Computing with Floating Point
It's not Dark Magic, it's Science

Florent de Dinechin, Arénaire Project, ENS-Lyon
Florent.de.Dinechin@ens-lyon.fr

CERN seminar, January 11, 2004.99999

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
ECOLE NORMALE SUPERIEURE DE LYON
INRIA

First some advertising

This seminar will only survey the topic of floating-point computing.
To probe further:

- *What Every Computer Scientist Should Know About Floating-Point Arithmetic* par Goldberg (Google will find you several copies)
- The web page of William Kahan at Berkeley.
- The web page of the Arénaire group.

Introduction: Floating point ?

Also known as "scientific notation"

A real number $\widehat{x}$ is approximated in machine by a rational:

$$x = (-1)^s \times m \times \beta^e$$

where

- β is the radix
  - 10 in your calculator and (usually) your head
  - 2 in most computers
  - Some IBM financial mainframes use radix 10, why ?
- $s \in \{0, 1\}$ is a sign bit
- m is the *mantissa*, a rational number of n_m digits in radix β, or

$$m = d_0, d_1 d_2 ... d_{n_m - 1}$$

- e is the exponent, a signed integer on n_e bits

n_m specifies the *precision* of the format, and n_e its *dynamic*.

Imposing $d_0 \neq 0$ ensures unicity of representation.

In programming languages

- sometimes `real`, `real*8`,
- sometimes `float`,
- sometimes silly names like `double` or even `long double` (what's the semantic ?)

Some common misconceptions (1)

Floating-point arithmetic is fuzzily defined, programs involving floating-point should ne be expected to be deterministic.

- ⊕ Since 1985 there is a IEEE standard for floating-point arithmetic.
- ⊕ Everybody agrees it is a good thing and will do his best to comply
- ⊖ ... but full compliance requires more cooperation between processor, OS, languages, and compilers than the world is able to provide.
- ⊖ Besides full compliance has a cost in terms of performance.
- ⊖ There are holes in the standard (under revision)

Floating-point programs are deterministic, but should not be expected to be spontaneously portable...

Some common misconceptions (2)

A floating-point number somehow represents an interval of values around the “real value”.

- $\oplus$ An FP number only represents itself (a rational), and that is difficult enough
- $\ominus$ If there is an epsilon or an incertainty somewhere in your data, it is your job (as a programmer) to model and handle it.
- $\oplus$ This is much easier if an FP number only represents itself.

Some common misconceptions (3)

All floating-point operations involve a (somehow fuzzy) rounding error.

- ⊕ Many are exact, we know who they are and we may even force them into our programs
- ⊕ Since the IEEE-754 standard, rounding is well defined, and you can do maths about it

I need 3 significant digits in the end,
a `double` holds 15 decimal digits,
therefore I shouldn't worry about precision.

- $\ominus$ You can destroy 14 significant digits in one subtraction
- $\ominus$ it will happen to you if you do not expect it
- $\oplus$ It is relatively easy to avoid if you expect it

A variant of the previous: `PI=3.1416`

- $\oplus$ sometimes it's enough
- $\ominus$ to compute a correctly rounded sine, I need to store 1440 bits (420 decimal digits) of π...

Floating-point as it should be: The IEEE-754 standard

In the beginnings, floating-point computing was a mess

- no hope of portability
- little hope of proving results e.g. on the numerical stability of a program
- horror stories : $\arcsin\left(\frac{x}{\sqrt{x^2+y^2}}\right)$ could segfault on a Cray
- therefore, little trust in FP-heavy programs

Motivations and rationale behind the IEEE-754 standard

- Ensure portability
- Ensure provability
- Ensure that some important mathematical properties hold
  - People will assume that $x + y == y + x$
  - People will assume that $x + 0 == x$
  - People will assume that $x == y \iff x - y == 0$
  - People will assume that $\frac{x}{\sqrt{x^2+y^2}} \leq 1$
  - ...
- These benefits should not come at a significant performance cost

Obviously, we need to specify not only the formats but also the operations.

