Can we trust floating-point numbers?

Paul Zimmermann, INRIA LORRAINE Loria

Who is NOT using floating-point numbers?

Who is NOT using floating-point numbers?

linear algebra (BLAS library)

Who is NOT using floating-point numbers?

linear algebra (BLAS library)

Microsoft Excel

Who is NOT using floating-point numbers?

linear algebra (BLAS library)

Microsoft Excel

bank accounting, interest rates

Who is NOT using floating-point numbers?

linear algebra (BLAS library)

Microsoft Excel

bank accounting, interest rates

plotting graphs

Who is NOT using floating-point numbers?

linear algebra (BLAS library)

Microsoft Excel

bank accounting, interest rates

plotting graphs

google (page rank)

Who is NOT using floating-point numbers?

linear algebra (BLAS library)

Microsoft Excel

bank accounting, interest rates

plotting graphs

google (page rank)

travel costs . . .

“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

The Pentium bug (1994)

Revealed by Thomas Nicely (Univ. Virginia), a mathematician.

Twin primes: p and $p+2$ are prime, e.g. 5 and 7, 11 and 13, $\ldots$

Theorem (Brun, 1919) The sum of the inverses of twin primes is finite:

$$B_2 = \left(\frac{1}{3} + \frac{1}{5}\right) + \left(\frac{1}{5} + \frac{1}{7}\right) + \left(\frac{1}{11} + \frac{1}{13}\right) + \cdots$$

The Pentium bug (cont'd)

Reciprocal sums are computed with two methods:

- to 19 significant digits using the FPU
- to 53 decimal places using arrays of long integers

Nicely used half a dozen 486's, and added a **Pentium-60** in March 1994.

October 4: reciprocal sum on the Pentium differed from the 486:

$$\frac{1}{824633702441} + \frac{1}{824633702443}$$

Tim Coe found the worst case:

$$\frac{4195835.0}{3145727.0}$$ gives 1.333**73906802** instead of 1.333**8204491**

There was a bug because …

there was a **specification** (IEEE 754)

There was a bug because ...

there was a **specification** (IEEE 754)

no specification $\Longrightarrow$ **no bug!**

The IEEE 754 standard

Approved by IEEE and ANSI in 1985.

The IEEE 754 standard

Approved by IEEE and ANSI in 1985.

Defines four **binary** formats: single (24 significand bits), single-extended (deprecated), double (53 significand bits), double-extended (≥ 64 significand bits).

The IEEE 754 standard

Approved by IEEE and ANSI in 1985.

Defines four **binary** formats: single (24 significand bits), single-extended (deprecated), double (53 significand bits), double-extended (≥ 64 significand bits).

Requires **correct rounding** for $+, -, \times, \mathrm{div}, \sqrt{\cdot}$.

The IEEE 754 standard

Approved by IEEE and ANSI in 1985.

Defines four **binary** formats: single (24 significand bits), single-extended (deprecated), double (53 significand bits), double-extended (≥ 64 significand bits).

Requires **correct rounding** for $+, -, \times, \mathrm{div}, \sqrt{\cdot}$.

Four rounding modes: toward zero, $+\infty$, $-\infty$, nearest.

The IEEE 754 standard

Approved by IEEE and ANSI in 1985.

Defines four **binary** formats: single (24 significand bits), single-extended (deprecated), double (53 significand bits), double-extended (≥ 64 significand bits).

Requires **correct rounding** for $+, -, \times, \mathrm{div}, \sqrt{\cdot}$.

Four rounding modes: toward zero, $+\infty$, $-\infty$, nearest.

Special values: NaN, $\pm\infty$, ± 0.

The IEEE 754 standard

Approved by IEEE and ANSI in 1985.

Defines four **binary** formats: single (24 significand bits), single-extended (deprecated), double (53 significand bits), double-extended (≥ 64 significand bits).

Requires **correct rounding** for $+, -, \times, \mathrm{div}, \sqrt{\cdot}$.

Four rounding modes: toward zero, $+\infty$, $-\infty$, nearest.

Special values: NaN, $\pm\infty$, ± 0.

Exceptions: invalid operation, division by zero, overflow, underflow, inexact.

The IEEE double precision format

64-bit encoding

1-bit sign, 53-bit mantissa (implicit leading bit), 11-bit exponent

$$x = (-1)^s \cdot 1.b_1 b_2 \ldots b_{52} \cdot 2^e$$

$-1022 \leqslant e \leqslant 1023$

Largest value is $1.11\ldots 11 \cdot 2^{1023} \approx 1.79 \cdot 10^{308}$

Smallest (normal) value is $2.22 \cdot 10^{-308}$

Correct Rounding

Let $\mathbb{R}$ be the set of real numbers, $\mathbb{F} \in \mathbb{R}$ the set of floating-point numbers.

