1)
0.0
>>> ellipf(0,0)
0.0
>>> ellipf(1,0); ellipf(2+3j,0)
1.0
(2.0 + 3.0j)
>>> ellipf(1,1); log(sec(1)+tan(1))
1.2261911708835170708130610
1.2261911708835170708130610
>>> ellipf(pi/2, -0.5); ellipk(-0.5)
1.4157372084259561988921660
1.4157372084259561988921660
>>> ellipf(pi/2+eps, 1); ellipf(-pi/2-eps, 1)
+inf
+inf
>>> ellipf(1.5, 1)
3.3406775427983110033208130
```

Comparing with numerical integration:

```
>>> z,m = 0.5, 1.25
>>> ellipf(z,m)
0.52872192022063278729782 55
>>> quad(lambda t: (1-m*sin(t)**2)**(-0.5), [0,z])
0.52872192022063278729782 55
```

The arguments may be complex numbers:

```
>>> ellipf(3j, 0.5)
(0.0 + 1.7136024078415902348041 43j)
>>> ellipf(3+4j, 5-6j)
(1.2691312419503513233057 41 - 0.3561052815014558335412538j)
>>> z,m = 2+3j, 1.25
>>> k = 1011
>>> ellipf(z+pi*k,m); ellipf(z,m) + 2*k*ellipk(m)
(4086.184383622179764082821 - 3003.0035389237493965468 71j)
(4086.184383622179764082821 - 3003.0035389237493965468 71j)
```

For $|\Re(z)| < \pi/2$, the function can be expressed as a hypergeometric series of two variables (see *appellf1()*):

```
>>> z,m = 0.5, 0.25
>>> ellipf(z,m)
0.50508872757864807888310 83
>>> sin(z)*appellf1(0.5,0.5,0.5,1.5,sin(z)**2,m*sin(z)**2)
0.50508872757864807888310 83
```

ellipe()

mpmath.**ellipe**(**args*)

Called with a single argument m, evaluates the Legendre complete elliptic integral of the second kind, $E(m)$, defined by

$$E(m) = \int_0^{\pi/2} \sqrt{1 - m \sin^2 t}\, dt = \frac{\pi}{2}\, {}_2F_1\left(\frac{1}{2}, -\frac{1}{2}, 1, m\right).$$

Called with two arguments ϕ, m, evaluates the incomplete elliptic integral of the second kind

$$E(\phi, m) = \int_0^{\phi} \sqrt{1 - m \sin^2 t}\, dt = \int_0^{\sin z} \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}}\, dt.$$

The incomplete integral reduces to a complete integral when $\phi = \frac{\pi}{2}$; that is,

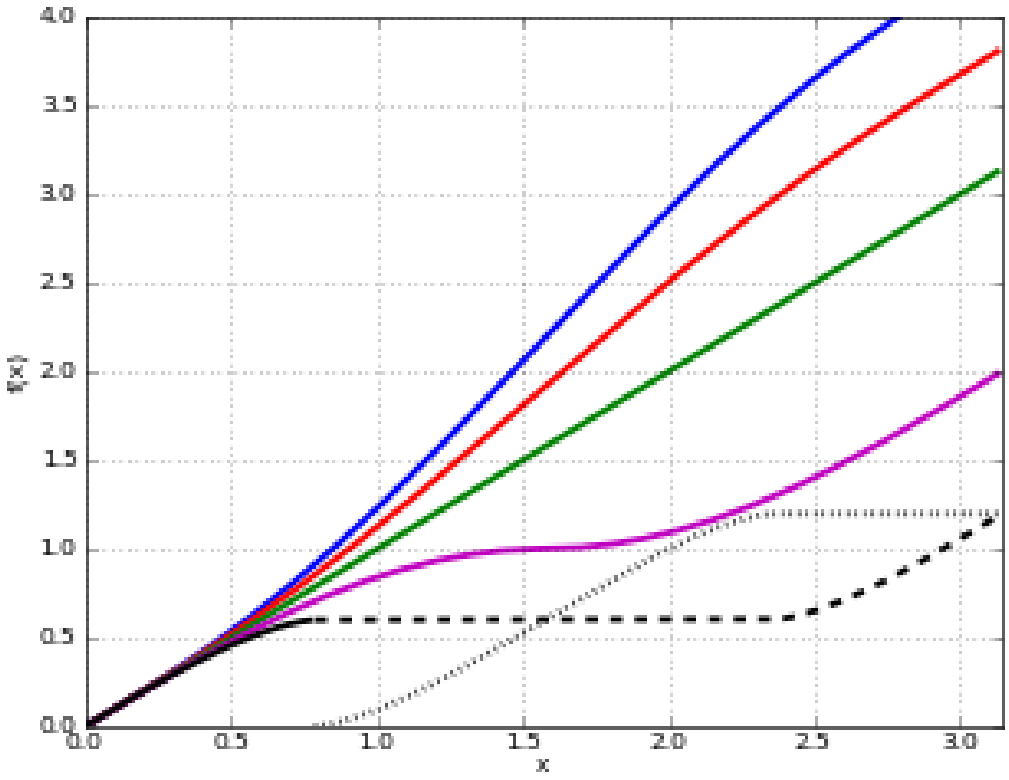
$$E\left(\frac{\pi}{2}, m\right) = E(m).$$

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside $-\pi/2 \le \Re(z) \le \pi/2$, the function extends quasi-periodically as

$$E(\phi + n\pi, m) = 2nE(m) + F(\phi, m), n \in \mathbb{Z}.$$

Plots

```
# Elliptic integral E(z,m) for some different m
f1 = lambda z: ellipe(z,-2)
f2 = lambda z: ellipe(z,-1)
f3 = lambda z: ellipe(z,0)
f4 = lambda z: ellipe(z,1)
f5 = lambda z: ellipe(z,2)
plot([f1,f2,f3,f4,f5], [0,pi], [0,4])
```



Examples for the complete integral

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipe(0)
1.570796326794896619231322
>>> ellipe(1)
1.0
>>> ellipe(-1)
1.910098894513856008952381
>>> ellipe(2)
(0.5990701173677961037199612 + 0.5990701173677961037199612j)
>>> ellipe(inf)
(0.0 + +infj)
>>> ellipe(-inf)
+inf
```

Verifying the defining integral and hypergeometric representation:

```
>>> ellipe(0.5)
1.3506438810476755025201750
>>> quad(lambda t: sqrt(1-0.5*sin(t)**2), [0, pi/2])
1.3506438810476755025201750
>>> pi/2*hyp2f1(0.5,-0.5,1,0.5)
1.3506438810476755025201750
```

Evaluation is supported for arbitrary complex m:

```
>>> ellipe(0.5+0.25j)
(1.360868682163129682716687 - 0.12387334425617868435573150j)
>>> ellipe(3+4j)
(1.4995535209333469543336120 - 1.5778790079127582745333090j)
```

A definite integral:

```
>>> quad(ellipe, [0,1])
1.3333333333333333333333333
```

Examples for the incomplete integral

Basic values and limits:

```
>>> ellipe(0,1)
0.0
>>> ellipe(0,0)
0.0
>>> ellipe(1,0)
1.0
>>> ellipe(2+3j,0)
(2.0 + 3.0j)
>>> ellipe(1,1); sin(1)
0.84147098480789650665250230
0.84147098480789650665250230
>>> ellipe(pi/2, -0.5); ellipe(-0.5)
1.7517712756948178620265020
1.7517712756948178620265020
>>> ellipe(pi/2, 1); ellipe(-pi/2, 1)
1.0
-1.0
>>> ellipe(1.5, 1)
0.99749498660405443094172337
```

Comparing with numerical integration:

```
>>> z,m = 0.5, 1.25
>>> ellipe(z,m)
0.47401521826526283942644494
>>> quad(lambda t: sqrt(1-m*sin(t)**2), [0,z])
0.47401521826526283942644494
```

The arguments may be complex numbers:

```
>>> ellipe(3j, 0.5)
(0.0 + 7.5519912348903718735021050j)
>>> ellipe(3+4j, 5-6j)
(24.152990225742205024244660 + 75.250367048032599741815600j)
>>> k = 35
>>> z,m = 2+3j, 1.25
>>> ellipe(z+pi*k,m); ellipe(z,m) + 2*k*ellipe(m)
```

```
(48.30138799412005235090766 + 17.47255216721987688224357j)
(48.30138799412005235090766 + 17.47255216721987688224357j)
```

For $|\Re(z)| < \pi/2$, the function can be expressed as a hypergeometric series of two variables (see *appellf1()*):

```
>>> z,m = 0.5, 0.25
>>> ellipe(z,m)
0.49500170301641519288870375
>>> sin(z)*appellf1(0.5,0.5,-0.5,1.5,sin(z)**2,m*sin(z)**2)
0.49500170301641519288870376
```

ellippi()

mpmath.**ellippi**(**args*)

Called with three arguments n, ϕ, m, evaluates the Legendre incomplete elliptic integral of the third kind

$$\Pi(n;\phi,m) = \int_0^{\phi} \frac{dt}{(1-n\sin^2 t)\sqrt{1-m\sin^2 t}} = \int_0^{\sin\phi} \frac{dt}{(1-nt^2)\sqrt{1-t^2}\sqrt{1-mt^2}}.$$

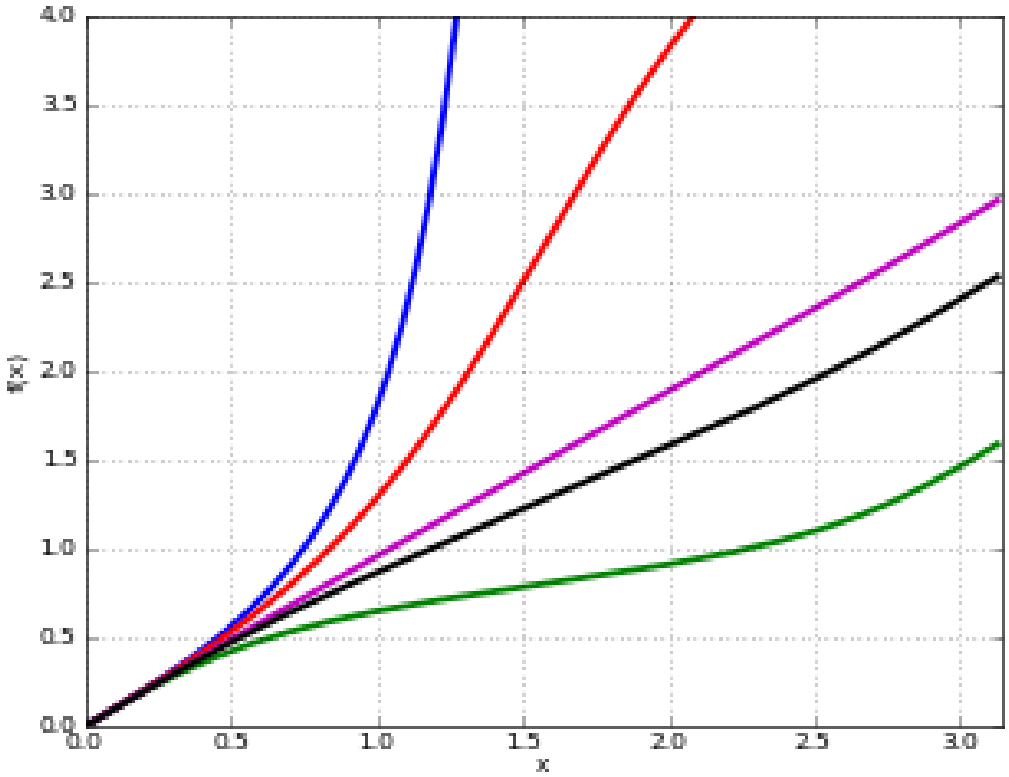
Called with two arguments n, m, evaluates the complete elliptic integral of the third kind $\Pi(n,m) = \Pi(n;\frac{\pi}{2},m)$.

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside $-\pi/2 \le \Re(\phi) \le \pi/2$, the function extends quasi-periodically as

$$\Pi(n,\phi+k\pi,m) = 2k\Pi(n,m) + \Pi(n,\phi,m), k \in \mathbb{Z}.$$

Plots

```
# Elliptic integral Pi(n,z,m) for some different n, m
f1 = lambda z: ellippi(0.9,z,0.9)
f2 = lambda z: ellippi(0.5,z,0.5)
f3 = lambda z: ellippi(-2,z,-0.9)
f4 = lambda z: ellippi(-0.5,z,0.5)
f5 = lambda z: ellippi(-1,z,0.5)
plot([f1,f2,f3,f4,f5], [0,pi], [0,4])
```



Examples for the complete integral

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellippi(0,-5); ellipk(-5)
0.95550392706404393373793334
0.95550392706404393373793334
>>> ellippi(inf,2)
0.0
>>> ellippi(2,inf)
0.0
>>> abs(ellippi(1,5))
+inf
>>> abs(ellippi(0.25,1))
+inf
```

Evaluation in terms of simpler functions:

```
>>> ellippi(0.25,0.25); ellipe(0.25)/(1-0.25)
1.9566162791192362072797 27
1.9566162791192362072797 27
>>> ellippi(3,0); pi/(2*sqrt(-2))
(0.0 - 1.1107207345395915617539 7j)
(0.0 - 1.1107207345395915617539 7j)
>>> ellippi(-3,0); pi/(2*sqrt(4))
0.7853981633974483096156609
0.7853981633974483096156609
```

Examples for the incomplete integral

Basic values and limits:

```
>>> ellippi(0.25,-0.5); ellippi(0.25,pi/2,-0.5)
1.6229447609547416037105555
1.6229447609547416037105555
>>> ellippi(1,0,1)
0.0
>>> ellippi(inf,0,1)
0.0
>>> ellippi(0,0.25,0.5); ellipf(0.25,0.5)
0.25130400865449257941345910
0.25130400865449257941345910
>>> ellippi(1,1,1); (log(sec(1)+tan(1))+sec(1)*tan(1))/2
2.0543329332562486686924520
2.0543329332562486686924520
>>> ellippi(0.25, 53*pi/2, 0.75); 53*ellippi(0.25,0.75)
135.24086875789084075505800
135.24086875789084075505800
>>> ellippi(0.5,pi/4,0.5); 2*ellipe(pi/4,0.5)-1/sqrt(3)
0.91902273916569699039872690
0.91902273916569699039872690
```

Complex arguments are supported:

```
>>> ellippi(0.5, 5+6j-2*pi, -7-8j)
(-0.36128566200767476604101670 + 0.52177353399848078297558150j)
```

Some degenerate cases:

```
>>> ellippi(1,1)
+inf
>>> ellippi(1,0)
+inf
>>> ellippi(1,2,0)
+inf
>>> ellippi(1,2,1)
+inf
>>> ellippi(1,0,1)
0.0
```

Carlson symmetric elliptic integrals

elliprf()

mpmath.**elliprf**(*x*, *y*, *z*)

Evaluates the Carlson symmetric elliptic integral of the first kind

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}$$

which is defined for $x, y, z \notin (-\infty, 0)$, and with at most one of x, y, z being zero.

For real $x, y, z \geq 0$, the principal square root is taken in the integrand. For complex x, y, z, the principal square root is taken as $t \to \infty$ and as $t \to 0$ non-principal branches are chosen as necessary so as to make the integrand continuous.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprf(0,1,1); pi/2
1.570796326794896619231322
1.570796326794896619231322
>>> elliprf(0,1,inf)
0.0
>>> elliprf(1,1,1)
1.0
>>> elliprf(2,2,2)**2
0.5
>>> elliprf(1,0,0); elliprf(0,0,1); elliprf(0,1,0); elliprf(0,0,0)
+inf
+inf
+inf
+inf
```

Representing complete elliptic integrals in terms of R_F:

```
>>> m = mpf(0.75)
>>> ellipk(m); elliprf(0,1-m,1)
2.156515647499643235438675
2.156515647499643235438675
>>> ellipe(m); elliprf(0,1-m,1)-m*elliprd(0,1-m,1)/3
1.211056027568459524803563
1.211056027568459524803563
```

Some symmetries and argument transformations:

```
>>> x,y,z = 2,3,4
>>> elliprf(x,y,z); elliprf(y,x,z); elliprf(z,y,x)
0.58408284167715170669284920
0.58408284167715170669284920
0.58408284167715170669284920
>>> k = mpf(100000)
>>> elliprf(k*x,k*y,k*z); k**(-0.5) * elliprf(x,y,z)
0.00184703212192332125321928400
0.00184703212192332125321928400
>>> l = sqrt(x*y) + sqrt(y*z) + sqrt(z*x)
>>> elliprf(x,y,z); 2*elliprf(x+l,y+l,z+l)
0.58408284167715170669284920
0.58408284167715170669284920
>>> elliprf((x+l)/4,(y+l)/4,(z+l)/4)
0.58408284167715170669284920
```

Comparing with numerical integration:

```
>>> x,y,z = 2,3,4
>>> elliprf(x,y,z)
0.58408284167715170669284920
>>> f = lambda t: 0.5*((t+x)*(t+y)*(t+z))**(-0.5)
>>> q = extradps(25)(quad)
>>> q(f, [0,inf])
0.58408284167715170669284920
```

With the following arguments, the square root in the integrand becomes discontinuous at $t = 1/2$ if the principal branch is used. To obtain the right value, $-\sqrt{r}$ must be taken instead of $\sqrt{r}$ on $t \in (0, 1/2)$:

```
>>> x,y,z = j-1,j,0
>>> elliprf(x,y,z)
(0.79612586584233913293056940 - 1.21385666983649598643009 4j)
>>> -q(f, [0,0.5]) + q(f, [0.5,inf])
(0.79612586584233913293056940 - 1.21385666983649598643009 4j)
```

The so-called *first lemniscate constant*, a transcendental number:

```
>>> elliprf(0,1,2)
1.3110287771460599052324 2
>>> extradps(25)(quad)(lambda t: 1/sqrt(1-t**4), [0,1])
1.3110287771460599052324 2
>>> gamma('1/4')**2/(4*sqrt(2*pi))
1.3110287771460599052324 2
```

References

1. *[Carlson]*
2. *[DLMF]* Chapter 19. Elliptic Integrals

elliprc()

mpmath.**elliprc**(*x*, *y*, *pv=True*)

Evaluates the degenerate Carlson symmetric elliptic integral of the first kind

$$R_C(x,y) = R_F(x,y,y) = \frac{1}{2}\int_0^\infty \frac{dt}{(t+y)\sqrt{(t+x)}}.$$

If $y \in (-\infty, 0)$, either a value defined by continuity, or with *pv=True* the Cauchy principal value, can be computed.

If $x \geq 0, y > 0$, the value can be expressed in terms of elementary functions as

$$R_C(x,y) = \begin{cases} \dfrac{1}{\sqrt{y-x}}\cos^{-1}\left(\sqrt{\dfrac{x}{y}}\right), & x < y \\ \dfrac{1}{\sqrt{y}}, & x = y \\ \dfrac{1}{\sqrt{x-y}}\cosh^{-1}\left(\sqrt{\dfrac{x}{y}}\right), & x > y \end{cases}.$$

Examples

Some special values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprc(1,2)*4; elliprc(0,1)*2; +pi
3.14159265358979323846264
3.14159265358979323846264
3.14159265358979323846264
>>> elliprc(1,0)
+inf
>>> elliprc(5,5)**2
0.2
>>> elliprc(1,inf); elliprc(inf,1); elliprc(inf,inf)
0.0
0.0
0.0
```

Comparing with the elementary closed-form solution:

```
>>> elliprc('1/3', '1/5'); sqrt(7.5)*acosh(sqrt('5/3'))
2.0416307789834983907512380
2.0416307789834983907512380
>>> elliprc('1/5', '1/3'); sqrt(7.5)*acos(sqrt('3/5'))
1.8751807652065470651110850
1.8751807652065470651110850
```

Comparing with numerical integration:

```
>>> q = extradps(25)(quad)
>>> elliprc(2, -3, pv=True)
0.33339691011136726707493340
>>> elliprc(2, -3, pv=False)
(0.33339691011136726707493340 + 0.70248147310407263931563750j)
>>> 0.5*q(lambda t: 1/(sqrt(t+2)*(t-3)), [0,3-j,6,inf])
(0.33339691011136726707493340 + 0.70248147310407263931563750j)
```

elliprj()

mpmath.**elliprj**(*x*, *y*, *z*, *p*)

Evaluates the Carlson symmetric elliptic integral of the third kind

$$R_J(x,y,z,p) = \frac{3}{2}\int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}}.$$

Like *elliprf()*, the branch of the square root in the integrand is defined so as to be continuous along the path of integration for complex values of the arguments.

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprj(1,1,1,1)
1.0
>>> elliprj(2,2,2,2); 1/(2*sqrt(2))
0.3535533905932737622004222
0.3535533905932737622004222
>>> elliprj(0,1,2,2)
1.067937989667395702268688
>>> 3*(2*gamma('5/4')**2-pi**2/gamma('1/4')**2)/(sqrt(2*pi))
1.067937989667395702268688
>>> elliprj(0,1,1,2); 3*pi*(2-sqrt(2))/4
1.380226776765915172432054
1.380226776765915172432054
>>> elliprj(1,3,2,0); elliprj(0,1,1,0); elliprj(0,0,0,0)
+inf
+inf
+inf
>>> elliprj(1,inf,1,0); elliprj(1,1,1,inf)
0.0
0.0
>>> chop(elliprj(1+j, 1-j, 1, 1))
0.8505007163686739432927844
```

Scale transformation:

```
>>> x,y,z,p = 2,3,4,5
>>> k = mpf(100000)
>>> elliprj(k*x,k*y,k*z,k*p); k**(-1.5)*elliprj(x,y,z,p)
4.5212916775927455278511 68e-9
4.5212916775927455278511 68e-9
```

Comparing with numerical integration:

```
>>> elliprj(1,2,3,4)
0.23984809974956776217586 17
>>> f = lambda t: 1/((t+4)*sqrt((t+1)*(t+2)*(t+3)))
>>> 1.5*quad(f, [0,inf])
0.23984809974956776217586 17
>>> elliprj(1,2+1j,3,4-2j)
(0.21688890601463349873995 2 + 0.04081912627366673332369512j)
>>> f = lambda t: 1/((t+4-2j)*sqrt((t+1)*(t+2+1j)*(t+3)))
>>> 1.5*quad(f, [0,inf])
(0.21688890601463349873995 2 + 0.04081912627366673332369511j)
```

elliprd()

mpmath.**elliprd**(*x*, *y*, *z*)

Evaluates the degenerate Carlson symmetric elliptic integral of the third kind or Carlson elliptic integral of the second kind $R_D(x,y,z) = R_J(x,y,z,z)$.

See *elliprj()* for additional information.

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprd(1,2,3)
0.2904602810289906442326534
>>> elliprj(1,2,3,3)
0.2904602810289906442326534
```

The so-called *second lemniscate constant*, a transcendental number:

```
>>> elliprd(0,2,1)/3
0.5990701173677961037199612
>>> extradps(25)(quad)(lambda t: t**2/sqrt(1-t**4), [0,1])
0.5990701173677961037199612
>>> gamma('3/4')**2/sqrt(2*pi)
0.5990701173677961037199612
```

elliprg()

mpmath.**elliprg**(*x*, *y*, *z*)

Evaluates the Carlson completely symmetric elliptic integral of the second kind

$$R_G(x,y,z) = \frac{1}{4}\int_0^\infty \frac{t}{\sqrt{(t+x)(t+y)(t+z)}}\left(\frac{x}{t+x}+\frac{y}{t+y}+\frac{z}{t+z}\right)dt.$$

Examples

Evaluation for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprg(0,1,1)*4; +pi
3.1415926535897932384626433
3.1415926535897932384626433
>>> elliprg(0,0.5,1)
0.67532194052383775126008740
>>> chop(elliprg(1+j, 1-j, 2))
1.1724313276764166045328220
```

A double integral that can be evaluated in terms of R_G:

```
>>> x,y,z = 2,3,4
>>> def f(t,u):
...     st = fp.sin(t); ct = fp.cos(t)
...     su = fp.sin(u); cu = fp.cos(u)
...     return (x*(st*cu)**2 + y*(st*su)**2 + z*ct**2)**0.5 * st
...
>>> nprint(mpf(fp.quad(f, [0,fp.pi], [0,2*fp.pi])/(4*fp.pi)), 13)
1.725503028069
>>> nprint(elliprg(x,y,z), 13)
1.725503028069
```

Jacobi theta functions

jtheta()

mpmath.**jtheta**(*n*, *z*, *q*, *derivative=0*)

Computes the Jacobi theta function $\vartheta_n(z, q)$, where $n = 1, 2, 3, 4$, defined by the infinite series:

$$\vartheta_1(z,q) = 2q^{1/4}\sum_{n=0}^{\infty}(-1)^n q^{n^2+n}\sin((2n+1)z)$$

$$\vartheta_2(z,q) = 2q^{1/4}\sum_{n=0}^{\infty} q^{n^2+n}\cos((2n+1)z)$$

$$\vartheta_3(z,q) = 1 + 2\sum_{n=1}^{\infty} q^{n^2}\cos(2nz)$$

$$\vartheta_4(z,q) = 1 + 2\sum_{n=1}^{\infty}(-q)^{n^2}\cos(2nz)$$

The theta functions are functions of two variables:

- z is the *argument*, an arbitrary real or complex number
- q is the *nome*, which must be a real or complex number in the unit disk (i.e. $|q| < 1$). For $|q| \ll 1$, the series converge very quickly, so the Jacobi theta functions can efficiently be evaluated to high precision.

The compact notations $\vartheta_n(q) = \vartheta_n(0, q)$ and $\vartheta_n = \vartheta_n(0, q)$ are also frequently encountered. Finally, Jacobi theta functions are frequently considered as functions of the half-period ratio τ and then usually denoted by $\vartheta_n(z|\tau)$.

Optionally, `jtheta(n, z, q, derivative=d)` with $d > 0$ computes a d-th derivative with respect to z.

Examples and basic properties

Considered as functions of z, the Jacobi theta functions may be viewed as generalizations of the ordinary trigonometric functions cos and sin. They are periodic functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> jtheta(1, 0.25, '0.2')
0.29451207986273000450531040
>>> jtheta(1, 0.25 + 2*pi, '0.2')
0.29451207986273000450531040
```

Indeed, the series defining the theta functions are essentially trigonometric Fourier series. The coefficients can be retrieved using *fourier()*:

```
>>> mp.dps = 10
>>> nprint(fourier(lambda x: jtheta(2, x, 0.5), [-pi, pi], 4))
([0.0, 1.68179, 0.0, 0.420448, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0])
```

The Jacobi theta functions are also so-called quasiperiodic functions of z and τ, meaning that for fixed τ, $\vartheta_n(z,q)$ and $\vartheta_n(z+\pi\tau,q)$ are the same except for an exponential factor:

```
>>> mp.dps = 25
>>> tau = 3*j/10
>>> q = exp(pi*j*tau)
>>> z = 10
>>> jtheta(4, z+tau*pi, q)
(-0.6824202807860346875205680 + 1.526683999721399103332021j)
>>> -exp(-2*j*z)/q * jtheta(4, z, q)
(-0.6824202807860346875205680 + 1.526683999721399103332021j)
```

The Jacobi theta functions satisfy a huge number of other functional equations, such as the following identity (valid for any q):

```
>>> q = mpf(3)/10
>>> jtheta(3,0,q)**4
6.823744089352763305137427
>>> jtheta(2,0,q)**4 + jtheta(4,0,q)**4
6.823744089352763305137427
```

Extensive listings of identities satisfied by the Jacobi theta functions can be found in standard reference works.

The Jacobi theta functions are related to the gamma function for special arguments:

```
>>> jtheta(3, 0, exp(-pi))
1.086434811213308014575316
>>> pi**(1/4.) / gamma(3/4.)
1.086434811213308014575316
```

jtheta() supports arbitrary precision evaluation and complex arguments:

```
>>> mp.dps = 50
>>> jtheta(4, sqrt(2), 0.5)
2.0549510717571539127004115835148878097035750653737
>>> mp.dps = 25
>>> jtheta(4, 1+2j, (1+j)/5)
(7.1803317601468059263566340 - 1.6342928581191624173016830j)
```

Evaluation of derivatives:

```
>>> mp.dps = 25
>>> jtheta(1, 7, 0.25, 1); diff(lambda z: jtheta(1, z, 0.25), 7)
1.209857192844475388637236
1.209857192844475388637236
>>> jtheta(1, 7, 0.25, 2); diff(lambda z: jtheta(1, z, 0.25), 7, 2)
```

```
    -0.25987187916502017206533052
    -0.25987187916502017206533052
    >>> jtheta(2, 7, 0.25, 1); diff(lambda z: jtheta(2, z, 0.25), 7)
    -1.1502314370702596444614 74
    -1.1502314370702596444614 74
    >>> jtheta(2, 7, 0.25, 2); diff(lambda z: jtheta(2, z, 0.25), 7, 2)
    -0.6226636990043777445898114
    -0.6226636990043777445898114
    >>> jtheta(3, 7, 0.25, 1); diff(lambda z: jtheta(3, z, 0.25), 7)
    -0.9990312046096634316587882
    -0.9990312046096634316587882
    >>> jtheta(3, 7, 0.25, 2); diff(lambda z: jtheta(3, z, 0.25), 7, 2)
    -0.1530388693066334936151174
    -0.1530388693066334936151174
    >>> jtheta(4, 7, 0.25, 1); diff(lambda z: jtheta(4, z, 0.25), 7)
    0.98209959672627939435711 39
    0.98209959672627939435711 39
    >>> jtheta(4, 7, 0.25, 2); diff(lambda z: jtheta(4, z, 0.25), 7, 2)
    0.39369028502914370816677 55
    0.39369028502914370816677 55
```

Possible issues

For $|q| \geq 1$ or $\Im(\tau) \leq 0$, *jtheta()* raises `ValueError`. This exception is also raised for $|q|$ extremely close to 1 (or equivalently τ very close to 0), since the series would converge too slowly:

```
    >>> jtheta(1, 10, 0.99999999 * exp(0.5*j))
    Traceback (most recent call last):
      ...
    ValueError: abs(q) > THETA_Q_LIM = 1.000000
```

Jacobi elliptic functions

ellipfun()

mpmath.**ellipfun**(*kind*, *u=None*, *m=None*, *q=None*, *k=None*, *tau=None*)

Computes any of the Jacobi elliptic functions, defined in terms of Jacobi theta functions as

$$\begin{aligned}\operatorname{sn}(u,m) &= \frac{\vartheta_3(0,q)}{\vartheta_2(0,q)}\frac{\vartheta_1(t,q)}{\vartheta_4(t,q)} \\ \operatorname{cn}(u,m) &= \frac{\vartheta_4(0,q)}{\vartheta_2(0,q)}\frac{\vartheta_2(t,q)}{\vartheta_4(t,q)} \\ \operatorname{dn}(u,m) &= \frac{\vartheta_4(0,q)}{\vartheta_3(0,q)}\frac{\vartheta_3(t,q)}{\vartheta_4(t,q)},\end{aligned}$$

or more generally computes a ratio of two such functions. Here $t = u/\vartheta_3(0,q)^2$, and $q = q(m)$ denotes the nome (see `nome()`). Optionally, you can specify the nome directly instead of m by passing `q=<value>`, or you can directly specify the elliptic parameter k with `k=<value>`.

The first argument should be a two-character string specifying the function using any combination of `'s'`, `'c'`, `'d'`, `'n'`. These letters respectively denote the basic functions $\operatorname{sn}(u,m)$, $\operatorname{cn}(u,m)$, $\operatorname{dn}(u,m)$, and 1. The identifier specifies the ratio of two such functions. For example, `'ns'` identifies the function

$$\operatorname{ns}(u,m) = \frac{1}{\operatorname{sn}(u,m)}$$

and `'cd'` identifies the function

$$\mathrm{cd}(u,m) = \frac{\mathrm{cn}(u,m)}{\mathrm{dn}(u,m)}.$$

If called with only the first argument, a function object evaluating the chosen function for given arguments is returned.

Examples

Basic evaluation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipfun('cd', 3.5, 0.5)
-0.9891101840595543931308394
>>> ellipfun('cd', 3.5, q=0.25)
0.07111979240214668158441418
```

The sn-function is doubly periodic in the complex plane with periods $4K(m)$ and $2iK(1-m)$ (see `ellipk()`):

```
>>> sn = ellipfun('sn')
>>> sn(2, 0.25)
0.9628981775982774425751399
>>> sn(2+4*ellipk(0.25), 0.25)
0.9628981775982774425751399
>>> chop(sn(2+2*j*ellipk(1-0.25), 0.25))
0.9628981775982774425751399
```

The cn-function is doubly periodic with periods $4K(m)$ and $4iK(1-m)$:

```
>>> cn = ellipfun('cn')
>>> cn(2, 0.25)
-0.2698649654510865792581416
>>> cn(2+4*ellipk(0.25), 0.25)
-0.2698649654510865792581416
>>> chop(cn(2+4*j*ellipk(1-0.25), 0.25))
-0.2698649654510865792581416
```

The dn-function is doubly periodic with periods $2K(m)$ and $4iK(1-m)$:

```
>>> dn = ellipfun('dn')
>>> dn(2, 0.25)
0.8764740583123262286931578
>>> dn(2+2*ellipk(0.25), 0.25)
0.8764740583123262286931578
>>> chop(dn(2+4*j*ellipk(1-0.25), 0.25))
0.8764740583123262286931578
```

Modular functions

kleinj()

mpmath.**kleinj**(*tau=None*, ***kwargs*)

Evaluates the Klein j-invariant, which is a modular function defined for τ in the upper half-plane as

$$J(\tau) = \frac{g_2^3(\tau)}{g_2^3(\tau) - 27g_3^2(\tau)}$$

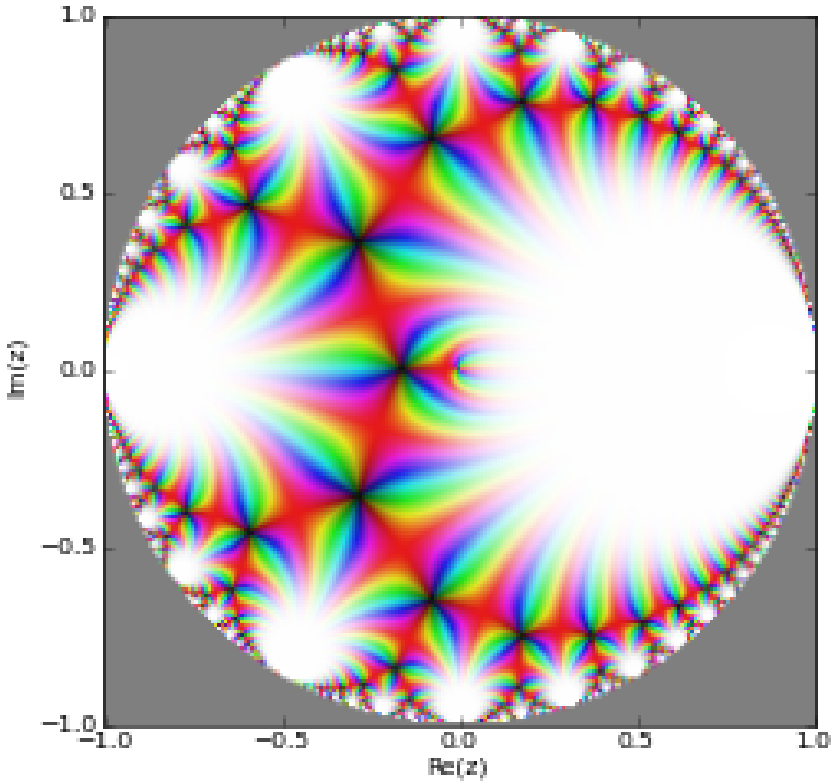
where g_2 and g_3 are the modular invariants of the Weierstrass elliptic function,

$$g_2(\tau) = 60 \sum_{(m,n)\in\mathbb{Z}^2\setminus(0,0)} (m\tau+n)^{-4}$$
$$g_3(\tau) = 140 \sum_{(m,n)\in\mathbb{Z}^2\setminus(0,0)} (m\tau+n)^{-6}.$$

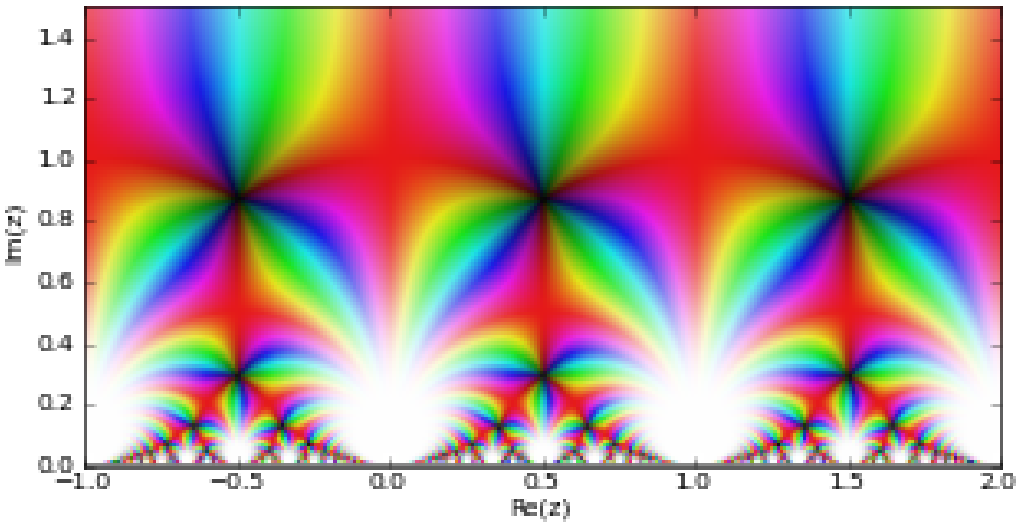
An alternative, common notation is that of the j-function $j(\tau) = 1728J(\tau)$.

