Why transcendentals and arbitrary precision?

Paul Zimmermann, INRIA LORRAINE Loria

December 15th, 2005

Why Transcendentals?

Some transcendentals today

Opteron, Linux 2.6.12, gcc 4.0.1, libc 2.3.5:

```
Testing function atan for exponent 0.
 rounding mode GMP_RNDU:
  1.507141 ulp(s) for x=5.27348750514293418412e-01
  wrong DR: x=8.71159292701253917812e-01 [-0.505215]

Testing function cbrt for exponent 0.
 rounding mode GMP_RNDN:
  wrong monotonicity for x=8.90550497574918109578e-01
            f(x-)=9.62098454219197263271e-01
      not <= f(x)=9.62098454219197152248e-01
```

Sparc, SunOS 5.7, cc Sun WorkShop 6:

```
Testing function exp for exponent 0.
 rounding mode GMP_RNDN:
  0.659120 ulp(s) for x=9.43344491255437844757e-01
 rounding mode GMP_RNDU:
  wrong DR: x=5.33824498679617898134e-01 [-0.295496]
Testing function pow for exponents 0 and 0.
 rounding mode GMP_RNDN:
  -0.522792 ulp(s) for x=9.91109071895216686698e-01
                     t=6.06627312254989226048e-01
Testing function tanh for exponent 0.
 rounding mode GMP_RNDN:
  1.771299 ulp(s) for x=5.19240368581155742334e-01
```

MIPS R16000, IRIX64 6.5, gcc 3.3:

```
Testing function tan for exponent 10.
 rounding mode GMP_RNDZ:
  -6.143332 ulp(s) for x=5.25427198389763360000e+02
  wrong DR: x=7.56078520967298570000e+02 [-4.523771]
```

Itanium 1, Linux 2.4.20, gcc 3.2.3, libc 2.2.5:

```
Testing function gamma for exponent 7.
 rounding mode GMP_RNDN:
  -610.873724 ulp(s) for x=1.22201576631543275653e+02
```

Pentium 4, Linux 2.6.11, gcc 3.4.3, libc 2.3.5:

```
Testing function acos for exponent 0.
 rounding mode GMP_RNDN:
  174.666207 ulp(s) for x=9.99579327845287135546e-01
```

Test case for:

`sources.redhat.com/bugzilla/show_bug.cgi?id=706`

```
Linux/x86, glibc 2.3.5, gcc 4.0.3 20051201 (pre):
  pow (0X1.FFFFFFFFFFFFFP-1, -0X1.000000000004P+54)
     MPFR (correct rounding): 0X1.D8E64B8D4DDBDP+2
       default-precision FPU: 0X1.D8E64B899BEB7P+2
      extended-precision FPU: 0X1.D8E64B899BEB7P+2
        double-precision FPU: 0X1.B4C90207CD76AP+5
```

Why correct rounding is needed?

- better accuracy
- better portability
- interval arithmetic (at least correct direction)
- error analysis (rigorous bound enough)
- many people believe transcendentals are reliable
- those who know write their own implementation (CATIA, Dassault Systèmes)

A lot of code involving a little floating-point will be written by many people who have ***never attended*** *my (nor anyone else's) numerical analysis classes. We had to enhance the likelihood that* ***their programs would get correct results.*** *At the same time we had to ensure that people who really are expert in floating-point could write* ***portable software*** *and prove that it worked, since so many of us would have to rely upon it. There were a lot of almost* ***conflicting requirements*** *on the way to a balanced design.*

William Kahan, An Interview with the Old Man of Floating-Point, February 1998.

Why not?

March 14, 2001: *"If **less than correct rounding** is specified [...] then is always possible that somebody will come up with a non-standard algorithm that is **more accurate AND faster**", "correctly-rounded transcendental functions seem to be **inherently much more expensive** than almost-correctly-rounded"*

June 20, 2001: *"with comparable care in coding, **correctly-rounded transcendental functions cost 2-4X more** than conventional functions that aim for perhaps 0.53 ulps worst error"*

*"If the cost could be gotten uniformly **down to 1.25X we'd just do it**"*

May 23, 2002 (Markstein's presentation): *"Darcy, Kahan, and Thomas all asked whether it would be* ***worth the costs, and who would benefit****"*

Zuras: "***difficulty of testing***", "*a set of* ***test cases*** *to verify whether an implementation conforms to the standard*"

Koev: *"difficulty of* ***proving correct rounding*** *for transcendentals"*

Less than CR makes no sense?

