Mpmath manual

Author: Fredrik Johansson
E-mail: fredrik.johansson@gmail.com
Updated: 2008-03-12
Mpmath version: 0.7

Table of contents

1 About this document

This document gives an introduction to usage of mpmath, a Python library for arbitrary-precision floating-point arithmetic. For general information about mpmath, see the website http://code.google.com/p/mpmath/. The most up-to-date version of this document is available at the mpmath website in the following formats:

- http://mpmath.googlecode.com/svn/trunk/doc/manual.html (HTML)
- http://mpmath.googlecode.com/svn/trunk/doc/manual.pdf (PDF)

This manual gives an introduction to mpmath's major features. Some supplementary documentation, FAQs, additional examples, etc may be available on the mpmath website.

2 Basics

For download and installation instructions, please refer to the README or the mpmath website (in most cases, installation should be as simple as running `python easy_install mpmath`). After the setup has completed, you can fire up the interactive Python interpreter and try the following:

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

In all interactive code examples that follow, it will be assumed that the main contents of the `mpmath` package have been imported with "`import *`".

2.1 Mpmath numbers

Mpmath provides two main numerical types: `mpf` and `mpc`. 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` 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')
```

An mpc 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:

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

Prettier output can be obtained by using str() or print, which hide the mpf and mpc constructor signatures and suppress small rounding artifacts:

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

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 special object called mp, which has the following default state:

```
>>> mp
Mpmath settings:
  mp.prec = 53                [default: 53]
  mp.dps = 15                 [default: 15]
  mp.rounding = 'nearest'     [default: 'nearest']
```

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).

The valid rounding modes are "nearest", "up", "down", "floor", and "ceiling". These modes are described in more detail in the section on rounding below. The default rounding mode (round to nearest) is the best setting for most purposes.

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('0.16666666666666666666666666666666666666666666666666656')
>>> mp.dps = 25
>>> mpf(2) ** mpf('0.5')
mpf('1.414213562373095048801688713')
```

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')
```

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 the nstr and nprint functions:

```
>>> mp.dps = 15
>>> a = mpf(1) / 6
>>> a
mpf('0.16666666666666666')
>>> nstr(a, 8)
'0.16666667'
>>> nprint(a, 8)
0.16666667
>>> nstr(a, 50)
'0.16666666666666665741480812812369549646973609924316'
```

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

```
>>> print mpf(2)**32582657 - 1
1.24575026015369e+9808357
```

Or why not 1 googolplex:

```
>>> print mpf(10) ** (10**100)  # doctest:+ELLIPSIS
1.0e+100000000000000000000000000000000000000000000000000...
```

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

2.2.1 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
```

In 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
>>> with workdps(20):  # doctest: +SKIP
...     print mpf(1)/7
...     with extradps(10):
...         print mpf(1)/7
...
0.14285714285714285714
0.142857142857142857142857142857
>>> 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')
```

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('10.9000000000000003552713678800501')
>>> mpf('10.9')  # good
mpf('10.8999999999999999999999999999997')
>>> mpf(109) / mpf(10)   # also good
mpf('10.8999999999999999999999999999997')
```

(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.)

2.4 Special numbers

Mpmath provides several special numbers, which are summarized in the following table.

Symbol	Description
`j`	Imaginary unit
`inf`	Positive infinity
`-inf`	Negative infinity
`nan`	Not-a-number
`pi`	pi = 3.14159
`degree`	1 deg = pi/180 = 0.0174532
`e`	Base of the natural logarithm, e = 2.71828
`euler`	Euler's constant, gamma = 0.577216
`catalan`	Catalan's constant, C or K = 0.915966
`ln2`	log(2) = 0.693147
`ln10`	log(10) = 2.30259
`eps`	Epsilon of working precision

The first four objects (`j`, `inf`, `-inf`, `nan`) are merely shortcuts to `mpc` and `mpf` instances with these fixed values.