Plots

```
# Klein J-function as function of the number-theoretic nome
fp.cplot(lambda q: fp.kleinj(qbar=q), [-1,1], [-1,1], points=50000)
```




```
# Klein J-function as function of the half-period ratio
fp.cplot(lambda t: fp.kleinj(tau=t), [-1,2], [0,1.5], points=50000)
```



Examples

Verifying the functional equation $J(\tau) = J(\tau+1) = J(-\tau^{-1})$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> tau = 0.625+0.75*j
>>> tau = 0.625+0.75*j
>>> kleinj(tau)
(-0.15074921665111822671252 42 + 0.075959483790845719272289 48j)
>>> kleinj(tau+1)
(-0.15074921665111822671252 42 + 0.075959483790845719272289 48j)
>>> kleinj(-1/tau)
(-0.15074921665111822671252 42 + 0.075959483790845719272289 46j)
```

The j-function has a famous Laurent series expansion in terms of the nome $\bar{q}$, $j(\tau) = \bar{q}^{-1} + 744 + 196884\bar{q} + \ldots$:

```
>>> mp.dps = 15
>>> taylor(lambda q: 1728*q*kleinj(qbar=q), 0, 5, singular=True)
[1.0, 744.0, 196884.0, 21493760.0, 864299970.0, 20245856256.0]
```

The j-function admits exact evaluation at special algebraic points related to the Heegner numbers 1, 2, 3, 7, 11, 19, 43, 67, 163:

```
>>> @extraprec(10)
... def h(n):
...     v = (1+sqrt(n)*j)
...     if n > 2:
...         v *= 0.5
```

```
...     return v
...
>>> mp.dps = 25
>>> for n in [1,2,3,7,11,19,43,67,163]:
...     n, chop(1728*kleinj(h(n)))
...
(1, 1728.0)
(2, 8000.0)
(3, 0.0)
(7, -3375.0)
(11, -32768.0)
(19, -884736.0)
(43, -884736000.0)
(67, -147197952000.0)
(163, -262537412640768000.0)
```

Also at other special points, the j-function assumes explicit algebraic values, e.g.:

```
>>> chop(1728*kleinj(j*sqrt(5)))
1264538.909475140509320227
>>> identify(cbrt(_))      # note: not simplified
'((100+sqrt(13520))/2)'
>>> (50+26*sqrt(5))**3
1264538.909475140509320227
```

3.1.11 Zeta functions, L-series and polylogarithms

This section includes the Riemann zeta functions and associated functions pertaining to analytic number theory.

Riemann and Hurwitz zeta functions

zeta()

mpmath.**zeta**(*s*, *a=1*, *derivative=0*)

Computes the Riemann zeta function

$$\zeta(s) = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \ldots$$

or, with $a \neq 1$, the more general Hurwitz zeta function

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

Optionally, `zeta(s, a, n)` computes the n-th derivative with respect to s,

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

Although these series only converge for $\Re(s) > 1$, the Riemann and Hurwitz zeta functions are defined through analytic continuation for arbitrary complex $s \neq 1$ ($s = 1$ is a pole).

The implementation uses three algorithms: the Borwein algorithm for the Riemann zeta function when s is close to the real line; the Riemann-Siegel formula for the Riemann zeta function when s is large imaginary, and Euler-Maclaurin summation in all other cases. The reflection formula for $\Re(s) < 0$ is implemented in some cases.

The algorithm can be chosen with `method = 'borwein'`, `method='riemann-siegel'` or `method = 'euler-maclaurin'`.

The parameter a is usually a rational number $a = p/q$, and may be specified as such by passing an integer tuple (p, q). Evaluation is supported for arbitrary complex a, but may be slow and/or inaccurate when $\Re(s) < 0$ for nonrational a or when computing derivatives.

Examples

Some values of the Riemann zeta function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> zeta(2); pi**2 / 6
1.6449340668482264364724152
1.6449340668482264364724152
>>> zeta(0)
-0.5
>>> zeta(-1)
-0.083333333333333333333333333
>>> zeta(-2)
0.0
```

For large positive s, $\zeta(s)$ rapidly approaches 1:

```
>>> zeta(50)
1.0000000000000008881784210
>>> zeta(100)
1.0
>>> zeta(inf)
1.0
>>> 1-sum((zeta(k)-1)/k for k in range(2,85)); +euler
0.57721566490153286060651209
0.57721566490153286060651209
>>> nsum(lambda k: zeta(k)-1, [2, inf])
1.0
```

Evaluation is supported for complex s and a:

```
>>> zeta(-3+4j)
(-0.033730573388277570675846980 + 0.27744992515570937452976770j)
>>> zeta(2+3j, -1+j)
(389.68412301408428163707410 + 295.26746101503053340259620j)
```

The Riemann zeta function has so-called nontrivial zeros on the critical line $s = 1/2 + it$:

```
>>> findroot(zeta, 0.5+14j); zetazero(1)
(0.5 + 14.134725141734693790457251j)
(0.5 + 14.134725141734693790457251j)
>>> findroot(zeta, 0.5+21j); zetazero(2)
(0.5 + 21.022039638771554992628479j)
(0.5 + 21.022039638771554992628479j)
>>> findroot(zeta, 0.5+25j); zetazero(3)
(0.5 + 25.010857580145688763213790j)
(0.5 + 25.010857580145688763213790j)
>>> chop(zeta(zetazero(10)))
0.0
```

Evaluation on and near the critical line is supported for large heights t by means of the Riemann-Siegel formula (currently for $a = 1, n \le 4$):

```
>>> zeta(0.5+100000j)
(1.073032014857753132114076 + 5.780848544363503984261041j)
>>> zeta(0.75+1000000j)
(0.95353160583751450203515 59 + 0.952594589483427306017 5651j)
>>> zeta(0.5+10000000j)
(11.4580406105770925450022 7 - 8.643437226836021723818215j)
>>> zeta(0.5+100000000j, derivative=1)
(51.12433106710194942681869 + 43.872211678723045205994 18j)
>>> zeta(0.5+100000000j, derivative=2)
(-444.2760822795430400549229 - 896.3789978119185981665403j)
>>> zeta(0.5+100000000j, derivative=3)
(3230.72682687670422215339 + 14374.369500736158976167 81j)
>>> zeta(0.5+100000000j, derivative=4)
(-11967.35573095046402130602 - 218945.78177892628392661 48j)
>>> zeta(1+10000000j)    # off the line
(2.85984648333253033700888 2 + 0.491808047480981808903986j)
>>> zeta(1+10000000j, derivative=1)
(-4.33383549467964791567320 5 - 0.084053379626029336360961 03j)
>>> zeta(1+10000000j, derivative=4)
(453.2764822702057701894278 - 581.9636258327681891409 95j)
```

For investigation of the zeta function zeros, the Riemann-Siegel Z-function is often more convenient than working with the Riemann zeta function directly (see *siegelz()*).

Some values of the Hurwitz zeta function:

```
>>> zeta(2, 3); -5./4 + pi**2/6
0.39493406684822643647241 52
0.39493406684822643647241 52
>>> zeta(2, (3,4)); pi**2 - 8*catalan
2.54187964767160649839766 3
2.54187964767160649839766 3
```

For positive integer values of s, the Hurwitz zeta function is equivalent to a polygamma function (except for a normalizing factor):

```
>>> zeta(4, (1,5)); psi(3, '1/5')/6
625.5408324774542966919938
625.5408324774542966919938
```

Evaluation of derivatives:

```
>>> zeta(0, 3+4j, 1); loggamma(3+4j) - ln(2*pi)/2
(-2.675565317808456852310934 + 4.742664438034657928194889j)
(-2.675565317808456852310934 + 4.742664438034657928194889j)
>>> zeta(2, 1, 20)
2432902008176640000.000242
>>> zeta(3+4j, 5.5+2j, 4)
(-0.140075548947797130681075 - 0.310926336027541325131363 4j)
>>> zeta(0.5+100000j, 1, 4)
(-10407.16081931495861539236 + 13777.786698628045085373 84j)
>>> zeta(-100+0.5j, (1,3), derivative=4)
(4.00718082109982394270224 9e+79 + 4.916117957092593868321778e+78j)
```

Generating a Taylor series at $s = 2$ using derivatives:

```
>>> for k in range(11): print("%s * (s-2)^%i" % (zeta(2,1,k)/fac(k), k))
...
1.64493406684822643647241 5 * (s-2)^0
-0.93754825431584375370257 41 * (s-2)^1
```

```
    0.99464011714945051171042 93 * (s-2)^2
    -1.00002430047384081094065 7 * (s-2)^3
    1.00006193307235256545751 2 * (s-2)^4
    -1.00000686944393180640894 1 * (s-2)^5
    1.00000017323376953182059 2 * (s-2)^6
    -0.99999995699898684934323 99 * (s-2)^7
    0.99999999372188445086842 06 * (s-2)^8
    -0.99999999963550139166082 84 * (s-2)^9
    1.00000000000461064502074 7 * (s-2)^10
```

Evaluation at zero and for negative integer s:

```
    >>> zeta(0, 10)
    -9.5
    >>> zeta(-2, (2,3)); mpf(1)/81
    0.012345679012345679012345 68
    0.012345679012345679012345 68
    >>> zeta(-3+4j, (5,4))
    (0.28992360376826951820859 88 + 0.06561206166091757973112783j)
    >>> zeta(-3.25, 1/pi)
    -0.00051172696275744304944 96877
    >>> zeta(-3.5, pi, 1)
    11.156360390440003294709
    >>> zeta(-100.5, (8,3))
    -4.68162300487989766727122e+77
    >>> zeta(-10.5, (-8,3))
    (-0.01521913704446246609237979 + 29907.725108742481616082 16j)
    >>> zeta(-1000.5, (-8,3))
    (1.03191194906233453820256 7e+1770 + 1.51955575055679421880472 4e+426j)
    >>> zeta(-1+j, 3+4j)
    (-16.32988355630802510888631 - 22.17706465801374033261383j)
    >>> zeta(-1+j, 3+4j, 2)
    (32.489852763920566415940 55 - 51.116044661573972670436 55j)
    >>> diff(lambda s: zeta(s, 3+4j), -1+j, 2)
    (32.489852763920566415940 55 - 51.116044661573972670436 55j)
```

References

1. http://mathworld.wolfram.com/RiemannZetaFunction.html
2. http://mathworld.wolfram.com/HurwitzZetaFunction.html
3. http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P155.pdf

Dirichlet L-series

altzeta()

mpmath.**altzeta**(*s*)

Gives the Dirichlet eta function, $\eta(s)$, also known as the alternating zeta function. This function is defined in analogy with the Riemann zeta function as providing the sum of the alternating series

$$\eta(s) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k^s} = 1 - \frac{1}{2^s} + \frac{1}{3^s} - \frac{1}{4^s} + \ldots$$

The eta function, unlike the Riemann zeta function, is an entire function, having a finite value for all complex s. The special case $\eta(1) = \log(2)$ gives the value of the alternating harmonic series.

The alternating zeta function may expressed using the Riemann zeta function as $\eta(s) = (1 - 2^{1-s})\zeta(s)$. It can also be expressed in terms of the Hurwitz zeta function, for example using *dirichlet()* (see documentation for that function).

Examples

Some special values are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> altzeta(1)
0.693147180559945
>>> altzeta(0)
0.5
>>> altzeta(-1)
0.25
>>> altzeta(-2)
0.0
```

An example of a sum that can be computed more accurately and efficiently via *altzeta()* than via numerical summation:

```
>>> sum(-(-1)**n / mpf(n)**2.5 for n in range(1, 100))
0.867204951503984
>>> altzeta(2.5)
0.867199889012184
```

At positive even integers, the Dirichlet eta function evaluates to a rational multiple of a power of π:

```
>>> altzeta(2)
0.822467033424113
>>> pi**2/12
0.822467033424113
```

Like the Riemann zeta function, $\eta(s)$, approaches 1 as s approaches positive infinity, although it does so from below rather than from above:

```
>>> altzeta(30)
0.999999999068682
>>> altzeta(inf)
1.0
>>> mp.pretty = False
>>> altzeta(1000, rounding='d')
mpf('0.99999999999999989')
>>> altzeta(1000, rounding='u')
mpf('1.0')
```

References

1. http://mathworld.wolfram.com/DirichletEtaFunction.html

2. http://en.wikipedia.org/wiki/Dirichlet_eta_function

dirichlet()

mpmath.**dirichlet**(*s*, *chi*, *derivative=0*)

Evaluates the Dirichlet L-function

$$L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s}.$$

where χ is a periodic sequence of length q which should be supplied in the form of a list $[\chi(0), \chi(1), \ldots, \chi(q-1)]$. Strictly, χ should be a Dirichlet character, but any periodic sequence will work.

For example, `dirichlet(s, [1])` gives the ordinary Riemann zeta function and `dirichlet(s, [-1,1])` gives the alternating zeta function (Dirichlet eta function).

Also the derivative with respect to s (currently only a first derivative) can be evaluated.

Examples

The ordinary Riemann zeta function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> dirichlet(3, [1]); zeta(3)
1.2020569031595942853997381
1.2020569031595942853997381
>>> dirichlet(1, [1])
+inf
```

The alternating zeta function:

```
>>> dirichlet(1, [-1,1]); ln(2)
0.6931471805599453094172321
0.6931471805599453094172321
```

The following defines the Dirichlet beta function $\beta(s) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^s}$ and verifies several values of this function:

```
>>> B = lambda s, d=0: dirichlet(s, [0, 1, 0, -1], d)
>>> B(0); 1./2
0.5
0.5
>>> B(1); pi/4
0.78539816339744830961566091
0.78539816339744830961566091
>>> B(2); +catalan
0.91596559417721901505460351
0.91596559417721901505460351
>>> B(2,1); diff(B, 2)
0.08158073611659279510291217
0.08158073611659279510291217
>>> B(-1,1); 2*catalan/pi
0.58312180806163756027676891
0.58312180806163756027676891
>>> B(0,1); log(gamma(0.25)**2/(2*pi*sqrt(2)))
0.39159439270683677647194530
0.39159439270683677647194541
>>> B(1,1); 0.25*pi*(euler+2*ln2+3*ln(pi)-4*ln(gamma(0.25)))
0.19290131679691242936318981
0.19290131679691242936318981
```

A custom L-series of period 3:

```
>>> dirichlet(2, [2,0,1])
0.70597150478390780921468310
>>> 2*nsum(lambda k: (3*k)**-2, [1,inf]) + \
...   nsum(lambda k: (3*k+2)**-2, [0,inf])
0.70597150478390780921468310
```

Stieltjes constants

stieltjes()

mpmath.**stieltjes**(*n*, *a=1*)

For a nonnegative integer n, `stieltjes(n)` computes the n-th Stieltjes constant γ_n, defined as the n-th coefficient in the Laurent series expansion of the Riemann zeta function around the pole at $s = 1$. That is, we have:

$$\zeta(s) = \frac{1}{s-1} \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n (s-1)^n$$

More generally, `stieltjes(n, a)` gives the corresponding coefficient $\gamma_n(a)$ for the Hurwitz zeta function $\zeta(s, a)$ (with $\gamma_n = \gamma_n(1)$).

Examples

The zeroth Stieltjes constant is just Euler's constant γ:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> stieltjes(0)
0.577215664901533
```

Some more values are:

```
>>> stieltjes(1)
-0.0728158454836767
>>> stieltjes(10)
0.000205332814909065
>>> stieltjes(30)
0.00355772885557316
>>> stieltjes(1000)
-1.57095384420474e+486
>>> stieltjes(2000)
2.680424678918e+1109
>>> stieltjes(1, 2.5)
-0.23747539175716
```

An alternative way to compute γ_1:

```
>>> diff(extradps(15)(lambda x: 1/(x-1) - zeta(x)), 1)
-0.0728158454836767
```

stieltjes() supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> stieltjes(2)
-0.0096903631928723184845303860352125293590658061013408
```

Algorithm

stieltjes() numerically evaluates the integral in the following representation due to Ainsworth, Howell and Coffey [1], [2]:

$$\gamma_n(a) = \frac{\log^n a}{2a} - \frac{\log^{n+1}(a)}{n+1} + \frac{2}{a} \Re \int_0^{\infty} \frac{(x/a - i)\log^n(a - ix)}{(1 + x^2/a^2)(e^{2\pi x} - 1)} dx.$$

For some reference values with $a = 1$, see e.g. [4].

References

1. O. R. Ainsworth & L. W. Howell, "An integral representation of the generalized Euler-Mascheroni constants", NASA Technical Paper 2456 (1985), http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19850014994_1985014994.pdf

2. M. W. Coffey, "The Stieltjes constants, their relation to the η_j coefficients, and representation of the Hurwitz zeta function", arXiv:0706.0343v1 http://arxiv.org/abs/0706.0343

3. http://mathworld.wolfram.com/StieltjesConstants.html

4. http://pi.lacim.uqam.ca/piDATA/stieltjesgamma.txt

Zeta function zeros

These functions are used for the study of the Riemann zeta function in the critical strip.

zetazero()

mpmath.**zetazero**(*n*, *verbose=False*)

Computes the n-th nontrivial zero of $\zeta(s)$ on the critical line, i.e. returns an approximation of the n-th largest complex number $s = \frac{1}{2} + ti$ for which $\zeta(s) = 0$. Equivalently, the imaginary part t is a zero of the Z-function (*siegelz()*).

Examples

The first few zeros:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> zetazero(1)
(0.5 + 14.134725141734693790457251j)
>>> zetazero(2)
(0.5 + 21.022039638771554992628479j)
>>> zetazero(20)
(0.5 + 77.144840068874805372682664j)
```

Verifying that the values are zeros:

```
>>> for n in range(1,5):
...     s = zetazero(n)
...     chop(zeta(s)), chop(siegelz(s.imag))
...
(0.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
```

Negative indices give the conjugate zeros ($n = 0$ is undefined):

```
>>> zetazero(-1)
(0.5 - 14.134725141734693790457251j)
```

zetazero() supports arbitrarily large n and arbitrary precision:

```
>>> mp.dps = 15
>>> zetazero(1234567)
(0.5 + 727690.906948208j)
>>> mp.dps = 50
>>> zetazero(1234567)
```

```
(0.5 + 727690.906948207539238942004114714209270839381993 5j)
>>> chop(zeta(_)/_)
0.0
```

with *info=True*, `zetazero()` gives additional information:

```
>>> mp.dps = 15
>>> zetazero(542964976,info=True)
((0.5 + 209039046.578535j), [542964969, 542964978], 6, '(013111110)')
```

This means that the zero is between Gram points 542964969 and 542964978; it is the 6-th zero between them. Finally (01311110) is the pattern of zeros in this interval. The numbers indicate the number of zeros in each Gram interval (Rosser blocks between parenthesis). In this case there is only one Rosser block of length nine.

nzeros()

mpmath.**nzeros**(*t*)

Computes the number of zeros of the Riemann zeta function in $(0,1) \times (0,t]$, usually denoted by $N(t)$.

Examples

The first zero has imaginary part between 14 and 15:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nzeros(14)
0
>>> nzeros(15)
1
>>> zetazero(1)
(0.5 + 14.1347251417347j)
```

Some closely spaced zeros:

```
>>> nzeros(10**7)
21136125
>>> zetazero(21136125)
(0.5 + 9999999.32718175j)
>>> zetazero(21136126)
(0.5 + 10000000.2400236j)
>>> nzeros(545439823.215)
1500000001
>>> zetazero(1500000001)
(0.5 + 545439823.201985j)
>>> zetazero(1500000002)
(0.5 + 545439823.325697j)
```

This confirms the data given by J. van de Lune, H. J. J. te Riele and D. T. Winter in 1986.

siegelz()

mpmath.**siegelz**(*t*)

Computes the Z-function, also known as the Riemann-Siegel Z function,

$$Z(t) = e^{i\theta(t)}\zeta(1/2+it)$$

where $\zeta(s)$ is the Riemann zeta function (`zeta()`) and where $\theta(t)$ denotes the Riemann-Siegel theta function (see `siegeltheta()`).

Evaluation is supported for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> siegelz(1)
-0.7363054628673177346778998
>>> siegelz(3+4j)
(-0.185289576436631497600393 6 - 0.27730991980555622469924 79j)
```

The first four derivatives are supported, using the optional *derivative* keyword argument:

```
>>> siegelz(1234567, derivative=3)
56.89689348495089294249178
>>> diff(siegelz, 1234567, n=3)
56.89689348495089294249178
```

The Z-function has a Maclaurin expansion:

```
>>> nprint(chop(taylor(siegelz, 0, 4)))
[-1.46035, 0.0, 2.73588, 0.0, -8.39357]
```

The Z-function $Z(t)$ is equal to $\pm|\zeta(s)|$ on the critical line $s = 1/2 + it$ (i.e. for real arguments t to Z). Its zeros coincide with those of the Riemann zeta function:

```
>>> findroot(siegelz, 14)
14.134725141734693790457 25
>>> findroot(siegelz, 20)
21.022039638771554992628 48
>>> findroot(zeta, 0.5+14j)
(0.5 + 14.134725141734693790457 25j)
>>> findroot(zeta, 0.5+20j)
(0.5 + 21.022039638771554992628 48j)
```

Since the Z-function is real-valued on the critical line (and unlike $|\zeta(s)|$ analytic), it is useful for investigating the zeros of the Riemann zeta function. For example, one can use a root-finding algorithm based on sign changes:

```
>>> findroot(siegelz, [100, 200], solver='bisect')
176.44143429771041888889 26
```

To locate roots, Gram points g_n which can be computed by *grampoint()* are useful. If $(-1)^n Z(g_n)$ is positive for two consecutive n, then $Z(t)$ must have a zero between those points:

```
>>> g10 = grampoint(10)
>>> g11 = grampoint(11)
>>> (-1)**10 * siegelz(g10) > 0
True
>>> (-1)**11 * siegelz(g11) > 0
True
>>> findroot(siegelz, [g10, g11], solver='bisect')
56.446247697063394804 36776
>>> g10, g11
(54.675237446853256266 32663, 57.545165179547254437 03014)
```

siegeltheta()

mpmath.**siegeltheta**(*t*)

Computes the Riemann-Siegel theta function,

$$\theta(t) = \frac{\log\Gamma\left(\frac{1+2it}{4}\right) - \log\Gamma\left(\frac{1-2it}{4}\right)}{2i} - \frac{\log\pi}{2}t.$$

The Riemann-Siegel theta function is important in providing the phase factor for the Z-function (see *siegelz()*). Evaluation is supported for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> siegeltheta(0)
0.0
>>> siegeltheta(inf)
+inf
>>> siegeltheta(-inf)
-inf
>>> siegeltheta(1)
-1.767547952812290388302216
>>> siegeltheta(10+0.25j)
(-3.068638039426838572528867 + 0.05804937947429712998395177j)
```

Arbitrary derivatives may be computed with derivative = k

```
>>> siegeltheta(1234, derivative=2)
0.0004051864079114053109473741
>>> diff(siegeltheta, 1234, n=2)
0.0004051864079114053109473741
```

The Riemann-Siegel theta function has odd symmetry around $t = 0$, two local extreme points and three real roots including 0 (located symmetrically):

```
>>> nprint(chop(taylor(siegeltheta, 0, 5)))
[0.0, -2.68609, 0.0, 2.69433, 0.0, -6.40218]
>>> findroot(diffun(siegeltheta), 7)
6.289835988836902779665 09
>>> findroot(siegeltheta, 20)
17.845599540410860816826 34
```

For large t, there is a famous asymptotic formula for $\theta(t)$, to first order given by:

```
>>> t = mpf(10**6)
>>> siegeltheta(t)
5488816.353078403444882823
>>> -t*log(2*pi/t)/2-t/2
5488816.745777464310273645
```

grampoint()

mpmath.**grampoint**(*n*)

Gives the n-th Gram point g_n, defined as the solution to the equation $\theta(g_n) = \pi n$ where $\theta(t)$ is the Riemann-Siegel theta function (*siegeltheta()*).

The first few Gram points are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> grampoint(0)
17.845599540410860816826340
>>> grampoint(1)
23.170282701246309278996640
>>> grampoint(2)
27.670182217816337960938490
>>> grampoint(3)
31.717979954764053179551490
```

Checking the definition:

```
>>> siegeltheta(grampoint(3))
9.4247779607693797153879301
>>> 3*pi
9.4247779607693797153879301
```

A large Gram point:

```
>>> grampoint(10**10)
3293531632.7283354545611530
```

Gram points are useful when studying the Z-function (*siegelz()*). See the documentation of that function for additional examples.

grampoint() can solve the defining equation for nonintegral n. There is a fixed point where $g(x) = x$:

```
>>> findroot(lambda x: grampoint(x) - x, 10000)
9146.6981931714592658661980
```

References

1. http://mathworld.wolfram.com/GramPoint.html

backlunds()

mpmath.**backlunds**(t)

Computes the function $S(t) = \arg \zeta(\frac{1}{2} + it)/\pi$.

See Titchmarsh Section 9.3 for details of the definition.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> backlunds(217.3)
0.16302205431184
```

Generally, the value is a small number. At Gram points it is an integer, frequently equal to 0:

```
>>> chop(backlunds(grampoint(200)))
0.0
>>> backlunds(extraprec(10)(grampoint)(211))
1.0
>>> backlunds(extraprec(10)(grampoint)(232))
-1.0
```

The number of zeros of the Riemann zeta function up to height t satisfies $N(t) = \theta(t)/\pi + 1 + S(t)$ (see :func:nzeros‘ and `siegeltheta()`):

```
>>> t = 1234.55
>>> nzeros(t)
842
>>> siegeltheta(t)/pi+1+backlunds(t)
842.0
```

Lerch transcendent

lerchphi()

mpmath.**lerchphi**(z, s, a)

Gives the Lerch transcendent, defined for $|z| < 1$ and $\Re a > 0$ by

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

and generally by the recurrence $\Phi(z,s,a) = z\Phi(z,s,a+1) + a^{-s}$ along with the integral representation valid for $\Re a > 0$

$$\Phi(z,s,a) = \frac{1}{2a^s} + \int_0^\infty \frac{z^t}{(a+t)^s} dt - 2\int_0^\infty \frac{\sin(t\log z - s\arctan(t/a)}{(a^2+t^2)^{s/2}(e^{2\pi t}-1)} dt.$$

The Lerch transcendent generalizes the Hurwitz zeta function `zeta()` ($z = 1$) and the polylogarithm `polylog()` ($a = 1$).

Examples

Several evaluations in terms of simpler functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> lerchphi(-1,2,0.5); 4*catalan
3.66386237670887606021841 4
3.66386237670887606021841 4
>>> diff(lerchphi, (-1,-2,1), (0,1,0)); 7*zeta(3)/(4*pi**2)
0.21313919940875289546176 07
0.21313919940875289546176 07
>>> lerchphi(-4,1,1); log(5)/4
0.40235947810852509365018 98
0.40235947810852509365018 98
>>> lerchphi(-3+2j,1,0.5); 2*atanh(sqrt(-3+2j))/sqrt(-3+2j)
(1.1424234471202571377740 02 + 0.2118232380980201350495795j)
(1.1424234471202571377740 02 + 0.2118232380980201350495795j)
```

Evaluation works for complex arguments and $|z| \geq 1$:

```
>>> lerchphi(1+2j, 3-j, 4+2j)
(0.00202500995700990860053 9469 + 0.00332789753681355880743 8089j)
>>> lerchphi(-2,2,-2.5)
-12.28676272353094275265944
>>> lerchphi(10,10,10)
(-4.46213072710218570181734 9e-11 + 1.57517219898109621882348 1e-12j)
>>> lerchphi(10,10,-10.5)
(112658784011940.5605789002 + 498113185.5756221777743631j)
```

Some degenerate cases:

```
>>> lerchphi(0,1,2)
0.5
>>> lerchphi(0,1,-2)
-0.5
```

Reduction to simpler functions:

```
>>> lerchphi(1, 4.25+1j, 1)
(1.044674457556746668033975 - 0.046745086540126589322712 26j)
>>> zeta(4.25+1j)
(1.044674457556746668033975 - 0.046745086540126589322712 26j)
>>> lerchphi(1 - 0.5**10, 4.25+1j, 1)
(1.044629338021507546737197 - 0.046677688139633881817081 01j)
>>> lerchphi(3, 4, 1)
(1.249503297023366545192592 - 0.231425241337566477647446 2j)
>>> polylog(4, 3) / 3
(1.249503297023366545192592 - 0.231425241337566477647446 2j)
>>> lerchphi(3, 4, 1 - 0.5**10)
(1.253978063946663945672674 + 0.231673662283653546876537 6j)
```

References

1. *[DLMF]* section 25.14

Polylogarithms and Clausen functions

polylog()

mpmath.**polylog**(*s*, *z*)

Computes the polylogarithm, defined by the sum

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}.$$

This series is convergent only for $|z| < 1$, so elsewhere the analytic continuation is implied.

The polylogarithm should not be confused with the logarithmic integral (also denoted by Li or li), which is implemented as *li()*.

Examples

The polylogarithm satisfies a huge number of functional identities. A sample of polylogarithm evaluations is shown below:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> polylog(1,0.5), log(2)
(0.693147180559945, 0.693147180559945)
>>> polylog(2,0.5), (pi**2-6*log(2)**2)/12
(0.582240526465012, 0.582240526465012)
>>> polylog(2,-phi), -log(phi)**2-pi**2/10
(-1.21852526068613, -1.21852526068613)
>>> polylog(3,0.5), 7*zeta(3)/8-pi**2*log(2)/12+log(2)**3/6
(0.53721319360804, 0.53721319360804)
```

polylog() can evaluate the analytic continuation of the polylogarithm when s is an integer:

```
>>> polylog(2, 10)
(0.536301287357863 - 7.23378441241546j)
>>> polylog(2, -10)
-4.1982778868581
>>> polylog(2, 10j)
(-3.05968879432873 + 3.71678149306807j)
>>> polylog(-2, 10)
-0.150891632373114
>>> polylog(-2, -10)
0.067618332081142
>>> polylog(-2, 10j)
(0.0384353698579347 + 0.0912451798066779j)
```

Some more examples, with arguments on the unit circle (note that the series definition cannot be used for computation here):

```
>>> polylog(2,j)
(-0.205616758356028 + 0.915965594177219j)
>>> j*catalan-pi**2/48
(-0.205616758356028 + 0.915965594177219j)
>>> polylog(3,exp(2*pi*j/3))
(-0.534247512515375 + 0.765587078525922j)
>>> -4*zeta(3)/9 + 2*j*pi**3/81
(-0.534247512515375 + 0.765587078525921j)
```

Polylogarithms of different order are related by integration and differentiation:

```
>>> s, z = 3, 0.5
>>> polylog(s+1, z)
0.517479061673899
>>> quad(lambda t: polylog(s,t)/t, [0, z])
0.517479061673899
>>> z*diff(lambda t: polylog(s+2,t), z)
0.517479061673899
```

Taylor series expansions around $z = 0$ are:

```
>>> for n in range(-3, 4):
...     nprint(taylor(lambda x: polylog(n,x), 0, 5))
...
[0.0, 1.0, 8.0, 27.0, 64.0, 125.0]
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0.0, 1.0, 0.5, 0.333333, 0.25, 0.2]
[0.0, 1.0, 0.25, 0.111111, 0.0625, 0.04]
[0.0, 1.0, 0.125, 0.037037, 0.015625, 0.008]
```

The series defining the polylogarithm is simultaneously a Taylor series and an L-series. For certain values of z, the polylogarithm reduces to a pure zeta function:

```
>>> polylog(pi, 1), zeta(pi)
(1.17624173838258, 1.17624173838258)
>>> polylog(pi, -1), -altzeta(pi)
(-0.909670702980385, -0.909670702980385)
```

Evaluation for arbitrary, nonintegral s is supported for z within the unit circle:

```
>>> polylog(3+4j, 0.25)
(0.24258605789446 - 0.00222938275488344j)
>>> nsum(lambda k: 0.25**k / k**(3+4j), [1,inf])
(0.24258605789446 - 0.00222938275488344j)
```

It is also currently supported outside of the unit circle for z not too large in magnitude:

```
>>> polylog(1+j, 20+40j)
(-7.1421172179728 - 3.927266977213 69j)
>>> polylog(1+j, 200+400j)
Traceback (most recent call last):
  ...
NotImplementedError: polylog for arbitrary s and z
```

References

1. Richard Crandall, "Note on fast polylogarithm computation" http://people.reed.edu/~crandall/papers/Polylog.pdf
2. http://en.wikipedia.org/wiki/Polylogarithm
3. http://mathworld.wolfram.com/Polylogarithm.html

clsin()

mpmath.**clsin**(s, z)

Computes the Clausen sine function, defined formally by the series

$$\mathrm{Cl}_s(z) = \sum_{k=1}^{\infty} \frac{\sin(kz)}{k^s}.$$

The special case $\mathrm{Cl}_2(z)$ (i.e. `clsin(2,z)`) is the classical "Clausen function". More generally, the Clausen function is defined for complex s and z, even when the series does not converge. The Clausen function is related to the polylogarithm (*polylog()*) as

$$\begin{aligned}\mathrm{Cl}_s(z) &= \frac{1}{2i}\left(\mathrm{Li}_s\left(e^{iz}\right) - \mathrm{Li}_s\left(e^{-iz}\right)\right) \\ &= \mathrm{Im}\left[\mathrm{Li}_s(e^{iz})\right] \quad (s, z \in \mathbb{R}),\end{aligned}$$

and this representation can be taken to provide the analytic continuation of the series. The complementary function *clcos()* gives the corresponding cosine sum.