Normal numbers

Desirable properties :

- an FP number has a unique representation
- every FP number has an opposite

Normal numbers:

$$x = (-1)^s \times 2^e \times 1.m$$

Imposing $d_0 \neq 0$ ensures unicity of representation.

In radix $\beta = 2$, $d_0 \neq 0 \Longrightarrow d_0 = 1$: It needn't be stored.

- single precision: 32 bits
  - 23+1-bit mantissa, 8-bit exponent, sign bit
- double precision: 64 bits
  - 52+1- bit mantissa, 12-bit exponent, sign bit
- double-extended: anything better than double
  - IA32: 80 bits
  - IA64: 80 or 82 bits
  - Sparc: 128 bits, aka "quad precision"

Exceptional numbers

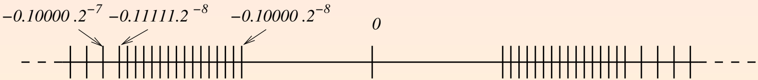
Desirable properties :

- representations of $\pm\infty$ (and therefore ± 0)
- standardized behaviour in case of overflow or underflow.
  - return ∞ or 0, and raise some flag/exception
- representations of *NaN*: Not a Number (result of 0^0, $\sqrt{-1}$, ...)
  - Quiet NaN
  - Signalling NaN

Infinities and NaNs are coded with the maximum exponent (you probably don't care).

Subnormal numbers

$$x = (-1)^s \times 2^e \times 1.m$$

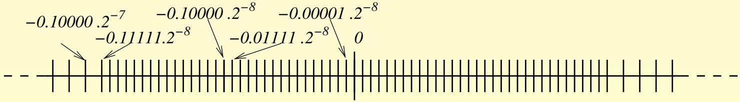


Desirable properties :

- $x == y \;\Leftrightarrow\; x - y == 0$
- Graceful degradation of precision around zero

Subnormal numbers: if $e = e_{\min}$, the implicit d_0 is equal to 0:

$$x = (-1)^s \times 2^e \times \textcolor{red}{0}.m$$



Operations

Desirable properties :

- if $a + b$ is a FP number, then $a \oplus b$ should return it
- Rounding should not introduce any statistical bias
- Sensible handling of infinities and NaNs

Correct rounding to the nearest:
The basic operations (noted $\oplus$, $\ominus$, $\otimes$, $\oslash$), and the square root should return the FP number closest to the mathematical result.
(in case of tie, round to the number with an even mantissa $\Longrightarrow$ no bias)

Three other rounding modes: to $+\infty$, to $-\infty$, to 0, with similar correct rounding requirement.

A few theorems (useful or not)

Let x and y be FP numbers.

- Sterbenz Lemma: if $x/2 < y < 2x$ then $x \ominus y = x - y$

- The rounding error when adding x and y: $r = x + y - (x \oplus y)$ is an FP number, and it may be computed as

$$r := b \ominus ((a \oplus b) \ominus a);$$

- The rounding error when multiplying x and y: $r = xy - (x \otimes y)$ is an FP number and may be computed by a (slightly more complex) sequence of $\otimes$, $\oplus$ and $\ominus$ operations.

- $\sqrt{x \otimes x + y \otimes y} \geq x$
- ...

The conclusion so far

- We have a standard for FP, and it is a good one

Floating point as it is

Who is in charge of ensuring the standard in my machine ?

- The processor
  - has internal FP registers,
  - performs FP operations,
  - raises exceptions,
  - writes results to memory.
- The operating system
  - handles exceptions
  - computes functions/operations not handled directly in hardware (subnormal numbers on Alpha)
  - handles floating-point status: precision, rounding mode, ...
- The programming language
  - should have a well-defined semantic
- The compiler
  - should preserve the well-defined semantic of the language
- The programmer
  - has to be an expert in all this ? Hey, we are physicists !