The remaining numbers are lazy implementations of numerical constants that can be computed with 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:

```
>>> mp.dps = 15
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.2831853071795862')
>>> +pi
mpf('3.1415926535897931')
>>> mp.dps = 40
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.283185307179586476925286766559005768394338')
>>> +pi
mpf('3.141592653589793238462643383279502884197169')
```

The special number `eps` is defined as the difference between 1 and the smallest floating-point number after 1 that can be represented with the current working precision:

```
>>> mp.dps = 15
>>> eps
<epsilon of working precision: 2.22045e-16~>
>>> 1 + eps
mpf('1.0000000000000002')
>>> 1 + eps/2    # Too small to make a difference
mpf('1.0')
```

```
>>>
>>> mp.dps = 100
>>> eps
<epsilon of working precision: 1.42873e-101~>
```

An useful application of eps is to perform approximate comparisons that work at any precision level, for example to check for convergence of iterative algorithms:

```
>>> def a_series():
...     s = 0
...     n = 1
...     while 1:
...         term = mpf(5) ** (-n)
...         s += term
...         if term < eps:
...             print "added", n, "terms"
...             return s
...         n += 1
...
>>> mp.dps = 15
>>> a_series()
added 23 terms
mpf('0.25000000000000011')
>>>
>>> mp.dps = 40
>>> a_series()
added 59 terms
mpf('0.2500000000000000000000000000000000000000057')
```

2.5 Mathematical functions

Mpmath implements the standard functions available in Python's `math` and `cmath` modules, for both real and complex numbers and with arbitrary precision:

```
>>> mp.dps = 25
>>> print cosh('1.234')
1.863033801698422589073644
>>> print asin(1)
1.570796326794896619231322
>>> print log(1+2j)
(0.8047189562170501873003797 + 1.107148717794090503017065j)
>>> print exp(2+3j)
(-7.315110094901102517486536 + 1.042743656235904414101504j)
```

Some functions that do not exist in the standard Python `math` library are available, such as factorials (with support for noninteger arguments):

```
>>> mp.dps = 20
>>> print factorial(10)
3628800.0
```

```
>>> print factorial(0.25)
0.90640247705547707798
>>> print factorial(2+3j)
(-0.44011340763700171113 - 0.06363724312631702183j)
```

The list of functions is given in the following table.

Function	Description
`sqrt(x)`	Square root
`hypot(x,y)`	Euclidean norm
`exp(x)`	Exponential function
`log(x,b)`	Natural logarithm (optionally base-b logarithm)
`power(x,y)`	Power, x^y
`cos(x)`	Cosine
`sin(x)`	Sine
`tan(x)`	Tangent
`cosh(x)`	Hyperbolic cosine
`sinh(x)`	Hyperbolic sine
`tanh(x)`	Hyperbolic tangent
`acos(x)`	Inverse cosine
`asin(x)`	Inverse sine
`atan(x)`	Inverse tangent
`atan2(y,x)`	Inverse tangent atan(y/x) with attention to signs of both x and y
`acosh(x)`	Inverse hyperbolic cosine
`asinh(x)`	Inverse hyperbolic sine
`atanh(x)`	Inverse hyperbolic tangent
`floor(x)`	Floor function (round to integer in the direction of -inf)
`ceil(x)`	Ceiling function (round to integer in the direction of +inf)
`arg(x)`	Complex argument
`rand()`	Generate a random number in [0, 1)
`factorial(x)`	Factorial
`gamma(x)`	Gamma function
`lower_gamma(a,x)`	Lower gamma function
`upper_gamma(a,x)`	Upper gamma function
`erf(x)`	Error function
`zeta(x)`	Riemann zeta function
`j0(x)`	Bessel function J_0(x)
`j1(x)`	Bessel function J_1(x)
`jn(n,x)`	Bessel function J_n(x)

The following functions do not accept complex input: `hypot`, `atan2`, `floor`, `ceil`, `j0`, `j1` and `jn`.

