Special functions in the Arb library

Fredrik Johansson

INRIA-Bordeaux

2015

What is Arb?

$\zeta(s)$ **Arb**

- Arbitrary-precision numerical library with rigorous error bounds
- $\sim$100,000 lines of C code, GPL v2+, developed since 2012
- Dependencies: GMP (or MPIR), MPFR, FLINT
- Python/Sage interfaces in progress
- `http://fredrikj.net/arb/`

Representing numbers

arb type (real mid-rad interval, "ball"):

$$[\underbrace{3.14159265358979323846264338328}_{\text{arbitrary-precision floating-point}} \pm \underbrace{8.65 \cdot 10^{-31}}_{\text{30-bit precision}}]$$

acb type (complex rectangular "ball"):

$$[1.414213562 \pm 3.74 \cdot 10^{-10}] + [1.732050808 \pm 4.32 \cdot 10^{-10}]i$$

Goal

- $\{\text{cost of interval arithmetic}\} = (1+\varepsilon) \cdot \{\text{cost of floating-point}\}$
- $\{\text{effort for error analysis}\} = \varepsilon \cdot \{\text{effort with floating-point}\}$

See also: iRRAM, Mathemagix, MPFS

Example: the partition function

$p(n)$: number of ways n can be written as a sum of positive integers
$p(10) = 42$
$p(100) = 190569292$
$p(1000) = 24061467864032622473692149727991$

Current record, set in 2014 with Arb:
$p(10^{20}) = 18381765\ldots88091448$ (11,140,086,260 digits)

The Hardy-Ramanujan-Rademacher formula

$$p(n) = \frac{1}{\pi\sqrt{2}} \sum_{k=1}^{\infty} \sqrt{k}\, A_k(n)\, \frac{d}{dn} \left(\frac{1}{\sqrt{n - \frac{1}{24}}} \sinh \left[\frac{\pi}{k} \sqrt{\frac{2}{3}\left(n - \frac{1}{24}\right)} \right] \right)$$

Compute approximation $y \in \mathbb{R}$ such that $|p(n) - y| < 0.5$, round.

For $n = 10^{20}$: 1.7 billion terms, 200 CPU hours, 130 GB memory

Semantics

Inclusion property

Mathematical function: $f(x) = y$
Implemented function: $F(X) = Y$
Guarantee: $x \in X \Rightarrow y \in Y$

- We make **no formal guarantees** about **tightness** of Y
- We can always fall back to $Y = [\pm\infty]$
- Usually behaves like floating-point + error analysis (but automatic)
- User can check if output is good enough, implement adaptivity

Example: easy rigorous numerical evaluation (via Python)

```
def N(f):
    prec = ctx.prec; guard = 10
    try:
        while 1:
            ctx.prec = prec + guard; y = f()
            if y.rel_accuracy_bits() >= prec:
                return y
            guard *= 2
    finally:
        ctx.prec = prec

>>> N(lambda: (arb(163).sqrt()*arb.pi()).exp() - 640320**3 - 744)
[-7.49927402801814e-13 ± 3.12e-28]

# 0.9 seconds in Arb
# 200 seconds in mpmath
# 100,000 seconds in Mathematica!
>>> N(lambda: acb.bessel_j(10000+10000j, 10000 * arb.pi()))
[-1.20973469401861e+5438 ± 4.77e+5423] + [1.21911522763864e+5438 ± 3.09e+5423]j

>>> ctx.dps = 30; N(lambda: acb(100).erf() - 1)
[-6.40596142492173203902133914859e-4346 ± 3.61e-4376]
```

Special functions in Arb

- Mathematical constants: π, γ, $\ldots$
- Elementary functions: $\exp(z), \log(z), \sin(z), \operatorname{asin}(z), \ldots$
- Gamma functions: $\Gamma(z)$, $\log\Gamma(z)$, $\psi(z)$
- Zeta functions: $\zeta(s, z)$, $\mathrm{Li}_s(z)$
- Bessel functions: $J_a(z)$, $K_a(z)$
- Exponential integrals: $\Gamma(a, z)$, $E_a(z)$, $\operatorname{erf}(z)$, $\operatorname{erfc}(z)$
- Hypergeometric functions (in progress): ${}_pF_q(\ldots, \ldots, z)$, $U(a, b, z)$
- Theta/elliptic/modular: $\theta(z, \tau)$, $\wp(z, \tau)$, $K(m)$, $j(\tau)$, $\operatorname{agm}(x, y)$
- Discrete: partition numbers $p(n)$, Bernoulli numbers B_n, $\ldots$