In 2005, I'm afraid you still have to be a little bit in charge.

Let us first review a few processors

… more precisely, a few families defined by their instruction sets.

The IA32 instruction set (aka x86)

Implemented in processors by Intel, AMD, Via/Cyrix, Transmeta...

- internal double-extended format on 80 bits:
  mantissa on 64 bits, exponent on 15 bits.
- (almost) perfect IEEE compliance on this double-extended format
- one status register which holds (among other things)
  - the current rounding mode
  - the precision to which operations round the mantissa: 24, 53 or 64 bits.
  - but the exponent is always 15 bits
- For single and double, IEEE-754-compliant rounding and overflow handling (including exponent) performed when writing back to memory

There is a rationale for all this.

What it means

Assume you want a portable programme, *i.e* use double-precision.

- Fully IEEE-754 compliant possible, but slow:
  - set the status flags to "round mantissa to 53 bits"
  - then write the result of every single operation to memory
  - (not every single but almost)
- Next best: compliant except for over/underflow handling:
  - set the status flags to "round mantissa to 53 bits"
  - but computations will use 15-bit exponents instead of 12
  - OK if if you may prove that your program doesn't generate huge nor tiny values
- Default behavior for C/gcc in Linux:
  - All the computations on registers are done in double-extended precision, even if the variables were declared as `double`.
  - Round to actual double only when writing to memory.
  - $\oplus$ More accurate in the common case (when portability not an issue)
  - $\ominus$ ... but it's the compiler who decides which variable is held in memory, and which is in register.
  - $\ominus$ Dangerous because of double rounding
  - $\ominus$ and because of the internal 15-bit exponent

Do you want to debug this ?

Compile this with gcc on whatever Intel or AMD processor under Linux:

```
double ref, index;

ref = 169.0 / 170.0;

for (i = 0; i < 250; i++) {
  index = i;
  if (ref == index / (index + 1)) break;
}

printf("i=%d\n",i);
```

Doesn't work either

```c
long double ref, index;

ref = 169.0 / 170.0;

for (i = 0; i < 250; i++) {
  index = i;
  if (ref == index / (index + 1)) break;
}

printf("i=%d\n",i);
```

This one is OK

```c
long double ref, index;

ref = (long double) 169.0 / 170.0;

for (i = 0; i < 250; i++) {
  index = i;
  if (ref == index / (index + 1)) break;
}

printf("i=%d\n",i);
```

Conclusion on this example

Solutions:

- live on the ege, and use explicitely double-extended (`long double`) everywhere
  - IA32 processors are perfectly IEEE-compliant when working only on double-extended.
  - a lot of work, as previous example shows
- set the processor flags to "round to 53 bits"
- run Solaris, and not Linux
  - Sparc hardware does not support double-extended,
  - and Sun people want portability accross their system range

This example also illustrates another FP adage:

Equality test between FP variables is dangerous.

Or,

If you can replace `a==b` with `(a-b)<epsilon` in your code, do it!

Quickly, the Macs

Power and PowerPC processors

- No double-extended hardware
- But one or two FMA: Fused Multiply-and-Add
  - Compute $\mathrm{round}(a \times b + c)$: Only one rounding instead of 2
  - Faster and more accurate
  - but breaks some expected mathematical properties: two ways of computing $\sqrt{a^2 + b^2}$ with different results
  - Also available on recents MIPS and HP PA-Risc, and on Itanium
- By default, gcc on MacOS X disables the use of FMA altogether
  - last time I checked. Your mileage may vary!
- In this case you may lose a factor 2 in performance to comply with IEEE-754
  - The FMA should be mentioned in the (ongoing) revision of the IEEE-754 standard

Quickly, IA64 (aka Itanium)

A commercial failure so far, but the best available FP architecture

- Two double-extended FMA (best of IA32, and best of Power)
- instead of one FP status register, 4 of them, selectable on an instruction-basis
  - you can mix round up and round down, double and double-extended
  - on all other architecture, changing the FP status requires flushing the pipeline (10-100 cycles)
- A register format with two more exponent bits (17).