3 High-level features

3.1 Integration

The function `quadts` performs numerical integration (quadrature) using the tanh-sinh algorithm. The syntax for integrating a function f between the endpoints a and b is `quadts(f, a, b)`. For example:

```
>>> print quadts(sin, 0, pi)
2.0
```

Tanh-sinh quadrature is extremely efficient for high-precision integration of analytic functions. Unlike the more well-known Gaussian quadrature algorithm, it is relatively insensitive to integrable singularities at the endpoints of the interval. The `quadts` function attempts to evaluate the integral to the full working precision; for example, it can calculate 100 digits of pi by integrating the area under the half circle arc `x^2 + y^2 = 1 (y > 0)`:

```
>>> mp.dps = 100
>>> print quadts(lambda x: 2*sqrt(1 - x**2), -1, 1)
... # doctest:+ELLIPSIS
3.141592653589793238462643383279502884197169399375105820974...
```

The tanh-sinh scheme is efficient enough that analytic 100-digit integrals like this one can often be evaluated in less than a second. The timings for computing this integral at various precision levels on the author's computer is:

dps	First evaluation	Second evaluation
15	0.029 seconds	0.0060 seconds
50	0.15 seconds	0.016 seconds
500	16.3 seconds	0.50 seconds

The second integration at the same precision level is much faster. The reason for this is that the tanh-sinh algorithm must be initalized by computing a set of nodes, and this initalization if often more expensive than actually evaluating the integral. Mpmath automatically caches all computed nodes to make subsequent integrations faster, but the cache is lost when Python shuts down, so if you would frequently like to use mpmath to calculate 1000-digit integrals, you may want to save the nodes to a file. The nodes are stored in a dict `TS_cache` located in the `mpmath.calculus` module, which can be pickled if desired.

3.1.1 Features and application examples

You can integrate over infinite or half-infinite intervals:

```
>>> mp.dps = 15
>>> print quadts(lambda x: 2/(x**2+1), 0, inf)
3.14159265358979
>>> print quadts(lambda x: exp(-x**2), -inf, inf)**2
3.14159265358979
```

Complex integrals are also supported. The next example computes Euler's constant gamma by using Cauchy's integral formula and looking at the pole of the Riemann zeta function at $z = 1$:

```
>>> print 1/(2*pi)*quadts(lambda x: zeta(exp(j*x)+1), 0, 2*pi)
(0.577215664901533 + 2.86444093843177e-25j)
```

Functions with integral representations, such as the gamma function, can be implemented directly from the definition:

```
>>> def Gamma(z):
...     return quadts(lambda t: exp(-t)*t**(z-
1), 0, inf)
...
>>> print Gamma(1)
1.0
>>> print Gamma(10)
362880.0
>>> print Gamma(1+1j)
(0.498015668118356 - 0.154949828301811j)
```

3.1.2 Double integrals

It is possible to calculate double integrals with `quadts`. To do this, simply provide a two-argument function and, instead of two endpoints, provide two intervals. The first interval specifies the range for the x variable and the second interval specifies the range of the y variable:

```
>>> f = lambda x, y: cos(x+y/2)
>>> print quadts(f, (-pi/2, pi/2), (0, pi))
4.0
```

Here are some more difficult examples taken from MathWorld (all except the second contain corner singularities):

```
>>> mp.dps = 30
>>> f = lambda x, y: (x-1)/((1-x*y)*log(x*y))
>>> print quadts(f, (0, 1), (0, 1))
0.577215664901532860606512090082
>>> print euler
0.577215664901532860606512090082

>>> f = lambda x, y: 1/sqrt(1+x**2+y**2)
>>> print quadts(f, (-1, 1), (-1, 1))
3.17343648530607134219175646705
>>> print 4*log(2+sqrt(3))-2*pi/3
3.17343648530607134219175646705

>>> f = lambda x, y: 1/(1-x**2 * y**2)
>>> print quadts(f, (0, 1), (0, 1))
1.23370055013616982735431137498
```

```
>>> print pi**2 / 8
1.2337005501361698273543113749

>>> print quadts(lambda x, y: 1/(1-x*y), (0, 1), (0, 1))
1.6449340668482264364724151666
>>> print pi**2 / 6
1.6449340668482264364724151666
```

There is currently no direct support for computing triple or higher dimensional integrals; if desired, this can be done easily by passing a function that calls `quadts` recursively:

```
>>> mp.dps = 15
>>> f = lambda x, y: quadts(lambda z: sin(x)/z+y*z, 1, 2)
>>> print quadts(f, (1, 2), (1, 2))
2.91296002641413
>>> print mpf(9)/4 + (cos(1)-cos(2))*log(2)
2.91296002641413
```

While double integrals are reasonably fast, even a simple triple integral at very low precision is likely to take several seconds to evaluate (harder integrals may take minutes). A quadruple integral will require a whole lot of patience.