- **Complex arguments** generally supported
- **Power series arguments** partially supported

Power series in Arb

Arithmetic in $\mathbb{R}[[x]]/\langle x^n \rangle$ and $\mathbb{C}[[x]]/\langle x^n \rangle$ is an essential feature

Applications

- Explicit computation of number sequences
- Limit computations (removing singularities)
- Numerical integration
- Isolating function zeros

- Fast + numerically stable multiplication using scaling, splitting into blocks and multiplying exactly over $\mathbb{Z}[x]$ via FLINT
- Effective bit complexity with $prec \sim n$ is typically $O(n^{2+\varepsilon})$
- Elementary functions (exp, log, ...) cost $O(1)$ multiplications
- Fast algorithms for some special functions

Example: high-order derivatives of $\zeta(s)$

Keiper-Li coefficients $\{\lambda_n\}_{n=1}^{\infty}$ are defined by

$$\log\left(2\,\xi\left(\frac{x}{x-1}\right)\right) = \sum_{n=1}^{\infty} \lambda_n x^n, \quad \xi(s) = \frac{1}{2}s(s-1)\pi^{-s/2}\Gamma(s/2)\zeta(s)$$

Keiper (1992):	Riemann hypothesis $\Rightarrow \forall n : \lambda_n > 0$
Li (1997):	Riemann hypothesis $\Leftarrow \forall n : \lambda_n > 0$

Keiper conjectured $2\lambda_n \approx \log n - \log(2\pi) + \gamma - 1$

Experimental verification

Evaluate at $s \to s + x \in \mathbb{C}[[x]]/\langle x^{n+1}\rangle$, read off coefficients.
Need working precision of $\approx n$ bits for accurate value of λ_n.
Complexity: $O(n^{2+\varepsilon})$ (theory), implemented $O(n^{3+\varepsilon})$.

Timings for Keiper-Li coefficients

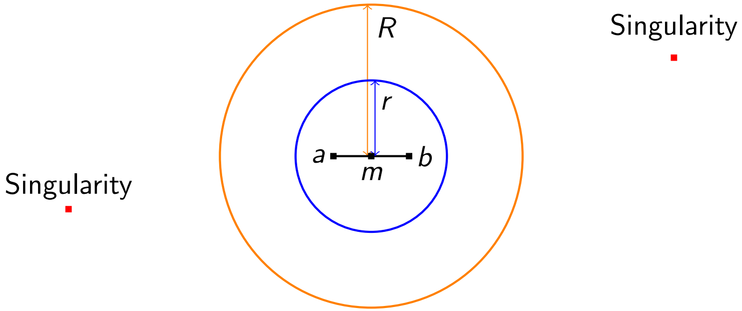
	$n = 1000$	$n = 10000$	$n = 100000$
Error bound	0.017	1.0	97
Power sum, 16 threads	0.048	47	65402
(Power sum, CPU time)	(0.65)	(693)	(1042210)
Bernoulli numbers	0.0020	0.19	59
Tail (binary splitting)	0.058	11	1972
Logarithm of power series	0.047	8.5	1126
$\log \Gamma(1+x)$ power series	0.019	3.0	1610
Power series composition	0.022	4.1	593
Total wall time (s)	0.23	84	71051
Memory	8 MB	730 MB	48700 MB

Computed: $\lambda_{100000} = 4.62580782406902231409416038\ldots$ (+2900 digits)
Keiper's asymptotic formula: $\lambda_{100000} \approx 4.62613$

Example: rigorous numerical integration

$$\int_a^b f(t)dt = F_N(c) - F_N(-c) + O\left(\left(\frac{r}{R}\right)^N\right), \quad c = (b-a)/2$$