Correct Rounding

Let $\mathbb{R}$ be the set of real numbers, $\mathbb{F} \in \mathbb{R}$ the set of floating-point numbers.

Let $f : \mathbb{R} \longrightarrow \mathbb{R}$ a mathematical function, $g : \mathbb{F} \longrightarrow \mathbb{F}$ its floating-point implementation for a given rounding mode.

Correct Rounding

Let $\mathbb{R}$ be the set of real numbers, $\mathbb{F} \in \mathbb{R}$ the set of floating-point numbers.

Let $f : \mathbb{R} \longrightarrow \mathbb{R}$ a mathematical function, $g : \mathbb{F} \longrightarrow \mathbb{F}$ its floating-point implementation for a given rounding mode.

Definition: g is *correctly rounded* if for all $x \in \mathbb{F}$, $g(x)$ is the number in $\mathbb{F}$ closest to $f(x)$ with respect to the given rounding mode.

Correct Rounding

Let $\mathbb{R}$ be the set of real numbers, $\mathbb{F} \in \mathbb{R}$ the set of floating-point numbers.

Let $f : \mathbb{R} \longrightarrow \mathbb{R}$ a mathematical function, $g : \mathbb{F} \longrightarrow \mathbb{F}$ its floating-point implementation for a given rounding mode.

Definition: g is *correctly rounded* if for all $x \in \mathbb{F}$, $g(x)$ is the number in $\mathbb{F}$ closest to $f(x)$ with respect to the given rounding mode.

Example: $1.0/3.0 \rightarrow 0.333$ for rounding toward zero, 0.334 for rounding towards $+\infty$.

The good news

- 1998: Intel hired John Harrison as a Senior Software Engineer specializing in the design and **formal verification** of mathematical algorithms.

Floating point verification in HOL Light: the exponential function, J. Harrison, Technical Report, Univ. Cambridge, 1997:

> *[...] error in the result is less than 0.54 units in the last place [...]*

The good news (cont'd)

Formal verification of IA-64 division algorithms, J. Harrison, Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2000:

- IA-64 floating-point and integer division done in software
- all available algorithms (subroutines, inline) checked with HOL Light
- better understanding of the underlying theory
- some significant efficiency improvements

The good news (cont’d)

Formal verification of IA-64 division algorithms, J. Harrison, Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2000:

- IA-64 floating-point and integer division done in software
- all available algorithms (subroutines, inline) checked with HOL Light
- better understanding of the underlying theory
- some significant efficiency improvements

AMD hired David Russinoff (proof of multiplication, division, square root on K5 and K7)

What you see is not what you get

When I write `double x=0.3` in C, why does it print as `0.2999999999`?

Fact 1: the IEEE formats are **binary** formats.

Fact 2: 0.3 is **not exactly representable** as $m \cdot 2^e$

Fact 3: the closest double-precision number is

$$5404319552844595 \cdot 2^{-54} \approx 0.299999999999999988897\ldots$$

Fact 4: when printed towards zero or with $\geqslant 17$ digits, one gets $0.2999999999\ldots$.

The current situation

Good confidence in IEEE 754 conformance

of processors/compilers/operating systems

The current situation

Good confidence in IEEE 754 conformance

of processors/compilers/operating systems

No need any more to write:

```
x = (x + x) - x;
```

The current situation

Good confidence in IEEE 754 conformance

of processors/compilers/operating systems

No need any more to write:

```
x = (x + x) - x;
```

or the following works as expected:

```
if (x != y)
   z = 1.0 / (x - y);
```

The bad news

IEEE 754 says nothing about:

- elementary functions: $\exp, \log, \sin, \cos, \ldots$
- arbitrary precision
- sequences of operations

Challenge 1. *Compute the sign of* $\sin(10^{22})$.

