mpmath: arbitrary-precision floating-point arithmetic and special functions

Fredrik Johansson

May 18, 2009

Overview of mpmath

- Started in 2007 as a SymPy module as a fast alternative to `decimal.Decimal`.
- Now standalone. Available in SymPy and Sage as `sympy.mpmath` (old version).
- Pure-Python (can optionally use GMPY), self-contained, BSD license.
- Latest release 0.11, January, 1200 downloads.
- Contributors: Vinzent Steinberg, Mario Pernici, Case Vanhorsen. Occasional patches from SymPy users.

Overview of mpmath - features

- Simple interface: drop-in (almost) replacement for `math` and `scipy`.
- Real and complex numbers, inf/nan, intervals, matrices
- Numbers are arbitrary-size
- Special functions (erf, gamma, ...)
- Calculus (limits, sums, derivatives, integrals, ODEs, ...),
- Goal: match arbitrary-precision numerics in Mathematica feature by feature (and ideally do more)
- Goal: do any series, integral, etc in reference tables

Present and future development

- Internals of mpmath (possible improvements)
- Adding mpmath to Sage
- Special functions (summer project)

Basic arithmetic

- Big floats: $x = m \cdot 2^e$, m, e both arbitrary-precision integers. m always `long`, e `int` or `long`.
- Arithmetic in pure Python is relatively fast at moderate precision, e.g. $x_1 x_2 = m_1 m_2 2^{e_1+e_2}$. About 2-4 times slower than C/GMP (`mpz` layer).
- Easy way to obtain faster high-precision arithmetic: use `sage.Integer` or `gmpy.mpz` for mantissa instead of `long`.
- Wrapper classes (`mpf`, `mpc`) provide correct rounding, type conversions, etc. Partial workaround for slowdown: write speed-critical functions (special functions, dot product, etc.) in "low level" code (unwrapped numbers).
- Current and future development: implement low level code as well as wrapper classes in Cython.

Backend comparison (mpmath unit tests)

```
pure python backend - 69 seconds
sage backend (yesterday) - 97 seconds
sage backend (today) - 67 seconds
gmpy backend - 37 seconds
```

Cython backend

Results from Mario Pernici (May 11):

```
Here is a benchmark with timings_mpmath.py using
gmpy in Cython with dps=100
     mpmath     sage
add  0.00046    0.00043
mul  0.00077    0.00052
div  0.0012     0.00080
sqrt 0.0018     0.0014
exp  0.011      0.012
log  0.012      0.017
sin  0.011      0.013
cos  0.010      0.0091
acos 0.024      0.075
atan 0.013      0.066
```

Algorithms: elementary functions

- Code minimization: all elementary functions (real and complex) can be reduced to the Taylor series of (for example) cos, cosh, atan and atanh of real variables.
- Implementation directly on top of `long` / `mpz`.
- Need to optimize for both small and large precisions. Currently working on tuning the code.

Exponential / trigonometric functions

- Use $e^x = \cosh x + \sinh x$, since Taylor series for cosh has half as many terms). Get sinh(x) [sin(x)] from cosh(x) [cos(x)] with a square root.
- Convergence acceleration: n-fold application of half-argument formula $\cosh x = 2\cosh(\frac{1}{2}x)^2 - 1$. Choosing $n = p^{1/2}$ reduces number of terms (= multiplication count) from $O(p)$ to $O(p^{1/2})$.
- Sum r series concurrently, e.g. $r = 4$:

$$\cosh x = \left(1 + \frac{x^8}{8!} + \ldots\right) + x^2\left(\frac{1}{2!} + \frac{x^8}{10!} + \ldots\right) +$$

$$x^4\left(\frac{1}{4!} + \frac{x^8}{12!} + \ldots\right) + x^6\left(\frac{1}{6!} + \frac{x^8}{14!} + \ldots\right).$$

Complexity (n and r chosen optimally): about $O(p^{1/3})$ multiplications.

Logarithm / arctangent

- Use $\log x = 2 \operatorname{atanh} \frac{x-1}{x+1}$.
- Use Taylor series for atan/atanh. Converges rapidly only for $|x| \ll 1$, so argument reductions mandatory.
- Reduction/convergence acceleration: n-fold application of half-argument formula (uses square roots); sum r series concurrently.
- Faster, low precision: cache $\log(m/2^n)$ and $\operatorname{atan}(m/2^n)$ rewrite x as $t + m/2^n$ using addition theorems.
- Can also use Newton's method (high overhead).
- Very high precision ($\gg 1000$ digits): use AGM for log. Compute exp from log using Newton's method. Complexity: $O(\log p)$ multiplications.

Hypergeometric functions

- Generalized hypergeometric function:

$$
{}_pF_q(a_1, \ldots, a_p; b_1, \ldots, b_q; z) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n \ldots (a_p)_n}{(b_1)_n (b_2)_n \ldots (b_q)_n} \frac{z^n}{n!}
$$

- Most common: ${}_0F_1$, ${}_1F_1$, ${}_2F_1$ (usually with $a_k, b_k \in \mathbb{Q}$)
- Particular cases: elementary functions, error functions, exponential/hyperbolic/trigonometric integrals, incomplete gamma function, Fresnel integrals, Bessel functions, Airy functions, Legendre/Chebyshev/Jacobi functions.
- Methods: direct summation, asymptotic expansions, continued fractions, expansions around poles, special-purpose code

Other functions

Many important functions are not of the hypergeometric type.
Examples:

- Gamma function
- Polygamma functions
- Theta functions
- Zeta functions
- ...

Methods: Euler-Maclaurin summation, special-purpose approximations, numerical integration
Difficulties: Hard to determine correct (let alone optimal) parameters and cutoffs

In progress: fast gamma function

- Use Maclaurin series for $\frac{1}{\Gamma(z)}$ with near-optimal truncation.
- Timings for gamma(3.7) (milliseconds)

digits	sage	mpmath(sage)	mpmath(gmpy)	new
50	0.30	1.25	0.41	0.09
150	1.65	3.62	1.45	0.19
500	33.9	21.5	15.5	1.84
1000	289	98.2	96.8	8.1

- Calculating n Maclaurin coefficients requires $\zeta(2), \zeta(3), \ldots, \zeta(n)$ and $O(n^2)$ multiplications. Precomputation time: 0.1 seconds @ 150 digits, 6 seconds @ 1000 digits.
- Separate algorithm for $\Gamma(p/q)$
- Separate algorithm for log gamma, and for $\Gamma(z)$, z large (Stirling's series, to be implemented)

Mixed machine-precision and arbitrary-precision

- Where appropriate, use abstract code that works with any number type (provided a suitable wrapper layer).
- Example (Bessel I function):

```
@defun_wrapped
def besseli(A,n,x):
    if A.isint(n):
        n = abs(int(n))
    hx = x/2
    return hx**n * A.hyp0f1(n+1, hx**2) / A.factorial(r
```

- Context `A` implements fundamental functions (e.g. ${}_0F_1$, $n!$) in an optimized fashion. Can also take care of adaptive evaluation (possibly requiring directives in the function description).
- Can support arbitrary-precision floats, Python floats, intervals, ...