3.1.3 Error detection

The tanh-sinh algorithm is not suitable for adaptive quadrature, and does not perform well if there are singularities between the endpoints or if the integrand is very bumpy or oscillatory (such integrals should manually be split into smaller pieces). If the `error` option is set, `quadts` will return an error estimate along with the result; although this estimate is not always correct, it can be useful for debugging. You can also pass `quadts` the option `verbose=True` to show detailed progress.

A simple example where the algorithm fails is the function $\mathrm{f}(x) = \mathrm{abs}(\sin(x))$, which is not smooth at $x = \mathrm{pi}$. In this case, a close value is calculated, but the result is nowhere near the target accuracy; however, `quadts` gives a good estimate of the magnitude of the error:

```
>>> mp.dps = 15
>>> quadts(lambda x: abs(sin(x)), 0, 2*pi, error=True)
(mpf('3.9990089417677899'), mpf('0.001'))
```

Attempting to evaluate oscillatory integrals on large intervals by means of the tanh-sinh method is generally futile. This integral should be pi/2 = 1.57:

```
>>> print quadts(lambda x: sin(x)/x, 0, inf, error=True)
(mpf('2.3840907358976544'), mpf('1.0'))
```

The next integral should be approximately 0.627 but `quadts` generates complete nonsense both in the result and the error estimate (the error estimate is somewhat arbitrarily capped at 1.0):

```
>>> print quadts(lambda x: sin(x**2), 0, inf, er-
ror=True)
(mpf('2.5190134849122411e+21'), mpf('1.0'))
```

However, oscillation is not a problem if suppressed by sufficiently fast (preferrably exponential) decay. This integral is exactly 1/2:

```
>>> print quadts(lambda x: exp(-x)*sin(x), 0, inf)
0.5
```

Another illustrative example is the following double integral, which `quadts` will process for several seconds before returning a value with very low accuracy:

```
>>> mpf.dps = 15
>>> f = lambda x, y: sqrt((x-0.5)**2+(y-0.5)**2)
>>> quadts(f, (0, 1), (0, 1), error=1)
(mpf('0.38259743528830826'), mpf('1.0e-6'))
```

The problem is due to the non-analytic behavior of the function at the midpoint (1/2, 1/2). We can do much better by splitting the area into four pieces (because of the symmetry, we only need to evaluate one of them):

```
>>> f = lambda x, y: 4*sqrt((x-0.5)**2 + (y-0.5)**2)
>>> print quadts(f, (0.5, 1), (0.5, 1))
0.382597858232106
>>> print (sqrt(2) + asinh(1))/6
0.382597858232106
```

The value agrees with the known answer and the running time in this case is just 0.7 seconds on the author's computer.

Even for analytic integrals on finite intervals, there is no guarantee that `quadts` will be successful. A few examples of integrals for which `quadts` currently fails to reach full accuracy are:

```
quadts(lambda x: sqrt(tan(x)), 0, pi/2)
quadts(lambda x: atan(x)/(x*sqrt(1-x**2)), 0, 1)
quadts(lambda x: log(1+x**2)/x**2, 0, 1)
quadts(lambda x: x**2/((1+x**4)*sqrt(1-x**4)), 0, 1)
```

(It is possible that future improvements to the `quadts` implementation will make these particular examples work.)