Examples

Evaluation for arbitrarily chosen s and z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> s, z = 3, 4
>>> clsin(s, z); nsum(lambda k: sin(z*k)/k**s, [1,inf])
-0.6533010136329338746275795
-0.6533010136329338746275795
```

Using $z + \pi$ instead of z gives an alternating series:

```
>>> clsin(s, z+pi)
0.8860032351260589402871624
>>> nsum(lambda k: (-1)**k*sin(z*k)/k**s, [1,inf])
0.8860032351260589402871624
```

With $s = 1$, the sum can be expressed in closed form using elementary functions:

```
>>> z = 1 + sqrt(3)
>>> clsin(1, z)
0.20477092301045797246759 85
>>> chop((log(1-exp(-j*z)) - log(1-exp(j*z)))/(2*j))
0.20477092301045797246759 85
>>> nsum(lambda k: sin(k*z)/k, [1,inf])
0.20477092301045797246759 85
```

The classical Clausen function $\mathrm{Cl}_2(\theta)$ gives the value of the integral $\int_0^\theta -\ln(2\sin(x/2))dx$ for $0 < \theta < 2\pi$:

```
>>> cl2 = lambda t: clsin(2, t)
>>> cl2(3.5)
-0.24650453023476942165342 55
>>> -quad(lambda x: ln(2*sin(0.5*x)), [0, 3.5])
-0.24650453023476942165342 55
```

This function is symmetric about $\theta = \pi$ with zeros and extreme points:

```
>>> cl2(0); cl2(pi/3); chop(cl2(pi)); cl2(5*pi/3); chop(cl2(2*pi))
0.0
1.0149416064096536250212 03
0.0
-1.0149416064096536250212 03
0.0
```

Catalan's constant is a special value:

```
>>> cl2(pi/2)
0.91596559417721901505460 35
>>> +catalan
0.91596559417721901505460 35
```

The Clausen sine function can be expressed in closed form when s is an odd integer (becoming zero when $s < 0$):

```
>>> z = 1 + sqrt(2)
>>> clsin(1, z); (pi-z)/2
0.36368954560834909483047 73
0.36368954560834909483047 73
>>> clsin(3, z); pi**2/6*z - pi*z**2/4 + z**3/12
0.56617515844511449917071 61
0.56617515844511449917071 61
>>> clsin(-1, z)
0.0
>>> clsin(-3, z)
0.0
```

It can also be expressed in closed form for even integer $s \leq 0$, providing a finite sum for series such as $\sin(z) + \sin(2z) + \sin(3z) + \ldots$:

```
>>> z = 1 + sqrt(2)
>>> clsin(0, z)
0.19031050295075138812758 65
>>> cot(z/2)/2
0.19031050295075138812758 65
>>> clsin(-2, z)
-0.10894061638415488175813 92
>>> -cot(z/2)*csc(z/2)**2/4
-0.10894061638415488175813 92
```

Call with `pi=True` to multiply z by π exactly:

```
>>> clsin(3, 3*pi)
-8.8923162249680724247328980e-26
>>> clsin(3, 3, pi=True)
0.0
```

Evaluation for complex s, z in a nonconvergent case:

```
>>> s, z = -1-j, 1+2j
>>> clsin(s, z)
(-0.59307948011737900251603 4 + 0.90386442333678682733624 46j)
>>> extraprec(20)(nsum)(lambda k: sin(k*z)/k**s, [1,inf])
(-0.59307948011737900251603 4 + 0.90386442333678682733624 46j)
```

clcos()

mpmath.**clcos**(s, z)

Computes the Clausen cosine function, defined formally by the series

$$\widetilde{\mathrm{Cl}}_s(z) = \sum_{k=1}^{\infty} \frac{\cos(kz)}{k^s}.$$

This function is complementary to the Clausen sine function *clsin()*. In terms of the polylogarithm,

$$\begin{aligned}\widetilde{\mathrm{Cl}}_s(z) &= \frac{1}{2}\left(\mathrm{Li}_s\left(e^{iz}\right) + \mathrm{Li}_s\left(e^{-iz}\right)\right) \\ &= \mathrm{Re}\left[\mathrm{Li}_s(e^{iz})\right] \quad (s, z \in \mathbb{R}).\end{aligned}$$

Examples

Evaluation for arbitrarily chosen s and z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> s, z = 3, 4
>>> clcos(s, z); nsum(lambda k: cos(z*k)/k**s, [1,inf])
-0.6518926267198991308332759
-0.6518926267198991308332759
```

Using $z + \pi$ instead of z gives an alternating series:

```
>>> s, z = 3, 0.5
>>> clcos(s, z+pi)
-0.8155530586502260817855618
>>> nsum(lambda k: (-1)**k*cos(z*k)/k**s, [1,inf])
-0.8155530586502260817855618
```

With $s = 1$, the sum can be expressed in closed form using elementary functions:

```
>>> z = 1 + sqrt(3)
>>> clcos(1, z)
-0.67203343733697148497979 18
>>> chop(-0.5*(log(1-exp(j*z))+log(1-exp(-j*z))))
-0.67203343733697148497979 18
>>> -log(abs(2*sin(0.5*z)))    # Equivalent to above when z is real
-0.67203343733697148497979 18
>>> nsum(lambda k: cos(k*z)/k, [1,inf])
-0.67203343733697148497979 18
```

It can also be expressed in closed form when s is an even integer. For example,

```
>>> clcos(2,z)
-0.78053590251355831188630 07
>>> pi**2/6 - pi*z/2 + z**2/4
-0.78053590251355831188630 07
```

The case $s = 0$ gives the renormalized sum of $\cos(z) + \cos(2z) + \cos(3z) + \ldots$ (which happens to be the same for any value of z):

```
>>> clcos(0, z)
-0.5
>>> nsum(lambda k: cos(k*z), [1,inf])
-0.5
```

Also the sums

$$\cos(z) + 2\cos(2z) + 3\cos(3z) + \ldots$$

and

$$\cos(z) + 2^n \cos(2z) + 3^n \cos(3z) + \ldots$$

for higher integer powers $n = -s$ can be done in closed form. They are zero when n is positive and even (s negative and even):

```
>>> clcos(-1, z); 1/(2*cos(z)-2)
-0.26078293752405424806941 26
-0.26078293752405424806941 26
>>> clcos(-3, z); (2+cos(z))*csc(z/2)**4/8
0.14726350549799443908480 06
0.14726350549799443908480 06
>>> clcos(-2, z); clcos(-4, z); clcos(-6, z)
0.0
0.0
0.0
```

With $z = \pi$, the series reduces to that of the Riemann zeta function (more generally, if $z = p\pi/q$, it is a finite sum over Hurwitz zeta function values):

```
>>> clcos(2.5, 0); zeta(2.5)
1.341487257250917179756 77
1.341487257250917179756 77
>>> clcos(2.5, pi); -altzeta(2.5)
-0.867199889012184138191 3472
-0.867199889012184138191 3472
```

Call with `pi=True` to multiply z by π exactly:

```
>>> clcos(-3, 2*pi)
2.997921055881167659267063e+102
>>> clcos(-3, 2, pi=True)
0.008333333333333333333333333333
```

Evaluation for complex s, z in a nonconvergent case:

```
>>> s, z = -1-j, 1+2j
>>> clcos(s, z)
(0.940743012156225147613 6807 + 0.715826296033590204557054j)
>>> extraprec(20)(nsum)(lambda k: cos(k*z)/k**s, [1,inf])
(0.940743012156225147613 6807 + 0.715826296033590204557054j)
```

polyexp()

mpmath.**polyexp**(*s*, *z*)

Evaluates the polyexponential function, defined for arbitrary complex s, z by the series

$$E_s(z) = \sum_{k=1}^{\infty} \frac{k^s}{k!} z^k.$$

$E_s(z)$ is constructed from the exponential function analogously to how the polylogarithm is constructed from the ordinary logarithm; as a function of s (with z fixed), E_s is an L-series It is an entire function of both s and z.

The polyexponential function provides a generalization of the Bell polynomials $B_n(x)$ (see *bell()*) to noninteger orders n. In terms of the Bell polynomials,

$$E_s(z) = e^z B_s(z) - \mathrm{sinc}(\pi s).$$

Note that $B_n(x)$ and $e^{-x} E_n(x)$ are identical if n is a nonzero integer, but not otherwise. In particular, they differ at $n = 0$.

Examples

Evaluating a series:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> nsum(lambda k: sqrt(k)/fac(k), [1,inf])
2.101755547733791780315904
>>> polyexp(0.5,1)
2.101755547733791780315904
```

Evaluation for arbitrary arguments:

```
>>> polyexp(-3-4j, 2.5+2j)
(2.351660261190434618268706 + 1.202966666673054671364215j)
```

Evaluation is accurate for tiny function values:

```
>>> polyexp(4, -100)
3.499471750566824369520223e-36
```

If n is a nonpositive integer, E_n reduces to a special instance of the hypergeometric function ${}_pF_q$:

```
>>> n = 3
>>> x = pi
>>> polyexp(-n,x)
4.042192318847986561771779
>>> x*hyper([1]*(n+1), [2]*(n+1), x)
4.042192318847986561771779
```

Zeta function variants

primezeta()

mpmath.**primezeta**(*s*)

Computes the prime zeta function, which is defined in analogy with the Riemann zeta function (*zeta()*) as

$$P(s) = \sum_p \frac{1}{p^s}$$

where the sum is taken over all prime numbers p. Although this sum only converges for $\mathrm{Re}(s) > 1$, the function is defined by analytic continuation in the half-plane $\mathrm{Re}(s) > 0$.

Examples

Arbitrary-precision evaluation for real and complex arguments is supported:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> primezeta(2)
0.452247420041065498506543364832
>>> primezeta(pi)
0.154837526988402842720364973 97
>>> mp.dps = 50
>>> primezeta(3)
0.17476263929944353642311331466570670097541212192615
>>> mp.dps = 20
>>> primezeta(3+4j)
(-0.12085382601645763295 - 0.013370403397787023602j)
```

The prime zeta function has a logarithmic pole at $s = 1$, with residue equal to the difference of the Mertens and Euler constants:

```
>>> primezeta(1)
+inf
>>> extradps(25)(lambda x: primezeta(1+x)+log(x))(+eps)
-0.31571845205389007685
>>> mertens-euler
-0.31571845205389007685
```

The analytic continuation to $0 < \mathrm{Re}(s) \le 1$ is implemented. In this strip the function exhibits very complex behavior; on the unit interval, it has poles at $1/n$ for every squarefree integer n:

```
>>> primezeta(0.5)          # Pole at s = 1/2
(-inf + 3.14159265358979323 85j)
>>> primezeta(0.25)
(-1.0416106801757269036 + 0.523598775598298873 08j)
>>> primezeta(0.5+10j)
(0.54892423556409790529 + 0.456268034234879342 64j)
```

Although evaluation works in principle for any $\mathrm{Re}(s) > 0$, it should be noted that the evaluation time increases exponentially as s approaches the imaginary axis.

For large $\mathrm{Re}(s)$, $P(s)$ is asymptotic to 2^{-s}:

```
>>> primezeta(inf)
0.0
>>> primezeta(10), mpf(2)**-10
(0.00099360357443698021786, 0.0009765625)
>>> primezeta(1000)
9.332636185032188789 9e-302
>>> primezeta(1000+1000j)
(-3.856544083365499594 9e-302 - 8.498539044755323430 5e-302j)
```

References

Carl-Erik Froberg, “On the prime zeta function”, BIT 8 (1968), pp. 187-202.

secondzeta()

mpmath.**secondzeta**(*s*, *a=0.015*, ***kwargs*)

Evaluates the secondary zeta function $Z(s)$, defined for $\mathrm{Re}(s) > 1$ by

$$Z(s) = \sum_{n=1}^{\infty} \frac{1}{\tau_n^s}$$

where $\frac{1}{2} + i\tau_n$ runs through the zeros of $\zeta(s)$ with imaginary part positive.

$Z(s)$ extends to a meromorphic function on $\mathbb{C}$ with a double pole at $s = 1$ and simple poles at the points $-2n$ for $n = 0, 1, 2, ...$

Examples

```
>>> from mpmath import *
>>> mp.pretty = True; mp.dps = 15
>>> secondzeta(2)
0.023104993115419
>>> xi = lambda s: 0.5*s*(s-1)*pi**(-0.5*s)*gamma(0.5*s)*zeta(s)
>>> Xi = lambda t: xi(0.5+t*j)
>>> -0.5*diff(Xi,0,n=2)/Xi(0)
(0.023104993115419 + 0.0j)
```

We may ask for an approximate error value:

```
>>> secondzeta(0.5+100j, error=True)
((-0.216272011276718 - 0.844952708937228j), 2.2044604925031e-16)
```

The function has poles at the negative odd integers, and dyadic rational values at the negative even integers:

```
>>> mp.dps = 30
>>> secondzeta(-8)
-0.67236328125
>>> secondzeta(-7)
+inf
```

Implementation notes

The function is computed as sum of four terms $Z(s) = A(s) - P(s) + E(s) - S(s)$ respectively main, prime, exponential and singular terms. The main term $A(s)$ is computed from the zeros of zeta. The prime term depends on the von Mangoldt function. The singular term is responsible for the poles of the function.

The four terms depends on a small parameter a. We may change the value of a. Theoretically this has no effect on the sum of the four terms, but in practice may be important.

A smaller value of the parameter a makes $A(s)$ depend on a smaller number of zeros of zeta, but $P(s)$ uses more values of von Mangoldt function.

We may also add a verbose option to obtain data about the values of the four terms.

```
>>> mp.dps = 10
>>> secondzeta(0.5 + 40j, error=True, verbose=True)
main term = (-30190318549.138656312556 - 13964804384.624622876523j)
    computed using 19 zeros of zeta
prime term = (132717176.89212754625045 + 188980555.17563978290601j)
    computed using 9 values of the von Mangoldt function
exponential term = (542447428666.07179812536 + 362434922978.80192435203j)
singular term = (512124392939.98154322355 + 348281138038.65531023921j)
((0.059471043 + 0.3463514534j), 1.455191523e-11)
```

```
>>> secondzeta(0.5 + 40j, a=0.04, error=True, verbose=True)
main term = (-151962888.19606243907725 - 217930683.90210294051982j)
    computed using 9 zeros of zeta
prime term = (2476659342.3038722372461 + 28711581821.921627163136j)
    computed using 37 values of the von Mangoldt function
exponential term = (178506047114.7838188264 + 819674143244.45677330576j)
singular term = (175877424884.22441310708 + 790744630738.28669174871j)
((0.059471043 + 0.3463514534j), 1.455191523e-11)
```

Notice the great cancellation between the four terms. Changing a, the four terms are very different numbers but the cancellation gives the good value of Z(s).

References

A. Voros, Zeta functions for the Riemann zeros, Ann. Institute Fourier, 53, (2003) 665–699.

A. Voros, Zeta functions over Zeros of Zeta Functions, Lecture Notes of the Unione Matematica Italiana, Springer, 2009.

3.1.12 Number-theoretical, combinatorial and integer functions

For factorial-type functions, including binomial coefficients, double factorials, etc., see the separate section Factorials and gamma functions.

Fibonacci numbers

fibonacci()/fib()

mpmath.**fibonacci**(*n*, ***kwargs*)

`fibonacci(n)` computes the n-th Fibonacci number, $F(n)$. The Fibonacci numbers are defined by the recurrence $F(n) = F(n-1) + F(n-2)$ with the initial values $F(0) = 0$, $F(1) = 1$. *`fibonacci()`* extends this definition to arbitrary real and complex arguments using the formula

$$F(z) = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

where ϕ is the golden ratio. *`fibonacci()`* also uses this continuous formula to compute $F(n)$ for extremely large n, where calculating the exact integer would be wasteful.

For convenience, `fib()` is available as an alias for *`fibonacci()`*.

Basic examples

Some small Fibonacci numbers are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for i in range(10):
...     print(fibonacci(i))
...
0.0
1.0
1.0
2.0
3.0
5.0
8.0
```

```
13.0
21.0
34.0
>>> fibonacci(50)
12586269025.0
```

The recurrence for $F(n)$ extends backwards to negative n:

```
>>> for i in range(10):
...     print(fibonacci(-i))
...
0.0
1.0
-1.0
2.0
-3.0
5.0
-8.0
13.0
-21.0
34.0
```

Large Fibonacci numbers will be computed approximately unless the precision is set high enough:

```
>>> fib(200)
2.8057117299251e+41
>>> mp.dps = 45
>>> fib(200)
280571172992510140037611932413038677189525.0
```

fibonacci() can compute approximate Fibonacci numbers of stupendous size:

```
>>> mp.dps = 15
>>> fibonacci(10**25)
3.49052338550226e+2089876402499787337692720
```

Real and complex arguments

The extended Fibonacci function is an analytic function. The property $F(z) = F(z-1) + F(z-2)$ holds for arbitrary z:

```
>>> mp.dps = 15
>>> fib(pi)
2.1170270579161
>>> fib(pi-1) + fib(pi-2)
2.1170270579161
>>> fib(3+4j)
(-5248.51130728372 - 14195.962288353j)
>>> fib(2+4j) + fib(1+4j)
(-5248.51130728372 - 14195.962288353j)
```

The Fibonacci function has infinitely many roots on the negative half-real axis. The first root is at 0, the second is close to -0.18, and then there are infinitely many roots that asymptotically approach $-n + 1/2$:

```
>>> findroot(fib, -0.2)
-0.183802359692956
>>> findroot(fib, -2)
-1.57077646820395
>>> findroot(fib, -17)
-16.4999999596115
```

```
>>> findroot(fib, -24)
-23.5000000000479
```

Mathematical relationships

For large n, $F(n+1)/F(n)$ approaches the golden ratio:

```
>>> mp.dps = 50
>>> fibonacci(101)/fibonacci(100)
1.6180339887498948482045868343656381177203127439638
>>> +phi
1.6180339887498948482045868343656381177203091798058
```

The sum of reciprocal Fibonacci numbers converges to an irrational number for which no closed form expression is known:

```
>>> mp.dps = 15
>>> nsum(lambda n: 1/fib(n), [1, inf])
3.35988566624318
```

Amazingly, however, the sum of odd-index reciprocal Fibonacci numbers can be expressed in terms of a Jacobi theta function:

```
>>> nsum(lambda n: 1/fib(2*n+1), [0, inf])
1.82451515740692
>>> sqrt(5)*jtheta(2,0,(3-sqrt(5))/2)**2/4
1.82451515740692
```

Some related sums can be done in closed form:

```
>>> nsum(lambda k: 1/(1+fib(2*k+1)), [0, inf])
1.11803398874989
>>> phi - 0.5
1.11803398874989
>>> f = lambda k:(-1)**(k+1) / sum(fib(n)**2 for n in range(1,int(k+1)))
>>> nsum(f, [1, inf])
0.618033988749895
>>> phi-1
0.618033988749895
```

References

1. http://mathworld.wolfram.com/FibonacciNumber.html

Bernoulli numbers and polynomials

bernoulli()

mpmath.**bernoulli**(*n*)

Computes the nth Bernoulli number, B_n, for any integer $n \geq 0$.

The Bernoulli numbers are rational numbers, but this function returns a floating-point approximation. To obtain an exact fraction, use *bernfrac()* instead.

Examples

Numerical values of the first few Bernoulli numbers:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(15):
...     print("%s %s" % (n, bernoulli(n)))
...
0 1.0
1 -0.5
2 0.166666666666667
3 0.0
4 -0.0333333333333333
5 0.0
6 0.0238095238095238
7 0.0
8 -0.0333333333333333
9 0.0
10 0.0757575757575758
11 0.0
12 -0.253113553113553
13 0.0
14 1.16666666666667
```

Bernoulli numbers can be approximated with arbitrary precision:

```
>>> mp.dps = 50
>>> bernoulli(100)
-2.8382249570693706959264156336481764738284680928013e+78
```

Arbitrarily large n are supported:

```
>>> mp.dps = 15
>>> bernoulli(10**20 + 2)
3.09136296657021e+1876752564973863312327
```

The Bernoulli numbers are related to the Riemann zeta function at integer arguments:

```
>>> -bernoulli(8) * (2*pi)**8 / (2*fac(8))
1.00407735619794
>>> zeta(8)
1.00407735619794
```

Algorithm

For small n ($n < 3000$) *bernoulli()* uses a recurrence formula due to Ramanujan. All results in this range are cached, so sequential computation of small Bernoulli numbers is guaranteed to be fast.

For larger n, B_n is evaluated in terms of the Riemann zeta function.

bernfrac()

mpmath.**bernfrac**(*n*)

Returns a tuple of integers (p, q) such that $p/q = B_n$ exactly, where B_n denotes the n-th Bernoulli number. The fraction is always reduced to lowest terms. Note that for $n > 1$ and n odd, $B_n = 0$, and $(0, 1)$ is returned.

Examples

The first few Bernoulli numbers are exactly:

```
>>> from mpmath import *
>>> for n in range(15):
...     p, q = bernfrac(n)
```

```
...     print("%s %s/%s" % (n, p, q))
...
0 1/1
1 -1/2
2 1/6
3 0/1
4 -1/30
5 0/1
6 1/42
7 0/1
8 -1/30
9 0/1
10 5/66
11 0/1
12 -691/2730
13 0/1
14 7/6
```

This function works for arbitrarily large n:

```
>>> p, q = bernfrac(10**4)
>>> print(q)
2338224387510
>>> print(len(str(p)))
27692
>>> mp.dps = 15
>>> print(mpf(p) / q)
-9.04942396360948e+27677
>>> print(bernoulli(10**4))
-9.04942396360948e+27677
```

Note: *bernoulli()* computes a floating-point approximation directly, without computing the exact fraction first. This is much faster for large n.

Algorithm

bernfrac() works by computing the value of B_n numerically and then using the von Staudt-Clausen theorem [1] to reconstruct the exact fraction. For large n, this is significantly faster than computing $B_1, B_2, \ldots, B_2$ recursively with exact arithmetic. The implementation has been tested for $n = 10^m$ up to $m = 6$.

In practice, *bernfrac()* appears to be about three times slower than the specialized program calcbn.exe [2]

References

1. MathWorld, von Staudt-Clausen Theorem: http://mathworld.wolfram.com/vonStaudt-ClausenTheorem.html

2. The Bernoulli Number Page: http://www.bernoulli.org/

bernpoly()

mpmath.**bernpoly**(*n*, *z*)

Evaluates the Bernoulli polynomial $B_n(z)$.

The first few Bernoulli polynomials are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
```

```
>>> for n in range(6):
...     nprint(chop(taylor(lambda x: bernpoly(n,x), 0, n)))
...
[1.0]
[-0.5, 1.0]
[0.166667, -1.0, 1.0]
[0.0, 0.5, -1.5, 1.0]
[-0.0333333, 0.0, 1.0, -2.0, 1.0]
[0.0, -0.166667, 0.0, 1.66667, -2.5, 1.0]
```

At $z = 0$, the Bernoulli polynomial evaluates to a Bernoulli number (see *bernoulli()*):

```
>>> bernpoly(12, 0), bernoulli(12)
(-0.253113553113553, -0.253113553113553)
>>> bernpoly(13, 0), bernoulli(13)
(0.0, 0.0)
```

Evaluation is accurate for large n and small z:

```
>>> mp.dps = 25
>>> bernpoly(100, 0.5)
2.838224957069370695926416e+78
>>> bernpoly(1000, 10.5)
5.318704469415522036482914e+1769
```

Euler numbers and polynomials

eulernum()

mpmath.**eulernum**(*n*)

Gives the n-th Euler number, defined as the n-th derivative of $\operatorname{sech}(t) = 1/\cosh(t)$ evaluated at $t = 0$. Equivalently, the Euler numbers give the coefficients of the Taylor series

$$\operatorname{sech}(t) = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n.$$

The Euler numbers are closely related to Bernoulli numbers and Bernoulli polynomials. They can also be evaluated in terms of Euler polynomials (see *eulerpoly()*) as $E_n = 2^n E_n(1/2)$.

Examples

Computing the first few Euler numbers and verifying that they agree with the Taylor series:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> [eulernum(n) for n in range(11)]
[1.0, 0.0, -1.0, 0.0, 5.0, 0.0, -61.0, 0.0, 1385.0, 0.0, -50521.0]
>>> chop(diffs(sech, 0, 10))
[1.0, 0.0, -1.0, 0.0, 5.0, 0.0, -61.0, 0.0, 1385.0, 0.0, -50521.0]
```

Euler numbers grow very rapidly. *eulernum()* efficiently computes numerical approximations for large indices:

```
>>> eulernum(50)
-6.053285248188621896314384e+54
>>> eulernum(1000)
3.887561841253070615257336e+2371
```

```
>>> eulernum(10**20)
4.346791453661149089338186e+193695856410665955133 1
```

Comparing with an asymptotic formula for the Euler numbers:

```
>>> n = 10**5
>>> (-1)**(n//2) * 8 * sqrt(n/(2*pi)) * (2*n/(pi*e))**n
3.69919063017432362805663e+436961
>>> eulernum(n)
3.69919371283446653794128 3e+436961
```

Pass `exact=True` to obtain exact values of Euler numbers as integers:

```
>>> print(eulernum(50, exact=True))
-6053285248188621896314383785111649088103498225146815121
>>> print(eulernum(200, exact=True) % 10**10)
1925859625
>>> eulernum(1001, exact=True)
0
```

eulerpoly()

mpmath.**eulerpoly**(*n*, *z*)

Evaluates the Euler polynomial $E_n(z)$, defined by the generating function representation

$$\frac{2e^{zt}}{e^t+1} = \sum_{n=0}^{\infty} E_n(z)\frac{t^n}{n!}.$$

The Euler polynomials may also be represented in terms of Bernoulli polynomials (see *bernpoly()*) using various formulas, for example

$$E_n(z) = \frac{2}{n+1}\left(B_n(z) - 2^{n+1}B_n\left(\frac{z}{2}\right)\right).$$

Special values include the Euler numbers $E_n = 2^n E_n(1/2)$ (see *eulernum()*).

Examples

Computing the coefficients of the first few Euler polynomials:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> for n in range(6):
...     chop(taylor(lambda z: eulerpoly(n,z), 0, n))
...
[1.0]
[-0.5, 1.0]
[0.0, -1.0, 1.0]
[0.25, 0.0, -1.5, 1.0]
[0.0, 1.0, 0.0, -2.0, 1.0]
[-0.5, 0.0, 2.5, 0.0, -2.5, 1.0]
```

Evaluation for arbitrary z:

```
>>> eulerpoly(2,3)
6.0
>>> eulerpoly(5,4)
423.5
>>> eulerpoly(35, 11111111112)
```

```
    3.994957561486776072734601e+351
    >>> eulerpoly(4, 10+20j)
    (-47990.0 - 235980.0j)
    >>> eulerpoly(2, '-3.5e-5')
    0.000035001225
    >>> eulerpoly(3, 0.5)
    0.0
    >>> eulerpoly(55, -10**80)
    -1.0e+4400
    >>> eulerpoly(5, -inf)
    -inf
    >>> eulerpoly(6, -inf)
    +inf
```

Computing Euler numbers:

```
    >>> 2**26 * eulerpoly(26,0.5)
    -408707250929312389 2361.0
    >>> eulernum(26)
    -408707250929312389 2361.0
```

Evaluation is accurate for large n and small z:

```
    >>> eulerpoly(100, 0.5)
    2.290479999881941141779 43e+108
    >>> eulerpoly(1000, 10.5)
    3.628120031122876847764566e+2070
    >>> eulerpoly(10000, 10.5)
    1.149364285543783412210773e+30688
```

Bell numbers and polynomials

bell()

mpmath.**bell**(*n*, *x*)

For n a nonnegative integer, `bell(n,x)` evaluates the Bell polynomial $B_n(x)$, the first few of which are

$$B_0(x) = 1$$
$$B_1(x) = x$$
$$B_2(x) = x^2 + x$$
$$B_3(x) = x^3 + 3x^2 + x$$

If $x = 1$ or *bell()* is called with only one argument, it gives the n-th Bell number B_n, which is the number of partitions of a set with n elements. By setting the precision to at least $\log_{10} B_n$ digits, *bell()* provides fast calculation of exact Bell numbers.

In general, *bell()* computes

$$B_n(x) = e^{-x} \left(\operatorname{sinc}(\pi n) + E_n(x)\right)$$

where $E_n(x)$ is the generalized exponential function implemented by *polyexp()*. This is an extension of Dobinski's formula [1], where the modification is the sinc term ensuring that $B_n(x)$ is continuous in n; *bell()* can thus be evaluated, differentiated, etc for arbitrary complex arguments.

Examples

Simple evaluations:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> bell(0, 2.5)
1.0
>>> bell(1, 2.5)
2.5
>>> bell(2, 2.5)
8.75
```

Evaluation for arbitrary complex arguments:

```
>>> bell(5.75+1j, 2-3j)
(-10767.71345136587098445143 - 15449.55065599872579097221j)
```

The first few Bell polynomials:

```
>>> for k in range(7):
...     nprint(taylor(lambda x: bell(k,x), 0, k))
...
[1.0]
[0.0, 1.0]
[0.0, 1.0, 1.0]
[0.0, 1.0, 3.0, 1.0]
[0.0, 1.0, 7.0, 6.0, 1.0]
[0.0, 1.0, 15.0, 25.0, 10.0, 1.0]
[0.0, 1.0, 31.0, 90.0, 65.0, 15.0, 1.0]
```

The first few Bell numbers and complementary Bell numbers:

```
>>> [int(bell(k)) for k in range(10)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
>>> [int(bell(k,-1)) for k in range(10)]
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

Large Bell numbers:

```
>>> mp.dps = 50
>>> bell(50)
185724268771078270438257767181908917499221852770.0
>>> bell(50,-1)
-29113173035759403920216141265491160286912.0
```

Some even larger values:

```
>>> mp.dps = 25
>>> bell(1000,-1)
-1.237132026969293954162816e+1869
>>> bell(1000)
2.989901335682408421480422e+1927
>>> bell(1000,2)
6.591553486811969380442171e+1987
>>> bell(1000,100.5)
9.101014101401543575679639e+2529
```

A determinant identity satisfied by Bell numbers:

```
>>> mp.dps = 15
>>> N = 8
>>> det([[bell(k+j) for j in range(N)] for k in range(N)])
125411328000.0
```

```
>>> superfac(N-1)
125411328000.0
```

References

1. http://mathworld.wolfram.com/DobinskisFormula.html

Stirling numbers

stirling1()

mpmath.**stirling1**(*n*, *k*, *exact=False*)

Gives the Stirling number of the first kind $s(n, k)$, defined by

$$x(x-1)(x-2)\cdots(x-n+1) = \sum_{k=0}^{n} s(n,k)x^k.$$

The value is computed using an integer recurrence. The implementation is not optimized for approximating large values quickly.

Examples

Comparing with the generating function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> taylor(lambda x: ff(x, 5), 0, 5)
[0.0, 24.0, -50.0, 35.0, -10.0, 1.0]
>>> [stirling1(5, k) for k in range(6)]
[0.0, 24.0, -50.0, 35.0, -10.0, 1.0]
```

Recurrence relation:

```
>>> n, k = 5, 3
>>> stirling1(n+1,k) + n*stirling1(n,k) - stirling1(n,k-1)
0.0
```

The matrices of Stirling numbers of first and second kind are inverses of each other:

```
>>> A = matrix(5, 5); B = matrix(5, 5)
>>> for n in range(5):
...     for k in range(5):
...         A[n,k] = stirling1(n,k)
...         B[n,k] = stirling2(n,k)
...
>>> A * B
[1.0  0.0  0.0  0.0  0.0]
[0.0  1.0  0.0  0.0  0.0]
[0.0  0.0  1.0  0.0  0.0]
[0.0  0.0  0.0  1.0  0.0]
[0.0  0.0  0.0  0.0  1.0]
```

Pass `exact=True` to obtain exact values of Stirling numbers as integers:

```
>>> stirling1(42, 5)
-2.86449897176850163373662803014001546235808317440000
>>> print stirling1(42, 5, exact=True)
-286449897176850163373662803014001546235808317440000
```

stirling2()

mpmath.**stirling2**(*n*, *k*, *exact=False*)

Gives the Stirling number of the second kind $S(n, k)$, defined by

$$x^n = \sum_{k=0}^{n} S(n,k) x(x-1)(x-2)\cdots(x-k+1)$$

The value is computed using integer arithmetic to evaluate a power sum. The implementation is not optimized for approximating large values quickly.

Examples

Comparing with the generating function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> taylor(lambda x: sum(stirling2(5,k) * ff(x,k) for k in range(6)), 0, 5)
[0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
```

Recurrence relation:

```
>>> n, k = 5, 3
>>> stirling2(n+1,k) - k*stirling2(n,k) - stirling2(n,k-1)
0.0
```

Pass `exact=True` to obtain exact values of Stirling numbers as integers:

```
>>> stirling2(52, 10)
2.641822121003543906807485e+45
>>> print stirling2(52, 10, exact=True)
2641822121003543906807485307053638921722527655
```

Prime counting functions

primepi()

mpmath.**primepi**(*x*)

Evaluates the prime counting function, $\pi(x)$, which gives the number of primes less than or equal to x. The argument x may be fractional.

The prime counting function is very expensive to evaluate precisely for large x, and the present implementation is not optimized in any way. For numerical approximation of the prime counting function, it is better to use *primepi2()* or *riemannr()*.

Some values of the prime counting function:

```
>>> from mpmath import *
>>> [primepi(k) for k in range(20)]
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 8]
>>> primepi(3.5)
2
>>> primepi(100000)
9592
```

primepi2()

mpmath.**primepi2**(*x*)

Returns an interval (as an `mpi` instance) providing bounds for the value of the prime counting function $\pi(x)$. For small x, *primepi2()* returns an exact interval based on the output of *primepi()*. For $x > 2656$, a loose interval based on Schoenfeld's inequality

$$|\pi(x) - \mathrm{li}(x)| < \frac{\sqrt{x}\log x}{8\pi}$$

is returned. This estimate is rigorous assuming the truth of the Riemann hypothesis, and can be computed very quickly.

Examples

Exact values of the prime counting function for small x:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> iv.dps = 15; iv.pretty = True
>>> primepi2(10)
[4.0, 4.0]
>>> primepi2(100)
[25.0, 25.0]
>>> primepi2(1000)
[168.0, 168.0]
```

Loose intervals are generated for moderately large x:

```
>>> primepi2(10000), primepi(10000)
([1209.0, 1283.0], 1229)
>>> primepi2(50000), primepi(50000)
([5070.0, 5263.0], 5133)
```

As x increases, the absolute error gets worse while the relative error improves. The exact value of $\pi(10^{23})$ is 1925320391606803968923, and *primepi2()* gives 9 significant digits:

```
>>> p = primepi2(10**23)
>>> p
[1.9253203909477020467e+21, 1.925320392280406229e+21]
>>> mpf(p.delta) / mpf(p.a)
6.9219865355293e-10
```

A more precise, nonrigorous estimate for $\pi(x)$ can be obtained using the Riemann R function (*riemannr()*). For large enough x, the value returned by *primepi2()* essentially amounts to a small perturbation of the value returned by *riemannr()*:

```
>>> primepi2(10**100)
[4.3619719871407024816e+97, 4.3619719871407032404e+97]
>>> riemannr(10**100)
4.3619719871407e+97
```

riemannr()

mpmath.**riemannr**(*x*)

Evaluates the Riemann R function, a smooth approximation of the prime counting function $\pi(x)$ (see *primepi()*). The Riemann R function gives a fast numerical approximation useful e.g. to roughly estimate the number of primes in a given interval.