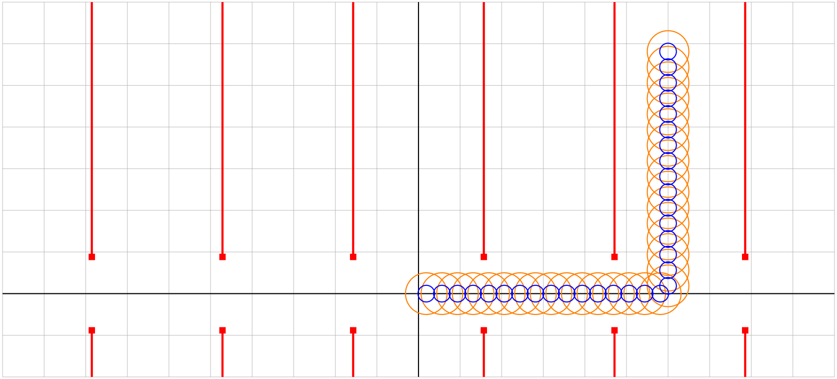
F_N: integrated order-N Taylor polynomial at $m = (a+b)/2$



Input: f, a, b, r, R. Algorithm subdivides until $|b-a| < r$, bounds error and chooses N using Cauchy integral formula ($|f(t)|$ on R-circle).

Example: incomplete elliptic integral “ab initio”

$$F(6+6i, 1/2), \qquad F(\phi, m) = \int_0^\phi \frac{1}{\sqrt{1 - m \sin^2 t}}\, dt$$



$R = 0.5, \quad r = 0.2 \quad$ branch points $(n+1/2)\pi \pm \log(1+\sqrt{2})i$

It's slow, but it works

```
30 digits:
    [7.41433... +/- 6.46e-29] + [1.84734... +/- 6.59e-30]j
    cpu/wall(s): 0.06 0.069
200 digits:
    [7.41433... +/- 2.58e-199] + [1.84734... +/- 6.21e-200]j
    cpu/wall(s): 1.24 1.268
1000 digits:
    [7.41433... +/- 5.05e-999] + [1.84734... +/- 4.09e-1000]j
    cpu/wall(s): 53.07 57.271
```

To do

- More automation, better adaptivity
- Singularities (e.g. $f(x) = \log^{\alpha}(x)\, x^{\beta}\, g(x)$)
- Faster algorithms (Gauss-Legendre, double exponential, ...)
- ODEs

What's in computing a function?

Example (simplified):

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

Input: $X = [m \pm r]$ with $x \in X$

$$f(X) \subseteq [A \pm E], \quad \underbrace{A = \sum_{k=0}^{N-1} \frac{X^k}{k!}}_{\text{Using ball arithmetic}}, \quad \underbrace{\left| \sum_{k=N}^{\infty} \frac{X^k}{k!} \right| \le E}_{\text{Upper bound}}$$

Better error propagation:

$$f(X) \subseteq [A \pm (E_1 + E_2)], \quad A = \sum_{k=0}^{N-1} \frac{m^k}{k!}, \quad \left| \frac{m^k}{k!} \right| \le E_1, \quad \sup_{t \in X} |e^m - e^t| \le E_2$$

Many possibilities for error propagation. . .

$f(x) = e^x$, input $X = [m \pm r]$ with $x \in X$

Best interval: $[\frac{1}{2}(a+b) \pm \frac{1}{2}(b-a)]$ where $a = e^{m-r}, b = e^{m+r}$

Best midpoint, best radius: $[e^m \pm r']$ where $r' = e^m(e^r - 1)$

Best midpoint, generic radius: $[e^m \pm r']$ where $r' = r \cdot (\sup_{t \in X} |f'(t)|) = r \cdot e^{m+r}$

Automatic propagation: $e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}$ (depends on algorithm). Acceptable for *point values* but may lead to *exponential slowdown* when used for *subdivision*!