3.2 Differentiation

The function `diff` computes a derivative of a given function. It uses a simple two-point finite difference approximation, but increases the working precision to get good results. The step size is chosen roughly equal to the `eps` of the working precision, and the function values are computed at twice the working precision; for reasonably smooth functions, this typically gives full accuracy:

```
>>> mp.dps = 15
>>> print diff(cos, 1)
-0.841470984807897
>>> print -sin(1)
-0.841470984807897
```

One-sided derivatives can be computed by specifying the `direction` parameter. With `direction = 0` (default), `diff` uses a central difference (`f(x-h)`, `f(x+h)`). With `direction = 1`, it uses a forward difference (`f(x)`, `f(x+h)`), and with `direction = -1`, a backward difference (`f(x-h)`, `f(x)`):

```
>>> print diff(abs, 0, direction=0)
0.0
>>> print diff(abs, 0, direction=1)
1.0
>>> print diff(abs, 0, direction=-1)
-1.0
```

Although the finite difference approximation can be applied recursively to compute n-th order derivatives, this is inefficient for large n since `2^n` evaluation points are required, using `2^n`-fold extra precision. As an alternative, the function `diffc` computes derivatives of arbitrary order by means of complex contour integration. It is for example able to compute a 13th-order derivative of sin (here at $x = 0$):

```
>>> print diffc(sin, 0, 13)
(0.999998702480854 + 6.05532349899064e-13j)
```

The accuracy can be improved by increasing the radius of the integration contour (provided that the function is well-behaved within this region):

```
>>> print diffc(sin, 0, 13, radius=5)
(1.0 - 3.3608728322706e-23j)
```

3.3 Root-finding

The function `secant` locates a root of a given function using the secant method. A simple example use of the secant method is to compute pi as the root of $\sin(x)$ closest to $x = 3$:

```
>>> mp.dps = 30
>>> print secant(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
>>> print secant(lambda x: x**3 + 2*x + 1, j)
(0.226698825758202 + 1.46771150871022j)
```

A good initial guess for the location of the root is required for the method to be effective, so it is somewhat more appropriate to think of the secant method as a root-polishing method than a root-finding method. When the rough location of the root is known, the secant method can be used to refine it to very high precision in only a few steps. If the root is a first-order root, only roughly log(prec) iterations are required. (The secant method is far less efficient for double roots.) It may be worthwhile to compute the initial approximation to a root using a machine precision solver (for example using one of SciPy's many solvers), and then refining it to high precision using mpmath's `secant` method.

3.3.1 Applications

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
>>> print secant(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
>>> print secant(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 $w\ \exp(w)$, given the first term of the solution's asymptotic expansion as the initial value:

```
>>> def lambert(x):
...     return secant(lambda w: w*exp(w) - x, log(1+x))
...
>>> mp.dps = 15
>>> print lambert(1)
0.567143290409784
>>> print lambert(1000)
5.2496028524016
```

3.3.2 Options

Strictly speaking, the secant method requires two initial values. By default, you only have to provide the first point `x0`; `secant` automatically sets the second point (somewhat arbitrarily) to `x0 + 1/4`. Manually providing also the second point can help in some cases if `secant` fails to converge.

By default, `secant` performs a maximum of 20 steps, which can be increased or decreased using the `maxsteps` keyword argument. You can pass `secant` the option `verbose=True` to show detailed progress.

3.4 Polynomials

3.4.1 Polynomial evaluation

Polynomial functions can be evaluated using `polyval`, which takes as input a list of coefficients and the desired evaluation point. The following example evaluates `2 + 5*x + x^3` at `x = 3.5`:

```
>>> mp.dps = 20
>>> polyval([2, 5, 0, 1], mpf('3.5'))
mpf('62.375')
```

With `derivative=True`, both the polynomial and its derivative are evaluated at the same point:

```
>>> polyval([2, 5, 0, 1], mpf('3.5'), derivative=True)
(mpf('62.375'), mpf('41.75'))
```

The point and coefficients may be complex numbers.

3.4.2 Finding roots of polynomials

The function `polyroots` computes all n real or complex roots of an n-th degree polynomial using complex arithmetic, and returns them along with an error estimate. As a simple example, it will successfully compute the two real roots of `3*x^2 - 7*x + 2` (which are 1/3 and 2):

```
>>> mp.dps = 15
>>> roots, err = polyroots([2, -7, 3])
>>> print err
2.66453525910038e-16
>>> for root in roots:
...     print root
...
(0.333333333333333 - 9.62964972193618e-35j)
(2.0 + 1.5395124730131e-50j)
```

As should be expected from the internal use of complex arithmetic, the calculated roots have small but nonzero imaginary parts.