Yes and No.

Implementations should provide rigorous bounds. Will enable to prove algorithms.

`man libm` under SunOS 5.10:

Double precision real functions (SPARC)

function	error bound (ulps)	largest error observed (ulps)
acosh	4.0	1.878
asinh	7.0	1.653
atan2	2.5	1.456
atanh	4.0	1.960

Much more expensive?

YES: worst cases for n-bit format require $\approx 2n$-bit working precision (Table Maker's Dilemma)

On average NO: cf Ziv, Muller, Defour, De Dinechin, Lauter, Ershov, Gast (ARITH'17 talk *"Towards the Post-ultimate libm"*)

(First implementation was ml4j by Ziv et al., in 1999.)

CR libm

- 2 steps are enough (1st to 2^{-63}, 2nd to WC accuracy)
- 4KB per function are enough
- 2nd step with double-double-extended, or triple-double

Crlibm: 1st step in DE, 2nd step in double-DE.

arctan-P4	avg time	max time
crlibm	350	1680
default libm	339	388

Slowdown: avg 1.03, max 4.3.

exp Itanium-2	avg time	max time
crlibm (two steps)	67	114
crlibm (2nd step only)	92	92
default libm	63	63

Slowdown: avg 1.06, max 1.81 (1.46 for 2nd step only).

Markstein: log on the Itanium in 40 cycles, plus 12 cycles to check the result.

The Rounding Test (to nearest)

Approximation: $h + l$ with relative error $< \delta$.

We want $h = \circ((h + l)(1 + \delta))$.

$$\overbrace{\boxed{1xx \ldots xxx}}^{h}\underbrace{000 \ldots 000}_{q}\overbrace{\boxed{\pm 1xx \ldots xxx}}^{l}$$

(a) only problem when $q \le 1$ and $l = \pm 111 \ldots xxx$

(b) then $|h| \le (2^{55} - 4)|l|$, thus $|h + l|\delta \le 2^{55}|l|\delta$

(c) $(h + l)(1 + \delta) \le h + \epsilon l$ with $\epsilon = 1 + 2^{55}\delta$

The Rounding Test (2)

Let $\epsilon = 1 + 2^{55}\delta$ (rounded away).

```
{ 1st step }
if (h == (h + l * epsilon))
   return h;
else
   { 2nd step }
```

Already in MathLib source code. First proof: Defour's PhD thesis.

Algorithm Design

$$T_{\text{avg}} = T_1 + T_{\text{rt}} + p_2 T_2$$

(1) design a fast 2nd step for the target function

(2) choose p_2 so that $p_2 T_2 \ll T_1$

(3) 1st step accuracy is $\epsilon = 2^{-53} \cdot p_2$

Typically: $T_{\text{rt}} \approx 10$ cycles, $p_2 \approx 1/1000$, $\epsilon = 2^{-63}$

- the average penalty can be made negligible
- incompressible cost of the rounding test (one FMA)

Difficulty of proving CR, undecidable?

NO! Ziv's strategy loops only when $y = f(x)$ is **exact.**

Most functions: **finite number** of such $x, y \in \mathbb{Q}$.

Find the **worst cases** (WC) by exhaustive search.

