Computing with precision

Fredrik Johansson
Inria Bordeaux

X, Mountain View, CA
January 24, 2019

Computing with real numbers

How can we represent

`3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862...`

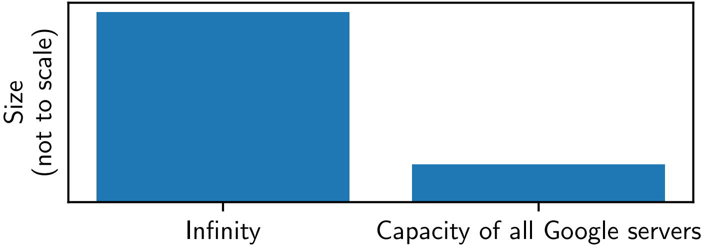
on a computer?

Computing with real numbers

How can we represent

`3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862...`

on a computer?



Consequences of numerical approximations

Mildly annoying:

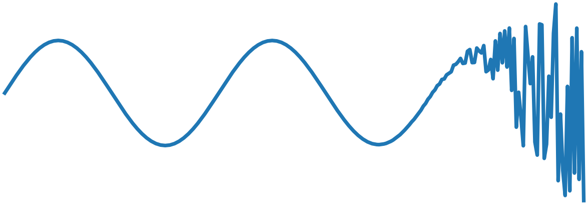
```
>>> 0.3 / 0.1
2.9999999999999996
```

Consequences of numerical approximations

Mildly annoying:

```
>>> 0.3 / 0.1
2.9999999999999996
```

Bad:

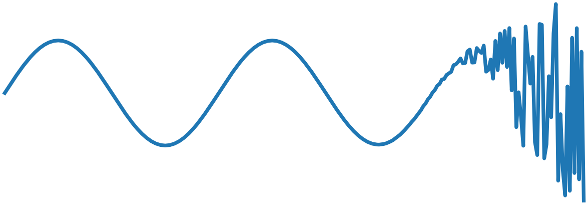


Consequences of numerical approximations

Mildly annoying:

```
>>> 0.3 / 0.1
2.9999999999999996
```

Bad:



Very bad:

Ariane 5 rocket explosion, Patriot missile accident, sinking of the Sleipner A offshore platform…

Precision in practice

Most scientific computing

`float` $p = 24$ ↓

`double` $p = 53$ ↓

3.14159265358979323846264338327950288419716939937...

Precision in practice

Most scientific computing

$$\frac{\text{Hydrogen atom}}{\text{Observable universe}} \approx 10^{-37}$$

`float` $p = 24$

`double` $p = 53$

`double-double` $p = 106$

`quad-double` $p = 212$

3.14159265358979323846264338327950288419716939937...

Precision in practice

Most scientific computing

$$\frac{\text{Hydrogen atom}}{\text{Observable universe}} \approx 10^{-37}$$

`float` $p = 24$

`double` $p = 53$

`double-double` $p = 106$

`quad-double` $p = 212$

3.14159265358979323846264338327950288419716939937...

`bfloat16` $p = 8$

(`int8`, `posit`, ...)

Computer graphics
Machine learning

Precision in practice

Most scientific computing

$$\frac{\text{Hydrogen atom}}{\text{Observable universe}} \approx 10^{-37}$$

`float` $p = 24$

`double` $p = 53$

`double-double` $p = 106$

`quad-double` $p = 212$

3.14159265358979323846264338327950288419716939937...

`bfloat16` $p = 8$

(`int8`, `posit`, ...)

Arbitrary-precision arithmetic

Computer graphics
Machine learning

Unstable algorithms
Dynamical systems
Computer algebra
Number theory

Different levels of strictness...

Error on sum of N terms with errors $|\varepsilon_k| \leq \varepsilon$?

Different levels of strictness...

Error on sum of N terms with errors $|\varepsilon_k| \leq \varepsilon$?

Worst-case error analysis

$N\varepsilon$ – will need $\log_2 N$ bits higher precision

Different levels of strictness...

Error on sum of N terms with errors $|\varepsilon_k| \leq \varepsilon$?

Worst-case error analysis

$N\varepsilon$ – will need $\log_2 N$ bits higher precision

Probabilistic error estimate

$O(\sqrt{N}\varepsilon)$ – assume errors probably cancel out

Different levels of strictness...

Error on sum of N terms with errors $|\varepsilon_k| \leq \varepsilon$?

Worst-case error analysis

$N\varepsilon$ – will need $\log_2 N$ bits higher precision

Probabilistic error estimate

$O(\sqrt{N}\varepsilon)$ – assume errors probably cancel out

Who cares?