The Riemann R function is computed using the rapidly convergent Gram series,

$$R(x) = 1 + \sum_{k=1}^{\infty} \frac{\log^k x}{kk!\zeta(k+1)}.$$

From the Gram series, one sees that the Riemann R function is a well-defined analytic function (except for a branch cut along the negative real half-axis); it can be evaluated for arbitrary real or complex arguments.

The Riemann R function gives a very accurate approximation of the prime counting function. For example, it is wrong by at most 2 for $x < 1000$, and for $x = 10^9$ differs from the exact value of $\pi(x)$ by 79, or less than two parts in a million. It is about 10 times more accurate than the logarithmic integral estimate (see `li()`), which however is even faster to evaluate. It is orders of magnitude more accurate than the extremely fast $x/\log x$ estimate.

Examples

For small arguments, the Riemann R function almost exactly gives the prime counting function if rounded to the nearest integer:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> primepi(50), riemannr(50)
(15, 14.9757023241462)
>>> max(abs(primepi(n)-int(round(riemannr(n)))) for n in range(100))
1
>>> max(abs(primepi(n)-int(round(riemannr(n)))) for n in range(300))
2
```

The Riemann R function can be evaluated for arguments far too large for exact determination of $\pi(x)$ to be computationally feasible with any presently known algorithm:

```
>>> riemannr(10**30)
1.46923988977204e+28
>>> riemannr(10**100)
4.3619719871407e+97
>>> riemannr(10**1000)
4.3448325764012e+996
```

A comparison of the Riemann R function and logarithmic integral estimates for $\pi(x)$ using exact values of $\pi(10^n)$ up to $n = 9$. The fractional error is shown in parentheses:

```
>>> exact = [4,25,168,1229,9592,78498,664579,5761455,50847534]
>>> for n, p in enumerate(exact):
...     n += 1
...     r, l = riemannr(10**n), li(10**n)
...     rerr, lerr = nstr((r-p)/p,3), nstr((l-p)/p,3)
...     print("%i %i %s(%s) %s(%s)" % (n, p, r, rerr, l, lerr))
...
1 4 4.56458314100509(0.141) 6.1655995047873(0.541)
2 25 25.6616332669242(0.0265) 30.1261415840796(0.205)
3 168 168.359446281167(0.00214) 177.609657990152(0.0572)
4 1229 1226.93121834343(-0.00168) 1246.13721589939(0.0139)
5 9592 9587.43173884197(-0.000476) 9629.8090010508(0.00394)
6 78498 78527.3994291277(0.000375) 78627.5491594622(0.00165)
7 664579 664667.447564748(0.000133) 664918.405048569(0.000511)
8 5761455 5761551.86732017(1.68e-5) 5762209.37544803(0.000131)
9 50847534 50847455.4277214(-1.55e-6) 50849234.9570018(3.35e-5)
```

The derivative of the Riemann R function gives the approximate probability for a number of magnitude x to be prime:

```
>>> diff(riemannr, 1000)
0.141903028110784
>>> mpf(primepi(1050) - primepi(950)) / 100
0.15
```

Evaluation is supported for arbitrary arguments and at arbitrary precision:

```
>>> mp.dps = 30
>>> riemannr(7.5)
3.72934743264966261918857135136
>>> riemannr(-4+2j)
(-0.551002208155486427591793957644 + 2.16966398138119450043195899746j)
```

Cyclotomic polynomials

cyclotomic()

mpmath.**cyclotomic**(*n*, *x*)

Evaluates the cyclotomic polynomial $\Phi_n(x)$, defined by

$$\Phi_n(x) = \prod_{\zeta} (x - \zeta)$$

where ζ ranges over all primitive n-th roots of unity (see *unitroots()*). An equivalent representation, used for computation, is

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)} = \Phi_n(x)$$

where $\mu(m)$ denotes the Moebius function. The cyclotomic polynomials are integer polynomials, the first of which can be written explicitly as

$$\Phi_0(x) = 1$$
$$\Phi_1(x) = x - 1$$
$$\Phi_2(x) = x + 1$$
$$\Phi_3(x) = x^3 + x^2 + 1$$
$$\Phi_4(x) = x^2 + 1$$
$$\Phi_5(x) = x^4 + x^3 + x^2 + x + 1$$
$$\Phi_6(x) = x^2 - x + 1$$

Examples

The coefficients of low-order cyclotomic polynomials can be recovered using Taylor expansion:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(9):
...     p = chop(taylor(lambda x: cyclotomic(n,x), 0, 10))
...     print("%s %s" % (n, nstr(p[:10+1-p[::-1].index(1)])))
...
0 [1.0]
1 [-1.0, 1.0]
2 [1.0, 1.0]
3 [1.0, 1.0, 1.0]
```

```
4 [1.0, 0.0, 1.0]
5 [1.0, 1.0, 1.0, 1.0, 1.0]
6 [1.0, -1.0, 1.0]
7 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
8 [1.0, 0.0, 0.0, 0.0, 1.0]
```

The definition as a product over primitive roots may be checked by computing the product explicitly (for a real argument, this method will generally introduce numerical noise in the imaginary part):

```
>>> mp.dps = 25
>>> z = 3+4j
>>> cyclotomic(10, z)
(-419.0 - 360.0j)
>>> fprod(z-r for r in unitroots(10, primitive=True))
(-419.0 - 360.0j)
>>> z = 3
>>> cyclotomic(10, z)
61.0
>>> fprod(z-r for r in unitroots(10, primitive=True))
(61.0 - 3.14604560508856860705545 4e-25j)
```

Up to permutation, the roots of a given cyclotomic polynomial can be checked to agree with the list of primitive roots:

```
>>> p = taylor(lambda x: cyclotomic(6,x), 0, 6)[:3]
>>> for r in polyroots(p[::-1]):
...     print(r)
...
(0.5 - 0.866025403784438646763723 2j)
(0.5 + 0.866025403784438646763723 2j)
>>>
>>> for r in unitroots(6, primitive=True):
...     print(r)
...
(0.5 + 0.866025403784438646763723 2j)
(0.5 - 0.866025403784438646763723 2j)
```

Arithmetic functions

mangoldt()

mpmath.**mangoldt**(*n*)

Evaluates the von Mangoldt function $\Lambda(n) = \log p$ if $n = p^k$ a power of a prime, and $\Lambda(n) = 0$ otherwise.

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> [mangoldt(n) for n in range(-2,3)]
[0.0, 0.0, 0.0, 0.0, 0.6931471805599453094172321]
>>> mangoldt(6)
0.0
>>> mangoldt(7)
1.945910149055313305105353
>>> mangoldt(8)
0.6931471805599453094172321
>>> fsum(mangoldt(n) for n in range(101))
```

```
    94.045311229357392246004 93
    >>> fsum(mangoldt(n) for n in range(10001))
    10013.396693263114783720 32
```

3.1.13 q-functions

q-Pochhammer symbol

qp()

mpmath.**qp**(*a*, *q=None*, *n=None*, ***kwargs*)

Evaluates the q-Pochhammer symbol (or q-rising factorial)

$$(a;q)_n = \prod_{k=0}^{n-1} (1 - aq^k)$$

where $n = \infty$ is permitted if $|q| < 1$. Called with two arguments, `qp(a,q)` computes $(a;q)_\infty$; with a single argument, `qp(q)` computes $(q;q)_\infty$. The special case

$$\phi(q) = (q;q)_\infty = \prod_{k=1}^{\infty} (1 - q^k) = \sum_{k=-\infty}^{\infty} (-1)^k q^{(3k^2-k)/2}$$

is also known as the Euler function, or (up to a factor $q^{-1/24}$) the Dedekind eta function.

Examples

If n is a positive integer, the function amounts to a finite product:

```
    >>> from mpmath import *
    >>> mp.dps = 25; mp.pretty = True
    >>> qp(2,3,5)
    -725305.0
    >>> fprod(1-2*3**k for k in range(5))
    -725305.0
    >>> qp(2,3,0)
    1.0
```

Complex arguments are allowed:

```
    >>> qp(2-1j, 0.75j)
    (0.46288422316601490899763 79 + 4.48182175355270309062879 3j)
```

The regular Pochhammer symbol $(a)_n$ is obtained in the following limit as $q \to 1$:

```
    >>> a, n = 4, 7
    >>> limit(lambda q: qp(q**a,q,n) / (1-q)**n, 1)
    604800.0
    >>> rf(a,n)
    604800.0
```

The Taylor series of the reciprocal Euler function gives the partition function $P(n)$, i.e. the number of ways of writing n as a sum of positive integers:

```
    >>> taylor(lambda q: 1/qp(q), 0, 10)
    [1.0, 1.0, 2.0, 3.0, 5.0, 7.0, 11.0, 15.0, 22.0, 30.0, 42.0]
```

Special values include:

```
>>> qp(0)
1.0
>>> findroot(diffun(qp), -0.4)    # location of maximum
-0.41124847917795547734440257
>>> qp(_)
1.2283488670385751125868 78
```

The q-Pochhammer symbol is related to the Jacobi theta functions. For example, the following identity holds:

```
>>> q = mpf(0.5)     # arbitrary
>>> qp(q)
0.28878809508660242127889 97
>>> root(3,-2)*root(q,-24)*jtheta(2,pi/6,root(q,6))
0.28878809508660242127889 97
```

q-gamma and factorial

qgamma()

mpmath.**qgamma**(*z*, *q*, ***kwargs*)

Evaluates the q-gamma function

$$\Gamma_q(z) = \frac{(q;q)_\infty}{(q^z;q)_\infty}(1-q)^{1-z}.$$

Examples

Evaluation for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qgamma(4,0.75)
4.046875
>>> qgamma(6,6)
121226245.0
>>> qgamma(3+4j, 0.5j)
(0.16630823822551998346300 88 + 0.01952474576025952984418217j)
```

The q-gamma function satisfies a functional equation similar to that of the ordinary gamma function:

```
>>> q = mpf(0.25)
>>> z = mpf(2.5)
>>> qgamma(z+1,q)
1.4282774248237609546859 12
>>> (1-q**z)/(1-q)*qgamma(z,q)
1.4282774248237609546859 12
```

qfac()

mpmath.**qfac**(*z*, *q*, ***kwargs*)

Evaluates the q-factorial,

$$[n]_q! = (1+q)(1+q+q^2)\cdots(1+q+\cdots+q^{n-1})$$

or more generally

$$[z]_q! = \frac{(q;q)_z}{(1-q)^z}.$$

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qfac(0,0)
1.0
>>> qfac(4,3)
2080.0
>>> qfac(5,6)
121226245.0
>>> qfac(1+1j, 2+1j)
(0.43705565513226724786136950 + 0.26097398392160392037089210j)
```

Hypergeometric q-series

qhyper()

mpmath.**qhyper**(*a_s*, *b_s*, *q*, *z*, ***kwargs*)

Evaluates the basic hypergeometric series or hypergeometric q-series

$$ {}_r\phi_s \left[\begin{matrix} a_1 & a_2 & \ldots & a_r \\ b_1 & b_2 & \ldots & b_s \end{matrix} ; q, z \right] = \sum_{n=0}^{\infty} \frac{(a_1;q)_n, \ldots, (a_r;q)_n}{(b_1;q)_n, \ldots, (b_s;q)_n} \left((-1)^n q^{\binom{n}{2}} \right)^{1+s-r} \frac{z^n}{(q;q)_n} $$

where $(a;q)_n$ denotes the q-Pochhammer symbol (see *qp()*).

Examples

Evaluation works for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qhyper([0.5], [2.25], 0.25, 4)
-0.19758490912633560095343850
>>> qhyper([0.5], [2.25], 0.25-0.25j, 4)
(2.80633024492571664983923700 + 3.56899762333794312176993800j)
>>> qhyper([1+j], [2,3+0.5j], 0.25, 3+4j)
(9.11288517177340001727022600 - 1.27275699716637505070038800j)
```

Comparing with a summation of the defining series, using *nsum()*:

```
>>> b, q, z = 3, 0.25, 0.5
>>> qhyper([], [b], q, z)
0.62211367482544955832283240
>>> nsum(lambda n: z**n / qp(q,q,n)/qp(b,q,n) * q**(n*(n-1)), [0,inf])
0.62211367482544955832283240
```

3.2 Numerical calculus

3.2.1 Polynomials

See also `taylor()` and `chebyfit()` for approximation of functions by polynomials.

Polynomial evaluation (polyval)

mpmath.**polyval**(*ctx*, *coeffs*, *x*, *derivative=False*)

Given coefficients $[c_n, \ldots, c_2, c_1, c_0]$ and a number x, `polyval()` evaluates the polynomial

$$P(x) = c_n x^n + \ldots + c_2 x^2 + c_1 x + c_0.$$

If *derivative=True* is set, `polyval()` simultaneously evaluates $P(x)$ with the derivative, $P'(x)$, and returns the tuple $(P(x), P'(x))$.

```
>>> from mpmath import *
>>> mp.pretty = True
>>> polyval([3, 0, 2], 0.5)
2.75
>>> polyval([3, 0, 2], 0.5, derivative=True)
(2.75, 3.0)
```

The coefficients and the evaluation point may be any combination of real or complex numbers.

Polynomial roots (polyroots)

mpmath.**polyroots**(*ctx*, *coeffs*, *maxsteps=50*, *cleanup=True*, *extraprec=10*, *error=False*)

Computes all roots (real or complex) of a given polynomial.

The roots are returned as a sorted list, where real roots appear first followed by complex conjugate roots as adjacent elements. The polynomial should be given as a list of coefficients, in the format used by `polyval()`. The leading coefficient must be nonzero.

With *error=True*, `polyroots()` returns a tuple *(roots, err)* where *err* is an estimate of the maximum error among the computed roots.

Examples

Finding the three real roots of $x^3 - x^2 - 14x + 24$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(polyroots([1,-1,-14,24]), 4)
[-4.0, 2.0, 3.0]
```

Finding the two complex conjugate roots of $4x^2 + 3x + 2$, with an error estimate:

```
>>> roots, err = polyroots([4,3,2], error=True)
>>> for r in roots:
...     print(r)
...
(-0.375 + 0.599478940414 09j)
(-0.375 - 0.599478940414 09j)
>>>
>>> err
2.22044604925031e-16
>>>
>>> polyval([4,3,2], roots[0])
(2.22044604925031e-16 + 0.0j)
>>> polyval([4,3,2], roots[1])
(2.22044604925031e-16 + 0.0j)
```

The following example computes all the 5th roots of unity; that is, the roots of $x^5 - 1$:

```
>>> mp.dps = 20
>>> for r in polyroots([1, 0, 0, 0, 0, -1]):
...     print(r)
...
1.0
(-0.8090169943749474241 + 0.58778525229247312917j)
(-0.8090169943749474241 - 0.58778525229247312917j)
(0.3090169943749474241 + 0.95105651629515357212j)
(0.3090169943749474241 - 0.95105651629515357212j)
```

Precision and conditioning

The roots are computed to the current working precision accuracy. If this accuracy cannot be achieved in $maxsteps$ steps, then a $NoConvergence$ exception is raised. The algorithm internally is using the current working precision extended by $extraprec$. If $NoConvergence$ was raised, that is caused either by not having enough extra precision to achieve convergence (in which case increasing $extraprec$ should fix the problem) or too low $maxsteps$ (in which case increasing $maxsteps$ should fix the problem), or a combination of both.

The user should always do a convergence study with regards to $extraprec$ to ensure accurate results. It is possible to get convergence to a wrong answer with too low $extraprec$.

Provided there are no repeated roots, *polyroots()* can typically compute all roots of an arbitrary polynomial to high precision:

```
>>> mp.dps = 60
>>> for r in polyroots([1, 0, -10, 0, 1]):
...     print r
...
-3.14626436994197234232913506571557044551247712918732870123249
-0.31783724519578224472575761729617428837313337843343255487912 7
0.31783724519578224472575761729617428837313337843343255487912 7
3.14626436994197234232913506571557044551247712918732870123249
>>>
>>> sqrt(3) + sqrt(2)
3.14626436994197234232913506571557044551247712918732870123249
>>> sqrt(3) - sqrt(2)
0.31783724519578224472575761729617428837313337843343255487912 7
```

Algorithm

polyroots() implements the Durand-Kerner method [1], which uses complex arithmetic to locate all roots simultaneously. The Durand-Kerner method can be viewed as approximately performing simultaneous Newton iteration for all the roots. In particular, the convergence to simple roots is quadratic, just like Newton's method.

Although all roots are internally calculated using complex arithmetic, any root found to have an imaginary part smaller than the estimated numerical error is truncated to a real number (small real parts are also chopped). Real roots are placed first in the returned list, sorted by value. The remaining complex roots are sorted by their real parts so that conjugate roots end up next to each other.

References

1. http://en.wikipedia.org/wiki/Durand-Kerner_method

3.2.2 Root-finding and optimization

Root-finding (findroot)

mpmath.**findroot**(*f*, *x0*, *solver=Secant*, *tol=None*, *verbose=False*, *verify=True*, ***kwargs*)

Find a solution to $f(x) = 0$, using *x0* as starting point or interval for *x*.

Multidimensional overdetermined systems are supported. You can specify them using a function or a list of functions.

If the found root does not satisfy $|f(x)|^2 \leq \mathrm{tol}$, an exception is raised (this can be disabled with *verify=False*).

Arguments

f one dimensional function

x0 starting point, several starting points or interval (depends on solver)

tol the returned solution has an error smaller than this

verbose print additional information for each iteration if true

verify verify the solution and raise a ValueError if $|f(x)|^2 > \mathrm{tol}$

solver a generator for *f* and *x0* returning approximative solution and error

maxsteps after how many steps the solver will cancel

df first derivative of *f* (used by some solvers)

d2f second derivative of *f* (used by some solvers)

multidimensional force multidimensional solving

J Jacobian matrix of *f* (used by multidimensional solvers)

norm used vector norm (used by multidimensional solvers)

solver has to be callable with `(f, x0, **kwargs)` and return an generator yielding pairs of approximative solution and estimated error (which is expected to be positive). You can use the following string aliases: 'secant', 'mnewton', 'halley', 'muller', 'illinois', 'pegasus', 'anderson', 'ridder', 'anewton', 'bisect'

See mpmath.calculus.optimization for their documentation.

Examples

The function *findroot()* locates a root of a given function using the secant method by default. A simple example use of the secant method is to compute π as the root of $\sin x$ closest to $x_0 = 3$:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> findroot(sin, 3)
3.14159265358979323846264338328
```

The secant method can be used to find complex roots of analytic functions, although it must in that case generally be given a nonreal starting value (or else it will never leave the real line):

```
>>> mp.dps = 15
>>> findroot(lambda x: x**3 + 2*x + 1, j)
(0.226698825758202 + 1.46771150871022j)
```

A nice application is to compute nontrivial roots of the Riemann zeta function with many digits (good initial values are needed for convergence):

```
>>> mp.dps = 30
>>> findroot(zeta, 0.5+14j)
(0.5 + 14.1347251417346937904572519836j)
```

The secant method can also be used as an optimization algorithm, by passing it a derivative of a function. The following example locates the positive minimum of the gamma function:

```
>>> mp.dps = 20
>>> findroot(lambda x: diff(gamma, x), 1)
1.4616321449683623413
```

Finally, a useful application is to compute inverse functions, such as the Lambert W function which is the inverse of we^w, given the first term of the solution's asymptotic expansion as the initial value. In basic cases, this gives identical results to mpmath's built-in `lambertw` function:

```
>>> def lambert(x):
...     return findroot(lambda w: w*exp(w) - x, log(1+x))
...
>>> mp.dps = 15
>>> lambert(1); lambertw(1)
0.567143290409784
0.567143290409784
>>> lambert(1000); lambert(1000)
5.2496028524016
5.2496028524016
```

Multidimensional functions are also supported:

```
>>> f = [lambda x1, x2: x1**2 + x2,
...      lambda x1, x2: 5*x1**2 - 3*x1 + 2*x2 - 3]
>>> findroot(f, (0, 0))
[-0.618033988749895]
[-0.381966011250105]
>>> findroot(f, (10, 10))
[ 1.61803398874989]
[-2.61803398874989]
```

You can verify this by solving the system manually.

Please note that the following (more general) syntax also works:

```
>>> def f(x1, x2):
...     return x1**2 + x2, 5*x1**2 - 3*x1 + 2*x2 - 3
...
>>> findroot(f, (0, 0))
[-0.618033988749895]
[-0.381966011250105]
```

Multiple roots

For multiple roots all methods of the Newtonian family (including secant) converge slowly. Consider this example:

```
>>> f = lambda x: (x - 1)**99
>>> findroot(f, 0.9, verify=False)
0.918073542444929
```

Even for a very close starting point the secant method converges very slowly. Use `verbose=True` to illustrate this.

It is possible to modify Newton's method to make it converge regardless of the root's multiplicity:

```
>>> findroot(f, -10, solver='mnewton')
1.0
```

This variant uses the first and second derivative of the function, which is not very efficient.

Alternatively you can use an experimental Newtonian solver that keeps track of the speed of convergence and accelerates it using Steffensen's method if necessary:

```
>>> findroot(f, -10, solver='anewton', verbose=True)
x:     -9.8888888888888888889
error: 0.11111111111111111111
converging slowly
x:     -9.77890011223344556678
error: 0.10998877665544332211
converging slowly
x:     -9.67002233332199662166
error: 0.108877778911448945119
converging slowly
accelerating convergence
x:     -9.5622443299551077669
error: 0.107778003366888854764
converging slowly
x:     0.99999999999999999214
error: 10.562244329955107759
x:     1.0
error: 7.8598304758094664213e-18
ZeroDivisionError: canceled with x = 1.0
1.0
```

Complex roots

For complex roots it's recommended to use Muller's method as it converges even for real starting points very fast:

```
>>> findroot(lambda x: x**4 + x + 1, (0, 1, 2), solver='muller')
(0.727136084491197 + 0.934099289460529j)
```

Intersection methods

When you need to find a root in a known interval, it's highly recommended to use an intersection-based solver like `'anderson'` or `'ridder'`. Usually they converge faster and more reliable. They have however problems with multiple roots and usually need a sign change to find a root:

```
>>> findroot(lambda x: x**3, (-1, 1), solver='anderson')
0.0
```

Be careful with symmetric functions:

```
>>> findroot(lambda x: x**2, (-1, 1), solver='anderson')
Traceback (most recent call last):
  ...
ZeroDivisionError
```

It fails even for better starting points, because there is no sign change:

```
>>> findroot(lambda x: x**2, (-1, .5), solver='anderson')
Traceback (most recent call last):
  ...
ValueError: Could not find root within given tolerance. (1 > 2.1684e-19)
Try another starting point or tweak arguments.
```

Solvers

class mpmath.calculus.optimization.**Secant**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Needs starting points x0 and x1 close to the root. x1 defaults to x0 + 0.25.

Pro:

- converges fast

Contra:

- converges slowly for multiple roots

class mpmath.calculus.optimization.**Newton**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Needs starting points x0 close to the root.

Pro:

- converges fast
- sometimes more robust than secant with bad second starting point

Contra:

- converges slowly for multiple roots
- needs first derivative
- 2 function evaluations per iteration

class mpmath.calculus.optimization.**MNewton**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Needs starting point x0 close to the root. Uses modified Newton's method that converges fast regardless of the multiplicity of the root.

Pro:

- converges fast for multiple roots

Contra:

- needs first and second derivative of f
- 3 function evaluations per iteration

class mpmath.calculus.optimization.**Halley**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Needs a starting point x0 close to the root. Uses Halley's method with cubic convergence rate.

Pro:

- converges even faster the Newton's method
- useful when computing with *many* digits

Contra:

- needs first and second derivative of f
- 3 function evaluations per iteration
- converges slowly for multiple roots

class mpmath.calculus.optimization.**Muller**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Needs starting points x0, x1 and x2 close to the root. x1 defaults to x0 + 0.25; x2 to x1 + 0.25. Uses Muller's method that converges towards complex roots.

Pro:

•converges fast (somewhat faster than secant)

•can find complex roots

Contra:

•converges slowly for multiple roots

•may have complex values for real starting points and real roots

http://en.wikipedia.org/wiki/Muller's_method

class mpmath.calculus.optimization.**Bisection**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Uses bisection method to find a root of f in [a, b]. Might fail for multiple roots (needs sign change).

Pro:

•robust and reliable

Contra:

•converges slowly

•needs sign change

class mpmath.calculus.optimization.**Illinois**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Uses Illinois method or similar to find a root of f in [a, b]. Might fail for multiple roots (needs sign change). Combines bisect with secant (improved regula falsi).

The only difference between the methods is the scaling factor m, which is used to ensure convergence (you can choose one using the 'method' keyword):

Illinois method ('illinois'): m = 0.5

Pegasus method ('pegasus'): m = fb/(fb + fz)

Anderson-Bjoerk method ('anderson'): m = 1 - fz/fb if positive else 0.5

Pro:

•converges very fast

Contra:

•has problems with multiple roots

•needs sign change

class mpmath.calculus.optimization.**Pegasus**
1d-solver generating pairs of approximative root and error.

Uses Pegasus method to find a root of f in [a, b]. Wrapper for illinois to use method='pegasus'.

class mpmath.calculus.optimization.**Anderson**
1d-solver generating pairs of approximative root and error.

Uses Anderson-Bjoerk method to find a root of f in [a, b]. Wrapper for illinois to use method='pegasus'.

class mpmath.calculus.optimization.**Ridder**(*ctx*, *f*, *x0*, ***kwargs*)
1d-solver generating pairs of approximative root and error.

Ridders' method to find a root of f in [a, b]. Is told to perform as well as Brent's method while being simpler.

Pro:

•very fast

•simpler than Brent's method

Contra:

•two function evaluations per step

•has problems with multiple roots

•needs sign change

http://en.wikipedia.org/wiki/Ridders'_method

class mpmath.calculus.optimization.**ANewton**(*ctx*, *f*, *x0*, ***kwargs*)
EXPERIMENTAL 1d-solver generating pairs of approximative root and error.

Uses Newton's method modified to use Steffensens method when convergence is slow. (I.e. for multiple roots.)

class mpmath.calculus.optimization.**MDNewton**(*ctx*, *f*, *x0*, ***kwargs*)
Find the root of a vector function numerically using Newton's method.

f is a vector function representing a nonlinear equation system.

x0 is the starting point close to the root.

J is a function returning the Jacobian matrix for a point.

Supports overdetermined systems.

Use the 'norm' keyword to specify which norm to use. Defaults to max-norm. The function to calculate the Jacobian matrix can be given using the keyword 'J'. Otherwise it will be calculated numerically.

Please note that this method converges only locally. Especially for high- dimensional systems it is not trivial to find a good starting point being close enough to the root.

It is recommended to use a faster, low-precision solver from SciPy [1] or OpenOpt [2] to get an initial guess. Afterwards you can use this method for root-polishing to any precision.

[1] http://scipy.org

[2] http://openopt.org/Welcome

3.2.3 Sums, products, limits and extrapolation

The functions listed here permit approximation of infinite sums, products, and other sequence limits. Use *mpmath.fsum()* and *mpmath.fprod()* for summation and multiplication of finite sequences.

Summation

nsum()

mpmath.**nsum**(*ctx*, *f*, **intervals*, ***options*)
Computes the sum

$$S = \sum_{k=a}^{b} f(k)$$

where (a, b) = *interval*, and where $a = -\infty$ and/or $b = \infty$ are allowed, or more generally

$$S = \sum_{k_1=a_1}^{b_1} \cdots \sum_{k_n=a_n}^{b_n} f(k_1, \ldots, k_n)$$

if multiple intervals are given.

Two examples of infinite series that can be summed by *nsum()*, where the first converges rapidly and the second converges slowly, are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nsum(lambda n: 1/fac(n), [0, inf])
2.71828182845905
>>> nsum(lambda n: 1/n**2, [1, inf])
1.64493406684823
```

When appropriate, *nsum()* applies convergence acceleration to accurately estimate the sums of slowly convergent series. If the series is finite, *nsum()* currently does not attempt to perform any extrapolation, and simply calls *fsum()*.

Multidimensional infinite series are reduced to a single-dimensional series over expanding hypercubes; if both infinite and finite dimensions are present, the finite ranges are moved innermost. For more advanced control over the summation order, use nested calls to *nsum()*, or manually rewrite the sum as a single-dimensional series.

Options

tol Desired maximum final error. Defaults roughly to the epsilon of the working precision.

method Which summation algorithm to use (described below). Default: `'richardson+shanks'`.

maxterms Cancel after at most this many terms. Default: 10*dps.

steps An iterable giving the number of terms to add between each extrapolation attempt. The default sequence is [10, 20, 30, 40, ...]. For example, if you know that approximately 100 terms will be required, efficiency might be improved by setting this to [100, 10]. Then the first extrapolation will be performed after 100 terms, the second after 110, etc.

verbose Print details about progress.

ignore If enabled, any term that raises `ArithmeticError` or `ValueError` (e.g. through division by zero) is replaced by a zero. This is convenient for lattice sums with a singular term near the origin.

Methods

Unfortunately, an algorithm that can efficiently sum any infinite series does not exist. *nsum()* implements several different algorithms that each work well in different cases. The *method* keyword argument selects a method.

The default method is `'r+s'`, i.e. both Richardson extrapolation and Shanks transformation is attempted. A slower method that handles more cases is `'r+s+e'`. For very high precision summation, or if the summation needs to be fast (for example if multiple sums need to be evaluated), it is a good idea to investigate which one method works best and only use that.

`'richardson'/'r'`: Uses Richardson extrapolation. Provides useful extrapolation when $f(k) \sim P(k)/Q(k)$ or when $f(k) \sim (-1)^k P(k)/Q(k)$ for polynomials P and Q. See *richardson()* for additional information.

`'shanks'/'s'`: Uses Shanks transformation. Typically provides useful extrapolation when $f(k) \sim c^k$ or when successive terms alternate signs. Is able to sum some divergent series. See *shanks()* for additional information.

'levin'/'l': Uses the Levin transformation. It performs better than the Shanks transformation for logarithmic convergent or alternating divergent series. The 'levin_variant'-keyword selects the variant. Valid choices are "u", "t", "v" and "all" whereby "all" uses all three u,t and v simultanously (This is good for performance comparison in conjunction with "verbose=True"). Instead of the Levin transform one can also use the Sidi-S transform by selecting the method 'sidi'. See *levin()* for additional details.

'alternating'/'a': This is the convergence acceleration of alternating series developped by Cohen, Villegras and Zagier. See *cohen_alt()* for additional details.

'euler-maclaurin'/'e': Uses the Euler-Maclaurin summation formula to approximate the remainder sum by an integral. This requires high-order numerical derivatives and numerical integration. The advantage of this algorithm is that it works regardless of the decay rate of f, as long as f is sufficiently smooth. See *sumem()* for additional information.

'direct'/'d': Does not perform any extrapolation. This can be used (and should only be used for) rapidly convergent series. The summation automatically stops when the terms decrease below the target tolerance.

Basic examples

A finite sum:

```
>>> nsum(lambda k: 1/k, [1, 6])
2.45
```

Summation of a series going to negative infinity and a doubly infinite series:

```
>>> nsum(lambda k: 1/k**2, [-inf, -1])
1.64493406684823
>>> nsum(lambda k: 1/(1+k**2), [-inf, inf])
3.15334809493716
```

nsum() handles sums of complex numbers:

```
>>> nsum(lambda k: (0.5+0.25j)**k, [0, inf])
(1.6 + 0.8j)
```

The following sum converges very rapidly, so it is most efficient to sum it by disabling convergence acceleration:

```
>>> mp.dps = 1000
>>> a = nsum(lambda k: -(-1)**k * k**2 / fac(2*k), [1, inf],
...     method='direct')
>>> b = (cos(1)+sin(1))/4
>>> abs(a-b) < mpf('1e-998')
True
```

Examples with Richardson extrapolation

Richardson extrapolation works well for sums over rational functions, as well as their alternating counterparts:

```
>>> mp.dps = 50
>>> nsum(lambda k: 1 / k**3, [1, inf],
...     method='richardson')
1.2020569031595942853997381615114499907649862923405
>>> zeta(3)
1.2020569031595942853997381615114499907649862923405

>>> nsum(lambda n: (n + 3)/(n**3 + n**2), [1, inf],
...     method='richardson')
2.9348022005446793094172454999380755676568497036204
>>> pi**2/2-2
2.9348022005446793094172454999380755676568497036204
```

```
>>> nsum(lambda k: (-1)**k / k**3, [1, inf],
...     method='richardson')
-0.90154267736969571404980362113358749307373971925537
>>> -3*zeta(3)/4
-0.90154267736969571404980362113358749307373971925538
```

Examples with Shanks transformation

The Shanks transformation works well for geometric series and typically provides excellent acceleration for Taylor series near the border of their disk of convergence. Here we apply it to a series for $\log(2)$, which can be seen as the Taylor series for $\log(1+x)$ with $x = 1$:

```
>>> nsum(lambda k: -(-1)**k/k, [1, inf],
...     method='shanks')
0.69314718055994530941723212145817656807550013436025
>>> log(2)
0.69314718055994530941723212145817656807550013436025
```

Here we apply it to a slowly convergent geometric series:

```
>>> nsum(lambda k: mpf('0.995')**k, [0, inf],
...     method='shanks')
200.0
```

Finally, Shanks' method works very well for alternating series where $f(k) = (-1)^k g(k)$, and often does so regardless of the exact decay rate of $g(k)$:

```
>>> mp.dps = 15
>>> nsum(lambda k: (-1)**(k+1) / k**1.5, [1, inf],
...     method='shanks')
0.765147024625408
>>> (2-sqrt(2))*zeta(1.5)/2
0.765147024625408
```

The following slowly convergent alternating series has no known closed-form value. Evaluating the sum a second time at higher precision indicates that the value is probably correct:

```
>>> nsum(lambda k: (-1)**k / log(k), [2, inf],
...     method='shanks')
0.924299897222939
>>> mp.dps = 30
>>> nsum(lambda k: (-1)**k / log(k), [2, inf],
...     method='shanks')
0.924299897222938853595957018136
```

Examples with Levin transformation

The following example calculates Euler's constant as the constant term in the Laurent expansion of zeta(s) at s=1. This sum converges extremly slow because of the logarithmic convergence behaviour of the Dirichlet series for zeta.

```
>>> mp.dps = 30
>>> z = mp.mpf(10) ** (-10)
>>> a = mp.nsum(lambda n: n**(-(1+z)), [1, mp.inf], method = "levin") - 1 / z
>>> print(mp.chop(a - mp.euler, tol = 1e-10))
0.0
```

Now we sum the zeta function outside its range of convergence (attention: This does not work at the negative integers!):

```
>>> mp.dps = 15
>>> w = mp.nsum(lambda n: n ** (2 + 3j), [1, mp.inf], method = "levin", levin_variant = "v")
>>> print(mp.chop(w - mp.zeta(-2-3j)))
0.0
```

The next example resummates an asymptotic series expansion of an integral related to the exponential integral.