Trimming: $[0.54402111 \pm 0.1] \to [0.544 \pm 0.101]$

Algorithm selection

For many functions, we want **different algorithms** for **different input**

Must also choose **internal parameters** (such as the number of terms N)

Philosophy

Choose heuristically, verify rigorously

- Heuristic step: choose N given $(x, prec)$
- Rigorous step: evaluate $\sum_{k=0}^{N-1} a_k x^k$, bound $\sum_{k=N}^{\infty} a_k x^k$

Make optimization orthogonal to correctness

Heuristics are easy to understand, fast (can use double precision)

It's sometimes fine to give up ($\sin(2^{2^{30}}) \in [-1, 1]$)

Code reliability – unit tests

- For each function, generate many ($10^3 - 10^6$) random inputs **non-uniformly**:
  - Integers, near-integers
  - Strange binary expansions like 0.1011111111111000000000000000001
  - Extremely large/small parts
- For each input, test **interval contract**:
  - Verify functional equation, for example: $X \subseteq \exp(\log(X))$
  - Compare different algorithms: $f_\mathrm{A}(X) \cap f_\mathrm{B}(X) \neq \emptyset$
  - Compare with different prec and N
- Deliberately **insert bugs** in the function. If the test code passes, the test is too weak. Strengthen the test until it fails!
- Tests must pass **valgrind** (no memory leaks, reading uninitialized memory, out of bounds accesses).

Elementary functions

Building blocks: exp, sin, cos, log, atan

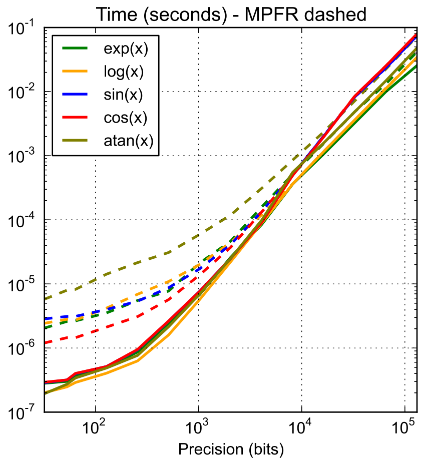
Below $\approx$ 4096 bits:

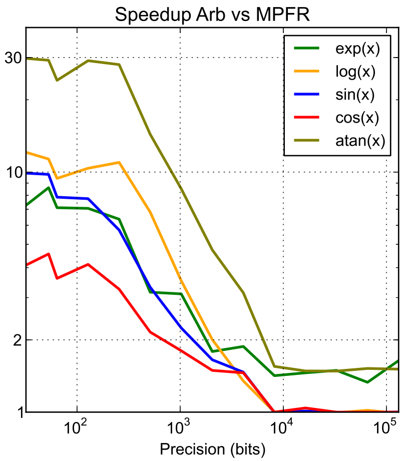
- Low-level code, fixed-point arithmetic using mpn layer of GMP
- Lookup tables and other dirty tricks
- Manual error analysis

Above $\approx$ 4096 bits:

- High-level code, ball arithmetic and big-integer arithmetic
- exp, atan using bit-burst evaluation (some speedup over MPFR)
- log, sin, cos currently via MPFR (should reimplement)

Elementary functions - performance





Input: $x = 1 + \sqrt{2}$

Performance at quad and quad-double precision

	prec	exp	sin	cos	log	atan
MPFR	113	5.76	7.29	3.42	8.01	21.30
libquadmath	113	4.51	4.71	4.57	5.39	4.32
QD	106	0.73	0.69	0.69	0.82	1.08
Arb	113	0.65	0.81	0.79	0.61	0.68
MPFR	212	7.87	9.23	5.06	12.60	33.00
QD	212	6.09	5.77	5.76	20.10	24.90
Arb	212	1.29	1.49	1.49	1.26	1.23

Timings in microseconds.

Recipe for elementary functions

$\exp(x)$ $\sin(x), \cos(x)$ $\log(1+x)$ $\operatorname{atan}(x)$

$\downarrow$

Domain reduction using π and $\log(2)$

$\downarrow$

$x \in [0, \log(2))$ $x \in [0, \pi/4)$ $x \in [0, 1)$ $x \in [0, 1)$

$\downarrow$

Argument-halving $r \approx 8$ times

$$\exp(x) = [\exp(x/2)]^2$$

$$\log(1+x) = 2 \log(\sqrt{1+x})$$

$\downarrow$

$x \in [0, 2^{-r})$

$\downarrow$

Taylor series (or bit-burst algorithm)