- *Can check that solution is reasonable once it's computed*
- *Don't need an accurate solution, because we are solving the wrong problem anyway* (said about ML)

Error analysis

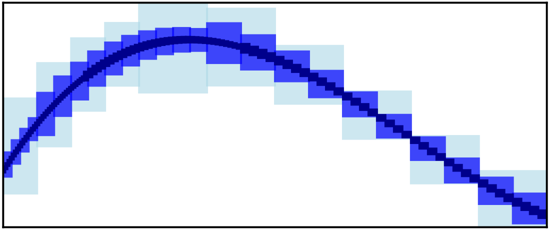
- Time-consuming, prone to human error
- Does not compose
  - $f(x), g(x)$ with error ε tells us nothing about $f(g(x))$
- Bounds are not enforced in code

```
y = g(x);        /* 0.01 error */
r = f(y);        /* amplifies error at most 10X */
/* now r has error <= 0.1 */

y = g_fast(x);  /* 0.02 error */
r = f(y);        /* amplifies error at most 10X */
/* now r has error <= 0.1 */     BUG
```

- Computer-assisted formal verification is improving – but still limited in scope

Interval arithmetic

Represent $x \in \mathbb{R}$ by an enclosure $x \in [a, b]$, and automatically propagate rigorous enclosures through calculations



If we are unlucky, the enclosure can be $[-\infty, +\infty]$

Dependency problem: $[-1, 1] - [-1, 1] = [-2, 2]$

Solutions:

- Higher precision
- Interval-aware algorithms

Lazy infinite-precision real arithmetic

Using functions

```
prec = 64
while True:
    y = f(prec)
    if is_accurate_enough(y):
        return y
    else:
        prec *= 2
```

Using symbolic expressions

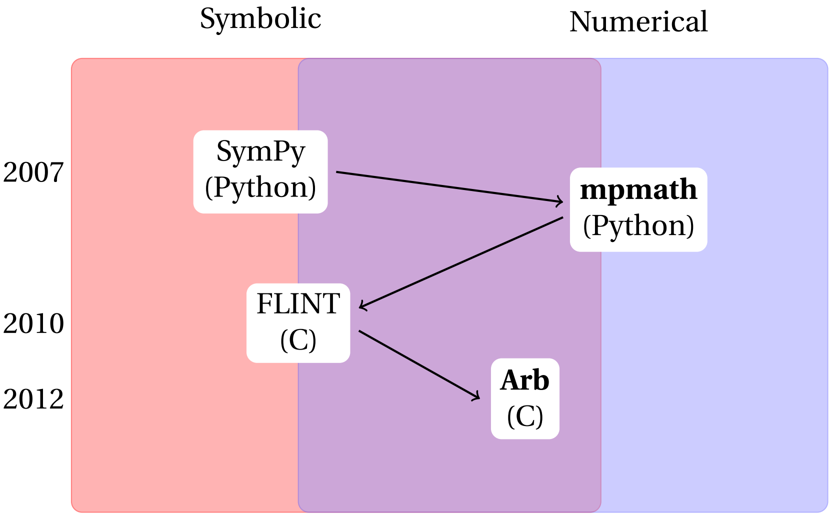
$\cos(2\pi) - 1$ becomes a DAG `(- (cos (* 2 pi)) 1)`

Tools for arbitrary-precision arithmetic

- Mathematica, Maple, Magma, Matlab Multiprecision Computing Toolbox (non-free)
- SageMath, Pari/GP, Maxima (open source computer algebra systems)
- ARPREC (C++/Fortran)
- CLN, Boost Multiprecision Library (C++)
- GMP, MPFR, MPC, MPFI (C)
- FLINT, Arb (C)
- GMPY, SymPy, mpmath, Python-FLINT (Python)
- BigFloat, Nemo.jl (Julia)

And many others...

My work on open source software



Also: SageMath, Nemo.jl (Julia), Python-FLINT (Python)

mpmath

`http://mpmath.org`, BSD, Python

- Real and complex arbitrary-precision floating-point
- Written in pure Python (portable, accessible, slow)
- Optional GMP backend (GMPY, SageMath)
- Designed for easy interactive use
  (inspired by Matlab and Mathematica)
- Plotting, linear algebra, calculus (limits, derivatives, integrals, infinite series, ODEs, root-finding, inverse Laplace transforms), Chebyshev and Fourier series, special functions
- 50 000 lines of code, $\approx$ 20 major contributors

mpmath

```
>>> from mpmath import *
>>> mp.dps = 50; mp.pretty = True
>>> +pi
3.1415926535897932384626433832795028841971693993751
>>> findroot(sin, 3)
3.1415926535897932384626433832795028841971693993751
```

mpmath

```
>>> from mpmath import *
>>> mp.dps = 50; mp.pretty = True
>>> +pi
3.1415926535897932384626433832795028841971693993751
>>> findroot(sin, 3)
3.1415926535897932384626433832795028841971693993751
```

(More: `http://fredrikj.net/blog/2011/03/100-mpmath-one-liners-for-pi/`)

```
>>> 16*acot(5)-4*acot(239)
>>> 8/(hyp2f1(0.5,0.5,1,0.5)*gamma(0.75)/gamma(1.25))**2
>>> nsum(lambda k: 4*(-1)**(k+1)/(2*k-1), [1,inf])
>>> quad(lambda x: exp(-x**2), [-inf,inf])**2
>>> limit(lambda k: 16**k/(k*binomial(2*k,k)**2), inf)
>>> (2/diff(erf, 0))**2
...
```

FLINT (Fast Library for Number Theory)