```
>>> mp.dps = 15
>>> z = mp.mpf(10)
>>> # exact = mp.quad(lambda x: mp.exp(-x)/(1+x/z),[0,mp.inf])
>>> exact = z * mp.exp(z) * mp.expint(1,z) # this is the symbolic expression for the integral
>>> w = mp.nsum(lambda n: (-1) ** n * mp.fac(n) * z ** (-n), [0, mp.inf], method = "sidi", levin
>>> print(mp.chop(w - exact))
0.0
```

Following highly divergent asymptotic expansion needs some care. Firstly we need copious amount of working precision. Secondly the stepsize must not be chosen to large, otherwise nsum may miss the point where the Levin transform converges and reach the point where only numerical garbage is produced due to numerical cancellation.

```
>>> mp.dps = 15
>>> z = mp.mpf(2)
>>> # exact = mp.quad(lambda x: mp.exp( -x * x / 2 - z * x ** 4), [0,mp.inf]) * 2 / mp.sqrt(2 *
>>> exact = mp.exp(mp.one / (32 * z)) * mp.besselk(mp.one / 4, mp.one / (32 * z)) / (4 * mp.sqrt
>>> w = mp.nsum(lambda n: (-z)**n * mp.fac(4 * n) / (mp.fac(n) * mp.fac(2 * n) * (4 ** n)),
...   [0, mp.inf], method = "levin", levin_variant = "t", workprec = 8*mp.prec, steps = [2] + [1
>>> print(mp.chop(w - exact))
0.0
```

The hypergeoemtric function can also be summed outside its range of convergence:

```
>>> mp.dps = 15
>>> z = 2 + 1j
>>> exact = mp.hyp2f1(2 / mp.mpf(3), 4 / mp.mpf(3), 1 / mp.mpf(3), z)
>>> f = lambda n: mp.rf(2 / mp.mpf(3), n) * mp.rf(4 / mp.mpf(3), n) * z**n / (mp.rf(1 / mp.mpf(3
>>> v = mp.nsum(f, [0, mp.inf], method = "levin", steps = [10 for x in xrange(1000)])
>>> print(mp.chop(exact-v))
0.0
```

Examples with Cohen's alternating series resummation

The next example sums the alternating zeta function:

```
>>> v = mp.nsum(lambda n: (-1)**(n-1) / n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.log(2)))
0.0
```

The derivate of the alternating zeta function outside its range of convergence:

```
>>> v = mp.nsum(lambda n: (-1)**n * mp.log(n) * n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.diff(lambda s: mp.altzeta(s), -1)))
0.0
```

Examples with Euler-Maclaurin summation

The sum in the following example has the wrong rate of convergence for either Richardson or Shanks to be effective.

```
>>> f = lambda k: log(k)/k**2.5
>>> mp.dps = 15
>>> nsum(f, [1, inf], method='euler-maclaurin')
```

```
0.38734195032621
>>> -diff(zeta, 2.5)
0.38734195032621
```

Increasing `steps` improves speed at higher precision:

```
>>> mp.dps = 50
>>> nsum(f, [1, inf], method='euler-maclaurin', steps=[250])
0.38734195032620997271199237593105101319948228874688
>>> -diff(zeta, 2.5)
0.38734195032620997271199237593105101319948228874688
```

Divergent series

The Shanks transformation is able to sum some *divergent* series. In particular, it is often able to sum Taylor series beyond their radius of convergence (this is due to a relation between the Shanks transformation and Pade approximations; see *pade()* for an alternative way to evaluate divergent Taylor series). Furthermore the Levin-transform examples above contain some divergent series resummation.

Here we apply it to $\log(1+x)$ far outside the region of convergence:

```
>>> mp.dps = 50
>>> nsum(lambda k: -(-9)**k/k, [1, inf],
...     method='shanks')
2.3025850929940456840179914546843642076011014886288
>>> log(10)
2.3025850929940456840179914546843642076011014886288
```

A particular type of divergent series that can be summed using the Shanks transformation is geometric series. The result is the same as using the closed-form formula for an infinite geometric series:

```
>>> mp.dps = 15
>>> for n in range(-8, 8):
...     if n == 1:
...         continue
...     print("%s %s %s" % (mpf(n), mpf(1)/(1-n),
...         nsum(lambda k: n**k, [0, inf], method='shanks')))
...
-8.0 0.111111111111111 0.111111111111111
-7.0 0.125 0.125
-6.0 0.142857142857143 0.142857142857143
-5.0 0.166666666666667 0.166666666666667
-4.0 0.2 0.2
-3.0 0.25 0.25
-2.0 0.333333333333333 0.333333333333333
-1.0 0.5 0.5
0.0 1.0 1.0
2.0 -1.0 -1.0
3.0 -0.5 -0.5
4.0 -0.333333333333333 -0.333333333333333
5.0 -0.25 -0.25
6.0 -0.2 -0.2
7.0 -0.166666666666667 -0.166666666666667
```

Multidimensional sums

Any combination of finite and infinite ranges is allowed for the summation indices:

```
>>> mp.dps = 15
>>> nsum(lambda x,y: x+y, [2,3], [4,5])
28.0
```

```
>>> nsum(lambda x,y: x/2**y, [1,3], [1,inf])
6.0
>>> nsum(lambda x,y: y/2**x, [1,inf], [1,3])
6.0
>>> nsum(lambda x,y,z: z/(2**x*2**y), [1,inf], [1,inf], [3,4])
7.0
>>> nsum(lambda x,y,z: y/(2**x*2**z), [1,inf], [3,4], [1,inf])
7.0
>>> nsum(lambda x,y,z: x/(2**z*2**y), [3,4], [1,inf], [1,inf])
7.0
```

Some nice examples of double series with analytic solutions or reductions to single-dimensional series (see [1]):

```
>>> nsum(lambda m, n: 1/2**(m*n), [1,inf], [1,inf])
1.60669515241529
>>> nsum(lambda n: 1/(2**n-1), [1,inf])
1.60669515241529

>>> nsum(lambda i,j: (-1)**(i+j)/(i**2+j**2), [1,inf], [1,inf])
0.278070510848213
>>> pi*(pi-3*ln2)/12
0.278070510848213

>>> nsum(lambda i,j: (-1)**(i+j)/(i+j)**2, [1,inf], [1,inf])
0.129319852864168
>>> altzeta(2) - altzeta(1)
0.129319852864168

>>> nsum(lambda i,j: (-1)**(i+j)/(i+j)**3, [1,inf], [1,inf])
0.0790756439455825
>>> altzeta(3) - altzeta(2)
0.0790756439455825

>>> nsum(lambda m,n: m**2*n/(3**m*(n*3**m+m*3**n)),
...     [1,inf], [1,inf])
0.28125
>>> mpf(9)/32
0.28125

>>> nsum(lambda i,j: fac(i-1)*fac(j-1)/fac(i+j),
...     [1,inf], [1,inf], workprec=400)
1.64493406684823
>>> zeta(2)
1.64493406684823
```

A hard example of a multidimensional sum is the Madelung constant in three dimensions (see [2]). The defining sum converges very slowly and only conditionally, so *nsum()* is lucky to obtain an accurate value through convergence acceleration. The second evaluation below uses a much more efficient, rapidly convergent 2D sum:

```
>>> nsum(lambda x,y,z: (-1)**(x+y+z)/(x*x+y*y+z*z)**0.5,
...     [-inf,inf], [-inf,inf], [-inf,inf], ignore=True)
-1.74756459463318
>>> nsum(lambda x,y: -12*pi*sech(0.5*pi * \
...     sqrt((2*x+1)**2+(2*y+1)**2))**2, [0,inf], [0,inf])
-1.74756459463318
```

Another example of a lattice sum in 2D:

```
>>> nsum(lambda x,y: (-1)**(x+y) / (x**2+y**2), [-inf,inf],
...     [-inf,inf], ignore=True)
-2.1775860903036
>>> -pi*ln2
-2.1775860903036
```

An example of an Eisenstein series:

```
>>> nsum(lambda m,n: (m+n*1j)**(-4), [-inf,inf], [-inf,inf],
...     ignore=True)
(3.1512120021539 + 0.0j)
```

References

1. *[Weisstein]* http://mathworld.wolfram.com/DoubleSeries.html,
2. *[Weisstein]* http://mathworld.wolfram.com/MadelungConstants.html

sumem()

mpmath.**sumem**(*ctx*, *f*, *interval*, *tol=None*, *reject=10*, *integral=None*, *adiffs=None*, *bdiffs=None*, *verbose=False*, *error=False*, *_fast_abort=False*)

Uses the Euler-Maclaurin formula to compute an approximation accurate to within `tol` (which defaults to the present epsilon) of the sum

$$S = \sum_{k=a}^{b} f(k)$$

where (a, b) are given by `interval` and a or b may be infinite. The approximation is

$$S \sim \int_a^b f(x)\,dx + \frac{f(a)+f(b)}{2} + \sum_{k=1}^{\infty} \frac{B_{2k}}{(2k)!}\left(f^{(2k-1)}(b) - f^{(2k-1)}(a)\right).$$

The last sum in the Euler-Maclaurin formula is not generally convergent (a notable exception is if f is a polynomial, in which case Euler-Maclaurin actually gives an exact result).

The summation is stopped as soon as the quotient between two consecutive terms falls below *reject*. That is, by default (*reject* = 10), the summation is continued as long as each term adds at least one decimal.

Although not convergent, convergence to a given tolerance can often be "forced" if $b = \infty$ by summing up to $a + N$ and then applying the Euler-Maclaurin formula to the sum over the range $(a + N + 1, \ldots, \infty)$. This procedure is implemented by *nsum()*.

By default numerical quadrature and differentiation is used. If the symbolic values of the integral and endpoint derivatives are known, it is more efficient to pass the value of the integral explicitly as `integral` and the derivatives explicitly as `adiffs` and `bdiffs`. The derivatives should be given as iterables that yield $f(a), f'(a), f''(a), \ldots$ (and the equivalent for b).

Examples

Summation of an infinite series, with automatic and symbolic integral and derivative values (the second should be much faster):

```
>>> from mpmath import *
>>> mp.dps = 50; mp.pretty = True
>>> sumem(lambda n: 1/n**2, [32, inf])
0.031743366520302090126581680438741427141321886413417
>>> I = mpf(1)/32
```

```
>>> D = adiffs=((-1)**n*fac(n+1)*32**(-2-n) for n in range(999))
>>> sumem(lambda n: 1/n**2, [32, inf], integral=I, adiffs=D)
0.031743366520302090126581680438741427141321886413417
```

An exact evaluation of a finite polynomial sum:

```
>>> sumem(lambda n: n**5-12*n**2+3*n, [-100000, 200000])
10500155000624963999742499550000.0
>>> print(sum(n**5-12*n**2+3*n for n in range(-100000, 200001)))
10500155000624963999742499550000
```

sumap()

mpmath.**sumap**(*ctx*, *f*, *interval*, *integral=None*, *error=False*)

Evaluates an infinite series of an analytic summand f using the Abel-Plana formula

$$\sum_{k=0}^{\infty} f(k) = \int_0^{\infty} f(t)dt + \frac{1}{2}f(0) + i\int_0^{\infty} \frac{f(it) - f(-it)}{e^{2\pi t} - 1}dt.$$

Unlike the Euler-Maclaurin formula (see *sumem()*), the Abel-Plana formula does not require derivatives. However, it only works when $|f(it) - f(-it)|$ does not increase too rapidly with t.

Examples

The Abel-Plana formula is particularly useful when the summand decreases like a power of k; for example when the sum is a pure zeta function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sumap(lambda k: 1/k**2.5, [1,inf])
1.341487257250917179756775
>>> zeta(2.5)
1.341487257250917179756775
>>> sumap(lambda k: 1/(k+1j)**(2.5+2.5j), [1,inf])
(-3.385361068546473342286084 - 0.7432082105196321803869551j)
>>> zeta(2.5+2.5j, 1+1j)
(-3.385361068546473342286084 - 0.7432082105196321803869551j)
```

If the series is alternating, numerical quadrature along the real line is likely to give poor results, so it is better to evaluate the first term symbolically whenever possible:

```
>>> n=3; z=-0.75
>>> I = expint(n,-log(z))
>>> chop(sumap(lambda k: z**k / k**n, [1,inf], integral=I))
-0.69170360369045945101414480
>>> polylog(n,z)
-0.69170360369045945101414480
```

Products

nprod()

mpmath.**nprod**(*ctx*, *f*, *interval*, *nsum=False*, ***kwargs*)

Computes the product

$$P = \prod_{k=a}^{b} f(k)$$

where (a, b) = *interval*, and where $a = -\infty$ and/or $b = \infty$ are allowed.

By default, `nprod()` uses the same extrapolation methods as `nsum()`, except applied to the partial products rather than partial sums, and the same keyword options as for `nsum()` are supported. If `nsum=True`, the product is instead computed via `nsum()` as

$$P = \exp\left(\sum_{k=a}^{b} \log(f(k))\right).$$

This is slower, but can sometimes yield better results. It is also required (and used automatically) when Euler-Maclaurin summation is requested.

Examples

A simple finite product:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> nprod(lambda k: k, [1, 4])
24.0
```

A large number of infinite products have known exact values, and can therefore be used as a reference. Most of the following examples are taken from MathWorld [1].

A few infinite products with simple values are:

```
>>> 2*nprod(lambda k: (4*k**2)/(4*k**2-1), [1, inf])
3.14159265358979323846264
>>> nprod(lambda k: (1+1/k)**2/(1+2/k), [1, inf])
2.0
>>> nprod(lambda k: (k**3-1)/(k**3+1), [2, inf])
0.66666666666666666666667
>>> nprod(lambda k: (1-1/k**2), [2, inf])
0.5
```

Next, several more infinite products with more complicated values:

```
>>> nprod(lambda k: exp(1/k**2), [1, inf]); exp(pi**2/6)
5.18066831789711574841662
5.18066831789711574841662

>>> nprod(lambda k: (k**2-1)/(k**2+1), [2, inf]); pi*csch(pi)
0.272029054982133162950236
0.272029054982133162950236

>>> nprod(lambda k: (k**4-1)/(k**4+1), [2, inf])
0.848054049352900392129650
>>> pi*sinh(pi)/(cosh(sqrt(2)*pi)-cos(sqrt(2)*pi))
0.848054049352900392129650
```

```
>>> nprod(lambda k: (1+1/k+1/k**2)**2/(1+2/k+3/k**2), [1, inf])
1.848936182858244485224927
>>> 3*sqrt(2)*cosh(pi*sqrt(3)/2)**2*csch(pi*sqrt(2))/pi
1.848936182858244485224927

>>> nprod(lambda k: (1-1/k**4), [2, inf]); sinh(pi)/(4*pi)
0.919019477593744430173924 4
0.919019477593744430173924 4

>>> nprod(lambda k: (1-1/k**6), [2, inf])
0.982684277742192518324475 9
>>> (1+cosh(pi*sqrt(3)))/(12*pi**2)
0.982684277742192518324475 9

>>> nprod(lambda k: (1+1/k**2), [2, inf]); sinh(pi)/(2*pi)
1.838038955187488860347849
1.838038955187488860347849

>>> nprod(lambda n: (1+1/n)**n * exp(1/(2*n)-1), [1, inf])
1.447255926890365298959138
>>> exp(1+euler/2)/sqrt(2*pi)
1.447255926890365298959138
```

The following two products are equivalent and can be evaluated in terms of a Jacobi theta function. Pi can be replaced by any value (as long as convergence is preserved):

```
>>> nprod(lambda k: (1-pi**-k)/(1+pi**-k), [1, inf])
0.383845120748167240477868 6
>>> nprod(lambda k: tanh(k*log(pi)/2), [1, inf])
0.383845120748167240477868 6
>>> jtheta(4,0,1/pi)
0.383845120748167240477868 6
```

This product does not have a known closed form value:

```
>>> nprod(lambda k: (1-1/2**k), [1, inf])
0.288788095086602421278899 7
```

A product taken from $-\infty$:

```
>>> nprod(lambda k: 1-k**(-3), [-inf,-2])
0.809396597366290109578680 5
>>> cosh(pi*sqrt(3)/2)/(3*pi)
0.809396597366290109578680 5
```

A doubly infinite product:

```
>>> nprod(lambda k: exp(1/(1+k**2)), [-inf, inf])
23.414326882318643374200 35
>>> exp(pi/tanh(pi))
23.414326882318643374200 35
```

A product requiring the use of Euler-Maclaurin summation to compute an accurate value:

```
>>> nprod(lambda k: (1-1/k**2.5), [2, inf], method='e')
0.696155111336231052898125
```

References

1. *[Weisstein]* http://mathworld.wolfram.com/InfiniteProduct.html

Limits (`limit`)

`limit()`

mpmath.**limit**(*ctx*, *f*, *x*, *direction=1*, *exp=False*, ***kwargs*)

Computes an estimate of the limit

$$\lim_{t \to x} f(t)$$

where x may be finite or infinite.

For finite x, `limit()` evaluates $f(x + d/n)$ for consecutive integer values of n, where the approach direction d may be specified using the *direction* keyword argument. For infinite x, `limit()` evaluates values of $f(\mathrm{sign}(x) \cdot n)$.

If the approach to the limit is not sufficiently fast to give an accurate estimate directly, `limit()` attempts to find the limit using Richardson extrapolation or the Shanks transformation. You can select between these methods using the *method* keyword (see documentation of `nsum()` for more information).

Options

The following options are available with essentially the same meaning as for `nsum()`: *tol*, *method*, *maxterms*, *steps*, *verbose*.

If the option *exp=True* is set, f will be sampled at exponentially spaced points $n = 2^1, 2^2, 2^3, \ldots$ instead of the linearly spaced points $n = 1, 2, 3, \ldots$. This can sometimes improve the rate of convergence so that `limit()` may return a more accurate answer (and faster). However, do note that this can only be used if f supports fast and accurate evaluation for arguments that are extremely close to the limit point (or if infinite, very large arguments).

Examples

A basic evaluation of a removable singularity:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> limit(lambda x: (x-sin(x))/x**3, 0)
0.166666666666666666666666666667
```

Computing the exponential function using its limit definition:

```
>>> limit(lambda n: (1+3/n)**n, inf)
20.0855369231876677409285296546
>>> exp(3)
20.0855369231876677409285296546
```

A limit for π:

```
>>> f = lambda n: 2**(4*n+1)*fac(n)**4/(2*n+1)/fac(2*n)**2
>>> limit(f, inf)
3.14159265358979323846264338328
```

Calculating the coefficient in Stirling's formula:

```
>>> limit(lambda n: fac(n) / (sqrt(n)*(n/e)**n), inf)
2.50662827463100050241576528481
>>> sqrt(2*pi)
2.50662827463100050241576528481
```

Evaluating Euler's constant γ using the limit representation

$$\gamma = \lim_{n\to\infty} \left[\left(\sum_{k=1}^{n} \frac{1}{k} \right) - \log(n) \right]$$

(which converges notoriously slowly):

```
>>> f = lambda n: sum([mpf(1)/k for k in range(1,int(n)+1)]) - log(n)
>>> limit(f, inf)
0.57721566490153286060606512090082
>>> +euler
0.57721566490153286060606512090082
```

With default settings, the following limit converges too slowly to be evaluated accurately. Changing to exponential sampling however gives a perfect result:

```
>>> f = lambda x: sqrt(x**3+x**2)/(sqrt(x**3)+x)
>>> limit(f, inf)
0.99283115855833028112924968649 1
>>> limit(f, inf, exp=True)
1.0
```

Extrapolation

The following functions provide a direct interface to extrapolation algorithms. `nsum()` and `limit()` essentially work by calling the following functions with an increasing number of terms until the extrapolated limit is accurate enough.

The following functions may be useful to call directly if the precise number of terms needed to achieve a desired accuracy is known in advance, or if one wishes to study the convergence properties of the algorithms.

richardson()

mpmath.**richardson**(*ctx*, *seq*)

Given a list `seq` of the first N elements of a slowly convergent infinite sequence, *richardson()* computes the N-term Richardson extrapolate for the limit.

richardson() returns (v, c) where v is the estimated limit and c is the magnitude of the largest weight used during the computation. The weight provides an estimate of the precision lost to cancellation. Due to cancellation effects, the sequence must be typically be computed at a much higher precision than the target accuracy of the extrapolation.

Applicability and issues

The N-step Richardson extrapolation algorithm used by *richardson()* is described in [1].

Richardson extrapolation only works for a specific type of sequence, namely one converging like partial sums of $P(1)/Q(1) + P(2)/Q(2) + \ldots$ where P and Q are polynomials. When the sequence does not convergence at such a rate *richardson()* generally produces garbage.

Richardson extrapolation has the advantage of being fast: the N-term extrapolate requires only $O(N)$ arithmetic operations, and usually produces an estimate that is accurate to $O(N)$ digits. Contrast with the Shanks transformation (see *shanks()*), which requires $O(N^2)$ operations.

richardson() is unable to produce an estimate for the approximation error. One way to estimate the error is to perform two extrapolations with slightly different N and comparing the results.

Richardson extrapolation does not work for oscillating sequences. As a simple workaround, *richardson()* detects if the last three elements do not differ monotonically, and in that case applies extrapolation only to the even-index elements.

Example

Applying Richardson extrapolation to the Leibniz series for π:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> S = [4*sum(mpf(-1)**n/(2*n+1) for n in range(m))
...     for m in range(1,30)]
>>> v, c = richardson(S[:10])
>>> v
3.2126984126984126984126984127
>>> nprint([v-pi, c])
[0.0711058, 2.0]

>>> v, c = richardson(S[:30])
>>> v
3.1415926546862405282995420622 6
>>> nprint([v-pi, c])
[1.09645e-9, 20833.3]
```

References

1. *[BenderOrszag]* pp. 375-376

shanks()

mpmath.**shanks**(*ctx*, *seq*, *table=None*, *randomized=False*)

Given a list `seq` of the first N elements of a slowly convergent infinite sequence (A_k), *shanks()* computes the iterated Shanks transformation $S(A), S(S(A)), \ldots, S^{N/2}(A)$. The Shanks transformation often provides strong convergence acceleration, especially if the sequence is oscillating.

The iterated Shanks transformation is computed using the Wynn epsilon algorithm (see [1]). *shanks()* returns the full epsilon table generated by Wynn's algorithm, which can be read off as follows:

- The table is a list of lists forming a lower triangular matrix, where higher row and column indices correspond to more accurate values.
- The columns with even index hold dummy entries (required for the computation) and the columns with odd index hold the actual extrapolates.
- The last element in the last row is typically the most accurate estimate of the limit.
- The difference to the third last element in the last row provides an estimate of the approximation error.
- The magnitude of the second last element provides an estimate of the numerical accuracy lost to cancellation.

For convenience, so the extrapolation is stopped at an odd index so that `shanks(seq)[-1][-1]` always gives an estimate of the limit.

Optionally, an existing table can be passed to *shanks()*. This can be used to efficiently extend a previous computation after new elements have been appended to the sequence. The table will then be updated in-place.

The Shanks transformation

The Shanks transformation is defined as follows (see [2]): given the input sequence $(A_0, A_1, \ldots)$, the transformed sequence is given by

$$S(A_k) = \frac{A_{k+1}A_{k-1} - A_k^2}{A_{k+1} + A_{k-1} - 2A_k}$$

The Shanks transformation gives the exact limit A_∞ in a single step if $A_k = A + aq^k$. Note in particular that it extrapolates the exact sum of a geometric series in a single step.

Applying the Shanks transformation once often improves convergence substantially for an arbitrary sequence, but the optimal effect is obtained by applying it iteratively: $S(S(A_k)), S(S(S(A_k))), \ldots$.

Wynn's epsilon algorithm provides an efficient way to generate the table of iterated Shanks transformations. It reduces the computation of each element to essentially a single division, at the cost of requiring dummy elements in the table. See [1] for details.

Precision issues

Due to cancellation effects, the sequence must be typically be computed at a much higher precision than the target accuracy of the extrapolation.

If the Shanks transformation converges to the exact limit (such as if the sequence is a geometric series), then a division by zero occurs. By default, *shanks()* handles this case by terminating the iteration and returning the table it has generated so far. With *randomized=True*, it will instead replace the zero by a pseudorandom number close to zero. (TODO: find a better solution to this problem.)

Examples

We illustrate by applying Shanks transformation to the Leibniz series for π:

```
>>> from mpmath import *
>>> mp.dps = 50
>>> S = [4*sum(mpf(-1)**n/(2*n+1) for n in range(m))
...     for m in range(1,30)]
>>>
>>> T = shanks(S[:7])
>>> for row in T:
...     nprint(row)
...
[-0.75]
[1.25, 3.16667]
[-1.75, 3.13333, -28.75]
[2.25, 3.14524, 82.25, 3.14234]
[-2.75, 3.13968, -177.75, 3.14139, -969.937]
[3.25, 3.14271, 327.25, 3.14166, 3515.06, 3.14161]
```

The extrapolated accuracy is about 4 digits, and about 4 digits may have been lost due to cancellation:

```
>>> L = T[-1]
>>> nprint([abs(L[-1] - pi), abs(L[-1] - L[-3]), abs(L[-2])])
[2.22532e-5, 4.78309e-5, 3515.06]
```

Now we extend the computation:

```
>>> T = shanks(S[:25], T)
>>> L = T[-1]
>>> nprint([abs(L[-1] - pi), abs(L[-1] - L[-3]), abs(L[-2])])
[3.75527e-19, 1.48478e-19, 2.96014e+17]
```

The value for pi is now accurate to 18 digits. About 18 digits may also have been lost to cancellation.

Here is an example with a geometric series, where the convergence is immediate (the sum is exactly 1):

```
>>> mp.dps = 15
>>> for row in shanks([0.5, 0.75, 0.875, 0.9375, 0.96875]):
...     nprint(row)
[4.0]
[8.0, 1.0]
```

References

1.*[GravesMorris]*

2.*[BenderOrszag]* pp. 368-375

levin()

mpmath.**levin**(*ctx*, *method='levin'*, *variant='u'*)

This interface implements Levin's (nonlinear) sequence transformation for convergence acceleration and summation of divergent series. It performs better than the Shanks/Wynn-epsilon algorithm for logarithmic convergent or alternating divergent series.

Let *A* be the series we want to sum:

$$A = \sum_{k=0}^{\infty} a_k$$

Attention: all a_k must be non-zero!

Let s_n be the partial sums of this series:

$$s_n = \sum_{k=0}^{n} a_k.$$

Methods

Calling `levin` returns an object with the following methods.

`update(...)` works with the list of individual terms a_k of *A*, and `update_step(...)` works with the list of partial sums s_k of *A*:

```
v, e = ...update([a_0, a_1,..., a_k])
v, e = ...update_psum([s_0, s_1,..., s_k])
```

`step(...)` works with the individual terms a_k and `step_psum(...)` works with the partial sums s_k:

```
v, e = ...step(a_k)
v, e = ...step_psum(s_k)
```

v is the current estimate for *A*, and *e* is an error estimate which is simply the difference between the current estimate and the last estimate. One should not mix `update`, `update_psum`, `step` and `step_psum`.

A word of caution

One can only hope for good results (i.e. convergence acceleration or resummation) if the s_n have some well defind asymptotic behavior for large n and are not erratic or random. Furthermore one usually needs very high working precision because of the numerical cancellation. If the working precision is insufficient, levin may produce silently numerical garbage. Furthermore even if the Levin-transformation converges, in the general case there is no proof that the result is mathematically sound. Only for very special classes of problems one can prove that the Levin-transformation converges to the expected result (for example Stieltjes-type integrals). Furthermore the Levin-transform is quite expensive (i.e. slow) in comparison to Shanks/Wynn-epsilon, Richardson

& co. In summary one can say that the Levin-transformation is powerful but unreliable and that it may need a copious amount of working precision.

The Levin transform has several variants differing in the choice of weights. Some variants are better suited for the possible flavours of convergence behaviour of A than other variants:

```
convergence behaviour   levin-u   levin-t   levin-v   shanks/wynn-epsilon

logarithmic               +         -         +          -
linear                    +         +         +          +
alternating divergent     +         +         +          +

  "+" means the variant is suitable,"-" means the variant is not suitable;
  for comparison the Shanks/Wynn-epsilon transform is listed, too.
```

The variant is controlled though the variant keyword (i.e. `variant="u"`, `variant="t"` or `variant="v"`). Overall "u" is probably the best choice.

Finally it is possible to use the Sidi-S transform instead of the Levin transform by using the keyword `method='sidi'`. The Sidi-S transform works better than the Levin transformation for some divergent series (see the examples).

Parameters:

```
method      "levin" or "sidi" chooses either the Levin or the Sidi-S transformation
variant     "u","t" or "v" chooses the weight variant.
```

The Levin transform is also accessible through the nsum interface. `method="l"` or `method="levin"` select the normal Levin transform while `method="sidi"` selects the Sidi-S transform. The variant is in both cases selected through the levin_variant keyword. The stepsize in *nsum()* must not be chosen too large, otherwise it will miss the point where the Levin transform converges resulting in numerical overflow/garbage. For highly divergent series a copious amount of working precision must be chosen.

Examples

First we sum the zeta function:

```
>>> from mpmath import mp
>>> mp.prec = 53
>>> eps = mp.mpf(mp.eps)
>>> with mp.extraprec(2 * mp.prec): # levin needs a high working precision
...     L = mp.levin(method = "levin", variant = "u")
...     S, s, n = [], 0, 1
...     while 1:
...         s += mp.one / (n * n)
...         n += 1
...         S.append(s)
...         v, e = L.update_psum(S)
...         if e < eps:
...             break
...         if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - mp.pi ** 2 / 6))
0.0
>>> w = mp.nsum(lambda n: 1 / (n*n), [1, mp.inf], method = "levin", levin_variant = "u")
>>> print(mp.chop(v - w))
0.0
```

Now we sum the zeta function outside its range of convergence (attention: This does not work at the negative integers!):

```
>>> eps = mp.mpf(mp.eps)
>>> with mp.extraprec(2 * mp.prec): # levin needs a high working precision
...     L = mp.levin(method = "levin", variant = "v")
...     A, n = [], 1
...     while 1:
...         s = mp.mpf(n) ** (2 + 3j)
...         n += 1
...         A.append(s)
...         v, e = L.update(A)
...         if e < eps:
...             break
...         if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - mp.zeta(-2-3j)))
0.0
>>> w = mp.nsum(lambda n: n ** (2 + 3j), [1, mp.inf], method = "levin", levin_variant = "v")
>>> print(mp.chop(v - w))
0.0
```

Now we sum the divergent asymptotic expansion of an integral related to the exponential integral (see also [2] p.373). The Sidi-S transform works best here:

```
>>> z = mp.mpf(10)
>>> exact = mp.quad(lambda x: mp.exp(-x)/(1+x/z),[0,mp.inf])
>>> # exact = z * mp.exp(z) * mp.expint(1,z) # this is the symbolic expression for the integral
>>> eps = mp.mpf(mp.eps)
>>> with mp.extraprec(2 * mp.prec): # high working precisions are mandatory for divergent resumm
...     L = mp.levin(method = "sidi", variant = "t")
...     n = 0
...     while 1:
...         s = (-1)**n * mp.fac(n) * z ** (-n)
...         v, e = L.step(s)
...         n += 1
...         if e < eps:
...             break
...         if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - exact))
0.0
>>> w = mp.nsum(lambda n: (-1) ** n * mp.fac(n) * z ** (-n), [0, mp.inf], method = "sidi", levin
>>> print(mp.chop(v - w))
0.0
```

Another highly divergent integral is also summable:

```
>>> z = mp.mpf(2)
>>> eps = mp.mpf(mp.eps)
>>> exact = mp.quad(lambda x: mp.exp( -x * x / 2 - z * x ** 4), [0,mp.inf]) * 2 / mp.sqrt(2 * mp
>>> # exact = mp.exp(mp.one / (32 * z)) * mp.besselk(mp.one / 4, mp.one / (32 * z)) / (4 * mp.sq
>>> with mp.extraprec(7 * mp.prec):  # we need copious amount of precision to sum this highly di
...     L = mp.levin(method = "levin", variant = "t")
...     n, s = 0, 0
...     while 1:
...         s += (-z)**n * mp.fac(4 * n) / (mp.fac(n) * mp.fac(2 * n) * (4 ** n))
...         n += 1
...         v, e = L.step_psum(s)
...         if e < eps:
...             break
...         if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - exact))
0.0
```

```
>>> w = mp.nsum(lambda n: (-z)**n * mp.fac(4 * n) / (mp.fac(n) * mp.fac(2 * n) * (4 ** n)),
...   [0, mp.inf], method = "levin", levin_variant = "t", workprec = 8*mp.prec, steps = [2] + [1
>>> print(mp.chop(v - w))
0.0
```

These examples run with 15-20 decimal digits precision. For higher precision the working precision must be raised.

Examples for nsum

Here we calculate Euler's constant as the constant term in the Laurent expansion of $\zeta(s)$ at $s = 1$. This sum converges extremly slowly because of the logarithmic convergence behaviour of the Dirichlet series for zeta:

```
>>> mp.dps = 30
>>> z = mp.mpf(10) ** (-10)
>>> a = mp.nsum(lambda n: n**(-(1+z)), [1, mp.inf], method = "l") - 1 / z
>>> print(mp.chop(a - mp.euler, tol = 1e-10))
0.0
```

The Sidi-S transform performs excellently for the alternating series of $\log(2)$:

```
>>> a = mp.nsum(lambda n: (-1)**(n-1) / n, [1, mp.inf], method = "sidi")
>>> print(mp.chop(a - mp.log(2)))
0.0
```

Hypergeometric series can also be summed outside their range of convergence. The stepsize in *nsum()* must not be chosen too large, otherwise it will miss the point where the Levin transform converges resulting in numerical overflow/garbage:

```
>>> z = 2 + 1j
>>> exact = mp.hyp2f1(2 / mp.mpf(3), 4 / mp.mpf(3), 1 / mp.mpf(3), z)
>>> f = lambda n: mp.rf(2 / mp.mpf(3), n) * mp.rf(4 / mp.mpf(3), n) * z**n / (mp.rf(1 / mp.mpf(3
>>> v = mp.nsum(f, [0, mp.inf], method = "levin", steps = [10 for x in xrange(1000)])
>>> print(mp.chop(exact-v))
0.0
```

References:

[1] E.J. Weniger - "Nonlinear Sequence Transformations for the Acceleration of Convergence and the Summation of Divergent Series" arXiv:math/0306302

[2] A. Sidi - "Pratical Extrapolation Methods"

[3] H.H.H. Homeier - "Scalar Levin-Type Sequence Transformations" arXiv:math/0005209

cohen_alt()

mpmath.**cohen_alt**(*ctx*)

This interface implements the convergence acceleration of alternating series as described in H. Cohen, F.R. Villegas, D. Zagier - "Convergence Acceleration of Alternating Series". This series transformation works only well if the individual terms of the series have an alternating sign. It belongs to the class of linear series transformations (in contrast to the Shanks/Wynn-epsilon or Levin transform). This series transformation is also able to sum some types of divergent series. See the paper under which conditions this resummation is mathematical sound.

Let A be the series we want to sum:

$$A = \sum_{k=0}^{\infty} a_k$$

Let s_n be the partial sums of this series:

$$s_n = \sum_{k=0}^{n} a_k.$$

Interface

Calling `cohen_alt` returns an object with the following methods.

Then `update(...)` works with the list of individual terms a_k and `update_psum(...)` works with the list of partial sums s_k:

```
v, e = ...update([a_0, a_1,..., a_k])
v, e = ...update_psum([s_0, s_1,..., s_k])
```

v is the current estimate for *A*, and *e* is an error estimate which is simply the difference between the current estimate and the last estimate.

Examples

Here we compute the alternating zeta function using `update_psum`:

```
>>> from mpmath import mp
>>> AC = mp.cohen_alt()
>>> S, s, n = [], 0, 1
>>> while 1:
...     s += -((-1) ** n) * mp.one / (n * n)
...     n += 1
...     S.append(s)
...     v, e = AC.update_psum(S)
...     if e < mp.eps:
...         break
...     if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - mp.pi ** 2 / 12))
0.0
```

Here we compute the product $\prod_{n=1}^{\infty} \Gamma(1+1/(2n-1))/\Gamma(1+1/(2n))$:

```
>>> A = []
>>> AC = mp.cohen_alt()
>>> n = 1
>>> while 1:
...     A.append( mp.loggamma(1 + mp.one / (2 * n - 1)))
...     A.append(-mp.loggamma(1 + mp.one / (2 * n)))
...     n += 1
...     v, e = AC.update(A)
...     if e < mp.eps:
...         break
...     if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> v = mp.exp(v)
>>> print(mp.chop(v - 1.06215090557106, tol = 1e-12))
0.0
```

`cohen_alt` is also accessible through the *`nsum()`* interface:

```
>>> v = mp.nsum(lambda n: (-1)**(n-1) / n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.log(2)))
0.0
>>> v = mp.nsum(lambda n: (-1)**n / (2 * n + 1), [0, mp.inf], method = "a")
>>> print(mp.chop(v - mp.pi / 4))
0.0
```

```
>>> v = mp.nsum(lambda n: (-1)**n * mp.log(n) * n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.diff(lambda s: mp.altzeta(s), -1)))
0.0
```

3.2.4 Differentiation

Numerical derivatives (`diff`, `diffs`)

mpmath.**diff**(*ctx*, *f*, *x*, *n=1*, ***options*)

Numerically computes the derivative of f, $f'(x)$, or generally for an integer $n \geq 0$, the n-th derivative $f^{(n)}(x)$. A few basic examples are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> diff(lambda x: x**2 + x, 1.0)
3.0
>>> diff(lambda x: x**2 + x, 1.0, 2)
2.0
>>> diff(lambda x: x**2 + x, 1.0, 3)
0.0
>>> nprint([diff(exp, 3, n) for n in range(5)])    # exp'(x) = exp(x)
[20.0855, 20.0855, 20.0855, 20.0855, 20.0855]
```

Even more generally, given a tuple of arguments $(x_1, \ldots, x_k)$ and order $(n_1, \ldots, n_k)$, the partial derivative $f^{(n_1,\ldots,n_k)}(x_1, \ldots, x_k)$ is evaluated. For example:

```
>>> diff(lambda x,y: 3*x*y + 2*y - x, (0.25, 0.5), (0,1))
2.75
>>> diff(lambda x,y: 3*x*y + 2*y - x, (0.25, 0.5), (1,1))
3.0
```

Options

The following optional keyword arguments are recognized:

`method` Supported methods are `'step'` or `'quad'`: derivatives may be computed using either a finite difference with a small step size h (default), or numerical quadrature.