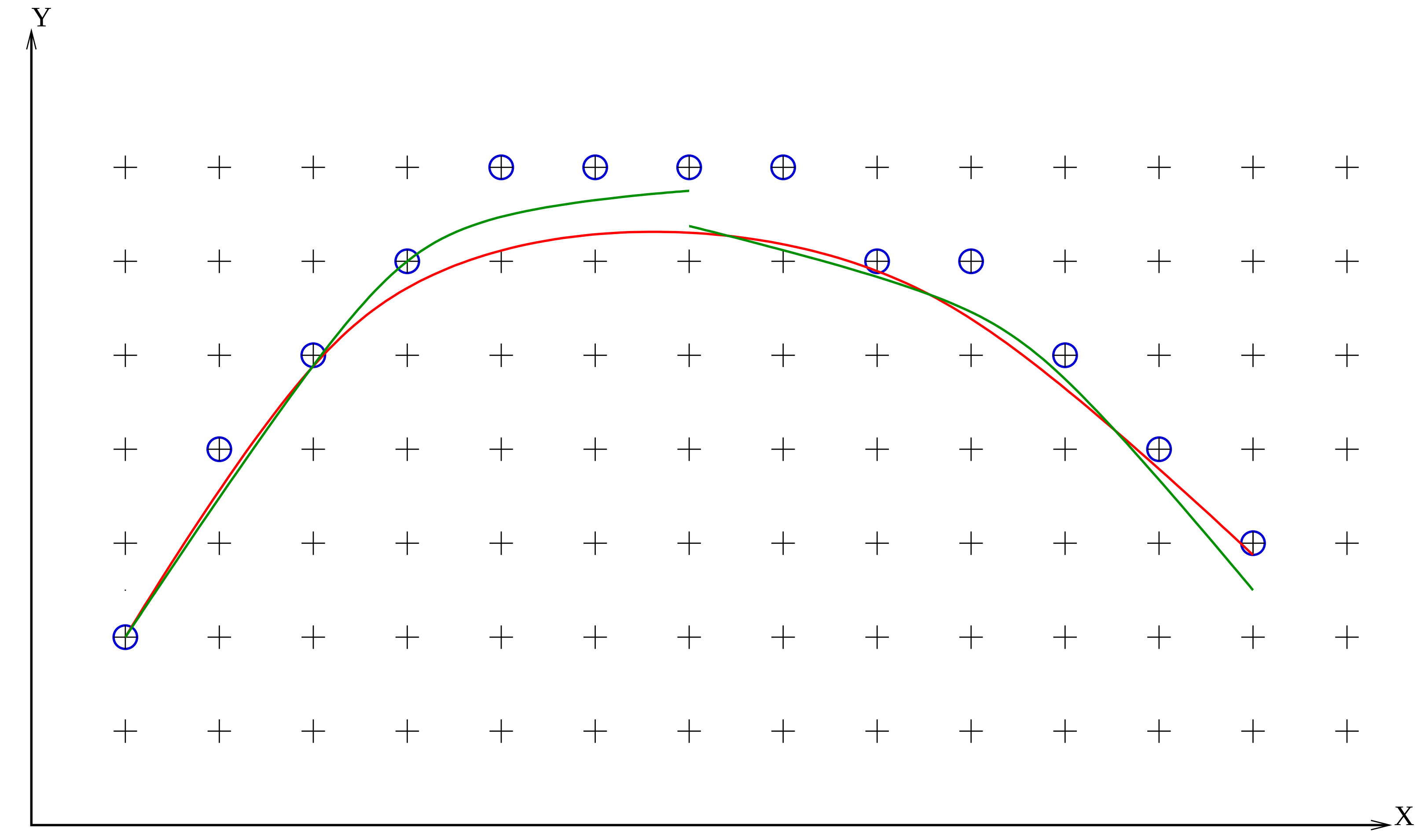
WC known for most functions in double precision (Lefèvre, Muller).

New algorithms make it possible for double-extended (Stehlé, Lefèvre, Z.):

$$t = 3990454322510295554$$

$$2^{63} \cdot 2^{\frac{1}{2} + \frac{t}{2^{64}}} = 15153900280214575669.000000000000000000036\ldots$$

Use polynomial approximations!



Simultaneous worst-cases: double precision,

$t_0 = 8736447851783651$

$$2^{53} \sin \frac{t_0}{2^{53}} = 7429607293621962.00000000327\ldots$$

$$2^{53} \cos \frac{t_0}{2^{53}} = 5092207171469561.00000000374\ldots$$

Two-variable functions:

$$2^{10} \cdot \left(\frac{560}{2^{10}}\right)^{\frac{947}{2^{10}}} = 585.999999253\ldots$$

Difficulty of testing

Yes: we can't check every input, but the same was true for the basic arithmetic functions (cf the Pentium bug). We may have the COSH bug...

No: we can easily construct reference tables (cf FPgen from IBM). Compare wrt arbitrary precision.

Algorithms for worst-cases can also be used to produce corner cases. Cf "invisible bits" work at IBM (Aharoni, Asaf, Maharik, Nehama, Nikulshin, Ziv, ARITH'17).