`http://flintlib.org`, LGPL, C, maintained by William Hart

- Exact arithmetic
  - Integers, rationals, integers mod n, finite fields
  - Polynomials and matrices over all the above types
  - Exact linear algebra
  - Number theory functions (factorization, etc.)
- Backend library for computer algebra systems (including SageMath, Singular, Nemo)
- Combine asymptotically fast algorithms with low-level optimizations (design for both tiny and huge operands)
- Builds on GMP and MPFR
- 400 000 lines of code, 5000 functions, many contributors
- Extensive randomized testing

Arb (arbitrary-precision ball arithmetic)

http://arblib.org, LGPL, C

- Mid-rad interval ("ball") arithmetic:

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

- Goal: extend FLINT to real and complex numbers
- Goal: all arbitrary-precision numerical functionality in mpmath/Mathematica/Maple. . ., but with rigorous error bounds **and** faster (often 10-10000$\times$)
- Linear algebra, polynomials, power series, root-finding, integrals, special functions
- 170 000 lines of code, 3000 functions, $\approx$ 5 major contributors

Interfaces

Example: Python-FLINT

```
>>> from flint import *
>>> ctx.dps = 25
>>> arb("0.3") / arb("0.1")
[3.000000000000000000000000 +/- 2.17e-25]

>>> (arb.pi()*10**100 + arb(1)/1000).sin()
[+/- 1.01]
>>> f = lambda: (arb.pi()*10**100 + arb(1)/1000).sin()
>>> good(f)
[0.0009999998333333416666664683 +/- 4.61e-29]

>>> a = fmpz_poly([1,2,3])
>>> b = fmpz_poly([2,3,4])
>>> a.gcd(a * b)
3*x^2 + 2*x + 1
```

Examples

- Linear algebra
- Special functions
- Integrals, derivatives

Example: linear algebra

Solve $Ax = b$
$A = n \times n$ Hilbert matrix, $A_{i,j} = 1/(i + j + 1)$
$b =$ vector of ones

What is the middle element of x?

$$
\begin{pmatrix}
1 & 1/2 & 1/3 & 1/4 & \dots \\
1/2 & 1/3 & 1/4 & 1/5 & \dots \\
1/3 & 1/4 & 1/5 & 1/6 & \dots \\
1/4 & 1/5 & 1/6 & 1/7 & \dots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{pmatrix}
\begin{pmatrix}
x_0 \\ \vdots \\ \boxed{x_{\lfloor n/2 \rfloor}} \\ \vdots \\ x_{n-1}
\end{pmatrix}
=
\begin{pmatrix}
1 \\ 1 \\ \vdots \\ 1 \\ 1
\end{pmatrix}
$$

Example: linear algebra

SciPy, standard (53-bit) precision:

```
>>> from scipy import ones
>>> from scipy.linalg import hilbert, solve
>>> def scipy_sol(n):
...     A = hilbert(n)
...     return solve(A, ones(n))[n//2]
```

mpmath, 24-digit precision:

```
>>> from mpmath import mp
>>> mp.dps = 24
>>> def mpmath_sol(n):
...     A = mp.hilbert(n)
...     return mp.lu_solve(A, mp.ones(n,1))[n//2,0]
```

Example: linear algebra

```
>>> for n in range(1,15):
...     a = scipy_sol(n); b = mpmath_sol(n)
...     print("{0: <2} {1: <15}  {2}".format(n, a, b))
...
1   1.0              1.0
2   6.0              6.0
3   -24.0            -24.000000000000000000002
4   -180.0           -180.000000000000000000013
5   630.000000005    630.00000000000000001195
6   5040.00000066    5040.00000000000000029801
7   -16800.0000559   -16799.9999999999999952846
8   -138600.003817   -138599.999999999992072999
9   450448.757784    450449.99999999326221191
10  3783740.26705    3783779.99999993033735503
11  -12112684.2704   -12108095.9999902703235601
12  -98905005.0899   -102918815.993729874568379
13  -937054504.99    325909583.09253012248934
14  -312986201.415   2793510502.10076485899567
```

Example: linear algebra

Using Arb (via Python-FLINT)
Default precision is 53 bits (15 digits)

```
>>> from flint import *
>>> def arb_sol(n):
...     A = arb_mat.hilbert(n,n)
...     return A.solve(arb_mat(n,1,[1]*n),nonstop=True)[n//2,0]
```

Example: linear algebra

```
>>> for n in range(1,15):
...     c = arb_sol(n)
...     print("{0: <2} {1}".format(n, c))
...
1    1.00000000000000
2    [6.00000000000000 +/- 5.78e-15]
3    [-24.0000000000 +/- 1.65e-12]
4    [-180.000000000 +/- 4.87e-10]
5    [630.00000 +/- 1.03e-6]
6    [5040.00000 +/- 2.81e-6]
7    [-16800.000 +/- 3.03e-4]
8    [-138600.0 +/- 0.0852]
9    [4.505e+5 +/- 57.5]
10   [3.78e+6 +/- 6.10e+3]
11   [-1.2e+7 +/- 3.37e+5]
12   nan
13   nan
14   nan
```