`direction` Direction of finite difference: can be -1 for a left difference, 0 for a central difference (default), or +1 for a right difference; more generally can be any complex number.

`addprec` Extra precision for h used to account for the function's sensitivity to perturbations (default = 10).

`relative` Choose h relative to the magnitude of x, rather than an absolute value; useful for large or tiny x (default = False).

`h` As an alternative to `addprec` and `relative`, manually select the step size h.

`singular` If True, evaluation exactly at the point x is avoided; this is useful for differentiating functions with removable singularities. Default = False.

`radius` Radius of integration contour (with `method = 'quad'`). Default = 0.25. A larger radius typically is faster and more accurate, but it must be chosen so that f has no singularities within the radius from the evaluation point.

A finite difference requires $n+1$ function evaluations and must be performed at $(n+1)$ times the target precision. Accordingly, f must support fast evaluation at high precision.

With integration, a larger number of function evaluations is required, but not much extra precision is required. For high order derivatives, this method may thus be faster if f is very expensive to evaluate at high precision.

Further examples

The direction option is useful for computing left- or right-sided derivatives of nonsmooth functions:

```
>>> diff(abs, 0, direction=0)
0.0
>>> diff(abs, 0, direction=1)
1.0
>>> diff(abs, 0, direction=-1)
-1.0
```

More generally, if the direction is nonzero, a right difference is computed where the step size is multiplied by sign(direction). For example, with direction=+j, the derivative from the positive imaginary direction will be computed:

```
>>> diff(abs, 0, direction=j)
(0.0 - 1.0j)
```

With integration, the result may have a small imaginary part even even if the result is purely real:

```
>>> diff(sqrt, 1, method='quad')
(0.5 - 4.59...e-26j)
>>> chop(_)
0.5
```

Adding precision to obtain an accurate value:

```
>>> diff(cos, 1e-30)
0.0
>>> diff(cos, 1e-30, h=0.0001)
-9.99999998328279e-31
>>> diff(cos, 1e-30, addprec=100)
-1.0e-30
```

mpmath.**diffs**(*ctx*, *f*, *x*, *n=None*, ***options*)

Returns a generator that yields the sequence of derivatives

$$f(x), f'(x), f''(x), \ldots, f^{(k)}(x), \ldots$$

With `method='step'`, *diffs()* uses only $O(k)$ function evaluations to generate the first k derivatives, rather than the roughly $O(k^2)$ evaluations required if one calls *diff()* k separate times.

With $n < \infty$, the generator stops as soon as the n-th derivative has been generated. If the exact number of needed derivatives is known in advance, this is further slightly more efficient.

Options are the same as for *diff()*.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15
>>> nprint(list(diffs(cos, 1, 5)))
[0.540302, -0.841471, -0.540302, 0.841471, 0.540302, -0.841471]
>>> for i, d in zip(range(6), diffs(cos, 1)):
...     print("%s %s" % (i, d))
...
0 0.54030230586814
1 -0.841470984807897
2 -0.54030230586814
3 0.841470984807897
4 0.54030230586814
5 -0.841470984807897
```

Composition of derivatives (diffs_prod, diffs_exp)

mpmath.**diffs_prod**(*ctx*, *factors*)

Given a list of N iterables or generators yielding $f_k(x), f_k'(x), f_k''(x), \ldots$ for $k = 1, \ldots, N$, generate $g(x), g'(x), g''(x), \ldots$ where $g(x) = f_1(x) f_2(x) \cdots f_N(x)$.

At high precision and for large orders, this is typically more efficient than numerical differentiation if the derivatives of each $f_k(x)$ admit direct computation.

Note: This function does not increase the working precision internally, so guard digits may have to be added externally for full accuracy.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = lambda x: exp(x)*cos(x)*sin(x)
>>> u = diffs(f, 1)
>>> v = mp.diffs_prod([diffs(exp,1), diffs(cos,1), diffs(sin,1)])
>>> next(u); next(v)
1.23586333600241
1.23586333600241
>>> next(u); next(v)
0.104658952245596
0.104658952245596
>>> next(u); next(v)
-5.96999877552086
-5.96999877552086
>>> next(u); next(v)
-12.4632923122697
-12.4632923122697
```

mpmath.**diffs_exp**(*ctx*, *fdiffs*)

Given an iterable or generator yielding $f(x), f'(x), f''(x), \ldots$ generate $g(x), g'(x), g''(x), \ldots$ where $g(x) = \exp(f(x))$.

At high precision and for large orders, this is typically more efficient than numerical differentiation if the derivatives of $f(x)$ admit direct computation.

Note: This function does not increase the working precision internally, so guard digits may have to be added externally for full accuracy.

Examples

The derivatives of the gamma function can be computed using logarithmic differentiation:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>>
>>> def diffs_loggamma(x):
...     yield loggamma(x)
...     i = 0
...     while 1:
...         yield psi(i,x)
...         i += 1
...
>>> u = diffs_exp(diffs_loggamma(3))
>>> v = diffs(gamma, 3)
>>> next(u); next(v)
2.0
2.0
```

```
>>> next(u); next(v)
1.84556867019693
1.84556867019693
>>> next(u); next(v)
2.49292999190269
2.49292999190269
>>> next(u); next(v)
3.44996501352367
3.44996501352367
```

Fractional derivatives / differintegration (`differint`)

mpmath.**differint**(*ctx*, *f*, *x*, *n=1*, *x0=0*)

Calculates the Riemann-Liouville differintegral, or fractional derivative, defined by

$$ {}_{x_0}\mathbb{D}_x^n f(x) \frac{1}{\Gamma(m-n)} \frac{d^m}{dx^m} \int_{x_0}^{x} (x-t)^{m-n-1} f(t) dt $$

where f is a given (presumably well-behaved) function, x is the evaluation point, n is the order, and x_0 is the reference point of integration (m is an arbitrary parameter selected automatically).

With $n = 1$, this is just the standard derivative $f'(x)$; with $n = 2$, the second derivative $f''(x)$, etc. With $n = -1$, it gives $\int_{x_0}^{x} f(t)dt$, with $n = -2$ it gives $\int_{x_0}^{x} \left(\int_{x_0}^{t} f(u) du \right) dt$, etc.

As n is permitted to be any number, this operator generalizes iterated differentiation and iterated integration to a single operator with a continuous order parameter.

Examples

There is an exact formula for the fractional derivative of a monomial x^p, which may be used as a reference. For example, the following gives a half-derivative (order 0.5):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> x = mpf(3); p = 2; n = 0.5
>>> differint(lambda t: t**p, x, n)
7.81764019044672
>>> gamma(p+1)/gamma(p-n+1) * x**(p-n)
7.81764019044672
```

Another useful test function is the exponential function, whose integration / differentiation formula easy generalizes to arbitrary order. Here we first compute a third derivative, and then a triply nested integral. (The reference point x_0 is set to $-\infty$ to avoid nonzero endpoint terms.):

```
>>> differint(lambda x: exp(pi*x), -1.5, 3)
0.278538406900792
>>> exp(pi*-1.5) * pi**3
0.278538406900792
>>> differint(lambda x: exp(pi*x), 3.5, -3, -inf)
1922.50563031149
>>> exp(pi*3.5) / pi**3
1922.50563031149
```

However, for noninteger n, the differentiation formula for the exponential function must be modified to give the same result as the Riemann-Liouville differintegral:

```
>>> x = mpf(3.5)
>>> c = pi
```

```
>>> n = 1+2*j
>>> differint(lambda x: exp(c*x), x, n)
(-123295.005390743 + 140955.117867654j)
>>> x**(-n) * exp(c)**x * (x*c)**n * gammainc(-n, 0, x*c) / gamma(-n)
(-123295.005390743 + 140955.117867654j)
```

3.2.5 Numerical integration (quadrature)

Standard quadrature (`quad`)

mpmath.**quad**(*ctx*, *f*, **points*, ***kwargs*)

Computes a single, double or triple integral over a given 1D interval, 2D rectangle, or 3D cuboid. A basic example:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> quad(sin, [0, pi])
2.0
```

A basic 2D integral:

```
>>> f = lambda x, y: cos(x+y/2)
>>> quad(f, [-pi/2, pi/2], [0, pi])
4.0
```

Interval format

The integration range for each dimension may be specified using a list or tuple. Arguments are interpreted as follows:

`quad(f, [x1, x2])` – calculates $\int_{x_1}^{x_2} f(x)\,dx$

`quad(f, [x1, x2], [y1, y2])` – calculates $\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x,y)\,dy\,dx$

`quad(f, [x1, x2], [y1, y2], [z1, z2])` – calculates $\int_{x_1}^{x_2} \int_{y_1}^{y_2} \int_{z_1}^{z_2} f(x,y,z)\,dz\,dy\,dx$

Endpoints may be finite or infinite. An interval descriptor may also contain more than two points. In this case, the integration is split into subintervals, between each pair of consecutive points. This is useful for dealing with mid-interval discontinuities, or integrating over large intervals where the function is irregular or oscillates.

Options

quad() recognizes the following keyword arguments:

method Chooses integration algorithm (described below).

error If set to true, *quad()* returns (v, e) where v is the integral and e is the estimated error.

maxdegree Maximum degree of the quadrature rule to try before quitting.

verbose Print details about progress.

Algorithms

Mpmath presently implements two integration algorithms: tanh-sinh quadrature and Gauss-Legendre quadrature. These can be selected using *method='tanh-sinh'* or *method='gauss-legendre'* or by passing the classes *method=TanhSinh*, *method=GaussLegendre*. The functions `quadts()` and `quadgl()` are also available as shortcuts.

Both algorithms have the property that doubling the number of evaluation points roughly doubles the accuracy, so both are ideal for high precision quadrature (hundreds or thousands of digits).

At high precision, computing the nodes and weights for the integration can be expensive (more expensive than computing the function values). To make repeated integrations fast, nodes are automatically cached.

The advantages of the tanh-sinh algorithm are that it tends to handle endpoint singularities well, and that the nodes are cheap to compute on the first run. For these reasons, it is used by *quad()* as the default algorithm.

Gauss-Legendre quadrature often requires fewer function evaluations, and is therefore often faster for repeated use, but the algorithm does not handle endpoint singularities as well and the nodes are more expensive to compute. Gauss-Legendre quadrature can be a better choice if the integrand is smooth and repeated integrations are required (e.g. for multiple integrals).

See the documentation for `TanhSinh` and `GaussLegendre` for additional details.

Examples of 1D integrals

Intervals may be infinite or half-infinite. The following two examples evaluate the limits of the inverse tangent function ($\int 1/(1+x^2) = \tan^{-1} x$), and the Gaussian integral $\int_{\infty}^{\infty} \exp(-x^2)\,dx = \sqrt{\pi}$:

```
>>> mp.dps = 15
>>> quad(lambda x: 2/(x**2+1), [0, inf])
3.14159265358979
>>> quad(lambda x: exp(-x**2), [-inf, inf])**2
3.14159265358979
```

Integrals can typically be resolved to high precision. The following computes 50 digits of π by integrating the area of the half-circle defined by $x^2 + y^2 \le 1$, $-1 \le x \le 1$, $y \ge 0$:

```
>>> mp.dps = 50
>>> 2*quad(lambda x: sqrt(1-x**2), [-1, 1])
3.1415926535897932384626433832795028841971693993751
```

One can just as well compute 1000 digits (output truncated):

```
>>> mp.dps = 1000
>>> 2*quad(lambda x: sqrt(1-x**2), [-1, 1])
3.14159265358979323846264338327950288419...216420198
```

Complex integrals are supported. The following computes a residue at $z = 0$ by integrating counterclockwise along the diamond-shaped path from 1 to $+i$ to -1 to $-i$ to 1:

```
>>> mp.dps = 15
>>> chop(quad(lambda z: 1/z, [1,j,-1,-j,1]))
(0.0 + 6.28318530717959j)
```

Examples of 2D and 3D integrals

Here are several nice examples of analytically solvable 2D integrals (taken from MathWorld [1]) that can be evaluated to high precision fairly rapidly by *quad()*:

```
>>> mp.dps = 30
>>> f = lambda x, y: (x-1)/((1-x*y)*log(x*y))
>>> quad(f, [0, 1], [0, 1])
0.577215664901532860606512090082
>>> +euler
0.577215664901532860606512090082

>>> f = lambda x, y: 1/sqrt(1+x**2+y**2)
>>> quad(f, [-1, 1], [-1, 1])
3.17343648530607134219175646705
>>> 4*log(2+sqrt(3))-2*pi/3
3.17343648530607134219175646705
```

```
>>> f = lambda x, y: 1/(1-x**2 * y**2)
>>> quad(f, [0, 1], [0, 1])
1.2337005501361698273543113749
>>> pi**2 / 8
1.2337005501361698273543113749

>>> quad(lambda x, y: 1/(1-x*y), [0, 1], [0, 1])
1.6449340668482264364724151666
>>> pi**2 / 6
1.6449340668482264364724151666
```

Multiple integrals may be done over infinite ranges:

```
>>> mp.dps = 15
>>> print(quad(lambda x,y: exp(-x-y), [0, inf], [1, inf]))
0.367879441171442
>>> print(1/e)
0.367879441171442
```

For nonrectangular areas, one can call *quad()* recursively. For example, we can replicate the earlier example of calculating π by integrating over the unit-circle, and actually use double quadrature to actually measure the area circle:

```
>>> f = lambda x: quad(lambda y: 1, [-sqrt(1-x**2), sqrt(1-x**2)])
>>> quad(f, [-1, 1])
3.14159265358979
```

Here is a simple triple integral:

```
>>> mp.dps = 15
>>> f = lambda x,y,z: x*y/(1+z)
>>> quad(f, [0,1], [0,1], [1,2], method='gauss-legendre')
0.101366277027041
>>> (log(3)-log(2))/4
0.101366277027041
```

Singularities

Both tanh-sinh and Gauss-Legendre quadrature are designed to integrate smooth (infinitely differentiable) functions. Neither algorithm copes well with mid-interval singularities (such as mid-interval discontinuities in $f(x)$ or $f'(x)$). The best solution is to split the integral into parts:

```
>>> mp.dps = 15
>>> quad(lambda x: abs(sin(x)), [0, 2*pi])    # Bad
3.99900894176779
>>> quad(lambda x: abs(sin(x)), [0, pi, 2*pi])  # Good
4.0
```

The tanh-sinh rule often works well for integrands having a singularity at one or both endpoints:

```
>>> mp.dps = 15
>>> quad(log, [0, 1], method='tanh-sinh')  # Good
-1.0
>>> quad(log, [0, 1], method='gauss-legendre')  # Bad
-0.999932197413801
```

However, the result may still be inaccurate for some functions:

```
>>> quad(lambda x: 1/sqrt(x), [0, 1], method='tanh-sinh')
1.99999999946942
```

This problem is not due to the quadrature rule per se, but to numerical amplification of errors in the nodes. The problem can be circumvented by temporarily increasing the precision:

```
>>> mp.dps = 30
>>> a = quad(lambda x: 1/sqrt(x), [0, 1], method='tanh-sinh')
>>> mp.dps = 15
>>> +a
2.0
```

Highly variable functions

For functions that are smooth (in the sense of being infinitely differentiable) but contain sharp mid-interval peaks or many "bumps", *quad()* may fail to provide full accuracy. For example, with default settings, *quad()* is able to integrate $\sin(x)$ accurately over an interval of length 100 but not over length 1000:

```
>>> quad(sin, [0, 100]); 1-cos(100)    # Good
0.137681127712316
0.137681127712316
>>> quad(sin, [0, 1000]); 1-cos(1000)    # Bad
-37.8587612408485
0.437620923709297
```

One solution is to break the integration into 10 intervals of length 100:

```
>>> quad(sin, linspace(0, 1000, 10))    # Good
0.437620923709297
```

Another is to increase the degree of the quadrature:

```
>>> quad(sin, [0, 1000], maxdegree=10)    # Also good
0.437620923709297
```

Whether splitting the interval or increasing the degree is more efficient differs from case to case. Another example is the function $1/(1+x^2)$, which has a sharp peak centered around $x = 0$:

```
>>> f = lambda x: 1/(1+x**2)
>>> quad(f, [-100, 100])    # Bad
3.64804647105268
>>> quad(f, [-100, 100], maxdegree=10)    # Good
3.12159332021646
>>> quad(f, [-100, 0, 100])    # Also good
3.12159332021646
```

References

1. http://mathworld.wolfram.com/DoubleIntegral.html

Oscillatory quadrature (quadosc)

mpmath.**quadosc**(*ctx*, *f*, *interval*, *omega=None*, *period=None*, *zeros=None*)

Calculates

$$I = \int_a^b f(x)dx$$

where at least one of a and b is infinite and where $f(x) = g(x)\cos(\omega x + \phi)$ for some slowly decreasing function $g(x)$. With proper input, *quadosc()* can also handle oscillatory integrals where the oscillation rate is different from a pure sine or cosine wave.

In the standard case when $|a| < \infty, b = \infty$, `quadosc()` works by evaluating the infinite series

$$I = \int_a^{x_1} f(x)dx + \sum_{k=1}^{\infty} \int_{x_k}^{x_{k+1}} f(x)dx$$

where x_k are consecutive zeros (alternatively some other periodic reference point) of $f(x)$. Accordingly, `quadosc()` requires information about the zeros of $f(x)$. For a periodic function, you can specify the zeros by either providing the angular frequency ω (*omega*) or the *period* $2\pi/\omega$. In general, you can specify the n-th zero by providing the *zeros* arguments. Below is an example of each:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = lambda x: sin(3*x)/(x**2+1)
>>> quadosc(f, [0,inf], omega=3)
0.37833007080198
>>> quadosc(f, [0,inf], period=2*pi/3)
0.37833007080198
>>> quadosc(f, [0,inf], zeros=lambda n: pi*n/3)
0.37833007080198
>>> (ei(3)*exp(-3)-exp(3)*ei(-3))/2  # Computed by Mathematica
0.37833007080198
```

Note that *zeros* was specified to multiply n by the *half-period*, not the full period. In theory, it does not matter whether each partial integral is done over a half period or a full period. However, if done over half-periods, the infinite series passed to `nsum()` becomes an *alternating series* and this typically makes the extrapolation much more efficient.

Here is an example of an integration over the entire real line, and a half-infinite integration starting at $-\infty$:

```
>>> quadosc(lambda x: cos(x)/(1+x**2), [-inf, inf], omega=1)
1.15572734979092
>>> pi/e
1.15572734979092
>>> quadosc(lambda x: cos(x)/x**2, [-inf, -1], period=2*pi)
-0.0844109505595739
>>> cos(1)+si(1)-pi/2
-0.0844109505595738
```

Of course, the integrand may contain a complex exponential just as well as a real sine or cosine:

```
>>> quadosc(lambda x: exp(3*j*x)/(1+x**2), [-inf,inf], omega=3)
(0.156410688228254 + 0.0j)
>>> pi/e**3
0.156410688228254
>>> quadosc(lambda x: exp(3*j*x)/(2+x+x**2), [-inf,inf], omega=3)
(0.00317486988463794 - 0.0447701735209082j)
>>> 2*pi/sqrt(7)/exp(3*(j+sqrt(7))/2)
(0.00317486988463794 - 0.0447701735209082j)
```

Non-periodic functions

If $f(x) = g(x)h(x)$ for some function $h(x)$ that is not strictly periodic, *omega* or *period* might not work, and it might be necessary to use *zeros*.

A notable exception can be made for Bessel functions which, though not periodic, are "asymptotically periodic" in a sufficiently strong sense that the sum extrapolation will work out:

```
>>> quadosc(j0, [0, inf], period=2*pi)
1.0
>>> quadosc(j1, [0, inf], period=2*pi)
1.0
```

More properly, one should provide the exact Bessel function zeros:

```
>>> j0zero = lambda n: findroot(j0, pi*(n-0.25))
>>> quadosc(j0, [0, inf], zeros=j0zero)
1.0
```

For an example where *zeros* becomes necessary, consider the complete Fresnel integrals

$$\int_0^\infty \cos x^2\,dx = \int_0^\infty \sin x^2\,dx = \sqrt{\frac{\pi}{8}}.$$

Although the integrands do not decrease in magnitude as $x \to \infty$, the integrals are convergent since the oscillation rate increases (causing consecutive periods to asymptotically cancel out). These integrals are virtually impossible to calculate to any kind of accuracy using standard quadrature rules. However, if one provides the correct asymptotic distribution of zeros ($x_n \sim \sqrt{n}$), *quadosc()* works:

```
>>> mp.dps = 30
>>> f = lambda x: cos(x**2)
>>> quadosc(f, [0,inf], zeros=lambda n:sqrt(pi*n))
0.626657068657750125603941321203
>>> f = lambda x: sin(x**2)
>>> quadosc(f, [0,inf], zeros=lambda n:sqrt(pi*n))
0.626657068657750125603941321203
>>> sqrt(pi/8)
0.626657068657750125603941321203
```

(Interestingly, these integrals can still be evaluated if one places some other constant than π in the square root sign.)

In general, if $f(x) \sim g(x)\cos(h(x))$, the zeros follow the inverse-function distribution $h^{-1}(x)$:

```
>>> mp.dps = 15
>>> f = lambda x: sin(exp(x))
>>> quadosc(f, [1,inf], zeros=lambda n: log(n))
-0.25024394235267
>>> pi/2-si(e)
-0.250243942352671
```

Non-alternating functions

If the integrand oscillates around a positive value, without alternating signs, the extrapolation might fail. A simple trick that sometimes works is to multiply or divide the frequency by 2:

```
>>> f = lambda x: 1/x**2+sin(x)/x**4
>>> quadosc(f, [1,inf], omega=1)   # Bad
1.28642190869861
>>> quadosc(f, [1,inf], omega=0.5)   # Perfect
1.28652953559617
>>> 1+(cos(1)+ci(1)+sin(1))/6
1.28652953559617
```

Fast decay

quadosc() is primarily useful for slowly decaying integrands. If the integrand decreases exponentially or faster, *quad()* will likely handle it without trouble (and generally be much faster than *quadosc()*):

```
>>> quadosc(lambda x: cos(x)/exp(x), [0, inf], omega=1)
0.5
>>> quad(lambda x: cos(x)/exp(x), [0, inf])
0.5
```

Quadrature rules

class mpmath.calculus.quadrature.**QuadratureRule**(*ctx*)

Quadrature rules are implemented using this class, in order to simplify the code and provide a common infrastructure for tasks such as error estimation and node caching.

You can implement a custom quadrature rule by subclassing `QuadratureRule` and implementing the appropriate methods. The subclass can then be used by *quad()* by passing it as the *method* argument.

`QuadratureRule` instances are supposed to be singletons. `QuadratureRule` therefore implements instance caching in `__new__()`.

calc_nodes(*degree*, *prec*, *verbose=False*)

Compute nodes for the standard interval $[-1, 1]$. Subclasses should probably implement only this method, and use `get_nodes()` method to retrieve the nodes.

clear()

Delete cached node data.

estimate_error(*results*, *prec*, *epsilon*)

Given results from integrations $[I_1, I_2, \ldots, I_k]$ done with a quadrature of rule of degree $1, 2, \ldots, k$, estimate the error of I_k.

For $k = 2$, we estimate $|I_\infty - I_2|$ as $|I_2 - I_1|$.

For $k > 2$, we extrapolate $|I_\infty - I_k| \approx |I_{k+1} - I_k|$ from $|I_k - I_{k-1}|$ and $|I_k - I_{k-2}|$ under the assumption that each degree increment roughly doubles the accuracy of the quadrature rule (this is true for both `TanhSinh` and `GaussLegendre`). The extrapolation formula is given by Borwein, Bailey & Girgensohn. Although not very conservative, this method seems to be very robust in practice.

get_nodes(*a*, *b*, *degree*, *prec*, *verbose=False*)

Return nodes for given interval, degree and precision. The nodes are retrieved from a cache if already computed; otherwise they are computed by calling `calc_nodes()` and are then cached.

Subclasses should probably not implement this method, but just implement `calc_nodes()` for the actual node computation.

guess_degree(*prec*)

Given a desired precision p in bits, estimate the degree m of the quadrature required to accomplish full accuracy for typical integrals. By default, *quad()* will perform up to m iterations. The value of m should be a slight overestimate, so that "slightly bad" integrals can be dealt with automatically using a few extra iterations. On the other hand, it should not be too big, so *quad()* can quit within a reasonable amount of time when it is given an "unsolvable" integral.

The default formula used by `guess_degree()` is tuned for both `TanhSinh` and `GaussLegendre`. The output is roughly as follows:

p	m
50	6
100	7
500	10
3000	12

This formula is based purely on a limited amount of experimentation and will sometimes be wrong.

sum_next(*f*, *nodes*, *degree*, *prec*, *previous*, *verbose=False*)

Evaluates the step sum $\sum w_k f(x_k)$ where the *nodes* list contains the (w_k, x_k) pairs.

`summation()` will supply the list *results* of values computed by `sum_next()` at previous degrees, in case the quadrature rule is able to reuse them.

summation(*f*, *points*, *prec*, *epsilon*, *max_degree*, *verbose=False*)
Main integration function. Computes the 1D integral over the interval specified by *points*. For each subinterval, performs quadrature of degree from 1 up to *max_degree* until `estimate_error()` signals convergence.

`summation()` transforms each subintegration to the standard interval and then calls `sum_next()`.

transform_nodes(*nodes*, *a*, *b*, *verbose=False*)
Rescale standardized nodes (for $[-1, 1]$) to a general interval $[a, b]$. For a finite interval, a simple linear change of variables is used. Otherwise, the following transformations are used:

$$[a, \infty] : t = \frac{1}{x} + (a - 1)$$
$$[-\infty, b] : t = (b + 1) - \frac{1}{x}$$
$$[-\infty, \infty] : t = \frac{x}{\sqrt{1 - x^2}}$$

Tanh-sinh rule

class mpmath.calculus.quadrature.**TanhSinh**(*ctx*)
This class implements "tanh-sinh" or "doubly exponential" quadrature. This quadrature rule is based on the Euler-Maclaurin integral formula. By performing a change of variables involving nested exponentials / hyperbolic functions (hence the name), the derivatives at the endpoints vanish rapidly. Since the error term in the Euler-Maclaurin formula depends on the derivatives at the endpoints, a simple step sum becomes extremely accurate. In practice, this means that doubling the number of evaluation points roughly doubles the number of accurate digits.

Comparison to Gauss-Legendre:

- Initial computation of nodes is usually faster
- Handles endpoint singularities better
- Handles infinite integration intervals better
- Is slower for smooth integrands once nodes have been computed

The implementation of the tanh-sinh algorithm is based on the description given in Borwein, Bailey & Girgensohn, "Experimentation in Mathematics - Computational Paths to Discovery", A K Peters, 2003, pages 312-313. In the present implementation, a few improvements have been made:

•A more efficient scheme is used to compute nodes (exploiting recurrence for the exponential function)

•The nodes are computed successively instead of all at once

Various documents describing the algorithm are available online, e.g.:

•http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-tanh-sinh.pdf

•http://users.cs.dal.ca/~jborwein/tanh-sinh.pdf

calc_nodes(*degree*, *prec*, *verbose=False*)
The abscissas and weights for tanh-sinh quadrature of degree m are given by

$$x_k = \tanh(\pi/2 \sinh(t_k))$$
$$w_k = \pi/2 \cosh(t_k) / \cosh(\pi/2 \sinh(t_k))^2$$

where $t_k = t_0 + hk$ for a step length $h \sim 2^{-m}$. The list of nodes is actually infinite, but the weights die off so rapidly that only a few are needed.

sum_next (*f*, *nodes*, *degree*, *prec*, *previous*, *verbose=False*)
Step sum for tanh-sinh quadrature of degree m. We exploit the fact that half of the abscissas at degree m are precisely the abscissas from degree $m-1$. Thus reusing the result from the previous level allows a 2x speedup.

Gauss-Legendre rule

class mpmath.calculus.quadrature.**GaussLegendre** (*ctx*)
This class implements Gauss-Legendre quadrature, which is exceptionally efficient for polynomials and polynomial-like (i.e. very smooth) integrands.

The abscissas and weights are given by roots and values of Legendre polynomials, which are the orthogonal polynomials on $[-1, 1]$ with respect to the unit weight (see *legendre()*).

In this implementation, we take the "degree" m of the quadrature to denote a Gauss-Legendre rule of degree $3 \cdot 2^m$ (following Borwein, Bailey & Girgensohn). This way we get quadratic, rather than linear, convergence as the degree is incremented.

Comparison to tanh-sinh quadrature:

- Is faster for smooth integrands once nodes have been computed
- Initial computation of nodes is usually slower
- Handles endpoint singularities worse
- Handles infinite integration intervals worse

calc_nodes (*degree*, *prec*, *verbose=False*)
Calculates the abscissas and weights for Gauss-Legendre quadrature of degree of given degree (actually $3 \cdot 2^m$).

3.2.6 Ordinary differential equations

Solving the ODE initial value problem (odefun)

mpmath.**odefun** (*ctx*, *F*, *x0*, *y0*, *tol=None*, *degree=None*, *method='taylor'*, *verbose=False*)
Returns a function $y(x) = [y_0(x), y_1(x), \ldots, y_n(x)]$ that is a numerical solution of the $n+1$-dimensional first-order ordinary differential equation (ODE) system

$$\begin{aligned} y_0'(x) &= F_0(x, [y_0(x), y_1(x), \ldots, y_n(x)]) \\ y_1'(x) &= F_1(x, [y_0(x), y_1(x), \ldots, y_n(x)]) \\ &\vdots \\ y_n'(x) &= F_n(x, [y_0(x), y_1(x), \ldots, y_n(x)]) \end{aligned}$$

The derivatives are specified by the vector-valued function *F* that evaluates $[y_0', \ldots, y_n'] = F(x, [y_0, \ldots, y_n])$. The initial point x_0 is specified by the scalar argument *x0*, and the initial value $y(x_0) = [y_0(x_0), \ldots, y_n(x_0)]$ is specified by the vector argument *y0*.

For convenience, if the system is one-dimensional, you may optionally provide just a scalar value for *y0*. In this case, *F* should accept a scalar *y* argument and return a scalar. The solution function *y* will return scalar values instead of length-1 vectors.

Evaluation of the solution function $y(x)$ is permitted for any $x \geq x_0$.

A high-order ODE can be solved by transforming it into first-order vector form. This transformation is described in standard texts on ODEs. Examples will also be given below.

Options, speed and accuracy

By default, *odefun()* uses a high-order Taylor series method. For reasonably well-behaved problems, the solution will be fully accurate to within the working precision. Note that *F* must be possible to evaluate to very high precision for the generation of Taylor series to work.

To get a faster but less accurate solution, you can set a large value for *tol* (which defaults roughly to *eps*). If you just want to plot the solution or perform a basic simulation, *tol = 0.01* is likely sufficient.

The *degree* argument controls the degree of the solver (with *method='taylor'*, this is the degree of the Taylor series expansion). A higher degree means that a longer step can be taken before a new local solution must be generated from *F*, meaning that fewer steps are required to get from x_0 to a given x_1. On the other hand, a higher degree also means that each local solution becomes more expensive (i.e., more evaluations of *F* are required per step, and at higher precision).

The optimal setting therefore involves a tradeoff. Generally, decreasing the *degree* for Taylor series is likely to give faster solution at low precision, while increasing is likely to be better at higher precision.

The function object returned by *odefun()* caches the solutions at all step points and uses polynomial interpolation between step points. Therefore, once $y(x_1)$ has been evaluated for some x_1, $y(x)$ can be evaluated very quickly for any $x_0 \le x \le x_1$. and continuing the evaluation up to $x_2 > x_1$ is also fast.