A formal proof is also possible (cf Daumas' talk in 2004), if we assume the worst-cases are correct.

Is it worth the cost? Who would benefit?

Maybe in 15 years we could say:

It was remarkable that so many hardware people there, knowing how difficult ***correctly-rounded transcendentals*** *would be, agreed that it should benefit the community at large. If it encouraged the production of floating-point software and eased the development of reliable software, it would help create a larger market [. . .]*

The original citation

"It was remarkable that so many hardware people there, knowing how difficult ***p754*** *would be, agreed that it should benefit the community at large. If it encouraged the production of floating-point software and eased the development of reliable software, it would help create a larger market for everyone's hardware. This degree of altruism was so astonishing that MATLAB's creator Dr. Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754."*

William Kahan, An Interview with the Old Man of Floating-Point, February 1998.

CR libraries (double precision)

• MathLib, Ziv (IBM). Implements sin, cos, tan, asin, acos, atan, atan2, exp, log, pow. Rounding to nearest only. No proof of correctness. Not supported anymore.

• CRLIBM (Dupont De Dinechin, INRIA). One function for each rounding mode (exp_rn, exp_rd, exp_ru, exp_rd). Based on Muller and Lefèvre worst-case bounds. Assumes machine rounding to nearest, to double precision. Implements exp, log, log2, log10, sin, cos, tan, atan, sinh, cosh, as of version 0.10beta (Sep 2005).

• LIBMCR (Sun). Uses the internal rounding state. Implements atan, cos, exp, log, pow, sin, tan, as of version 0.9 (Feb 2004).

```
With libmcr 0.9:


TONEAREST  pow(1.0000000000000004,0.5) = 1.0000000000000002
TOWARDZERO pow(1.0000000000000004,0.5) = 1
DOWNWARD   pow(1.0000000000000004,0.5) = 1
UPWARD     pow(1.0000000000000004,0.5) = 1.0000000000000004

TONEAREST  pow(1.0000000000000002,-0.5) = 0.99999999999999989
TOWARDZERO pow(1.0000000000000002,-0.5) = 0.99999999999999989
DOWNWARD   pow(1.0000000000000002,-0.5) = 0.99999999999999978
UPWARD     pow(1.0000000000000002,-0.5) = 1
```

• MPFR with precision 53 bits. Implements all C99 functions, as of version 2.2.0 (Sep 2005).

References

Minutes from the committee since 2001.

“French Proposal” (Defour, Hanrot, Lefèvre, Muller, Revol, Z.), presented by Peter Markstein on May 23, 2002, Numerical Algorithms, 2004.

Section T (Elementary Transcendental Functions) of the October 20, 2005 draft.

Open Problems

• Compute WC for decimal formats: not a big deal!

• trig(BIG) is still expensive: sin(MAXDBL) requires an argument reduction on $1024 + 128 = 1152$ bits!

$\Longrightarrow$ error bound $(\frac{1}{2} + \epsilon)$ ulp

$\Longrightarrow$ call arbitrary precision library?

• Two-variable functions like $x^y, \mathrm{atan}(y/x)$ are still out-of-reach for exhaustive worst-case search: 2^{128} cases!

$\Longrightarrow$ 1st step is still CR

$\Longrightarrow$ add rounding test in 2nd step (raise exception?)

$\Longrightarrow$ or call arbitrary precision library…

Nul n'est prophète en son pays

(or don't forget base conversion!)

"État de frais 9552" (PhD S. Boldo)

Type	Qté	Mnt. Unitaire	Tot.
Repas du soir	1,00	15,25	15,25
Frais de taxi	1,00	18,90	18,89
Bus, métro, RER	1,00	1,40	1,39

Summary

- want implementation-independent error bound
- correct direction for rounding towards $0, \pm\infty$
- correct rounding is our ultimate goal

Why Arbitrary Precision?