Example: linear algebra

```
>>> for n in range(1,15):
...     c = good(lambda: arb_sol(n))  # adaptive precision
...     print("{0: <2} {1}".format(n, c))
...
1    1.00000000000000
2    [6.00000000000000 +/- 2e-19]
3    [-24.0000000000000 +/- 1e-18]
4    [-180.000000000000 +/- 1e-17]
5    [630.000000000000 +/- 2e-16]
6    [5040.00000000000 +/- 1e-16]
7    [-16800.0000000000 +/- 1e-15]
8    [-138600.000000000 +/- 1e-14]
9    [450450.000000000 +/- 1e-14]
10   [3783780.00000000 +/- 3e-13]
11   [-12108096.0000000 +/- 3e-12]
12   [-102918816.000000 +/- 3e-11]
13   [325909584.000000 +/- 3e-11]
14   [2793510720.00000 +/- 3e-10]
```

Example: linear algebra

```
>>> n = 100
>>> good(lambda: arb_sol(n), maxprec=10000)
[-1.01540383154230e+71 +/- 3.01e+56]
```

Higher precision:

```
>>> ctx.dps = 75
>>> good(lambda: arb_sol(n), maxprec=10000)
[-1015403831542296990505387709805677848976826547302941869
33704066855192000.000 +/- 3e-8]
```

Exact solution using FLINT:

```
>>> fmpq_mat.hilbert(n,n).solve(fmpq_mat(n,1,[1]*n))[n//2,0]
-1015403831542296990505387709805677848976826547302941869
33704066855192000
```

Overhead of arbitrary-precision arithmetic

Time to multiply two 1000×1000 matrices?

OpenBLAS (1 thread): 0.066 s

Overhead of arbitrary-precision arithmetic

Time to multiply two 1000×1000 matrices?

OpenBLAS (1 thread): 0.066 s

mpmath, $p = 53$: 4102 s (60 000 times slower)
mpmath, $p = 212$: 4334 s
mpmath, $p = 3392$: 6475 s

Overhead of arbitrary-precision arithmetic

Time to multiply two 1000×1000 matrices?

OpenBLAS (1 thread): 0.066 s

mpmath, $p = 53$: 4102 s (60 000 times slower)
mpmath, $p = 212$: 4334 s
mpmath, $p = 3392$: 6475 s

Julia BigFloat, $p = 53$: 405 s (6 000 times slower)
Julia BigFloat, $p = 212$: 462 s
Julia BigFloat, $p = 3392$: 2586 s

Overhead of arbitrary-precision arithmetic

Time to multiply two 1000×1000 matrices?

OpenBLAS (1 thread): 0.066 s

mpmath, $p = 53$: 4102 s (60 000 times slower)
mpmath, $p = 212$: 4334 s
mpmath, $p = 3392$: 6475 s

Julia BigFloat, $p = 53$: 405 s (6 000 times slower)
Julia BigFloat, $p = 212$: 462 s
Julia BigFloat, $p = 3392$: 2586 s

Arb, $p = 53$: 3.6 s (50 times slower)
Arb, $p = 212$: 8.2 s
Arb, $p = 3392$: 115 s

Overhead of arbitrary-precision arithmetic

Time to multiply two 1000×1000 matrices?

OpenBLAS (1 thread): 0.066 s

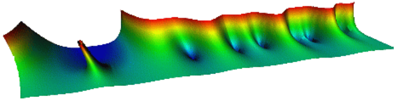
mpmath, $p = 53$: 4102 s (60 000 times slower)
mpmath, $p = 212$: 4334 s
mpmath, $p = 3392$: 6475 s

Julia BigFloat, $p = 53$: 405 s (6 000 times slower)
Julia BigFloat, $p = 212$: 462 s
Julia BigFloat, $p = 3392$: 2586 s

Arb, $p = 53$: 3.6 s (50 times slower)
Arb, $p = 212$: 8.2 s
Arb, $p = 3392$: 115 s

State of the art (small p): floating-point expansions on GPUs
(ex.: Joldes, Popescu and Tucker, 2016) – but limited scope

Special functions



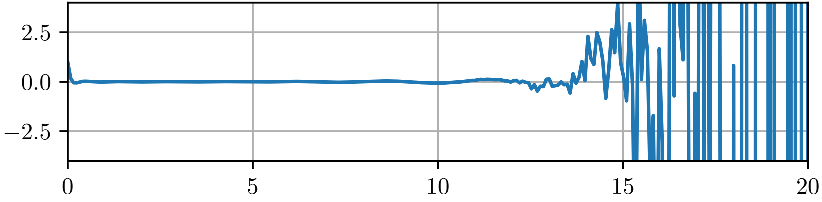
NIST Digital Library of Mathematical Functions

mpmath

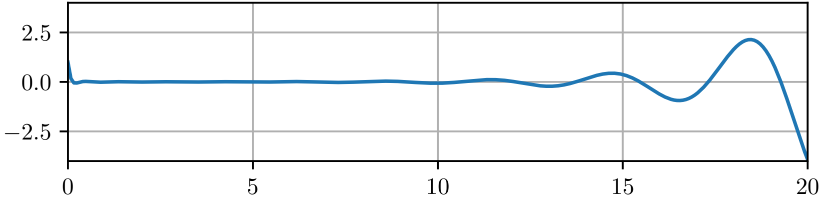
Arb

A good case for arbitrary-precision arithmetic...