Examples of first-order ODEs

We will solve the standard test problem $y'(x) = y(x), y(0) = 1$ which has explicit solution $y(x) = \exp(x)$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = odefun(lambda x, y: y, 0, 1)
>>> for x in [0, 1, 2.5]:
...     print((f(x), exp(x)))
...
(1.0, 1.0)
(2.71828182845905, 2.71828182845905)
(12.1824939607035, 12.1824939607035)
```

The solution with high precision:

```
>>> mp.dps = 50
>>> f = odefun(lambda x, y: y, 0, 1)
>>> f(1)
2.7182818284590452353602874713526624977572470937
>>> exp(1)
2.7182818284590452353602874713526624977572470937
```

Using the more general vectorized form, the test problem can be input as (note that *f* returns a 1-element vector):

```
>>> mp.dps = 15
>>> f = odefun(lambda x, y: [y[0]], 0, [1])
>>> f(1)
[2.71828182845905]
```

odefun() can solve nonlinear ODEs, which are generally impossible (and at best difficult) to solve analytically. As an example of a nonlinear ODE, we will solve $y'(x) = x \sin(y(x))$ for $y(0) = \pi/2$. An exact solution happens to be known for this problem, and is given by $y(x) = 2\tan^{-1}\left(\exp\left(x^2/2\right)\right)$:

```
>>> f = odefun(lambda x, y: x*sin(y), 0, pi/2)
>>> for x in [2, 5, 10]:
...     print((f(x), 2*atan(exp(mpf(x)**2/2))))
...
(2.87255666284091, 2.87255666284091)
```

```
(3.14158520028345, 3.14158520028345)
(3.14159265358979, 3.14159265358979)
```

If F is independent of y, an ODE can be solved using direct integration. We can therefore obtain a reference solution with *quad()*:

```
>>> f = lambda x: (1+x**2)/(1+x**3)
>>> g = odefun(lambda x, y: f(x), pi, 0)
>>> g(2*pi)
0.72128263801696
>>> quad(f, [pi, 2*pi])
0.72128263801696
```

Examples of second-order ODEs

We will solve the harmonic oscillator equation $y''(x) + y(x) = 0$. To do this, we introduce the helper functions $y_0 = y, y_1 = y_0'$ whereby the original equation can be written as $y_1' + y_0' = 0$. Put together, we get the first-order, two-dimensional vector ODE

$$\begin{cases} y_0' = y_1 \\ y_1' = -y_0 \end{cases}$$

To get a well-defined IVP, we need two initial values. With $y(0) = y_0(0) = 1$ and $-y'(0) = y_1(0) = 0$, the problem will of course be solved by $y(x) = y_0(x) = \cos(x)$ and $-y'(x) = y_1(x) = \sin(x)$. We check this:

```
>>> f = odefun(lambda x, y: [-y[1], y[0]], 0, [1, 0])
>>> for x in [0, 1, 2.5, 10]:
...     nprint(f(x), 15)
...     nprint([cos(x), sin(x)], 15)
...     print("---")
...
[1.0, 0.0]
[1.0, 0.0]
---
[0.54030230586814, 0.841470984807897]
[0.54030230586814, 0.841470984807897]
---
[-0.801143615546934, 0.598472144103957]
[-0.801143615546934, 0.598472144103957]
---
[-0.839071529076452, -0.54402111088937]
[-0.839071529076452, -0.54402111088937]
---
```

Note that we get both the sine and the cosine solutions simultaneously.

TODO

- Better automatic choice of degree and step size
- Make determination of Taylor series convergence radius more robust
- Allow solution for $x < x_0$
- Allow solution for complex x
- Test for difficult (ill-conditioned) problems
- Implement Runge-Kutta and other algorithms

3.2.7 Function approximation

Taylor series (`taylor`)

mpmath.**taylor**(*ctx*, *f*, *x*, *n*, ***options*)

Produces a degree-n Taylor polynomial around the point x of the given function f. The coefficients are returned as a list.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(chop(taylor(sin, 0, 5)))
[0.0, 1.0, 0.0, -0.166667, 0.0, 0.00833333]
```

The coefficients are computed using high-order numerical differentiation. The function must be possible to evaluate to arbitrary precision. See *diff()* for additional details and supported keyword options.

Note that to evaluate the Taylor polynomial as an approximation of f, e.g. with *polyval()*, the coefficients must be reversed, and the point of the Taylor expansion must be subtracted from the argument:

```
>>> p = taylor(exp, 2.0, 10)
>>> polyval(p[::-1], 2.5 - 2.0)
12.1824939606092
>>> exp(2.5)
12.1824939607035
```

Pade approximation (`pade`)

mpmath.**pade**(*ctx*, *a*, *L*, *M*)

Computes a Pade approximation of degree (L, M) to a function. Given at least $L + M + 1$ Taylor coefficients a approximating a function $A(x)$, *pade()* returns coefficients of polynomials P, Q satisfying

$$P = \sum_{k=0}^{L} p_k x^k$$
$$Q = \sum_{k=0}^{M} q_k x^k$$
$$Q_0 = 1$$
$$A(x)Q(x) = P(x) + O(x^{L+M+1})$$

$P(x)/Q(x)$ can provide a good approximation to an analytic function beyond the radius of convergence of its Taylor series (example from G.A. Baker 'Essentials of Pade Approximants' Academic Press, Ch.1A):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> one = mpf(1)
>>> def f(x):
...     return sqrt((one + 2*x)/(one + x))
...
>>> a = taylor(f, 0, 6)
>>> p, q = pade(a, 3, 3)
>>> x = 10
>>> polyval(p[::-1], x)/polyval(q[::-1], x)
1.38169105566806
>>> f(x)
1.38169855941551
```

Chebyshev approximation (`chebyfit`)

mpmath.**chebyfit**(*ctx*, *f*, *interval*, *N*, *error=False*)

Computes a polynomial of degree $N-1$ that approximates the given function f on the interval $[a, b]$. With `error=True`, *chebyfit()* also returns an accurate estimate of the maximum absolute error; that is, the maximum value of $|f(x) - P(x)|$ for $x \in [a, b]$.

chebyfit() uses the Chebyshev approximation formula, which gives a nearly optimal solution: that is, the maximum error of the approximating polynomial is very close to the smallest possible for any polynomial of the same degree.

Chebyshev approximation is very useful if one needs repeated evaluation of an expensive function, such as function defined implicitly by an integral or a differential equation. (For example, it could be used to turn a slow mpmath function into a fast machine-precision version of the same.)

Examples

Here we use *chebyfit()* to generate a low-degree approximation of $f(x) = \cos(x)$, valid on the interval $[1, 2]$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> poly, err = chebyfit(cos, [1, 2], 5, error=True)
>>> nprint(poly)
[0.00291682, 0.146166, -0.732491, 0.174141, 0.949553]
>>> nprint(err, 12)
1.61351758081e-5
```

The polynomial can be evaluated using `polyval`:

```
>>> nprint(polyval(poly, 1.6), 12)
-0.0291858904138
>>> nprint(cos(1.6), 12)
-0.0291995223013
```

Sampling the true error at 1000 points shows that the error estimate generated by `chebyfit` is remarkably good:

```
>>> error = lambda x: abs(cos(x) - polyval(poly, x))
>>> nprint(max([error(1+n/1000.) for n in range(1000)]), 12)
1.61349954245e-5
```

Choice of degree

The degree N can be set arbitrarily high, to obtain an arbitrarily good approximation. As a rule of thumb, an N-term Chebyshev approximation is good to $N/(b-a)$ decimal places on a unit interval (although this depends on how well-behaved f is). The cost grows accordingly: `chebyfit` evaluates the function $(N^2)/2$ times to compute the coefficients and an additional N times to estimate the error.

Possible issues

One should be careful to use a sufficiently high working precision both when calling `chebyfit` and when evaluating the resulting polynomial, as the polynomial is sometimes ill-conditioned. It is for example difficult to reach 15-digit accuracy when evaluating the polynomial using machine precision floats, no matter the theoretical accuracy of the polynomial. (The option to return the coefficients in Chebyshev form should be made available in the future.)

It is important to note the Chebyshev approximation works poorly if f is not smooth. A function containing singularities, rapid oscillation, etc can be approximated more effectively by multiplying it by a weight function that cancels out the nonsmooth features, or by dividing the interval into several segments.

Fourier series (`fourier`, `fourierval`)

mpmath.**fourier**(*ctx*, *f*, *interval*, *N*)

Computes the Fourier series of degree N of the given function on the interval $[a, b]$. More precisely, *fourier()* returns two lists (c, s) of coefficients (the cosine series and sine series, respectively), such that

$$f(x) \sim \sum_{k=0}^{N} c_k \cos(kmx) + s_k \sin(kmx)$$

where $m = 2\pi/(b-a)$.

Note that many texts define the first coefficient as $2c_0$ instead of c_0. The easiest way to evaluate the computed series correctly is to pass it to *fourierval()*.

Examples

The function $f(x) = x$ has a simple Fourier series on the standard interval $[-\pi, \pi]$. The cosine coefficients are all zero (because the function has odd symmetry), and the sine coefficients are rational numbers:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> c, s = fourier(lambda x: x, [-pi, pi], 5)
>>> nprint(c)
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint(s)
[0.0, 2.0, -1.0, 0.666667, -0.5, 0.4]
```

This computes a Fourier series of a nonsymmetric function on a nonstandard interval:

```
>>> I = [-1, 1.5]
>>> f = lambda x: x**2 - 4*x + 1
>>> cs = fourier(f, I, 4)
>>> nprint(cs[0])
[0.583333, 1.12479, -1.27552, 0.904708, -0.441296]
>>> nprint(cs[1])
[0.0, -2.6255, 0.580905, 0.219974, -0.540057]
```

It is instructive to plot a function along with its truncated Fourier series:

```
>>> plot([f, lambda x: fourierval(cs, I, x)], I)
```

Fourier series generally converge slowly (and may not converge pointwise). For example, if $f(x) = \cosh(x)$, a 10-term Fourier series gives an L^2 error corresponding to 2-digit accuracy:

```
>>> I = [-1, 1]
>>> cs = fourier(cosh, I, 9)
>>> g = lambda x: (cosh(x) - fourierval(cs, I, x))**2
>>> nprint(sqrt(quad(g, I)))
0.00467963
```

fourier() uses numerical quadrature. For nonsmooth functions, the accuracy (and speed) can be improved by including all singular points in the interval specification:

```
>>> nprint(fourier(abs, [-1, 1], 0), 10)
([0.5000441648], [0.0])
>>> nprint(fourier(abs, [-1, 0, 1], 0), 10)
([0.5], [0.0])
```

mpmath.**fourierval**(*ctx*, *series*, *interval*, *x*)

Evaluates a Fourier series (in the format computed by by *fourier()* for the given interval) at the point x.

The series should be a pair (c, s) where c is the cosine series and s is the sine series. The two lists need not have the same length.

3.3 Matrices

3.3.1 Creating matrices

Basic methods

Matrices in mpmath are implemented using dictionaries. Only non-zero values are stored, so it is cheap to represent sparse matrices.

The most basic way to create one is to use the `matrix` class directly. You can create an empty matrix specifying the dimensions:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> matrix(2)
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
>>> matrix(2, 3)
matrix(
[['0.0', '0.0', '0.0'],
 ['0.0', '0.0', '0.0']])
```

Calling `matrix` with one dimension will create a square matrix.

To access the dimensions of a matrix, use the `rows` or `cols` keyword:

```
>>> A = matrix(3, 2)
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0'],
 ['0.0', '0.0']])
>>> A.rows
3
>>> A.cols
2
```

You can also change the dimension of an existing matrix. This will set the new elements to 0. If the new dimension is smaller than before, the concerning elements are discarded:

```
>>> A.rows = 2
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
```

Internally `convert` is applied every time an element is set. This is done using the syntax A[row,column], counting from 0:

```
>>> A = matrix(2)
>>> A[1,1] = 1 + 1j
>>> print A
```

```
[0.0            0.0]
[0.0  (1.0 + 1.0j)]
```

A more comfortable way to create a matrix lets you use nested lists:

```
>>> matrix([[1, 2], [3, 4]])
matrix(
[['1.0', '2.0'],
 ['3.0', '4.0']])
```

Advanced methods

Convenient functions are available for creating various standard matrices:

```
>>> zeros(2)
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
>>> ones(2)
matrix(
[['1.0', '1.0'],
 ['1.0', '1.0']])
>>> diag([1, 2, 3]) # diagonal matrix
matrix(
[['1.0', '0.0', '0.0'],
 ['0.0', '2.0', '0.0'],
 ['0.0', '0.0', '3.0']])
>>> eye(2) # identity matrix
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
```

You can even create random matrices:

```
>>> randmatrix(2)
matrix(
[['0.53491598236191806', '0.57195669543302752'],
 ['0.85589992269513615', '0.82444367501382143']])
```

Vectors

Vectors may also be represented by the `matrix` class (with rows = 1 or cols = 1). For vectors there are some things which make life easier. A column vector can be created using a flat list, a row vectors using an almost flat nested list:

```
>>> matrix([1, 2, 3])
matrix(
[['1.0'],
 ['2.0'],
 ['3.0']])
>>> matrix([[1, 2, 3]])
matrix(
[['1.0', '2.0', '3.0']])
```

Optionally vectors can be accessed like lists, using only a single index:

```
>>> x = matrix([1, 2, 3])
>>> x[1]
mpf('2.0')
>>> x[1,0]
mpf('2.0')
```

Other

Like you probably expected, matrices can be printed:

```
>>> print randmatrix(3)
[ 0.782963853573023  0.802057689719883  0.427895717335467]
[0.0541876859348597  0.708243266653103  0.615134039977379]
[ 0.856151514955773  0.544759264818486  0.686210904770947]
```

Use `nstr` or `nprint` to specify the number of digits to print:

```
>>> nprint(randmatrix(5), 3)
[2.07e-1  1.66e-1  5.06e-1  1.89e-1  8.29e-1]
[6.62e-1  6.55e-1  4.47e-1  4.82e-1  2.06e-2]
[4.33e-1  7.75e-1  6.93e-2  2.86e-1  5.71e-1]
[1.01e-1  2.53e-1  6.13e-1  3.32e-1  2.59e-1]
[1.56e-1  7.27e-2  6.05e-1  6.67e-2  2.79e-1]
```

As matrices are mutable, you will need to copy them sometimes:

```
>>> A = matrix(2)
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
>>> B = A.copy()
>>> B[0,0] = 1
>>> B
matrix(
[['1.0', '0.0'],
 ['0.0', '0.0']])
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
```

Finally, it is possible to convert a matrix to a nested list. This is very useful, as most Python libraries involving matrices or arrays (namely NumPy or SymPy) support this format:

```
>>> B.tolist()
[[mpf('1.0'), mpf('0.0')], [mpf('0.0'), mpf('0.0')]]
```

3.3.2 Matrix operations

You can add and substract matrices of compatible dimensions:

```
>>> A = matrix([[1, 2], [3, 4]])
>>> B = matrix([[-2, 4], [5, 9]])
>>> A + B
matrix(
```

```
[['-1.0', '6.0'],
 ['8.0', '13.0']])
>>> A - B
matrix(
[['3.0', '-2.0'],
 ['-2.0', '-5.0']])
>>> A + ones(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...", line 238, in __add__
    raise ValueError('incompatible dimensions for addition')
ValueError: incompatible dimensions for addition
```

It is possible to multiply or add matrices and scalars. In the latter case the operation will be done element-wise:

```
>>> A * 2
matrix(
[['2.0', '4.0'],
 ['6.0', '8.0']])
>>> A / 4
matrix(
[['0.25', '0.5'],
 ['0.75', '1.0']])
>>> A - 1
matrix(
[['0.0', '1.0'],
 ['2.0', '3.0']])
```

Of course you can perform matrix multiplication, if the dimensions are compatible:

```
>>> A * B
matrix(
[['8.0', '22.0'],
 ['14.0', '48.0']])
>>> matrix([[1, 2, 3]]) * matrix([[-6], [7], [-2]])
matrix(
[['2.0']])
```

You can raise powers of square matrices:

```
>>> A**2
matrix(
[['7.0', '10.0'],
 ['15.0', '22.0']])
```

Negative powers will calculate the inverse:

```
>>> A**-1
matrix(
[['-2.0', '1.0'],
 ['1.5', '-0.5']])
>>> nprint(A * A**-1, 3)
[      1.0  1.08e-19]
[-2.17e-19        1.0]
```

Matrix transposition is straightforward:

```
>>> A = ones(2, 3)
>>> A
matrix(
```

```
[['1.0', '1.0', '1.0'],
 ['1.0', '1.0', '1.0']])
>>> A.T
matrix(
[['1.0', '1.0'],
 ['1.0', '1.0'],
 ['1.0', '1.0']])
```

Norms

Sometimes you need to know how "large" a matrix or vector is. Due to their multidimensional nature it's not possible to compare them, but there are several functions to map a matrix or a vector to a positive real number, the so called norms.

mpmath.**norm**(*ctx*, *x*, *p=2*)

Gives the entrywise p-norm of an iterable x, i.e. the vector norm $(\sum_k |x_k|^p)^{1/p}$, for any given $1 \le p \le \infty$.

Special cases:

If x is not iterable, this just returns `absmax(x)`.

`p=1` gives the sum of absolute values.

`p=2` is the standard Euclidean vector norm.

`p=inf` gives the magnitude of the largest element.

For x a matrix, `p=2` is the Frobenius norm. For operator matrix norms, use *mnorm()* instead.

You can use the string 'inf' as well as float('inf') or mpf('inf') to specify the infinity norm.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> x = matrix([-10, 2, 100])
>>> norm(x, 1)
mpf('112.0')
>>> norm(x, 2)
mpf('100.5186549850325')
>>> norm(x, inf)
mpf('100.0')
```

mpmath.**mnorm**(*ctx*, *A*, *p=1*)

Gives the matrix (operator) p-norm of A. Currently `p=1` and `p=inf` are supported:

`p=1` gives the 1-norm (maximal column sum)

`p=inf` gives the ∞-norm (maximal row sum). You can use the string 'inf' as well as float('inf') or mpf('inf')

`p=2` (not implemented) for a square matrix is the usual spectral matrix norm, i.e. the largest singular value.

`p='f'` (or 'F', 'fro', 'Frobenius, 'frobenius') gives the Frobenius norm, which is the elementwise 2-norm. The Frobenius norm is an approximation of the spectral norm and satisfies

$$\frac{1}{\sqrt{\mathrm{rank}(A)}}\|A\|_F \le \|A\|_2 \le \|A\|_F$$

The Frobenius norm lacks some mathematical properties that might be expected of a norm.

For general elementwise p-norms, use *norm()* instead.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> A = matrix([[1, -1000], [100, 50]])
>>> mnorm(A, 1)
mpf('1050.0')
>>> mnorm(A, inf)
mpf('1001.0')
>>> mnorm(A, 'F')
mpf('1006.2310867787777')
```

3.3.3 Linear algebra

Decompositions

mpmath.**cholesky**(*ctx*, *A*, *tol=None*)

Cholesky decomposition of a symmetric positive-definite matrix A. Returns a lower triangular matrix L such that $A = L \times L^T$. More generally, for a complex Hermitian positive-definite matrix, a Cholesky decomposition satisfying $A = L \times L^H$ is returned.

The Cholesky decomposition can be used to solve linear equation systems twice as efficiently as LU decomposition, or to test whether A is positive-definite.

The optional parameter `tol` determines the tolerance for verifying positive-definiteness.

Examples

Cholesky decomposition of a positive-definite symmetric matrix:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> A = eye(3) + hilbert(3)
>>> nprint(A)
[     2.0      0.5  0.333333]
[     0.5  1.33333      0.25]
[0.333333     0.25       1.2]
>>> L = cholesky(A)
>>> nprint(L)
[ 1.41421      0.0      0.0]
[0.353553  1.09924      0.0]
[0.235702  0.15162  1.05899]
>>> chop(A - L*L.T)
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
```

Cholesky decomposition of a Hermitian matrix:

```
>>> A = eye(3) + matrix([[0,0.25j,-0.5j],[-0.25j,0,0],[0.5j,0,0]])
>>> L = cholesky(A)
>>> nprint(L)
[          1.0                0.0                0.0]
[(0.0 - 0.25j)  (0.968246 + 0.0j)                0.0]
[ (0.0 + 0.5j)  (0.129099 + 0.0j)  (0.856349 + 0.0j)]
>>> chop(A - L*L.H)
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
```

Attempted Cholesky decomposition of a matrix that is not positive definite:

```
>>> A = -eye(3) + hilbert(3)
>>> L = cholesky(A)
Traceback (most recent call last):
  ...
ValueError: matrix is not positive-definite
```

References

1. *[Wikipedia]* http://en.wikipedia.org/wiki/Cholesky_decomposition

Linear equations

Basic linear algebra is implemented; you can for example solve the linear equation system:

```
  x + 2*y = -10
3*x + 4*y =  10
```

using `lu_solve`:

```
>>> A = matrix([[1, 2], [3, 4]])
>>> b = matrix([-10, 10])
>>> x = lu_solve(A, b)
>>> x
matrix(
[['30.0'],
 ['-20.0']])
```

If you don't trust the result, use `residual` to calculate the residual ||A*x-b||:

```
>>> residual(A, x, b)
matrix(
[['3.46944695195361e-18'],
 ['3.46944695195361e-18']])
>>> str(eps)
'2.22044604925031e-16'
```

As you can see, the solution is quite accurate. The error is caused by the inaccuracy of the internal floating point arithmetic. Though, it's even smaller than the current machine epsilon, which basically means you can trust the result.

If you need more speed, use NumPy, or use `fp` instead `mp` matrices and methods:

```
>>> A = fp.matrix([[1, 2], [3, 4]])
>>> b = fp.matrix([-10, 10])
>>> fp.lu_solve(A, b)
matrix(
[['30.0'],
 ['-20.0']])
```

`lu_solve` accepts overdetermined systems. It is usually not possible to solve such systems, so the residual is minimized instead. Internally this is done using Cholesky decomposition to compute a least squares approximation. This means that that `lu_solve` will square the errors. If you can't afford this, use `qr_solve` instead. It is twice as slow but more accurate, and it calculates the residual automatically.

Matrix factorization

The function `lu` computes an explicit LU factorization of a matrix:

```
>>> P, L, U = lu(matrix([[0,2,3],[4,5,6],[7,8,9]]))
>>> print P
[0.0  0.0  1.0]
[1.0  0.0  0.0]
[0.0  1.0  0.0]
>>> print L
[              1.0                0.0  0.0]
[              0.0                1.0  0.0]
[0.571428571428571  0.214285714285714  1.0]
>>> print U
[7.0  8.0                9.0]
[0.0  2.0                3.0]
[0.0  0.0  0.214285714285714]
>>> print P.T*L*U
[0.0  2.0  3.0]
[4.0  5.0  6.0]
[7.0  8.0  9.0]
```

The function `qr` computes a QR factorization of a matrix:

```
>>> A = matrix([[1, 2], [3, 4], [1, 1]])
>>> Q, R = qr(A)
>>> print Q
[-0.301511344577764   0.861640436855329   0.408248290463863]
[-0.904534033733291  -0.123091490979333  -0.408248290463863]
[-0.301511344577764  -0.492365963917331   0.816496580927726]
>>> print R
[-3.3166247903554  -4.52267016866645]
[             0.0  0.738548945875996]
[             0.0                0.0]
>>> print Q * R
[1.0  2.0]
[3.0  4.0]
[1.0  1.0]
>>> print chop(Q.T * Q)
[1.0  0.0  0.0]
[0.0  1.0  0.0]
[0.0  0.0  1.0]
```

The singular value decomposition

The routines `svd_r` and `svd_c` compute the singular value decomposition of a real or complex matrix A. `svd` is an unified interface calling either `svd_r` or `svd_c` depending on whether A is real or complex.

Given A, two orthogonal (A real) or unitary (A complex) matrices U and V are calculated such that

$$A = USV, \quad U'U = 1, \quad VV' = 1$$

where S is a suitable shaped matrix whose off-diagonal elements are zero. Here ' denotes the hermitian transpose (i.e. transposition and complex conjugation). The diagonal elements of S are the singular values of A, i.e. the square roots of the eigenvalues of $A'A$ or AA'.

Examples:

```
>>> from mpmath import mp
>>> A = mp.matrix([[2, -2, -1], [3, 4, -2], [-2, -2, 0]])
>>> S = mp.svd_r(A, compute_uv = False)
>>> print S
```

```
[6.0]
[3.0]
[1.0]
>>> U, S, V = mp.svd_r(A)
>>> print mp.chop(A - U * mp.diag(S) * V)
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
```

The Schur decomposition

This routine computes the Schur decomposition of a square matrix A. Given A, a unitary matrix Q is determined such that

$$Q'AQ = R, \quad Q'Q = QQ' = 1$$

where R is an upper right triangular matrix. Here ' denotes the hermitian transpose (i.e. transposition and conjugation).

Examples:

```
>>> from mpmath import mp
>>> A = mp.matrix([[3, -1, 2], [2, 5, -5], [-2, -3, 7]])
>>> Q, R = mp.schur(A)
>>> mp.nprint(R, 3)
[2.0  0.417  -2.53]
[0.0    4.0  -4.74]
[0.0    0.0    9.0]
>>> print(mp.chop(A - Q * R * Q.transpose_conj()))
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
```

The eigenvalue problem

The routine `eig` solves the (ordinary) eigenvalue problem for a real or complex square matrix A. Given A, a vector E and matrices ER and EL are calculated such that

```
        A ER[:,i] =          E[i] ER[:,i]
EL[i,:] A         = EL[i,:] E[i]
```

E contains the eigenvalues of A. The columns of ER contain the right eigenvectors of A whereas the rows of EL contain the left eigenvectors.

Examples:

```
>>> from mpmath import mp
>>> A = mp.matrix([[3, -1, 2], [2, 5, -5], [-2, -3, 7]])
>>> E, ER = mp.eig(A)
>>> print(mp.chop(A * ER[:,0] - E[0] * ER[:,0]))
[0.0]
[0.0]
[0.0]
>>> E, EL, ER = mp.eig(A,left = True, right = True)
>>> E, EL, ER = mp.eig_sort(E, EL, ER)
>>> mp.nprint(E)
[2.0, 4.0, 9.0]
>>> print(mp.chop(A * ER[:,0] - E[0] * ER[:,0]))
```

```
[0.0]
[0.0]
[0.0]
>>> print(mp.chop( EL[0,:] * A - EL[0,:] * E[0]))
[0.0  0.0  0.0]
```

The symmetric eigenvalue problem

The routines `eigsy` and `eighe` solve the (ordinary) eigenvalue problem for a real symmetric or complex hermitian square matrix *A*. `eigh` is an unified interface for this two functions calling either `eigsy` or `eighe` depending on whether *A* is real or complex.

Given *A*, an orthogonal (*A* real) or unitary matrix *Q* (*A* complex) is calculated which diagonalizes A:

$$Q'AQ = \mathrm{diag}(E), \quad QQ' = Q'Q = 1$$

Here diag(*E*) a is diagonal matrix whose diagonal is *E*. ' denotes the hermitian transpose (i.e. ordinary transposition and complex conjugation).

The columns of *Q* are the eigenvectors of *A* and *E* contains the eigenvalues:

```
A Q[:,i] = E[i] Q[:,i]
```

Examples:

```
>>> from mpmath import mp
>>> A = mp.matrix([[3, 2], [2, 0]])
>>> E = mp.eigsy(A, eigvals_only = True)
>>> print E
[-1.0]
[ 4.0]
>>> A = mp.matrix([[1, 2], [2, 3]])
>>> E, Q = mp.eigsy(A)                      # alternative: E, Q = mp.eigh(A)
>>> print mp.chop(A * Q[:,0] - E[0] * Q[:,0])
[0.0]
[0.0]
>>> A = mp.matrix([[1, 2 + 5j], [2 - 5j, 3]])
>>> E, Q = mp.eighe(A)                      # alternative: E, Q = mp.eigh(A)
>>> print mp.chop(A * Q[:,0] - E[0] * Q[:,0])
[0.0]
[0.0]
```

3.3.4 Interval and double-precision matrices

The `iv.matrix` and `fp.matrix` classes convert inputs to intervals and Python floating-point numbers respectively.

Interval matrices can be used to perform linear algebra operations with rigorous error tracking:

```
>>> a = iv.matrix([['0.1','0.3','1.0'],
...                ['7.1','5.5','4.8'],
...                ['3.2','4.4','5.6']])
>>>
>>> b = iv.matrix(['4','0.6','0.5'])
>>> c = iv.lu_solve(a, b)
>>> print c
[  [5.2582327113062393041, 5.2582327113062749951]]
[[-13.155049396267856583, -13.155049396267821167]]
```

```
[  [7.4206915477497212555, 7.4206915477497310922]]
>>> print a*c
[  [3.9999999999999866773, 4.0000000000000133227]]
[[0.59999999999972430942, 0.60000000000027142733]]
[[0.49999999999982236432, 0.50000000000018474111]]
```

3.3.5 Matrix functions

mpmath.**expm**(*ctx*, *A*, *method='taylor'*)

Computes the matrix exponential of a square matrix A, which is defined by the power series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots$$

With method='taylor', the matrix exponential is computed using the Taylor series. With method='pade', Pade approximants are used instead.

Examples

Basic examples:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> expm(zeros(3))
[1.0  0.0  0.0]
[0.0  1.0  0.0]
[0.0  0.0  1.0]
>>> expm(eye(3))
[2.71828182845905               0.0               0.0]
[             0.0  2.71828182845905               0.0]
[             0.0               0.0  2.71828182845905]
>>> expm([[1,1,0],[1,0,1],[0,1,0]])
[ 3.86814500615414  2.26812870852145  0.841130841230196]
[ 2.26812870852145  2.44114713886289   1.42699786729125]
[0.841130841230196  1.42699786729125    1.6000162976327]
>>> expm([[1,1,0],[1,0,1],[0,1,0]], method='pade')
[ 3.86814500615414  2.26812870852145  0.841130841230196]
[ 2.26812870852145  2.44114713886289   1.42699786729125]
[0.841130841230196  1.42699786729125    1.6000162976327]
>>> expm([[1+j, 0], [1+j,1]])
[(1.46869393991589 + 2.28735528717884j)                          0.0]
[  (1.03776739863568 + 3.536943175722j)  (2.71828182845905 + 0.0j)]
```

Matrices with large entries are allowed:

```
>>> expm(matrix([[1,2],[2,3]])**25)
[5.65024064048415e+2050488462815550  9.14228140091932e+2050488462815550]
[9.14228140091932e+2050488462815550  1.47925220414035e+2050488462815551]
```

The identity $\exp(A+B) = \exp(A)\exp(B)$ does not hold for noncommuting matrices:

```
>>> A = hilbert(3)
>>> B = A + eye(3)
>>> chop(mnorm(A*B - B*A))
0.0
>>> chop(mnorm(expm(A+B) - expm(A)*expm(B)))
0.0
>>> B = A + ones(3)
```

```
>>> mnorm(A*B - B*A)
1.8
>>> mnorm(expm(A+B) - expm(A)*expm(B))
42.0927851137247
```

mpmath.**cosm**(*ctx*, *A*)

Gives the cosine of a square matrix A, defined in analogy with the matrix exponential.

Examples:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> cosm(X)
[0.54030230586814               0.0               0.0]
[             0.0  0.54030230586814               0.0]
[             0.0               0.0  0.54030230586814]
>>> X = hilbert(3)
>>> cosm(X)
[ 0.424403834569555  -0.316643413047167  -0.221474945949293]
[-0.316643413047167   0.820646708837824  -0.127183694770039]
[-0.221474945949293  -0.127183694770039   0.909236687217541]
>>> X = matrix([[1+j,-2],[0,-j]])
>>> cosm(X)
[(0.833730025131149 - 0.988897705762865j)  (1.07485840848393 - 0.171921405442l3j)]
[                                     0.0               (1.54308063481524 + 0.0j)]
```

mpmath.**sinm**(*ctx*, *A*)

Gives the sine of a square matrix A, defined in analogy with the matrix exponential.

Examples:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> sinm(X)
[0.841470984807897                0.0                0.0]
[              0.0  0.841470984807897                0.0]
[              0.0                0.0  0.841470984807897]
>>> X = hilbert(3)
>>> sinm(X)
[0.711608512150994  0.339783913247439  0.220742837314741]
[0.339783913247439  0.244113865695532  0.187231271174372]
[0.220742837314741  0.187231271174372  0.155816730769635]
>>> X = matrix([[1+j,-2],[0,-j]])
>>> sinm(X)
[(1.29845758141598 + 0.634963914784736j)  (-1.96751511930922 + 0.3147000217613671)]
[                                    0.0                   (0.0 - 1.17520119364381)]
```

mpmath.**sqrtm**(*ctx*, *A*, *_may_rotate=2*)

Computes a square root of the square matrix A, i.e. returns a matrix $B = A^{1/2}$ such that $B^2 = A$. The square root of a matrix, if it exists, is not unique.