The conclusion so far

- We have a standard for FP, and it is a good one
- But it is difficult to trust the machine compliance

Now we shall see that even with perfect compliance, floating-point has intrinsic pitfalls anyway.

A few pitfalls

Beware of subtractions

- *Cancellation:* if you subtract numbers which were very close (example: 1.2345e0 - 1.2344e0 = 1.0000e-4)
  - you loose significant digits (and get meaningless zeroes)
  - although the operation is exact! (no rounding error)
- Problems may arise if such a subtraction is followed by multiplications or divisions
  - You may get meaningless digits in your result

  Two typical examples:
  - computing the area of a triangle
    - formula attributed to Heron of Alexandria:
      $A := \sqrt{(s(s-x)(s-y)(s-z))}$ with $s = (x+y+z)/2$
    - Kahan's algorithm:
      Sort x, y, z so that $x \geq y \geq z$;
      If $z < x - y$ then no such triangle exists;
      else $A :=$
      $\sqrt{((x+(y+z)) \times (z-(x-y)) \times (z+(x-y)) \times (x+(y-z)))/4}$
  - solving the quadratic equation by $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ (see references)

Beware of additions

In floating-point:

$$\mathit{BigNumber} + \mathit{SmallNumber} = \mathit{BigNumber}$$

if *BigNumber* is big enough.

If you have to add terms of know different magnitude, it may be a good idea to sort them (see triangle example)

Remark: This is also the recipe for not caring about cancellations!

Speaking of which

- The semantic of most recent languages is to respect your parentheses:
  - if you write $(a+b)+c$ the compiler should not replace it with $a+(b+c)$, unless it can prove that both computations always yield the same result.
  - Even if it would be faster!
  - if you write `r := b - ((a + b)- a) ;` the compiler shouldn't replace it with `r:=0 ;`
- Well-behaved compilers will respect the semantic of the language.
- Expect to be disappointed here...
  - `gcc` is best (not always compliant with standards, but in a sensible and documented way)
  - `icc` is sloppier, but OK if you know people at Intel who will tell you the undocumented parts.
  - I know nobody at Microsoft (Kahan has a lot of evil to say about their compilers).

Beware of flushing to zero/infinity

Typical examples:

- You compute $\dfrac{x^2}{\sqrt{x^3+1}}$ for a large value of x
- Instead of (large) $\sqrt{x}$ you get 0
- Here again, the solution is
  - to expect the problem before it hurts you
  - and to protect the computation with a test which returns $\sqrt{x}$ for large values
  - (a more accurate result, obtained faster...)

Extreme version of the previous

- $f(x) = \sqrt{\sqrt{....\sqrt{x}}}$ 128 times
- $g(x) = \left(((x^2)^2)\,...\right)^2$ 128 times
- Compute and plot $g(f(x))$ for $x \in [0, 2]$

$\sqrt{1-u} = 1 - u/2 - ...$

The conclusion so far

- We have a standard for FP, and it is a good one
- But it is difficult to trust the machine compliance
- Anyway even if with perfect compliance, the standard doesn't guarantee that the result of your program is close at all to the mathematical result it is supposed to compute.

... and how to avoid them

And now a little bit of modesty

We computer scientists won't do all the work.
Nothing replaces good old mathematicians.