The following example computes all the 5th roots of unity; i.e. the roots of `x^5 - 1`:

```
>>> mp.dps = 20
>>> for a in polyroots([-1, 0, 0, 0, 0, 1])[0]:
...     print a
...
(-0.8090169943749474241 + 0.58778525229247312917j)
(1.0 + 0.0j)
(0.3090169943749474241 + 0.95105651629515357212j)
(-0.8090169943749474241 - 0.58778525229247312917j)
(0.3090169943749474241 - 0.95105651629515357212j)
```

3.5 Interval arithmetic

The `mpi` type holds an interval defined by a pair of `mpf` values. Arithmetic on intervals uses conservative rounding so that, if an interval is interpreted as a numerical uncertainty interval 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.

You can create an `mpi` from a number (treated as a zero-width interval) or a pair of 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):

```
>>> mp.dps = 15
>>> mpi(3)
```

```
[3.0, 3.0]
>>> mpi(2, 3)
[2.0, 3.0]
>>> mpi(0.1)  # probably not what you want
[0.10000000000000000555, 0.10000000000000000555]
>>> mpi('0.1')  # good
[0.099999999999999991673, 0.10000000000000000555]
```

The fact that '0.1' results in an interval of nonzero width proves 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).

Some basic examples of interval arithmetic operations are:

```
>>> mpi(0,1) + 1
[1.0, 2.0]
>>> mpi(0,1) + mpi(4,6)
[4.0, 7.0]
>>> 2 * mpi(2, 3)
[4.0, 6.0]
>>> mpi(-1, 1) * mpi(10, 20)
[-20.0, 20.0]
```

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

```
>>> x = mpi(2, 5)
>>> x.a
mpf('2.0')
>>> x.b
mpf('5.0')
>>> x.mid
mpf('3.5')
>>> x.delta
mpf('3.0')
```

Intervals may be infinite or half-infinite:

```
>>> 1 / mpi(2, inf)
[0.0, 0.5]
```

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

```
>>> mpi(0, 2) in mpi(0, 10)
True
>>> 3 in mpi(-inf, 0)
False
```

Division is generally not an exact operation in floating-point arithmetic. Using interval arithmetic, we can track both the error from the division and the error that propagates if we follow up with the inverse operation:

```
>>> 1 / mpi(3)
[0.33333333333333331483, 0.33333333333333337034]
>>> 1 / (1 / mpi(3))
[2.9999999999999995559, 3.0000000000000004441]
```

The same goes for computing square roots:

```
>>> (mpi(2) ** 0.5) ** 2
[1.9999999999999995559, 2.0000000000000004441]
```

By design, interval arithmetic propagates errors, no matter how tiny, that would get rounded off in normal floating-point arithmetic:

```
>>> mpi(1) + mpi('1e-10000')
[1.0, 1.000000000000000222]
```

Interval arithmetic uses the same precision as the `mpf` class; if `mp.dps = 50` is set, all interval operations will be carried out with 50-digit precision. Of course, interval arithmetic is guaranteed to give correct bounds at any precision, but a higher precision makes the intervals narrower and hence more accurate:

```
>>> mp.dps = 5
>>> mpi(pi)
[3.141590118, 3.141593933]
>>> mp.dps = 30
>>> mpi(pi)  # doctest: +ELLIPSIS
[3.14159265358979...793333, 3.14159265358979...797277]
```

It should be noted that the support for interval arithmetic in mpmath is still somewhat primitive, but the standard arithmetic operators `+`, `-`, `*`, `/`, as well as integer powers should work correctly. It is not currently possible to use functions like `sin` or `log` with interval arguments. You can convert mathematical constants to intervals (as in the previous example) and compute fractional powers, but this is not currently guaranteed to give correct results (although it most likely will).