Examples

Square roots of some simple matrices:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sqrtm([[1,0], [0,1]])
```

```
[1.0  0.0]
[0.0  1.0]
>>> sqrtm([[0,0], [0,0]])
[0.0  0.0]
[0.0  0.0]
>>> sqrtm([[2,0],[0,1]])
[1.4142135623731  0.0]
[            0.0  1.0]
>>> sqrtm([[1,1],[1,0]])
[ (0.920442065259926 - 0.217286896751641j)  (0.568864481005783 + 0.351577584254143j)]
[(0.568864481005783 + 0.351577584254143j)  (0.351577584254143 - 0.568864481005783j)]
>>> sqrtm([[1,0],[0,1]])
[1.0  0.0]
[0.0  1.0]
>>> sqrtm([[-1,0],[0,1]])
[(0.0 - 1.0j)           0.0]
[         0.0  (1.0 + 0.0j)]
>>> sqrtm([[j,0],[0,j]])
[(0.707106781186547 + 0.707106781186547j)                                    0.0]
[                                     0.0  (0.707106781186547 + 0.707106781186547j)]
```

A square root of a rotation matrix, giving the corresponding half-angle rotation matrix:

```
>>> t1 = 0.75
>>> t2 = t1 * 0.5
>>> A1 = matrix([[cos(t1), -sin(t1)], [sin(t1), cos(t1)]])
>>> A2 = matrix([[cos(t2), -sin(t2)], [sin(t2), cos(t2)]])
>>> sqrtm(A1)
[0.930507621912314  -0.366272529086048]
[0.366272529086048   0.930507621912314]
>>> A2
[0.930507621912314  -0.366272529086048]
[0.366272529086048   0.930507621912314]
```

The identity $(A^2)^{1/2} = A$ does not necessarily hold:

```
>>> A = matrix([[4,1,4],[7,8,9],[10,2,11]])
>>> sqrtm(A**2)
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> sqrtm(A)**2
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> A = matrix([[-4,1,4],[7,-8,9],[10,2,11]])
>>> sqrtm(A**2)
[  7.43715112194995  -0.324127569985474   1.8481718827526]
[-0.251549715716942   9.32699765900402  2.48221180985147]
[  4.11609388833616  0.775751877098258   13.017955697342]
>>> chop(sqrtm(A)**2)
[-4.0   1.0   4.0]
[ 7.0  -8.0   9.0]
[10.0   2.0  11.0]
```

For some matrices, a square root does not exist:

```
>>> sqrtm([[0,1], [0,0]])
Traceback (most recent call last):
```

```
  ...
ZeroDivisionError: matrix is numerically singular
```

Two examples from the documentation for Matlab's `sqrtm`:

```
>>> mp.dps = 15; mp.pretty = True
>>> sqrtm([[7,10],[15,22]])
[1.56669890360128  1.74077655955698]
[2.61116483933547  4.17786374293675]
>>>
>>> X = matrix(\
...   [[5,-4,1,0,0],
...   [-4,6,-4,1,0],
...   [1,-4,6,-4,1],
...   [0,1,-4,6,-4],
...   [0,0,1,-4,5]])
>>> Y = matrix(\
...   [[2,-1,-0,-0,-0],
...   [-1,2,-1,0,-0],
...   [0,-1,2,-1,0],
...   [-0,0,-1,2,-1],
...   [-0,-0,-0,-1,2]])
>>> mnorm(sqrtm(X) - Y)
4.53155328326114e-19
```

mpmath.**logm**(*ctx*, *A*)

Computes a logarithm of the square matrix A, i.e. returns a matrix $B = \log(A)$ such that $\exp(B) = A$. The logarithm of a matrix, if it exists, is not unique.

Examples

Logarithms of some simple matrices:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> logm(X)
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
>>> logm(2*X)
[0.693147180559945                0.0                0.0]
[              0.0  0.693147180559945                0.0]
[              0.0                0.0  0.693147180559945]
>>> logm(expm(X))
[1.0  0.0  0.0]
[0.0  1.0  0.0]
[0.0  0.0  1.0]
```

A logarithm of a complex matrix:

```
>>> X = matrix([[2+j, 1, 3], [1-j, 1-2*j, 1], [-4, -5, j]])
>>> B = logm(X)
>>> nprint(B)
[ (0.808757 + 0.107759j)     (2.20752 + 0.202762j)    (1.07376 - 0.773874j)]
[ (0.905709 - 0.107795j)  (0.0287395 - 0.824993j)   (0.111619 + 0.514272j)]
[(-0.930151 + 0.399512j)    (-2.06266 - 0.674397j)   (0.791552 + 0.519839j)]
>>> chop(expm(B))
[(2.0 + 1.0j)           1.0           3.0]
[(1.0 - 1.0j)  (1.0 - 2.0j)           1.0]
```

```
[        -4.0          -5.0   (0.0 + 1.0j)]
```

A matrix X close to the identity matrix, for which $\log(\exp(X)) = \exp(\log(X)) = X$ holds:

```
>>> X = eye(3) + hilbert(3)/4
>>> X
[              1.25             0.125  0.0833333333333333]
[             0.125  1.08333333333333             0.0625]
[0.0833333333333333            0.0625               1.05]
>>> logm(expm(X))
[              1.25             0.125  0.0833333333333333]
[             0.125  1.08333333333333             0.0625]
[0.0833333333333333            0.0625               1.05]
>>> expm(logm(X))
[              1.25             0.125  0.0833333333333333]
[             0.125  1.08333333333333             0.0625]
[0.0833333333333333            0.0625               1.05]
```

A logarithm of a rotation matrix, giving back the angle of the rotation:

```
>>> t = 3.7
>>> A = matrix([[cos(t),sin(t)],[-sin(t),cos(t)]])
>>> chop(logm(A))
[             0.0  -2.58318530717959]
[2.58318530717959               0.0]
>>> (2*pi-t)
2.58318530717959
```

For some matrices, a logarithm does not exist:

```
>>> logm([[1,0], [0,0]])
Traceback (most recent call last):
  ...
ZeroDivisionError: matrix is numerically singular
```

Logarithm of a matrix with large entries:

```
>>> logm(hilbert(3) * 10**20).apply(re)
[ 45.5597513593433  1.27721006042799  0.317662687717978]
[ 1.27721006042799  42.5222778973542   2.24003708791604]
[0.317662687717978  2.24003708791604    42.395212822267]
```

mpmath.**powm**(*ctx*, *A*, *r*)

Computes $A^r = \exp(A \log r)$ for a matrix A and complex number r.

Examples

Powers and inverse powers of a matrix:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> A = matrix([[4,1,4],[7,8,9],[10,2,11]])
>>> powm(A, 2)
[ 63.0  20.0   69.0]
[174.0  89.0  199.0]
[164.0  48.0  179.0]
>>> chop(powm(powm(A, 4), 1/4.))
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
```

```
>>> powm(extraprec(20)(powm)(A, -4), -1/4.)
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> chop(powm(powm(A, 1+0.5j), 1/(1+0.5j)))
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> powm(extraprec(5)(powm)(A, -1.5), -1/(1.5))
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
```

A Fibonacci-generating matrix:

```
>>> powm([[1,1],[1,0]], 10)
[89.0  55.0]
[55.0  34.0]
>>> fib(10)
55.0
>>> powm([[1,1],[1,0]], 6.5)
[(16.5166626964253 - 0.0121089837381789j)  (10.2078589271083 + 0.0195927472575932j)]
[(10.2078589271083 + 0.0195927472575932j)  (6.30880376931698 - 0.0317017309957721j)]
>>> (phi**6.5 - (1-phi)**6.5)/sqrt(5)
(10.2078589271083 - 0.0195927472575932j)
>>> powm([[1,1],[1,0]], 6.2)
[ (14.3076953002666 - 0.008222855781077j)  (8.81733464837593 + 0.0133048601383712j)]
[(8.81733464837593 + 0.0133048601383712j)  (5.49036065189071 - 0.0215277159194482j)]
>>> (phi**6.2 - (1-phi)**6.2)/sqrt(5)
(8.81733464837593 - 0.0133048601383712j)
```

3.4 Number identification

Most function in mpmath are concerned with producing approximations from exact mathematical formulas. It is also useful to consider the inverse problem: given only a decimal approximation for a number, such as 0.73205080756887729352744634, is it possible to find an exact formula?

Subject to certain restrictions, such "reverse engineering" is indeed possible thanks to the existence of *integer relation algorithms*. Mpmath implements the PSLQ algorithm (developed by H. Ferguson), which is one such algorithm.

Automated number recognition based on PSLQ is not a silver bullet. Any occurring transcendental constants (π, e, etc) must be guessed by the user, and the relation between those constants in the formula must be linear (such as $x = 3\pi + 4e$). More complex formulas can be found by combining PSLQ with functional transformations; however, this is only feasible to a limited extent since the computation time grows exponentially with the number of operations that need to be combined.

The number identification facilities in mpmath are inspired by the Inverse Symbolic Calculator (ISC). The ISC is more powerful than mpmath, as it uses a lookup table of millions of precomputed constants (thereby mitigating the problem with exponential complexity).

3.4.1 Constant recognition

identify()

mpmath.**identify**(*ctx*, *x*, *constants=[]*, *tol=None*, *maxcoeff=1000*, *full=False*, *verbose=False*)

Given a real number x, `identify(x)` attempts to find an exact formula for x. This formula is returned as a string. If no match is found, `None` is returned. With `full=True`, a list of matching formulas is returned.

As a simple example, *identify()* will find an algebraic formula for the golden ratio:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> identify(phi)
'((1+sqrt(5))/2)'
```

identify() can identify simple algebraic numbers and simple combinations of given base constants, as well as certain basic transformations thereof. More specifically, *identify()* looks for the following:

1. Fractions
2. Quadratic algebraic numbers
3. Rational linear combinations of the base constants
4. Any of the above after first transforming x into $f(x)$ where $f(x)$ is $1/x$, $\sqrt{x}$, x^2, $\log x$ or $\exp x$, either directly or with x or $f(x)$ multiplied or divided by one of the base constants
5. Products of fractional powers of the base constants and small integers

Base constants can be given as a list of strings representing mpmath expressions (*identify()* will `eval` the strings to numerical values and use the original strings for the output), or as a dict of formula:value pairs.

In order not to produce spurious results, *identify()* should be used with high precision; preferably 50 digits or more.

Examples

Simple identifications can be performed safely at standard precision. Here the default recognition of rational, algebraic, and exp/log of algebraic numbers is demonstrated:

```
>>> mp.dps = 15
>>> identify(0.22222222222222222)
'(2/9)'
>>> identify(1.9662210973805663)
'sqrt(((24+sqrt(48))/8))'
>>> identify(4.1132503787829275)
'exp((sqrt(8)/2))'
>>> identify(0.881373587019543)
'log(((2+sqrt(8))/2))'
```

By default, *identify()* does not recognize π. At standard precision it finds a not too useful approximation. At slightly increased precision, this approximation is no longer accurate enough and *identify()* more correctly returns `None`:

```
>>> identify(pi)
'(2**(176/117)*3**(20/117)*5**(35/39))/(7**(92/117))'
>>> mp.dps = 30
>>> identify(pi)
>>>
```

Numbers such as π, and simple combinations of user-defined constants, can be identified if they are provided explicitly:

```
>>> identify(3*pi-2*e, ['pi', 'e'])
'(3*pi + (-2)*e)'
```

Here is an example using a dict of constants. Note that the constants need not be "atomic"; *identify()* can just as well express the given number in terms of expressions given by formulas:

```
>>> identify(pi+e, {'a':pi+2, 'b':2*e})
'((-2) + 1*a + (1/2)*b)'
```

Next, we attempt some identifications with a set of base constants. It is necessary to increase the precision a bit.

```
>>> mp.dps = 50
>>> base = ['sqrt(2)','pi','log(2)']
>>> identify(0.25, base)
'(1/4)'
>>> identify(3*pi + 2*sqrt(2) + 5*log(2)/7, base)
'(2*sqrt(2) + 3*pi + (5/7)*log(2))'
>>> identify(exp(pi+2), base)
'exp((2 + 1*pi))'
>>> identify(1/(3+sqrt(2)), base)
'((3/7) + (-1/7)*sqrt(2))'
>>> identify(sqrt(2)/(3*pi+4), base)
'sqrt(2)/(4 + 3*pi)'
>>> identify(5**(mpf(1)/3)*pi*log(2)**2, base)
'5**(1/3)*pi*log(2)**2'
```

An example of an erroneous solution being found when too low precision is used:

```
>>> mp.dps = 15
>>> identify(1/(3*pi-4*e+sqrt(8)), ['pi', 'e', 'sqrt(2)'])
'((11/25) + (-158/75)*pi + (76/75)*e + (44/15)*sqrt(2))'
>>> mp.dps = 50
>>> identify(1/(3*pi-4*e+sqrt(8)), ['pi', 'e', 'sqrt(2)'])
'1/(3*pi + (-4)*e + 2*sqrt(2))'
```

Finding approximate solutions

The tolerance `tol` defaults to 3/4 of the working precision. Lowering the tolerance is useful for finding approximate matches. We can for example try to generate approximations for pi:

```
>>> mp.dps = 15
>>> identify(pi, tol=1e-2)
'(22/7)'
>>> identify(pi, tol=1e-3)
'(355/113)'
>>> identify(pi, tol=1e-10)
'(5**(339/269))/(2**(64/269)*3**(13/269)*7**(92/269))'
```

With `full=True`, and by supplying a few base constants, `identify` can generate almost endless lists of approximations for any number (the output below has been truncated to show only the first few):

```
>>> for p in identify(pi, ['e', 'catalan'], tol=1e-5, full=True):
...     print(p)
...
e/log((6 + (-4/3)*e))
(3**3*5*e*catalan**2)/(2*7**2)
sqrt(((-13) + 1*e + 22*catalan))
log(((-6) + 24*e + 4*catalan)/e)
exp(catalan*((-1/5) + (8/15)*e))
catalan*(6 + (-6)*e + 15*catalan)
```

```
sqrt((5 + 26*e + (-3)*catalan))/e
e*sqrt(((-27) + 2*e + 25*catalan))
log(((-1) + (-11)*e + 59*catalan))
((3/20) + (21/20)*e + (3/20)*catalan)
...
```

The numerical values are roughly as close to π as permitted by the specified tolerance:

```
>>> e/log(6-4*e/3)
3.14157719846001
>>> 135*e*catalan**2/98
3.14166950419369
>>> sqrt(e-13+22*catalan)
3.14158000062992
>>> log(24*e-6+4*catalan)-1
3.14158791577159
```

Symbolic processing

The output formula can be evaluated as a Python expression. Note however that if fractions (like '2/3') are present in the formula, Python's `eval()` may erroneously perform integer division. Note also that the output is not necessarily in the algebraically simplest form:

```
>>> identify(sqrt(2))
'(sqrt(8)/2)'
```

As a solution to both problems, consider using SymPy's `sympify()` to convert the formula into a symbolic expression. SymPy can be used to pretty-print or further simplify the formula symbolically:

```
>>> from sympy import sympify
>>> sympify(identify(sqrt(2)))
2**(1/2)
```

Sometimes *identify()* can simplify an expression further than a symbolic algorithm:

```
>>> from sympy import simplify
>>> x = sympify('-1/(-3/2+(1/2)*5**(1/2))*(3/2-1/2*5**(1/2))**(1/2)')
>>> x
(3/2 - 5**(1/2)/2)**(-1/2)
>>> x = simplify(x)
>>> x
2/(6 - 2*5**(1/2))**(1/2)
>>> mp.dps = 30
>>> x = sympify(identify(x.evalf(30)))
>>> x
1/2 + 5**(1/2)/2
```

(In fact, this functionality is available directly in SymPy as the function `nsimplify()`, which is essentially a wrapper for *identify()*.)

Miscellaneous issues and limitations

The input x must be a real number. All base constants must be positive real numbers and must not be rationals or rational linear combinations of each other.

The worst-case computation time grows quickly with the number of base constants. Already with 3 or 4 base constants, *identify()* may require several seconds to finish. To search for relations among a large number of constants, you should consider using *pslq()* directly.

The extended transformations are applied to x, not the constants separately. As a result, `identify` will for example be able to recognize `exp(2*pi+3)` with `pi` given as a base constant, but not `2*exp(pi)+3`. It will

be able to recognize the latter if `exp(pi)` is given explicitly as a base constant.

3.4.2 Algebraic identification

findpoly()

mpmath.**findpoly**(*ctx*, *x*, *n=1*, ***kwargs*)

`findpoly(x, n)` returns the coefficients of an integer polynomial P of degree at most n such that $P(x) \approx 0$. If no polynomial having x as a root can be found, *findpoly()* returns `None`.

findpoly() works by successively calling *pslq()* with the vectors $[1, x]$, $[1, x, x^2]$, $[1, x, x^2, x^3]$, ..., $[1, x, x^2, .., x^n]$ as input. Keyword arguments given to *findpoly()* are forwarded verbatim to *pslq()*. In particular, you can specify a tolerance for $P(x)$ with `tol` and a maximum permitted coefficient size with `maxcoeff`.

For large values of n, it is recommended to run *findpoly()* at high precision; preferably 50 digits or more.

Examples

By default (degree $n = 1$), *findpoly()* simply finds a linear polynomial with a rational root:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> findpoly(0.7)
[-10, 7]
```

The generated coefficient list is valid input to `polyval` and `polyroots`:

```
>>> nprint(polyval(findpoly(phi, 2), phi), 1)
-2.0e-16
>>> for r in polyroots(findpoly(phi, 2)):
...     print(r)
...
-0.618033988749895
1.61803398874989
```

Numbers of the form $m + n\sqrt{p}$ for integers (m, n, p) are solutions to quadratic equations. As we find here, $1 + \sqrt{2}$ is a root of the polynomial $x^2 - 2x - 1$:

```
>>> findpoly(1+sqrt(2), 2)
[1, -2, -1]
>>> findroot(lambda x: x**2 - 2*x - 1, 1)
2.4142135623731
```

Despite only containing square roots, the following number results in a polynomial of degree 4:

```
>>> findpoly(sqrt(2)+sqrt(3), 4)
[1, 0, -10, 0, 1]
```

In fact, $x^4 - 10x^2 + 1$ is the *minimal polynomial* of $r = \sqrt{2} + \sqrt{3}$, meaning that a rational polynomial of lower degree having r as a root does not exist. Given sufficient precision, *findpoly()* will usually find the correct minimal polynomial of a given algebraic number.

Non-algebraic numbers

If *findpoly()* fails to find a polynomial with given coefficient size and tolerance constraints, that means no such polynomial exists.

We can verify that π is not an algebraic number of degree 3 with coefficients less than 1000:

```
>>> mp.dps = 15
>>> findpoly(pi, 3)
>>>
```

It is always possible to find an algebraic approximation of a number using one (or several) of the following methods:

1.Increasing the permitted degree

2.Allowing larger coefficients

3.Reducing the tolerance

One example of each method is shown below:

```
>>> mp.dps = 15
>>> findpoly(pi, 4)
[95, -545, 863, -183, -298]
>>> findpoly(pi, 3, maxcoeff=10000)
[836, -1734, -2658, -457]
>>> findpoly(pi, 3, tol=1e-7)
[-4, 22, -29, -2]
```

It is unknown whether Euler's constant is transcendental (or even irrational). We can use *findpoly()* to check that if is an algebraic number, its minimal polynomial must have degree at least 7 and a coefficient of magnitude at least 1000000:

```
>>> mp.dps = 200
>>> findpoly(euler, 6, maxcoeff=10**6, tol=1e-100, maxsteps=1000)
>>>
```

Note that the high precision and strict tolerance is necessary for such high-degree runs, since otherwise unwanted low-accuracy approximations will be detected. It may also be necessary to set maxsteps high to prevent a premature exit (before the coefficient bound has been reached). Running with `verbose=True` to get an idea what is happening can be useful.

3.4.3 Integer relations (PSLQ)

pslq()

mpmath.**pslq**(*ctx*, *x*, *tol=None*, *maxcoeff=1000*, *maxsteps=100*, *verbose=False*)

Given a vector of real numbers $x = [x_0, x_1, ..., x_n]$, `pslq(x)` uses the PSLQ algorithm to find a list of integers $[c_0, c_1, ..., c_n]$ such that

$$|c_1 x_1 + c_2 x_2 + ... + c_n x_n| < \text{tol}$$

and such that $\max |c_k| < \text{maxcoeff}$. If no such vector exists, *pslq()* returns `None`. The tolerance defaults to 3/4 of the working precision.

Examples

Find rational approximations for π:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> pslq([-1, pi], tol=0.01)
[22, 7]
>>> pslq([-1, pi], tol=0.001)
[355, 113]
```

```
>>> mpf(22)/7; mpf(355)/113; +pi
3.14285714285714
3.14159292035398
3.14159265358979
```

Pi is not a rational number with denominator less than 1000:

```
>>> pslq([-1, pi])
>>>
```

To within the standard precision, it can however be approximated by at least one rational number with denominator less than 10^{12}:

```
>>> p, q = pslq([-1, pi], maxcoeff=10**12)
>>> print(p); print(q)
238410049439
75888275702
>>> mpf(p)/q
3.14159265358979
```

The PSLQ algorithm can be applied to long vectors. For example, we can investigate the rational (in)dependence of integer square roots:

```
>>> mp.dps = 30
>>> pslq([sqrt(n) for n in range(2, 5+1)])
>>>
>>> pslq([sqrt(n) for n in range(2, 6+1)])
>>>
>>> pslq([sqrt(n) for n in range(2, 8+1)])
[2, 0, 0, 0, 0, 0, -1]
```

Machin formulas

A famous formula for π is Machin's,

$$\frac{\pi}{4} = 4\,\mathrm{acot}\,5 - \mathrm{acot}\,239$$

There are actually infinitely many formulas of this type. Two others are

$$\frac{\pi}{4} = \mathrm{acot}\,1$$

$$\frac{\pi}{4} = 12\,\mathrm{acot}\,49 + 32\,\mathrm{acot}\,57 + 5\,\mathrm{acot}\,239 + 12\,\mathrm{acot}\,110443$$

We can easily verify the formulas using the PSLQ algorithm:

```
>>> mp.dps = 30
>>> pslq([pi/4, acot(1)])
[1, -1]
>>> pslq([pi/4, acot(5), acot(239)])
[1, -4, 1]
>>> pslq([pi/4, acot(49), acot(57), acot(239), acot(110443)])
[1, -12, -32, 5, -12]
```

We could try to generate a custom Machin-like formula by running the PSLQ algorithm with a few inverse cotangent values, for example acot(2), acot(3) ... acot(10). Unfortunately, there is a linear dependence among these values, resulting in only that dependence being detected, with a zero coefficient for π:

```
>>> pslq([pi] + [acot(n) for n in range(2,11)])
[0, 1, -1, 0, 0, 0, -1, 0, 0, 0]
```

We get better luck by removing linearly dependent terms:

```
>>> pslq([pi] + [acot(n) for n in range(2,11) if n not in (3, 5)])
[1, -8, 0, 0, 4, 0, 0, 0]
```

In other words, we found the following formula:

```
>>> 8*acot(2) - 4*acot(7)
3.1415926535897932384626433832795
>>> +pi
3.1415926535897932384626433832795
```

Algorithm

This is a fairly direct translation to Python of the pseudocode given by David Bailey, "The PSLQ Integer Relation Algorithm": http://www.cecm.sfu.ca/organics/papers/bailey/paper/html/node3.html

The present implementation uses fixed-point instead of floating-point arithmetic, since this is significantly (about 7x) faster.

CHAPTER 4

End matter

4.1 Precision and representation issues

Most of the time, using mpmath is simply a matter of setting the desired precision and entering a formula. For verification purposes, a quite (but not always!) reliable technique is to calculate the same thing a second time at a higher precision and verifying that the results agree.

To perform more advanced calculations, it is important to have some understanding of how mpmath works internally and what the possible sources of error are. This section gives an overview of arbitrary-precision binary floating-point arithmetic and some concepts from numerical analysis.

The following concepts are important to understand:

- The main sources of numerical errors are rounding and cancellation, which are due to the use of finite-precision arithmetic, and truncation or approximation errors, which are due to approximating infinite sequences or continuous functions by a finite number of samples.
- Errors propagate between calculations. A small error in the input may result in a large error in the output.
- Most numerical algorithms for complex problems (e.g. integrals, derivatives) give wrong answers for sufficiently ill-behaved input. Sometimes virtually the only way to get a wrong answer is to design some very contrived input, but at other times the chance of accidentally obtaining a wrong result even for reasonable-looking input is quite high.
- Like any complex numerical software, mpmath has implementation bugs. You should be reasonably suspicious about any results computed by mpmath, even those it claims to be able to compute correctly! If possible, verify results analytically, try different algorithms, and cross-compare with other software.

4.1.1 Precision, error and tolerance

The following terms are common in this documentation:

- *Precision* (or *working precision*) is the precision at which floating-point arithmetic operations are performed.
- *Error* is the difference between a computed approximation and the exact result.
- *Accuracy* is the inverse of error.
- *Tolerance* is the maximum error (or minimum accuracy) desired in a result.

Error and accuracy can be measured either directly, or logarithmically in bits or digits. Specifically, if a $\hat{y}$ is an approximation for y, then

- (Direct) absolute error = $|\hat{y} - y|$

- (Direct) relative error = $|\hat{y} - y||y|^{-1}$
- (Direct) absolute accuracy = $|\hat{y} - y|^{-1}$
- (Direct) relative accuracy = $|\hat{y} - y|^{-1}|y|$
- (Logarithmic) absolute error = $\log_b |\hat{y} - y|$
- (Logarithmic) relative error = $\log_b |\hat{y} - y| - \log_b |y|$
- (Logarithmic) absolute accuracy = $-\log_b |\hat{y} - y|$
- (Logarithmic) relative accuracy = $-\log_b |\hat{y} - y| + \log_b |y|$

where $b = 2$ and $b = 10$ for bits and digits respectively. Note that:

- The logarithmic error roughly equals the position of the first incorrect bit or digit
- The logarithmic accuracy roughly equals the number of correct bits or digits in the result

These definitions also hold for complex numbers, using $|a + bi| = \sqrt{a^2 + b^2}$.

Full accuracy means that the accuracy of a result at least equals *prec*-1, i.e. it is correct except possibly for the last bit.

4.1.2 Representation of numbers

Mpmath uses binary arithmetic. A binary floating-point number is a number of the form $man \times 2^{exp}$ where both *man* (the *mantissa*) and *exp* (the *exponent*) are integers. Some examples of floating-point numbers are given in the following table.

Number	Mantissa	Exponent
3	3	0
10	5	1
-16	-1	4
1.25	5	-2

The representation as defined so far is not unique; one can always multiply the mantissa by 2 and subtract 1 from the exponent with no change in the numerical value. In mpmath, numbers are always normalized so that *man* is an odd number, with the exception of zero which is always taken to have *man = exp = 0*. With these conventions, every representable number has a unique representation. (Mpmath does not currently distinguish between positive and negative zero.)

Simple mathematical operations are now easy to define. Due to uniqueness, equality testing of two numbers simply amounts to separately checking equality of the mantissas and the exponents. Multiplication of nonzero numbers is straightforward: $(m2^e) \times (n2^f) = (mn) \times 2^{e+f}$. Addition is a bit more involved: we first need to multiply the mantissa of one of the operands by a suitable power of 2 to obtain equal exponents.

More technically, mpmath represents a floating-point number as a 4-tuple *(sign, man, exp, bc)* where *sign* is 0 or 1 (indicating positive vs negative) and the mantissa is nonnegative; *bc* (*bitcount*) is the size of the absolute value of the mantissa as measured in bits. Though redundant, keeping a separate sign field and explicitly keeping track of the bitcount significantly speeds up arithmetic (the bitcount, especially, is frequently needed but slow to compute from scratch due to the lack of a Python built-in function for the purpose).

Contrary to popular belief, floating-point *numbers* do not come with an inherent "small uncertainty", although floating-point *arithmetic* generally is inexact. Every binary floating-point number is an exact rational number. With arbitrary-precision integers used for the mantissa and exponent, floating-point numbers can be added, subtracted and multiplied *exactly*. In particular, integers and integer multiples of 1/2, 1/4, 1/8, 1/16, etc. can be represented, added and multiplied exactly in binary floating-point arithmetic.

Floating-point arithmetic is generally approximate because the size of the mantissa must be limited for efficiency reasons. The maximum allowed width (bitcount) of the mantissa is called the precision or *prec* for short. Sums and

products of floating-point numbers are exact as long as the absolute value of the mantissa is smaller than 2^{prec}. As soon as the mantissa becomes larger than this, it is truncated to contain at most *prec* bits (the exponent is incremented accordingly to preserve the magnitude of the number), and this operation introduces a rounding error. Division is also generally inexact; although we can add and multiply exactly by setting the precision high enough, no precision is high enough to represent for example 1/3 exactly (the same obviously applies for roots, trigonometric functions, etc).

The special numbers `+inf`, `-inf` and `nan` are represented internally by a zero mantissa and a nonzero exponent.

Mpmath uses arbitrary precision integers for both the mantissa and the exponent, so numbers can be as large in magnitude as permitted by the computer's memory. Some care may be necessary when working with extremely large numbers. Although standard arithmetic operators are safe, it is for example futile to attempt to compute the exponential function of of 10^{100000}. Mpmath does not complain when asked to perform such a calculation, but instead chugs away on the problem to the best of its ability, assuming that computer resources are infinite. In the worst case, this will be slow and allocate a huge amount of memory; if entirely impossible Python will at some point raise `OverflowError: long int too large to convert to int`.

For further details on how the arithmetic is implemented, refer to the mpmath source code. The basic arithmetic operations are found in the `libmp` directory; many functions there are commented extensively.

4.1.3 Decimal issues

Mpmath uses binary arithmetic internally, while most interaction with the user is done via the decimal number system. Translating between binary and decimal numbers is a somewhat subtle matter; many Python novices run into the following "bug" (addressed in the General Python FAQ):

```
>>> 1.2 - 1.0
0.19999999999999996
```

Decimal fractions fall into the category of numbers that generally cannot be represented exactly in binary floating-point form. For example, none of the numbers 0.1, 0.01, 0.001 has an exact representation as a binary floating-point number. Although mpmath can approximate decimal fractions with any accuracy, it does not solve this problem for all uses; users who need *exact* decimal fractions should look at the `decimal` module in Python's standard library (or perhaps use fractions, which are much faster).

With *prec* bits of precision, an arbitrary number can be approximated relatively to within 2^{-prec}, or within 10^{-dps} for *dps* decimal digits. The equivalent values for *prec* and *dps* are therefore related proportionally via the factor $C = \log(10)/\log(2)$, or roughly 3.32. For example, the standard (binary) precision in mpmath is 53 bits, which corresponds to a decimal precision of 15.95 digits.

More precisely, mpmath uses the following formulas to translate between *prec* and *dps*:

```
dps(prec) = max(1, int(round(int(prec) / C - 1)))

prec(dps) = max(1, int(round((int(dps) + 1) * C)))
```

Note that the dps is set 1 decimal digit lower than the corresponding binary precision. This is done to hide minor rounding errors and artifacts resulting from binary-decimal conversion. As a result, mpmath interprets 53 bits as giving 15 digits of decimal precision, not 16.

The *dps* value controls the number of digits to display when printing numbers with `str()`, while the decimal precision used by `repr()` is set two or three digits higher. For example, with 15 dps we have:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> str(pi)
'3.14159265358979'
>>> repr(+pi)
"mpf('3.1415926535897931')"
```

The extra digits in the output from `repr` ensure that `x == eval(repr(x))` holds, i.e. that numbers can be converted to strings and back losslessly.

It should be noted that precision and accuracy do not always correlate when translating between binary and decimal. As a simple example, the number 0.1 has a decimal precision of 1 digit but is an infinitely accurate representation of 1/10. Conversely, the number 2^{-50} has a binary representation with 1 bit of precision that is infinitely accurate; the same number can actually be represented exactly as a decimal, but doing so requires 35 significant digits:

```
0.00000000000000088817841970012523233890533447265625
```

All binary floating-point numbers can be represented exactly as decimals (possibly requiring many digits), but the converse is false.

4.1.4 Correctness guarantees

Basic arithmetic operations (with the `mp` context) are always performed with correct rounding. Results that can be represented exactly are guranteed to be exact, and results from single inexact operations are guaranteed to be the best possible rounded values. For higher-level operations, mpmath does not generally guarantee correct rounding. In general, mpmath only guarantees that it will use at least the user-set precision to perform a given calculation. *The user may have to manually set the working precision higher than the desired accuracy for the result, possibly much higher.*

Functions for evaluation of transcendental functions, linear algebra operations, numerical integration, etc., usually automatically increase the working precision and use a stricter tolerance to give a correctly rounded result with high probability: for example, at 50 bits the temporary precision might be set to 70 bits and the tolerance might be set to 60 bits. It can often be assumed that such functions return values that have full accuracy, given inputs that are exact (or sufficiently precise approximations of exact values), but the user must exercise judgement about whether to trust mpmath.

The level of rigor in mpmath covers the entire spectrum from "always correct by design" through "nearly always correct" and "handling the most common errors" to "just computing blindly and hoping for the best". Of course, a long-term development goal is to successively increase the rigor where possible. The following list might give an idea of the current state.

Operations that are correctly rounded:

- Addition, subtraction and multiplication of real and complex numbers.
- Division and square roots of real numbers.
- Powers of real numbers, assuming sufficiently small integer exponents (huge powers are rounded in the right direction, but possibly farther than necessary).
- Conversion from decimal to binary, for reasonably sized numbers (roughly between 10^{-100} and 10^{100}).
- Typically, transcendental functions for exact input-output pairs.

Operations that should be fully accurate (however, the current implementation may be based on a heuristic error analysis):

- Radix conversion (large or small numbers).
- Mathematical constants like π.
- Both real and imaginary parts of exp, cos, sin, cosh, sinh, log.
- Other elementary functions (the largest of the real and imaginary part).
- The gamma and log-gamma functions (the largest of the real and the imaginary part; both, when close to real axis).
- Some functions based on hypergeometric series (the largest of the real and imaginary part).

Correctness of root-finding, numerical integration, etc. largely depends on the well-behavedness of the input functions. Specific limitations are sometimes noted in the respective sections of the documentation.

4.1.5 Double precision emulation

On most systems, Python's `float` type represents an IEEE 754 *double precision* number, with a precision of 53 bits and rounding-to-nearest. With default precision (`mp.prec = 53`), the mpmath `mpf` type roughly emulates the behavior of the `float` type. Sources of incompatibility include the following:

- In hardware floating-point arithmetic, the size of the exponent is restricted to a fixed range: regular Python floats have a range between roughly 10^{-300} and 10^{300}. Mpmath does not emulate overflow or underflow when exponents fall outside this range.
- On some systems, Python uses 80-bit (extended double) registers for floating-point operations. Due to double rounding, this makes the `float` type less accurate. This problem is only known to occur with Python versions compiled with GCC on 32-bit systems.
- Machine floats very close to the exponent limit round subnormally, meaning that they lose accuracy (Python may raise an exception instead of rounding a `float` subnormally).
- Mpmath is able to produce more accurate results for transcendental functions.

4.1.6 Further reading

There are many excellent textbooks on numerical analysis and floating-point arithmetic. Some good web resources are:

- David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic
- The Wikipedia article about numerical analysis

4.2 References

The following is a non-comprehensive list of works used in the development of mpmath or cited for examples or mathematical definitions used in this documentation. References not listed here can be found in the source code.

Bibliography

[AbramowitzStegun] M Abramowitz & I Stegun. *Handbook of Mathematical Functions, 9th Ed.*, Tenth Printing, December 1972, with corrections (electronic copy: http://people.math.sfu.ca/~cbm/aands/)

[Bailey] D H Bailey. "Tanh-Sinh High-Precision Quadrature", http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-tanh-sinh.pdf

[BenderOrszag] C M Bender & S A Orszag. *Advanced Mathematical Methods for Scientists and Engineers*, Springer 1999

[BorweinBailey] J Borwein, D H Bailey & R Girgensohn. *Experimentation in Mathematics - Computational Paths to Discovery*, A K Peters, 2003

[BorweinBorwein] J Borwein & P B Borwein. *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity*, Wiley 1987

[BorweinZeta] P Borwein. "An Efficient Algorithm for the Riemann Zeta Function", http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P155.pdf

[CabralRosetti] L G Cabral-Rosetti & M A Sanchis-Lozano. "Appell Functions and the Scalar One-Loop Three-point Integrals in Feynman Diagrams". http://arxiv.org/abs/hep-ph/0206081

[Carlson] B C Carlson. "Numerical computation of real or complex elliptic integrals". http://arxiv.org/abs/math/9409227v1

[Corless] R M Corless et al. "On the Lambert W function", Adv. Comp. Math. 5 (1996) 329-359. http://www.apmaths.uwo.ca/~djeffrey/Offprints/W-adv-cm.pdf

[DLMF] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/

[GradshteynRyzhik] I S Gradshteyn & I M Ryzhik, A Jeffrey & D Zwillinger (eds.), *Table of Integrals, Series and Products*, Seventh edition (2007), Elsevier

[GravesMorris] P R Graves-Morris, D E Roberts & A Salam. "The epsilon algorithm and related topics", *Journal of Computational and Applied Mathematics*, Volume 122, Issue 1-2 (October 2000)

[MPFR] The MPFR team. "The MPFR Library: Algorithms and Proofs", http://www.mpfr.org/algorithms.pdf

[Slater] L J Slater. *Generalized Hypergeometric Functions*. Cambridge University Press, 1966

[Spouge] J L Spouge. "Computation of the gamma, digamma, and trigamma functions", SIAM J. Numer. Anal. Vol. 31, No. 3, pp. 931-944, June 1994.

[SrivastavaKarlsson] H M Srivastava & P W Karlsson. *Multiple Gaussian Hypergeometric Series*. Ellis Horwood, 1985.

[Vidunas] R Vidunas. “Identities between Appell’s and hypergeometric functions”. http://arxiv.org/abs/0804.0655

[Weisstein] E W Weisstein. *MathWorld*. http://mathworld.wolfram.com/

[WhittakerWatson] E T Whittaker & G N Watson. *A Course of Modern Analysis*. 4th Ed. 1946 Cambridge University Press

[Wikipedia] *Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Main_Page

[WolframFunctions] Wolfram Research, Inc. *The Wolfram Functions Site*. http://functions.wolfram.com/

m

A

B

C

D

E

F

G

H

Q

R

S

T

U

W

Z