`scipy.special.hyp1f1(-50,3,x)`



`mpmath.hyp1f1(-50,3,x)`



Methods of computation

Taylor series, asymptotic series, integral representations (numerical integration), functional equations, ODEs, ...

Sources of error

Arithmetic error: $\sum_{k=0}^{N} \frac{x^k}{k!}$ (in finite precision)

Approximation error: $\left| \sum_{k=N+1}^{\infty} \frac{x^k}{k!} \right| \leq \varepsilon$

Composition: $f(x) = g(u(x), v(x)) \ldots$

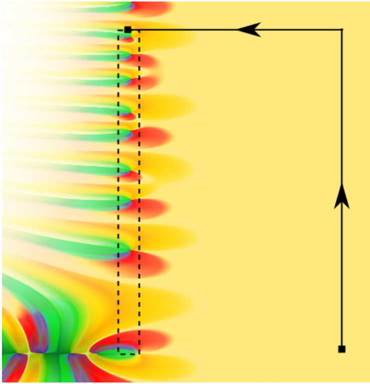
“Exact” numerical computing

Analytic formula → numerical solution → discrete solution

↑ Often involving special functions

- Complex path integrals → zero/pole count
- Special function values → integer sequences
- Numerical values → integer relations → exact formulas
- Constructing finite fields $GF(p^k)$: exponential sums → Gaussian period minimal polynomials
- Constructing elliptic curves with desired properties: modular forms → Hilbert class polynomials

Example: zeros of the Riemann zeta function



Number of zeros of $\zeta(s)$ on $R = [0,1] + [0,T]i$:

$$N(T) - 1 = \frac{1}{2\pi i} \int_\gamma \frac{\zeta'(s)}{\zeta(s)} ds = \frac{\theta(T)}{\pi} +$$

$$\frac{1}{\pi} \operatorname{Im}\left[\int_{1+\varepsilon}^{1+\varepsilon+Ti} \frac{\zeta'(s)}{\zeta(s)} ds + \int_{1+\varepsilon+Ti}^{\frac{1}{2}+Ti} \frac{\zeta'(s)}{\zeta(s)} ds \right]$$

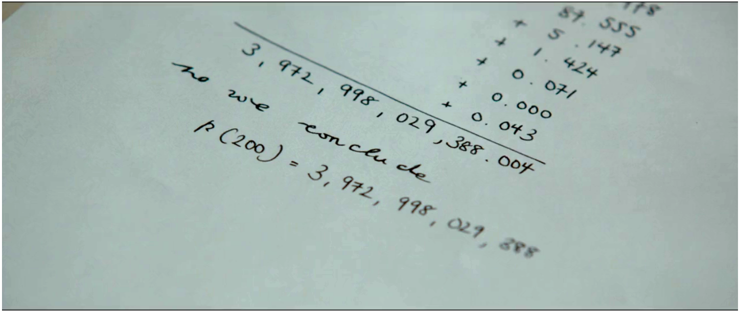
T	p	Time (s)	Eval	Sub	$N(T)$
10^3	32	0.51	1219	109	`[649.00000 +/- 7.78e-6]`
10^6	32	16	5326	440	`[1747146.00 +/- 4.06e-3]`
10^9	48	1590	8070	677	`[2846548032.000 +/- 1.95e-4]`

The integer partition function $p(n)$

$p(4) = 5$ since $(4) = (3+1) = (2+2) = (2+1+1) = (1+1+1+1)$

Hardy and Ramanujan, 1918; Rademacher 1937:

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



Scene from *The Man Who Knew Infinity*, 2015

Hold your horses...

THE ON–LINE ENCYCLOPEDIA
OF INTEGER SEQUENCES®

founded in 1964 by N. J. A. Sloane

A110375 Numbers n such that Maple 9.5, Maple 10, Maple 11 and Maple 12 give the wrong answers for the number of partitions of n. 2

11269, 11566, 12376, 12430, 12700, 12754, 15013, 17589, 17797, 18181, 18421, 18453, 18549, 18597, 18885, 18949, 18997, 20865, 21531, 21721, 21963, 22683, 23421, 23457, 23547, 23691, 23729, 23853, 24015, 24087, 24231, 24339, 24519, 24591, 24627, 24681, 24825, 24933, 25005, 25023, 25059, 25185, 25293, 27020 (list; graph; refs; listen; history; text; internal format)

OFFSET	1,1
COMMENTS	Based on various postings on the Web, sent to N. J. A. Sloane by R. J. Mathar. Thanks to several correspondents who sent information about other versions of Maple. Mathematica 6.0, DrScheme and pari-2.3.3 all give the correct answers. Ramanujan's congruence says that numbpart(5*k+4)==0 mod 5, so numbpart(11269)=...851==1 mod 5 can't be correct. [Robert Gerbicz, May 13 2008]
LINKS	Table of n, a(n) for n=1..44. Author?, Concerning this sequence
EXAMPLE	From PARI, the correct answer: numbpart(11269) 231139177231303975514411787649455628959060199360109972557851519105155 1761\ 80318215891795874905318274163248033071850 From Maple 11, incorrect: combinat[numbpart](11269); 2311391772313039755144117876494556289590601993601099725578515191051551761\ 80318215891795874905318274163248033071851 On the other hand, the old Maple 6 gives the correct answer.