3.5.1 Establishing inequalities

Interval arithmetic can be used to establish inequalities such as `exp(pi*sqrt(163)) < 640320**3 + 744`. The left-hand and right-hand sides in this inequality agree to over 30 digits, so low-precision arithmetic may give the wrong result:

```
>>> mp.dps = 25
>>> exp(pi*sqrt(163)) < (640320**3 + 744)
False
```

The answer should be True, but the rounding errors are larger than the difference between the numbers. To get the right answer, we can use interval arithmetic to check the sign of the difference between the two sides of the inequality. Interval arithmetic does not tell us the answer right away if we keep `mp.dps = 25`, but it is honest enough to admit it:

```
>>> mpi(e) ** (mpi(pi) * mpi(163)**0.5) -
(640320**3 + 744)
... # doctest: +ELLIPSIS
[-0.000000793..., 0.000000946...]
```

There is both a negative and a positive endpoint, so we cannot tell for certain whether the true difference is on one side or the other of zero. The solution is to increase the precision until the answer is strictly one-signed:

```
>>> mp.dps = 35
>>> mpi(e) ** (mpi(pi) * mpi(163)**0.5) -
(640320**3 + 744)
... # doctest: +ELLIPSIS
[-7.499745...e-13, -7.498606...-13]
```

4 Technical details

Doing a high-precision calculation in mpmath typically just amounts to setting the precision and entering a formula. However, some more details of mpmath's terminology and internal number model can be useful to avoid common errors, and is recommended for trying more advanced calculations.

4.1 Representation of numbers

Mpmath uses binary arithmetic. A binary floating-point number is a number of the form `man * 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: `(m*2^e) * (n*2^f) = (m*n) * 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).

The special numbers `+inf`, `-inf` and `nan` are represented internally by a zero mantissa and a nonzero exponent.

For further details on how the arithmetic is implemented, refer to the mpmath source code. The basic arithmetic operations are found in the `lib.py` module; many functions there are commented extensively.

4.2 Precision and accuracy

Contrary to popular superstition, floating-point numbers do not come with an inherent "small uncertainty". 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.

The reason why floating-point arithmetic is generally approximate is that we set a limit to the size of the mantissa for efficiency reasons. The maximum allowed width (bitcount) of the mantissa is called the precision or `prec` for short. Sums and products 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 threshold, we truncate it to have at most `prec` bits (the exponent is incremented accordingly to preserve the magnitude of the number), and it is this operation that typically introduces numerical errors. Division is also not generally exact; 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).

4.2.1 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):

```
>>> 0.1
0.10000000000000001
```

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 to within `2^(-prec)`. With `dps` decimal digits, the corresponding error is `10^-dps`. 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:

```
>>> 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 from binary to 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
```

In fact, all binary floating-point numbers can be represented exactly as decimals (despite the converse not being true), but displaying more than `dps` digits is usually not useful, since typically only at most `dps` digits will be correct when the floating-point number is an approximation for some computed quantity.

4.3 Rounding

There are several different strategies for rounding a too large mantissa or a result that cannot at all be represented exactly in binary floating-point form (such as 1/3 or log(2)). Mpmath supports the following rounding modes:

Name	**Direction**
Floor	Towards negative infinity
Ceiling	Towards positive infinity
Down	Towards 0
Up	Away from 0
Nearest	To nearest; to the nearest even number on a tie

The first four modes are called *directed* rounding schemes and are useful for implementing interval arithmetic; they are also fast. Rounding to nearest, which mpmath uses by default, is the slowest but most accurate method.

The arithmetic operations `+`, `-`, `*` and `/` acting on real floating-point numbers always round their results *correctly* in mpmath; that is, they are guaranteed to give exact results when possible, they always round in the intended direction, and they don't round to a number farther away than necessary. Exponentiation by an integer n preserves directions but may round too far if either the mantissa or n is very large.

Evaluation of transcendental functions (as well as square roots) is generally performed by computing an approximation with finite precision slightly higher than the target precision, and rounding the result. This gives correctly rounded results with a high probability, but can be wrong in exceptional cases.

Rounding for radix conversion is a slightly tricky business. When converting to a binary floating-point number from a decimal string, mpmath writes the number as an exact fraction and performs correct rounding division if the number is of reasonable size (roughly, larger than 10ˆ-100 and smaller than 10ˆ100), guaranteeing correct rounding. If the exponent is enormous, mpmath first performs a floating-point division to reduce it to a manageable size; this can produce a (tiny) rounding error.