Better recipe at medium precision

$\exp(x)$ $\sin(x), \cos(x)$ $\log(1+x)$ $\operatorname{atan}(x)$

$$\downarrow$$

Domain reduction using π and $\log(2)$

$$\downarrow$$

$x \in [0, \log(2))$ $x \in [0, \pi/4)$ $x \in [0, 1)$ $x \in [0, 1)$

$$\downarrow$$

Lookup table with $2^r \approx 2^8$ entries

$$\exp(t+x) = \exp(t)\exp(x)$$

$$\log(1+t+x) = \log(1+t) + \log(1+x/(1+t))$$

$$\downarrow$$

$$x \in [0, 2^{-r})$$

$$\downarrow$$

Taylor series

Optimizing lookup tables

$m = 2$ tables with $2^5 + 2^5$ entries gives same reduction as
$m = 1$ table with 2^{10} entries

Function	Precision	m	r	Entries	Size (KiB)
exp	≤ 512	1	8	178	11.125
exp	≤ 4608	2	5	23+32	30.9375
sin	≤ 512	1	8	203	12.6875
sin	≤ 4608	2	5	26+32	32.625
cos	≤ 512	1	8	203	12.6875
cos	≤ 4608	2	5	26+32	32.625
log	≤ 512	2	7	128+128	16
log	≤ 4608	2	5	32+32	36
atan	≤ 512	1	8	256	16
atan	≤ 4608	2	5	32+32	36
Total					236.6875

Taylor series of elementary functions

Example: $\operatorname{atanh}(x)/x$

$$\begin{aligned} &1 + \frac{x^2}{3} + \frac{x^4}{5} + \frac{x^6}{7} \\ &+ \frac{x^8}{9} + \frac{x^{10}}{11} + \frac{x^{12}}{13} + \frac{x^{14}}{15} \\ &+ \frac{x^{16}}{17} + \frac{x^{18}}{19} + \frac{x^{20}}{21} + \frac{x^{22}}{23} \end{aligned}$$

(We use the Taylor series of exp, sinh, sin, cos, atan, atanh. The algorithms are almost identical.)

Rectangular splitting

$$\begin{aligned} &\left[1 + \frac{x^2}{3} + \frac{x^4}{5} + \frac{x^6}{7}\right] \\ + \; x^8 &\left[\frac{1}{9} + \frac{x^2}{11} + \frac{x^4}{13} + \frac{x^6}{15}\right] \\ + \; x^{16} &\left[\frac{1}{17} + \frac{x^2}{19} + \frac{x^4}{21} + \frac{x^6}{23}\right] \end{aligned}$$

Inner polynomials: powers x^2, x^4, x^6

Outer polynomial in x^8: Horner's rule

Optimal: split at $\sqrt{\text{number of terms}}$

Multiplying instead of dividing

$X + Y \cdot c$ is much cheaper than $X + Y/c$

$$\frac{1}{3\cdot 5\cdot 7}\left[(3\cdot 5\cdot 7) + (5\cdot 7)x^2 + (3\cdot 7)x^4 + (3\cdot 5)x^6\right]$$
$$+\frac{1}{9\cdot 11\cdot 13}\left[(11\cdot 13)x^8 + (9\cdot 13)x^{10} + (9\cdot 11)x^{12}\right]$$
$$+\frac{1}{15\cdot 17\cdot 19}\left[(17\cdot 19)x^{14} + (15\cdot 19)x^{16} + (15\cdot 17)x^{18}\right]$$
$$+\frac{1}{21\cdot 23}\left[23x^{20} + 21x^{22}\right]$$

Optimal: minimize number of divisions subject to $c < 2^{64}$ (or $c < 2^{32}$)

Combined rectangular splitting and denominator collection

Problem: optimal points for nonscalar multiplications and scalar divisions don't coincide!