Partition function in Arb

- Ball arithmetic guarantees the correct integer
- Optimal time complexity, ≈ 200 times faster than previous best implementation (Mathematica) in practice
- Used to prove 22 billion new congruences, for example:

$$p(999959^4 \cdot 29k + 28995221336976431135321047) \equiv 0$$

(mod 29) holds for all k

- Largest computed value of $p(n)$:

$$p(10^{20}) = \underbrace{18381765\ldots88091448}_{11\,140\,086\,260 \text{ digits}}$$

1 710 193 158 terms, 200 CPU hours, 130 GB memory

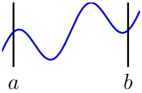
Numerical integration

$$\int_a^b f(x)dx$$

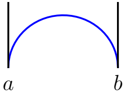
Methods specialized for high precision

- Degree-adaptive double exponential quadrature (mpmath)
- Convergence acceleration for oscillatory integrals (mpmath)
- Space/degree-adaptive Gauss-Legendre quadrature with error bounds based on complex magnitudes (Petras algorithm) (Arb)

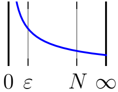
Typical integrals



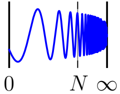
Analytic around $[a, b]$



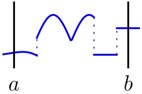
Bounded endpoint singularities (ex.: $\sqrt{1-x^2}$)



Smooth blow-up/decay (ex.: $\int_0^1 \log(x)dx$, $\int_0^\infty e^{-x}dx$)



Essential singularity, slow decay (ex.: $\int_1^\infty \frac{\sin(x)}{x}dx$)



Piecewise analytic (ex.: $\lfloor x \rfloor$, $|x|$, $\max(f(x), g(x))$)

Numerical integration with mpmath

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True

>>> quad(lambda x: exp(-x**2), [-inf, inf])**2
3.14159265358979323846264338328
>>> quad(lambda x: sqrt(1-x**2), [-1,1])*2
3.14159265358979323846264338328
```

Numerical integration with mpmath

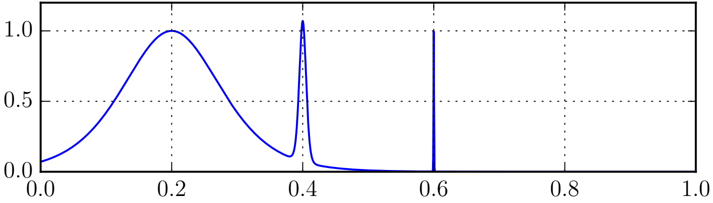
```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True

>>> quad(lambda x: exp(-x**2), [-inf, inf])**2
3.14159265358979323846264338328
>>> quad(lambda x: sqrt(1-x**2), [-1,1])*2
3.14159265358979323846264338328

>>> chop(quad(lambda z: 1/z, [1,j,-1,-j,1]))
(0.0 + 6.28318530717958647692528676656j)
```

Numerical integration with mpmath

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True

>>> quad(lambda x: exp(-x**2), [-inf, inf])**2
3.14159265358979323846264338328
>>> quad(lambda x: sqrt(1-x**2), [-1,1])*2
3.14159265358979323846264338328

>>> chop(quad(lambda z: 1/z, [1,j,-1,-j,1]))
(0.0 + 6.28318530717958647692528676656j)

>>> quadosc(lambda x: cos(x)/(1+x**2), [-inf, inf], omega=1)
1.15572734979092171791009318331
>>> pi/e
1.15572734979092171791009318331
```

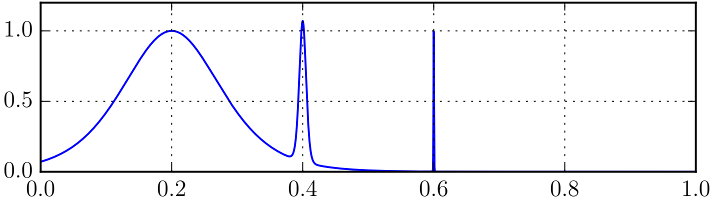
The spike integral (Cranley and Patterson, 1971)

$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6)) \; dx$$



The spike integral (Cranley and Patterson, 1971)

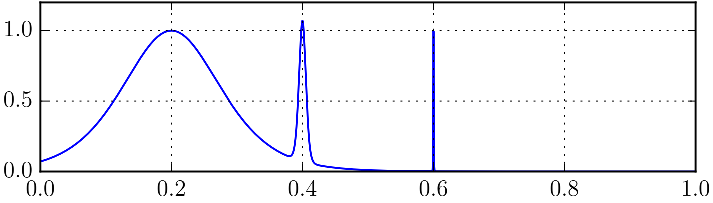
$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`: 0.209736