Classical example: Muller's recurrence

$$\begin{cases} x_0 & = & 4 \\ x_1 & = & 4.25 \\ x_{n+1} & = & 108 - (815 - 1500/x_{n-1})/x_n \end{cases}$$

- Any half-competent mathematician will find that it converges to 5
- On any calculator or computer system using non-exact arithmetic, it will converge to 100

$$x_n = \frac{\alpha 3^{n+1} + \beta 5^{n+1} + \gamma 100^{n+1}}{\alpha 3^n + \beta 5^n + \gamma 100^n}$$

Serious maths first

- Proving the absence of over/underflow may be relatively easy
  - when you compute energies, not when you compute areas
- Cancellation and under/overflow problems usually solved by
  - some tests, and
  - different, mathematically equivalent, formulae
  - provided you have detected the problem before it hurts you...
- Sensitivity and conditioning:

$$Cond = \frac{|\text{relative change in output}|}{|\text{relative change in input}|} = \lim_{\widehat{x}\to x} \frac{|(f(\widehat{x}) - f(x))/f(x)|}{|(\widehat{x}-x)/x|}$$

  - $Cond \geq 1$ problem is ill-conditionned / sensitive to rounding
  - $Cond \ll 1$ problem is well-conditionned / resistant to rounding
  - $Cond$ may depend on x: again, make cases...
- Error analysis techniques: how are your equations sensitive to roundoff errors ?
  - Forward error analysis: what errors did you make ?
  - Backward error analysis: which problem did you solve exactly ?
  - Several attempts to automate them (see Langlois' habilitation thesis @ ENS-Lyon)
- Warning: Real maths happen. Your mileage may vary.

Mindless schemes to evaluate numerical quality of your program

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.
  - Maple, Mathematica, GMP/MPFR
- Repeat the computation with same precision but different (IEEE-754) rounding modes, and compare the results.
  - all you need is change the processor status in the beginning
- Repeat the computation a few times with same precision, rounding each operation randomly, and compare the results.
  - stochastic arithmetic, CESTAC
- Repeat the computation a few times with same precision but slightly different inputs, and compare the results.
  - easy to do yourself

None of these schemes provide any guarantee. They may increase confidence, though.

See "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?" on Kahan's web page

Interval arithmetic

- Instead of computing $f(x)$, compute an interval $[f_l, f_u]$ which is guaranteed to contain $f(x)$
  - operation by operation
  - use directed rounding modes
  - several libraries exist
- This scheme does provide a guarantee
- ... which is often overly pessimistic
  ("Your result is in $[-\infty, +\infty]$, guaranteed")
- Limit interval bloat by being clever (changing your formula)
- ... and/or using bits of arbitrary precision when needed (MPFI library).
- Therefore not a mindless scheme
- Fair tradeoff between mindlessness and manual proof

The conclusion so far

- We have a standard for FP, and it is a good one
- But it is difficult to trust the machine compliance
- Anyway even if with perfect compliance, the standard doesn't guarantee that the result of your program is close at all to the mathematical result it is supposed to compute.
- But at least it makes it possible to do serious mathematics on it, and also to try various recipes

One drawback of the standard:

- In the 70s, when people ran the same program on different machines, they got widely different results.
  - They had to think about it and find what was wrong.
- Now they get the same result, and therefore trust it.
  - We have to educate them...

Arithmetic is not always the culprit

- Ask first-year students to write an n-body simulation
- Run it with one sun and one planet
- You always get rotating ellipses
- Analysing the simulation shows that it creates energy.

$$\mathbf{x}(t) := \mathbf{v}(t)\delta t$$

Elementary functions

I've been telling lies so far

The IEEE-754 standard for floating-point arithmetic enables *portability* and *provability* of FP algorithms
... at least, as long as no elementary function is used.

Logarithm, exponential, trigonometric, hyperbolic, ...

How does your PC compute elementary functions?

Rule of the game: use only $+$, $-$, $\times$ (and maybe $/$ and $\sqrt{\ }$ but they are expensive).

- Polynomial approximation on a small interval (degree 3 to 20)
- Argument reduction using mathematical identities

Remark: IA32 specifies hardware instructions for elementary functions. They are microcoded (barely faster than software equivalent) and often of poor quality.