$\sin 10^{22}$ with GCC 4.0.2

```
#include <stdio.h>
#include <math.h>

int
main()
{
  double x = 1e22;
  printf ("sin(1e22)=%1.16e\n", sin (x));
}
```

$\sin 10^{22}$ with GCC 4.0.2

```
#include <stdio.h>
#include <math.h>

int
main()
{
  double x = 1e22;
  printf ("sin(1e22)=%1.16e\n", sin (x));
}

bash-3.00$ ./a.out
sin(1e22)=4.6261304076460175e-01
```

$\sin 10^{22}$ with CAS

(GCC gives `4.6261304076460175e-01`)

Maple 6:

```
> sin(1E22);
                                        -.8522008498
```

$\sin 10^{22}$ with CAS

(GCC gives `4.6261304076460175e-01`)

Maple 6:

```
> sin(1E22);
                                   -.8522008498
```

Mathematica 5.0:

```
In[6]:= N[Sin[10^22]]

Out[6]= 0.462613
```

$\sin 10^{22}$ with CAS

(GCC gives `4.6261304076460175e-01`)

Maple 6:

```
> sin(1E22);
                                        -.8522008498
```

Mathematica 5.0:

```
In[6]:= N[Sin[10^22]]

Out[6]= 0.462613
```

PARI/GP 2.3.0:

```
? sin(1e22)
%1 = -0.852200749
```

$\sin 10^{22}$ with CAS

(GCC gives `4.6261304076460175e-01`)

Maple 6:

```
> sin(1E22);
                                    -.8522008498
```

Mathematica 5.0:

```
In[6]:= N[Sin[10^22]]

Out[6]= 0.462613
```

PARI/GP 2.3.0:

```
? sin(1e22)
%1 = -0.852200749
```

MuPAD 3.2.0:

```
>> sin(1e22);

                                  -0.9873536182
```

A much harder problem

Challenge 2. *Compute 10 digits of*

$$\sin(6303769153620408 \cdot 2^{971})$$

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.7193976429664643e-02`

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.7193976429664643e-02`

Maple 10 (10 digits) gives `-0.8021127471`

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.7193976429664643e-02`

Maple 10 (10 digits) gives `-0.8021127471`

Maple 10 (20 digits) gives `-0.9482478427...`

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.71939764296643e-02`

Maple 10 (10 digits) gives `-0.8021127471`

Maple 10 (20 digits) gives `-0.9482478427...`

Maple 10 (50 digits) gives `0.3915937923...`

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.7193976429664643e-02`

Maple 10 (10 digits) gives `-0.8021127471`

Maple 10 (20 digits) gives `-0.9482478427...`

Maple 10 (50 digits) gives `0.3915937923...`

Maple 10 (200 digits) gives `-0.3887412074...`

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.7193976429664643e-02`

Maple 10 (10 digits) gives `-0.8021127471`

Maple 10 (20 digits) gives `-0.9482478427...`

Maple 10 (50 digits) gives `0.3915937923...`

Maple 10 (200 digits) gives `-0.3887412074...`

T. Hoare: *How do we know the answers are correct?*

$\sin(6303769153620408 \cdot 2^{971})$

GCC 4.0.2 gives `-4.7193976429664643e-02`

Maple 10 (10 digits) gives `-0.8021127471`

Maple 10 (20 digits) gives `-0.9482478427...`

Maple 10 (50 digits) gives `0.3915937923...`

Maple 10 (200 digits) gives `-0.3887412074...`

T. Hoare: *How do we know the answers are correct?*

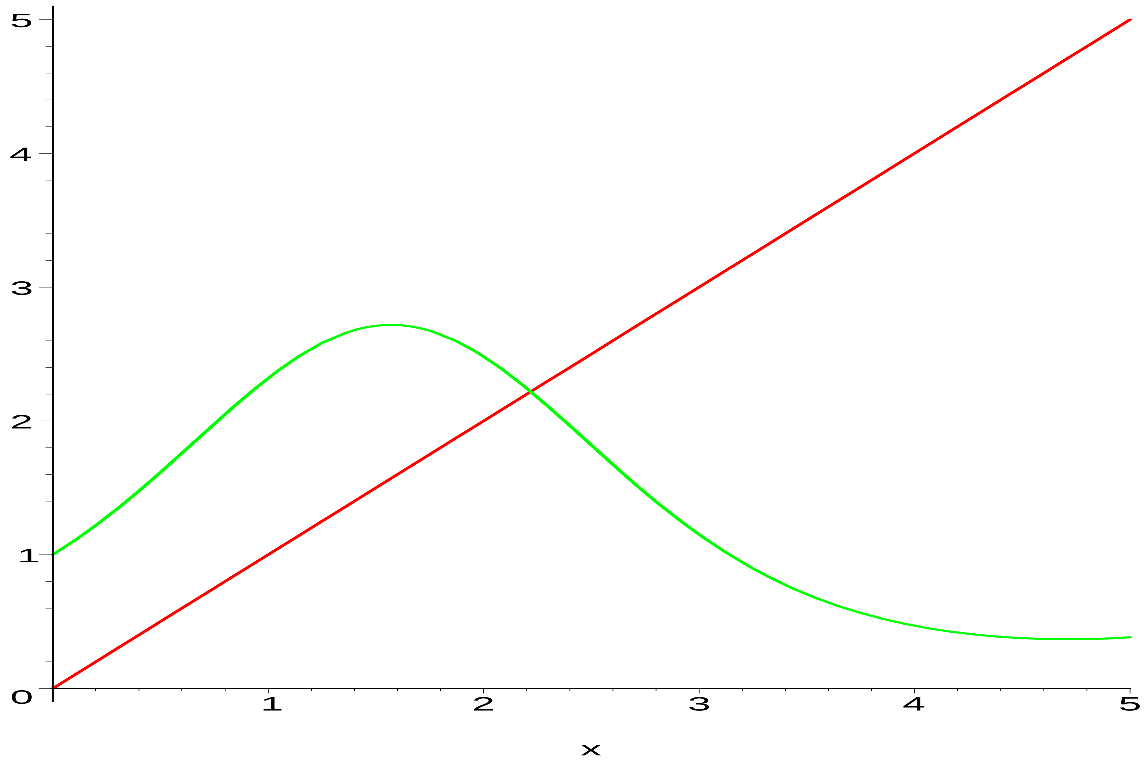
no specification $\Longrightarrow$ **no bug!**