Still the chaos ...

```
    |\^/|     Maple 9.5 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, ...
 \  MAPLE  /  All rights reserved. ...
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(480*Pi-1730*log(2)-367*sin(1));
                               -6
                         0.6 10


     GP/PARI CALCULATOR Version 2.2.10 (development)
i686 running linux (ix86/GMP-4.1.4) 32-bit version

? \p10
   realprecision = 19 sign. digits (10 dig. displayed)
? 480*Pi-1730*log(2)-367*sin(1)
%1 = -0.0000000701026491
```

Mathematica 5.0:

```
In[1]:= N[480*Pi-1730*Log[2]-367*Sin[1],10]

                         -8
Out[1]= -7.010264877 10

In[2]:= N[480*Pi-1730*Log[2]-367*Sin[1],29]

                                  -8
Out[2]= -7.010264877121109819 10

In[3]:= N[480*Pi-1730*Log[2]-367*Sin[1],30]

                                             -8
Out[3]= -7.01026487712110981868038112175 10

In[4]:= N[480*Pi-1730*Log[2]-367*Sin[1],29]

                                            -8
Out[4]= -7.0102648771211098186803811217 10
```

Magma V2.11-11

```
> R := RealField(10);
> 480*Pi(R)-1730*Log(R ! 2)-367*Sin(R ! 1);
-0.00000179

> R := RealField(20);
> 480*Pi(R)-1730*Log(R ! 2)-367*Sin(R ! 1);
-0.0000000701026672
```

Why Arbitrary Precision?

Section 3.5 (Varying-Width Floating-Point Formats) of the October 20, 2005 draft (was “Annex Z”).

Computer arithmetic:

- huge argument reduction like trig(BIG)
- 3rd step when WC are not known
- test/simulate hardware and/or fixed-precision libraries

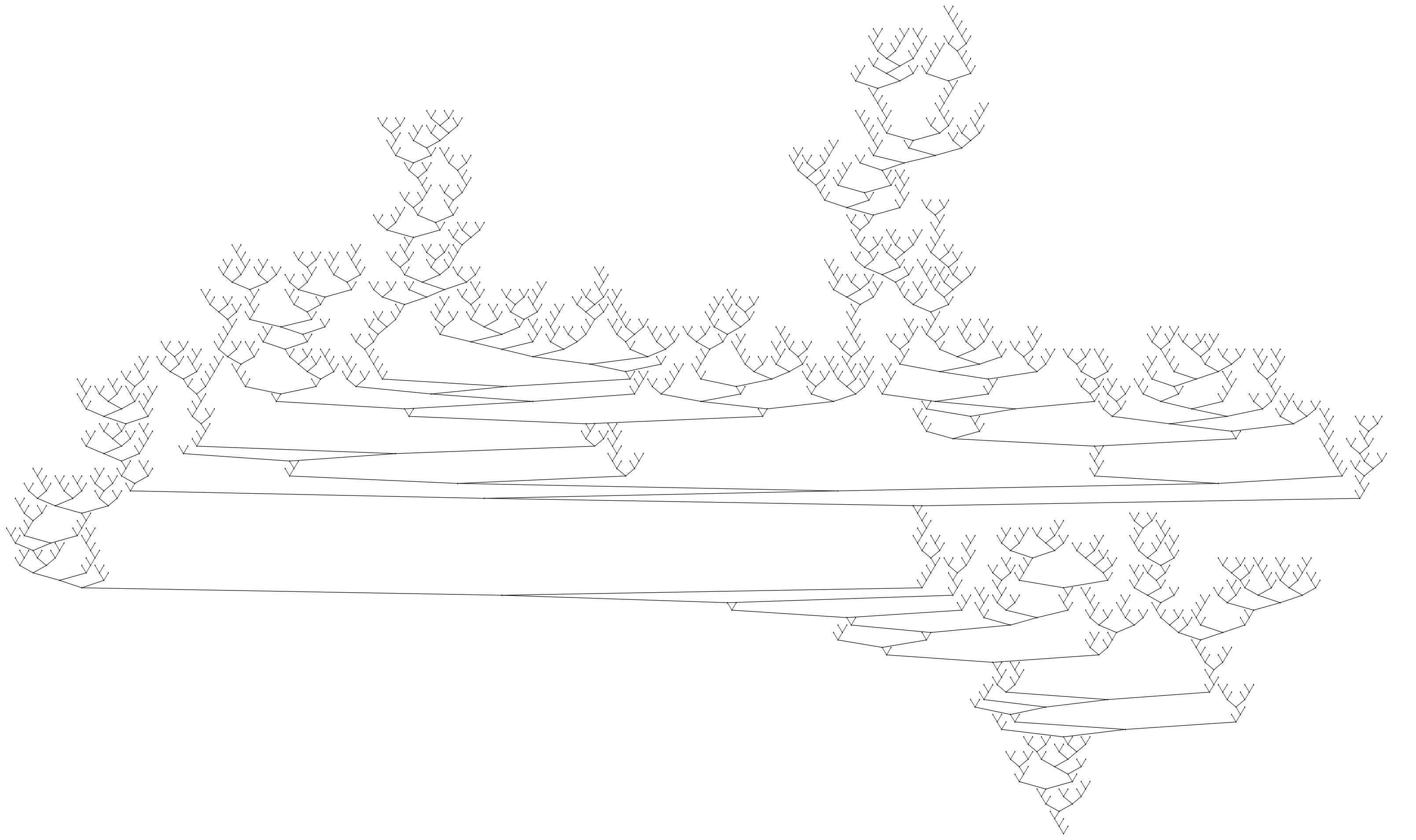
Numerical analysis:

- study sensibility of algorithms to roundoff errors
- implement arbitrary precision interval arithmetic

Computer Algebra and Algorithmic Geometry:

- prove symbolic inequalities (when decidable)
- Real RAM implementations
- “exact geometric computation” and “geometric rounding”

Random Generation of Combinatorial Structures



A random binary tree of size 1000

Binary trees with n leaves:

$$b_0 = 0, \quad b_1 = 1, \quad b_n = \sum_{k=0}^{n} b_k b_{n-k}$$

```
> b:=proc(n) option remember; if n<=1 then n
  else add(b(k)*b(n-k),k=1..n-1) fi end:
> seq(b(n), n=0..10);
        0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862
```

Catalan's numbers: $b_n = \frac{1}{n}\binom{2n-2}{n-1}$

$$\pi_{n,k} = \frac{b_k b_{n-k}}{b_n}$$

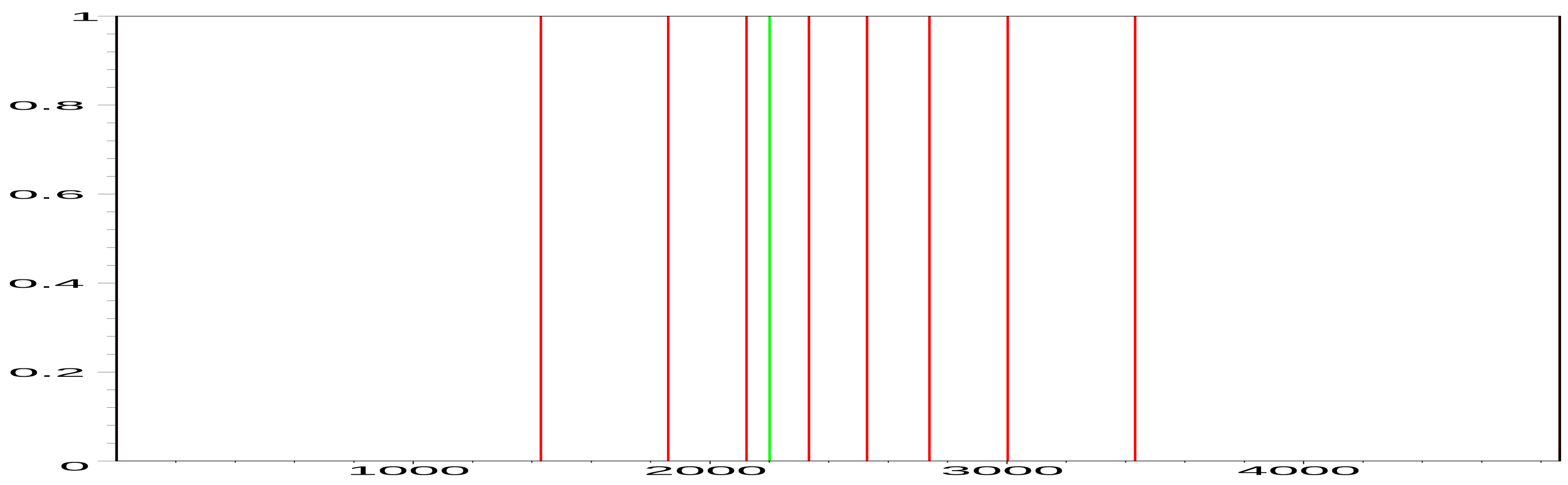
```
> n:=10: l:=[seq(b(k)*b(n-k),k=0..n)];
   [0, 1430, 429, 264, 210, 196, 210, 264, 429, 1430, 0]

> seq(add(l[i],i=1..k),k=1..n);


   0, 1430, 1859, 2123, 2333, 2529, 2739, 3003, 3432, 4862


> rand(b(n))();
```