Standardisation of the elementary functions so far

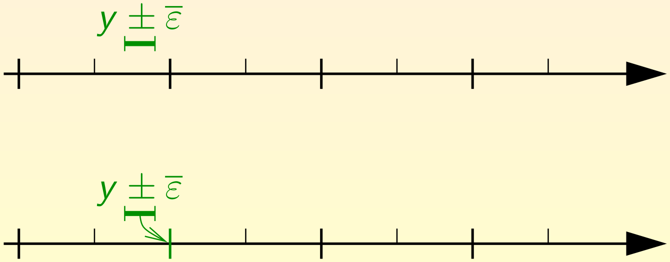
- Language standards give lists of functions
  - Example: appendix B.11 of the C99 standard:

```
...
double cos(double x);
float cosf(float x);
long double cosl(long double x);
...
```

- but they do not specify their behaviour...
- Current practice is to offer implementations in round-to-nearest mode, which are *accurate faithful*
  - or, 0.501 ulp accuracy
  - or, 99% correctly rounded.

  A few libraries do their best to support directed rounding.
- Rarer functions may behave badly (hyperbolic on Linux)
- 100% correct rounding is expensive because of the *Table Maker's Dilemma*

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound $\overline{\varepsilon}$.
- What we compute: y such that $f(x) \in [y - \overline{\varepsilon}, y + \overline{\varepsilon}]$



The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00	100001	0,00000,43429,2
2	0,30102,99956,6	100002	0,00000,86858,0
3	0,47712,12547,2	100003	0,00001,30286,4
4	0,60205,99903,3	100004	0,00001,73714,3
5	0,69897,00043,4	100005	0,00002,17141,8 *F*
6	0,77815,12503,8 *A*	100006	0,00002,60568,9
7	0,84509,80400,1	100007	0,00003,03995,5
8	0,90308,99869,9	100008	0,00003,47421,7
9	0,95424,25094,4	100009	0,00003,90847,4
11	0,04139,26851,6	1000001	0,00000,04342,9
12	0,07918,12460,5	1000002	0,00000,08685,9
13	0,11394,33523,1	1000003	0,00000,13028,8
14	0,14612,80356,8	1000004	0,00000,17371,7
15	0,17609,12590,6 *B*	1000005	0,00000,21714,7 *G*
16	0,20411,99826,6	1000006	0,00000,26057,6
17	0,23044,89213,8	1000007	0,00000,30400,5
18	0,25527,25051,0	1000008	0,00000,34743,4
19	0,27875,36009,5	1000009	0,00000,39086,3
101	0,00432,13737,8	10000001	0,00000,00434,3
102	0,00860,01717,6	10000002	0,00000,00868,6
103	0,01283,72247,1	10000003	0,00000,01302,9
104	0,01703,33393,0	10000004	0,00000,01737,2
105	0,02118,92990,7 *C*	10000005	0,00000,02171,5 *H*
106	0,02530,58652,6	10000006	0,00000,02605,8
107	0,02938,37776,9	10000007	0,00000,03040,1
108	0,03342,37554,9	10000008	0,00000,03474,4
109	0,03742,64979,4	10000009	0,00000,03908,6
1001	0,00043,40774,8	100000001	0,00000,00043,4
1002	0,00086,77215,3	100000002	0,00000,00086,9
1003	0,00130,09330,2	100000003	0,00000,00130,3
1004	0,00173,37128,1	100000004	0,02000,00173,7
1005	0,00216,60617,6 *D*	100000005	0,00000,00217,1 *I*
1006	0,00259,79807,2	100000006	0,00000,00260,6
1007	0,00302,94705,5	100000007	0,00000,00304,0
1008	0,00346,05321,1	160000008	0,00000,00347,4
1009	0,00389,11662,4	100000009	0,00000,00390,9
10001	0,00004,34272,8	1000000001	0,00000,00004,3
10002	0,00008,68502,1	1000000002	0,00000,00008,7
10003	0,00013,02688,1	1000000003	0,00000,00013,0
10004	0,00017,36830,6	1000000004	0,00000,00017,4
10005	0,00021,70029,7 *E*	1000000005	0,00000,00021,7 *K*
10006	0,00026,04985,5	1000000006	0,00000,00026,1
10007	0,00030,39997,8	1000000007	0,00000,00030,4
10008	0,00034,72966,9	1000000008	0,00000,00034,7
10009	0,00039,06892,5	1000000009	0,00000,00039,1