Challenge 3. *Obtain* 10 *digits of the solution of*

$$e^{\sin x} = x.$$

Problem P21 from the “Many Digits Competition” (Nijmegen, 2005).

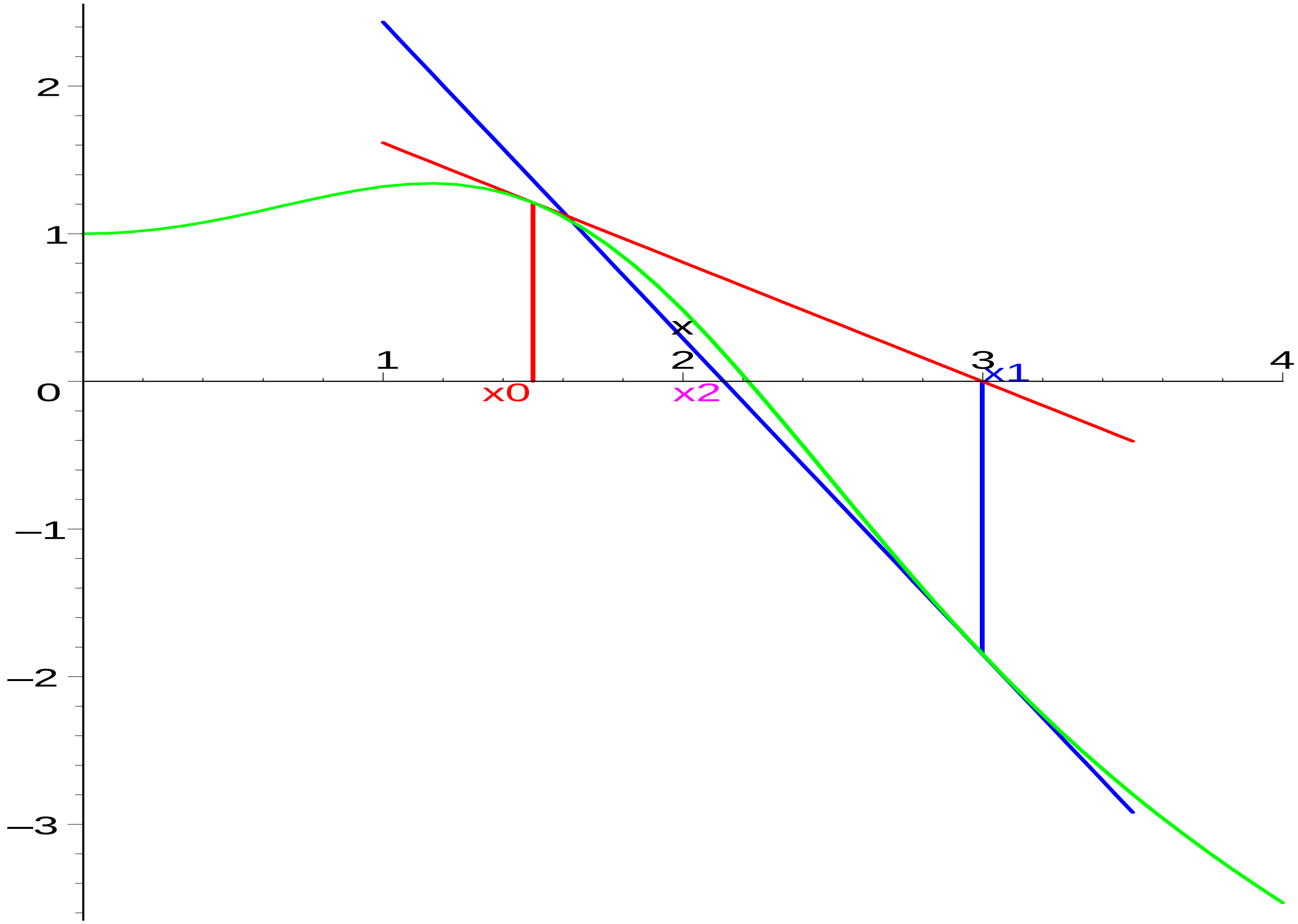
Newton's Method



Unique solution

$$\rho \approx 2.219$$

Newton's Method on $e^{\sin x} - x$



Newton's Method

infinite precision		finite precision
x_i [correct bits]	p	x_i [correct bits]
$x_0 = 1.5$ [2.4]	2	$x_0 = 1.5$ [2.4]
$x_1 = 2.998991444$ [2.3]	4	$x_1 = 3.25$ [1.9]
$x_2 = 2.136652643$ [5.6]	8	$x_2 = 1.9921875$ [4.1]
$x_3 = 2.220897155$ [11.1]	16	$x_3 = 2.239136$ [7.6]
$x_4 = 2.219107802$ [22.5]	32	$x_4 = 2.21918417$ [15.6]

Newton's Method

infinite precision		finite precision
x_i [correct bits]	p	x_i [correct bits]
$x_0 = 1.5$ [2.4]	2	$x_0 = 1.5$ [2.4]
$x_1 = 2.998991444$ [2.3]	4	$x_1 = 3.25$ [1.9]
$x_2 = 2.136652643$ [5.6]	8	$x_2 = 1.9921875$ [4.1]
$x_3 = 2.220897155$ [11.1]	16	$x_3 = 2.239136$ [7.6]
$x_4 = 2.219107802$ [22.5]	32	$x_4 = 2.21918417$ [15.6]

Can we know how many digits are correct?

Challenge 4. *Prove that* $\exp \pi > 23$.

The quick-and-dirty way

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(exp(Pi));
                                        23.14069264
```

The slow-and-correct way

Lemma1. $\pi > \frac{201}{64}$.

Proof: $\tan(2x) = \frac{2\tan x}{1-\tan^2 x}$ gives for $x = \pi/8$:

$$2\tan(\pi/8) = 1 - \tan^2(\pi/8)$$

i.e. $\pi = 8\arctan(\sqrt{2} - 1)$.

$$\arctan x > x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7.$$

Since $a := \frac{3393}{8192} \leqslant \sqrt{2} - 1 \leqslant b := \frac{3394}{8192}$,