Bad solution: extra nonscalar multiplications:

$$\frac{1}{A}[(\ldots) + x^8(\ldots)] + x^8\frac{1}{B}[(\ldots) + x^8(\ldots)]$$

Bad solution: extra scalar divisions:

$$\frac{1}{A}[(\ldots)] + x^8[\frac{1}{A}(\ldots) + \frac{1}{B}[(\ldots) + x^8(\ldots)]]$$

Better solution: 1 scalar mul + 1 scalar div instead of 1 scalar div

$$\frac{1}{A}[(\ldots) + x^8[(\ldots) + \frac{A}{B}[(\ldots) + x^8(\ldots)]]]$$

Implementation and correctness

- Fixed-point arithmetic: $n+1$ words for value in $[0, 2^{64})$ with ulp 2^{-64n}
- Signed values are implicit or use twos complement (tricky)
- Verification of Taylor series algorithm uses a separate computation
  - Summation is executed with symbolic variables that track possible range of values + possible range of error
  - Overflow is checked to be impossible at each step
  - Bound for the final ulp error at the end
- Error analysis by hand for all other steps
- Lookup tables are tested against MPFR
- Unit tests check 10^5 to 10^6 randomly generated inputs

Mathematical constants

π, e, $\log(2)$, Catalan's constant (etc.) use generic code for asymptotically fast (binary splitting) evaluation of hypergeometric series

$$\sum_{k=0}^{\infty} f(k), \quad f(k+1)/f(k) \in \mathbb{Q}(k)$$

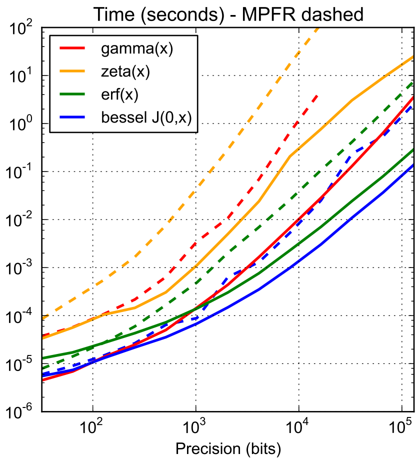
Euler's constant ($\gamma = 0.577\ldots$) uses the fastest known algorithm, due to Brent and McMillan (1980). Requires asymptotic expansions related to Bessel functions.

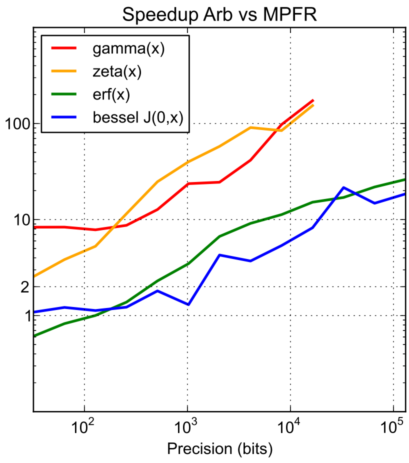
Until recently, this algorithm was heuristic! Error bounds derived in 2013 in joint work with Richard Brent.

R.P. Brent, F.J. *A bound for the error term in the Brent-McMillan algorithm*. To appear in *Math. Comp.*

Ball arithmetic useful for performance: p-bit numbers vs $O(p \log p)$

Special functions - performance





Input: $x = 1 + \sqrt{2}$

Gamma, zeta, polylogarithms

$\Gamma(s) \qquad \log \Gamma(s) \qquad \psi(s) = \Gamma'(s)/\Gamma(s)$

$\zeta(s,z) = \sum_{k=0}^{\infty} (z+k)^{-s} \qquad \mathrm{Li}_s(z) = \sum_{k=1}^{\infty} z^k k^{-s}$

Standard methods: Stirling's series, Euler-Maclaurin summation, functional equations

Some new bounds for error terms

Some new tricks to make things fast

More details in:

- F. J. *Rigorous high-precision computation of the Hurwitz zeta function and its derivatives*. Numerical Algorithms, 2014.
- F. J. *Evaluating parametric holonomic sequences using rectangular splitting*. ISSAC 2014.
- F. J. *Fast and rigorous computation of special functions to high precision*. PhD thesis, RISC, 2014.
- Arb documentation

An amazing algorithm for Bernoulli numbers

How to generate $B_0, B_1, \ldots, B_{10,000}$ in two seconds!

$$B_{2n} = (-1)^{n+1} \frac{2(2n)!}{(2\pi)^{2n}} \zeta(2n), \quad \zeta(s) = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \ldots$$

Go backwards and recycle the powers:

$$\frac{1}{k^{s-2}} = \frac{1}{k^s} \cdot k^2 \leftarrow \text{multiply bignum by single word}$$