The spike integral (Cranley and Patterson, 1971)

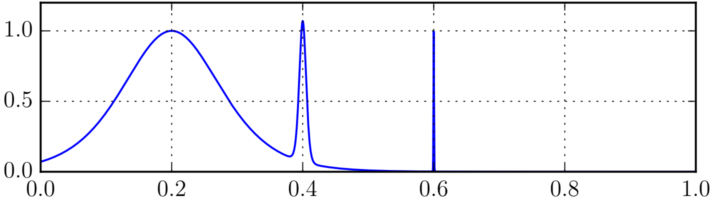
$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`:	0.209736
Octave `quad`:	0.209736, error estimate 10^{-9}

The spike integral (Cranley and Patterson, 1971)

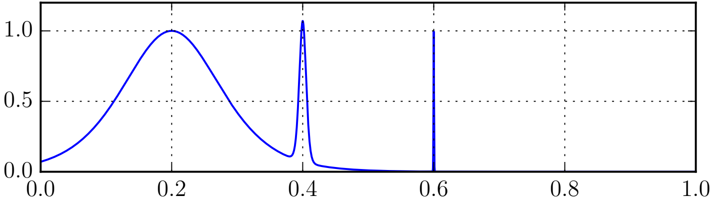
$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`:	0.209736
Octave `quad`:	0.209736, error estimate 10^{-9}
Sage `numerical_integral`:	0.209736, error estimate 10^{-14}

The spike integral (Cranley and Patterson, 1971)

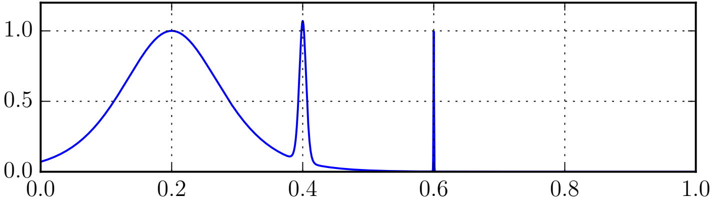
$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`:	0.209736
Octave `quad`:	0.209736, error estimate 10^{-9}
Sage `numerical_integral`:	0.209736, error estimate 10^{-14}
SciPy `quad`:	0.209736, error estimate 10^{-9}

The spike integral (Cranley and Patterson, 1971)

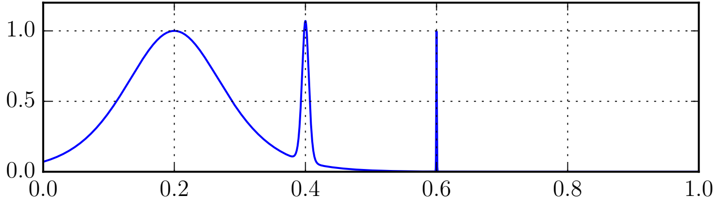
$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`:	0.209736
Octave `quad`:	0.209736, error estimate 10^{-9}
Sage `numerical_integral`:	0.209736, error estimate 10^{-14}
SciPy `quad`:	0.209736, error estimate 10^{-9}
mpmath `quad`:	0.209819

The spike integral (Cranley and Patterson, 1971)

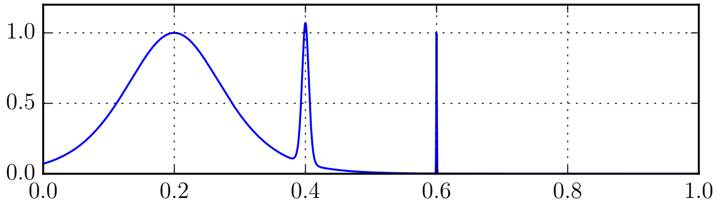
$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`:	0.209736
Octave `quad`:	0.209736, error estimate 10^{-9}
Sage `numerical_integral`:	0.209736, error estimate 10^{-14}
SciPy `quad`:	0.209736, error estimate 10^{-9}
mpmath `quad`:	0.209819
Pari/GP `intnum`:	0.211316

The spike integral (Cranley and Patterson, 1971)

$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx$$



Mathematica `NIntegrate`:	0.209736
Octave `quad`:	0.209736, error estimate 10^{-9}
Sage `numerical_integral`:	0.209736, error estimate 10^{-14}
SciPy `quad`:	0.209736, error estimate 10^{-9}
mpmath `quad`:	0.209819
Pari/GP `intnum`:	0.211316
Actual value:	0.210803

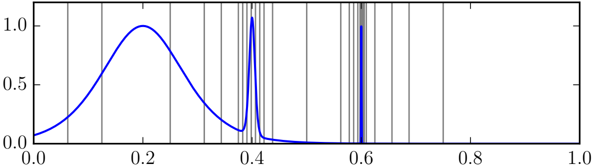
The spike integral

Using Arb:

```
>>> from flint import *
>>> f = lambda x, _: (10*x-2).sech()**2 +
...     (100*x-40).sech()**4 + (1000*x-600).sech()**6

>>> acb.integral(f, 0, 1)
[0.21080273550055 +/- 4.44e-15]

>>> ctx.dps = 300
>>> acb.integral(f, 0, 1)
[0.210802735500549277375643255705729154360909186436781190З
    478505058787206131281455002050586892615576418256930487
    967120600184392890901811133114479046741694620315482319
    853361121180728127354308183506890329305764794971077134
    710865180873848213386030655588722330743063348785462715
    31967986227310202562197239 8 +/- 3.29e-299]
```