$$\pi > 8(a - b^3/3 + a^5/5 - b^7/7) = \frac{797404939566065002745904209}{253874422119072126688296960} > \frac{201}{64}$$

The slow-and-correct way (cont'd)

$$\exp x > 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320}$$

$$\exp \pi > \exp \frac{201}{64} > \frac{2900419254646472870777}{126100789566373888 0} > 23$$

The slow-and-correct way (cont'd)

$$\exp x > 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320}$$

$$\exp \pi > \exp \frac{201}{64} > \frac{29004192546472870777}{1261007895663738880} > 23$$

Why proving a simple formula is so tedious?

Do you (still) trust floating-point numbers?

The Grand Challenges

Grand Challenge 1: design requirements for mathematical functions and arbitrary precision

The Grand Challenges

Grand Challenge 1: design requirements for mathematical functions and arbitrary precision

Grand Challenge 2: implement those requirements in software

The Grand Challenges

Grand Challenge 1: design requirements for mathematical functions and arbitrary precision

Grand Challenge 2: implement those requirements in software

Grand Challenge 3: prove those software are correct

The Grand Challenges

Grand Challenge 1: design requirements for mathematical functions and arbitrary precision

Grand Challenge 2: implement those requirements in software

Grand Challenge 3: prove those software are correct

no specification $\Longrightarrow$ **no bug!**

Partial Answers

Grand Challenge 1: 754R (Annex D)

Grand Challenge 2: MathLib (IBM), Libmcr (Sun), CRLIBM (ENS Lyon), IRRAM (Müller), RealLib (Lambov), MPFR/MPFI, . . .

Grand Challenge 3: CRLIBM (partly)

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.