Complexity is $O(n^{3+\varepsilon})$, but in practice an order of magnitude faster than the $O(n^{2+\varepsilon})$ power series algorithm

I first saw this algorithm in a blog post by Remco Bloemen
`http://2π.com/09/11/even-faster-zeta-calculation`

Elliptic and modular functions

Elliptic functions ($\wp(z,\tau)$ and derivatives), modular forms (Dedekind η-function, j-invariant, Eisenstein series, ...) are computed using theta series, e.g.:

$$\theta_3(z,\tau) = 1 + 2\sum_{n=1}^{\infty} q^{n^2}\cos(2n\pi z), \quad q = \exp(\pi i \tau)$$

Modular transformations $\tau \to (a\tau+b)/(c\tau+d)$ are applied until approximately $|q| < \exp(-\pi\sqrt{3}/2) \approx 0.0659$.

Complete elliptic integrals $K(m)$, $E(m)$ are computed using the arithmetic-geometric mean

To be done: incomplete elliptic integrals, more versions (Jacobian elliptic functions, etc.)

Hypergeometric functions

Many special functions can be expressed using

$$ {}_pF_q(a_1 \ldots a_p; b_1 \ldots b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \cdots (a_p)_k}{(b_1)_k \cdots (b_q)_k} \frac{z^k}{k!} $$

Most important special cases: confluent functions (${}_0F_1$, ${}_1F_1$, ${}_2F_0$), Gauss hypergeometric function ${}_2F_1$.

Arb supports evaluation for $a_i, b_i, z \in \mathbb{C}$ or $\mathbb{C}[[x]]/\langle x^n \rangle$, assuming that the sum is (rapidly) convergent. The tail is bounded nearly optimally by a geometric series:

$$ \left| \sum_{k=N}^{\infty} T(k) \right| \le |T(N)| \sum_{k=0}^{\infty} C^k $$

Confluent hypergeometric functions

Building blocks and glue

- Convergent series ${}_0F_1(a, z)$, ${}_1F_1(a, b, z)$
- Asymptotic series for $U(a, b, z)$ or ${}_2F_0(a, b, 1/z)$ as $|z| \to \infty$ (error bounds: Olver, DLMF 13.7)
- Ball arithmetic & power series arithmetic

$$\Downarrow \Downarrow \Downarrow$$

Many special cases

- Exponential integrals $\Gamma(a, z)$, $E_a(z)$, $\text{erf}(z)$, $\text{erfc}(z)$
- Bessel functions $K_a(z)$, $K_a(z)$
- (To be done: $Y_a(z)$, $I_a(z)$, Airy, Kelvin, Whittaker functions ...)

Example: modified Bessel function of the second kind

Case 1: $|z| \approx \infty$: asymptotic series

$$K_a(z) = \left(\frac{\pi}{2z}\right)^{1/2} e^{-z} U^*(a+\tfrac{1}{2}, 2a+1, 2z), \quad U^* \sim {}_2F_0(\ldots, -\tfrac{1}{2z})$$

Case 2: $|z| \approx 0$ and $a \notin \mathbb{Z}$: convergent series

$$K_a(z) = \frac{1}{2}\frac{\pi}{\sin(\pi a)}\left[\left(\frac{z}{2}\right)^{-a} {}_0\widetilde{F}_1\left(1-a, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^{a} {}_0\widetilde{F}_1\left(1+a, \frac{z^2}{4}\right)\right]$$

Case 3: $|z| \approx 0$ and $a \in \mathbb{Z}$: parameter limit

$$\lim_{\varepsilon \to 0} K_{a+\varepsilon}(z)$$

as in (Case 2), but with $\mathbb{C}[[\varepsilon]]/\langle \varepsilon^2 \rangle$ arithmetic

To do for hypergeometric functions

- Tune working precision, algorithm selection
- Reduced cancellation / better error propagation
  - **Bounds for function derivatives needed**
- Lots of low-level optimization (save precious CPU cycles)
- Complete algorithm for ${}_2F_1$ (and ${}_3F_2 \ldots$?)
- Bit-burst evaluation (see Mezzarobba's NumGfun)
- Asymptotic expansions for higher hypergeometrics?
  - **Error bounds needed**