2201



Using exact arithmetic:

- arithmetic complexity $O(n \log n)$
- numbers have $O(n)$ digits
- binary complexity $O(n^3 \log n)$

Using floating-point arithmetic:

- on average, $O(\log n)$ bits are enough
- binary complexity $O(n \log^3 n)$

Uniform Random Generation of Decomposable Structures Using Floating-Point Arithmetic, A. Denise, P. Z., TCS, 1999.

Problems

- double-precision is not enough for $n \approx 10^6$
- exponent range is too small: $b_{521} > 2^{1024}$

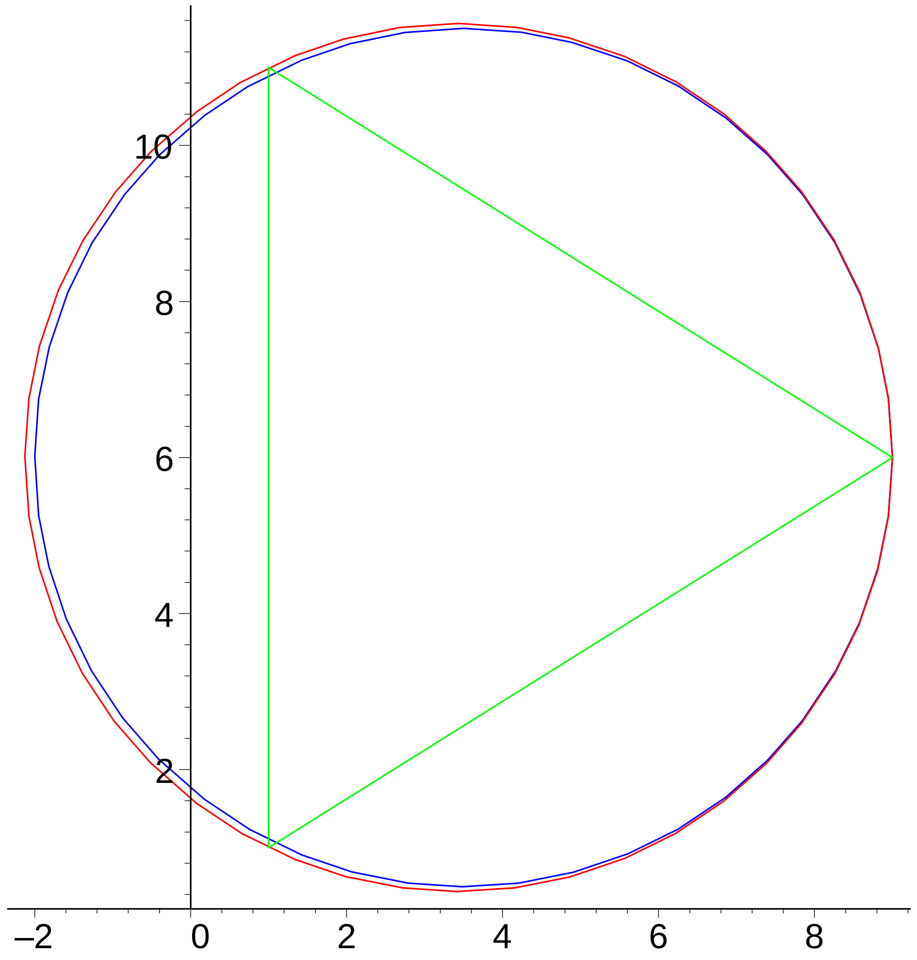
```
> b(521);
5593411376326319221069187523870184667647986304917005\
8679986671969984842889002358432008962209065120052816\
7004731897590718558233136622073615913210052132884741\
9038436415745407516057959285902579498653909532459374\
6275317931641965896668626251538543952636079551428205\
2629747942942913739553258875568360476666078672900
```

- double-extended: $b_{8204} > 2^{16384}$

Geometric Rounding

- simplify geometric objects
- while keeping geometric properties
- to be able to use floating-point coordinates
- to control the complexity

Smallest circle enclosing $A = (9, 6)$, $B = (1, 11)$, $C = (1, 1)$ with 4-bit precision:



Which Arbitrary Precision?