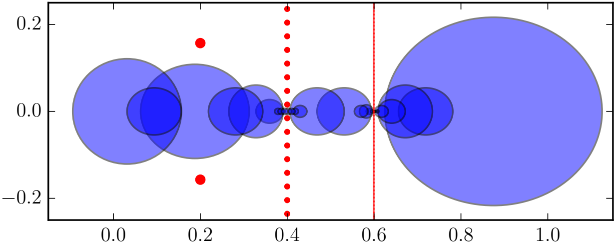
Adaptive subdivision

Arb chooses 31 subintervals, narrowest is 2^{-11}



Complex ellipses used for bounds

Red dots = poles



Derivatives

$$f'(x) \qquad f^{(n)}(x)$$

Integrating is easy, differentiating is hard

Derivatives

$$f'(x) \qquad f^{(n)}(x)$$

~~*Integrating is easy, differentiating is hard*~~

Everything is easy locally (for analytic functions)

Derivatives

$$f'(x) \qquad f^{(n)}(x)$$

~~*Integrating is easy, differentiating is hard*~~

Everything is easy locally (for analytic functions)

- Finite differences (mpmath)
- Complex integration (mpmath, Arb)
- Automatic differentiation (FLINT, Arb)

Numerical differentiation with mpmath

```
>>> mp.dps = 30; mp.pretty = True
>>> diff(exp, 1.0)
2.71828182845904523536028747135
>>> diff(exp, 1.0, 100) # 100th derivative
2.71828182845904523536028747135

>>> f = lambda x: nsum(lambda k: x**k/fac(k), [0,inf])
>>> diff(f, 1.0, 10)
2.71828182845904523536028747135

>>> diff(f, 1.0, 10, method="quad", radius=2)
(2.71828182845904523536028747135 + 9.52...e-37j)
```

Extreme differentiation

```
>>> ctx.cap = 1002            # set precision O(x^1002)
>>> x = arb_series([0,1])

>>> (x.sin() * x.cos())[1000] * arb.fac_ui(1000)
0
>>> (x.sin() * x.cos())[1001] * arb.fac_ui(1001)
[1.071508607186e+301 +/- 3.51e+288]

>>> x = fmpq_series([0,1])
>>> (x.sin() * x.cos())[1000] * fmpz.fac_ui(1000)
0
>>> (x.sin() * x.cos())[1001] * fmpz.fac_ui(1001)
1071508607186267320948425049060001810561404811705533607443
    750388370351051124936122493198378815695858127594672917
    553146825187145285692314043598457757469857480393456777
    482423098542107460506237114187795418215304647498358194
    126739876755916554394607706291457119647768654216766042
    9831652624386837205668069376
```

Example: testing the Riemann hypothesis

Define the *Keiper-Li coefficients* $\lambda_1, \lambda_2, \lambda_3, \ldots$ by

$$\log \xi \left(\frac{x}{x-1} \right) = -\log 2 + \sum_{n=1}^{\infty} \lambda_n x^n$$

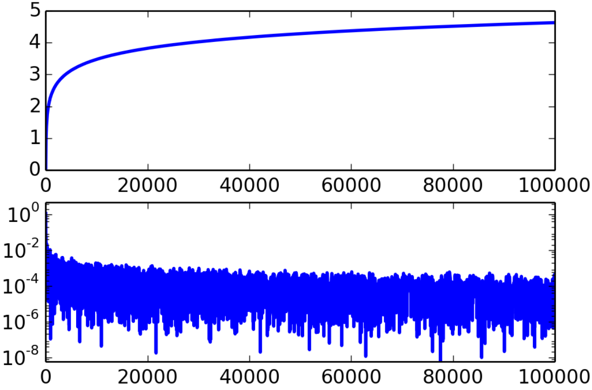
where $\xi(s) = \zeta(s) \cdot \frac{1}{2} s(s-1) \pi^{-s/2} \Gamma(s/2)$.

The Riemann hypothesis is equivalent to the statement

$$\lambda_n > 0 \text{ for all } n$$

(Keiper 1992 and Li 1997).

Example: testing the Riemann hypothesis



Top: computed λ_n values
Bottom: error of conjectured asymptote $(\log n - \log(2\pi) + \gamma - 1)/2$

Need $\approx n$ bits to get an accurate value for λ_n.

Conclusion

Arbitrary-precision arithmetic

- Power tool for difficult numerical problems
- Ball arithmetic is a natural framework for reliable numerics

Problems

- Mathematical problem $\to$ reliable numerical solution (often requires expertise in algorithms)
- Rigorous, performant algorithms for specific problems
- Performance on modern hardware (GPUs?)
- Formal verification