When converting from binary to decimal, mpmath first performs an approximate radix conversion with slightly increased precision, then truncates the resulting decimal number to remove long sequences of trailing 0's and 9's, and finally rounds to nearest, rounding up (away from zero) on a tie. The `decimal` library could be used to provide more control over the rounding in the binary-to-decimal conversion, and mpmath did do radix conversions via `decimal` in older versions, but this was far too slow compared to using a custom algorithm.

4.4 Exponent range

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 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`.

In some situations, it might be more convenient if mpmath could "round" extremely small numbers to 0 and extremely large numbers to `inf`, and directly raise an exception or return `nan` if there is no reasonable chance of finishing a computation. This option is not available, but could be implemented in the future on demand.

4.5 Compatibility

The floating-point arithmetic provided by processors that conform to the IEEE 754 *double precision* standard has a precision of 53 bits and rounds to nearest. (Additional precision and rounding modes are available, but regular double precision arithmetic should be the most familiar to Python users, since the Python `float` type corresponds to an IEEE double with rounding to nearest on most systems.)

This corresponds roughly to a decimal accuracy of 15 digits, and is the default precision used by mpmath. Thus, under normal circumstances, mpmath should produce identical results to Python `float` operations. This is not always true, mainly due to the simple fact that mpmath is able to produce more accurate results for transcendental functions. Machine floats very close to the exponent limit also round subnormally, meaning that they lose precision (Python may raise an exception instead of rounding a `float` subnormally).

5 Optimization tricks

There are a few tricks that can significantly speed up mpmath code at low to medium precision (up a hundred digits or so):

- Repeated type conversions from floats, strings and integers are expensive (exceptions: `n/x`, `n*x` and `x**n` are fast when `n` is an `int` and `x` is an `mpf`). Numerical constants that are used repeatedly, such as in the body of a function passed to `quadts`, should be pre-converted to `mpf` instances.
- The JIT compiler psyco fairly consistently speeds up mpmath about 2x.
- An additional 2x gain is possible by using the low-level functions in `mpmath.lib` instead of `mpf` instances.
- Changing the rounding mode to *floor* can give a slight speedup.

Here follows a simple example demonstrating some of these optimizations. Original algorithm (0.028 seconds):

```
>>> x = mpf(1)
>>> for i in range(1000):
...     x += 0.1
```

Preconverting the float constant (0.0080 seconds):

```
>>> x = mpf(1)
>>> one_tenth = mpf(0.1)
>>> for i in range(1000):
...     x += one_tenth
```

With psyco (0.0036 seconds):

```
>>> import psyco; psyco.full()
>>> x = mpf(1)
>>> one_tenth = mpf(0.1)
>>> for i in range(1000):
...     x += one_tenth
```

With psyco and low-level functions (0.0017 seconds):

```
>>> import psyco; psyco.full()
>>> from mpmath.lib im-
port from_int, from_float, fadd, round_nearest
>>> x = from_int(1)
>>> one_tenth = from_float(0.1)
>>> for i in range(1000):
...     x = fadd(x, one_tenth, 53, round_nearest)
```

The last version is 16.5 times faster than the first (however, this example is extreme; the gain will usually be smaller in realistic calculations).

Many calculations can be done with ordinary floating-point arithmetic, and only in special cases require multiprecision arithmetic (for example to avoid overflows in corner cases). In these situations, it may be possible to write code that uses fast regular floats by default, and automatically (or manually) falls backs to mpmath only when needed. Python's dynamic namespaces and ability to compile code on the fly are helpful. Here is a simple (probably not failsafe) example:

```
>>> import math
>>> import mpmath
>>>
>>> def evalmath(expr):
...     try:
...         r = eval(expr, math.__dict__)
...     except OverflowError:
...         r = eval(expr, mpmath.__dict__)
...         try:
...             r = float(r)
...         except OverflowError:
...             pass
...     return r
...
>>> evalmath('sin(3)')
0.14112000805986721
>>>
>>> evalmath('exp(10000)')
mpf('8.8068182256629216e+4342')
>>>
>>> evalmath('exp(10000) / exp(10000)')
1.0
```