- consider atomic operations only (no Real RAM)
- precision can be linked to variables or operations
- variables: global precision, or different for each var:

```
SetPrecision(a, prec);
Add (a, b, c, rnd);
```

- operations: working precision for the operation. Inputs are first rounded to that precision:

```
a = Add (b, c, prec, rnd);
```

- dense (digit array) vs sparse (f-p expansions) storage. For f-p expansions, “precision” is value-dependent.

Wanted

- simulate fixed formats (single, double, quad), including subnormals.
- available precisions and exponent ranges independent from hardware, system, compiler
- the internal representation may vary from one implementation to the other, what is important is to be able to exchange values exactly (Data Interchange Format proposal from Jeff Kidder)

Which base?

Base $\beta = 2$ (binary) would be useful to check binary16, binary32, binary64, binary 128.

Base $\beta = 10$ (decimal) would be useful to check decimal32, decimal64, decimal128.

I propose to add bases $\beta = 17$ or $\beta = 42$ in the revision :-)

Which precisions?

Any precision $n \geq 1$, only limit is memory.

Precision $n = 1$:

$$x = \pm 1. \cdot 2^e$$

No mantissa bit to store!

Round-even rule: $x = 1.5$ rounds to 1 or 2?

Which rounding modes?

- the four classical ones: to $\pm\infty$, to 0, to nearest
- away from zero (symmetry)
- faithful rounding (internal)

Existing implementations (CR)

Arithmos (Antwerp) - base-independent

Maple (WMI) and DecNumber (Mike Cowlishaw, IBM) - decimal

NTL RR class (Shoup) and MPFR - binary

Maple 9.5

```
> Digits:=3: Rounding:=0: 1.0 - 9e-5;
                                        1.0



> Digits:=2: Rounding:=infinity: csc(0.01);
                                        100.
```

(exact result is 100.001666687…)

```
> op(`evalf/csc`);
proc(r) local x; option `Copyright ...`;
    x := evalf(r);
    if x = 0 then 1./x
    elif type(x, 'complex(float)') then
        evalf(evalf[Digits + 2](1/'sin'(x)))
    else 'csc'(x)
    end if
end proc
```

The MPFR library

- binary arbitrary precision ($p \geq 2$)
- special values: ± 0, $\pm\infty$, NaN
- four 754 rounding modes
- correct rounding for all functions (basic arithmetic, conversions, transcendentals)
- allow mixed-precision operations (no widening)
- written in C, based on GNU MP (GMP)
- distributed under LGPL from `www.mpfr.org`
- project started in 1999, latest release is 2.2.0

The "ternary" flag

```
int t = mpfr_add (y, b, c, GMP_RNDN);
```

Idea: let $x = b + c$ be rounded to y. The ternary flag t is:

- $t < 0$ if $y < x$
- $t = 0$ if $y = x$
- $t > 0$ if $y > x$

Equivalent to the "inexact flag" for directed rounding, but gives more information for rounding to nearest.

How to emulate subnormals?

Easy for directed rounding: first round x to y with wide exponent range, then round y to next subnormal z

Rounding to nearest: y and the ternary flag t are enough!

(special case of double-rounding)

Software built on top of MPFR

MPFI - binary arbitrary precision floating-point intervals

MPC - binary arbitrary precision complex floating-point

MPCHECK - check mathematical libraries (CR, symmetry, monotonicity)

A small calculator:

```
$ ./calc -prec=18 -rnd=nearest ``exp(Pi*sqrt(163))''
2.62537412640768744e17
```

Software using MPFR

GFORTRAN - the GNU Fortran 95 compiler, part of GCC

iRRAM - a RealRAM implementation by Norbert Müller

Magma - a computational algebra system (Cannon, Sydney)

SAGE - Software for Algebra and Geometry Experimentation (William Stein, UCSD)

Summary

- CR arbitrary precision is most wanted
- at least for $+, -, \times, \div, \sqrt{\cdot}$
- transcendentals are welcome too!