- I want 12 significant digits
- I have an approximation scheme that gives 14
- or,

$$y = \log(x) \pm 10^{-14}$$

- "Usually" that's enough to round

$$y = x, xxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxx83 \pm 10^{-14}$$

- Dilemma when

$$y = x, xxxxxxxxxx50 \pm 10^{-14}$$

The first table-maker rounded these cases randomly, and recorded them to confound copiers.

Gal's probabilities

What is the probability of the Table's Maker Dilemma ?
(People who appreciate clean statistics should look away for a few slides)

- $y = \log(x) \pm 10^{-14}$ and we want 12 digits
- Assume that the digits after the 12th are uniformely distributed ...
- ... then the dilemma occurs once in 100 cases
(when the two last digits are 50).
- A more accurate scheme reduces this probability :
$y = \log(x) \pm 10^{-15} \quad \longrightarrow \quad$ once in 1000
- In general

$$y = \log(x) \pm 10^{-12-N} \quad \longrightarrow \quad p(\text{Dilemma}) \approx 10^{-N}$$

From the opposite point of view:

- The table has a finite number of entries, say 10^{10}.
- One of these entries holds the number that is the most difficult to round
- Under the previous flaky probabilistic hypotheses, I expect one of the 10^{10} logs to be like

$$log(x) = x, xxxxxxxxxxx50000000000zz...$$

- In other terms,
  - There probably exists a working precision which allows to round the whole table correctly
  - We expect it to be about 10 digits after the 12th.

With machine FP numbers it's just the same

Double-precision elementary functions:

- More or less 2^{64} numbers, at least 2^{62} entries for each function.
- Floating point correct rounding: at the 53th bit.
- Most libms compute about 60 exact bits, and round correctly most of the time, just like Renaissance tables.
- Statistics *à la* Gal predict worst cases requiring $53 + 64 = 117$ bits (more or less).

The first correctly rounded library: IBM Accurate Portable Library, or `libultim`, written by Ziv.

- one or two steps using double-doubles
- further steps using a multiple-precision package (up to 800 bits)

Drawbacks:

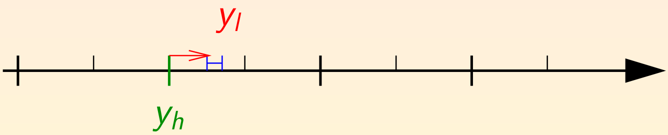
- unproven
  - theoretical reason: are 800 bits enough ?
  - practical reasons...
- very large worst-case time and memory
- only round-to-nearest mode
  - directed rounding modes may be more useful (interval arithmetic)

Initiated by David Defour's thesis

- Lefèvre and Muller computed worst-case required accuracy for several functions
  - this lifts off the theoretical obstacle to proven CR
  - as expected, correct rounding to double-precision (53 bits) typically requires 117 bits of internal precision (or $\overline{\varepsilon} = 2^{-117}$)
  - up to 150 bits in special cases.
- Two Ziv steps only
  - First step using double-double arithmetic
  - Second step "just right", always provide CR, uses an ad-hoc package for 200-bit precision.
- The four IEEE-754 rounding modes
- Less than 4KB / function
- A proof of the CR property is provided along with the code

Double-double ?

- Store a high-precision x number as two doubles x_h and x_l such as $x = x_h + x_l$



- Addition and subtraction fast
- Multiplication relatively fast
  - (fast if you have an FMA)

Proof of correct rounding ?

- Shared work:
  - many useful FP theorems (Sterbenz, etc)
  - double-double arithmetic well-known and well-proven
  - proof of correctness of rounding tests, including special cases (denormals etc)
  - Maple procedures *e.g.* for polynomial approximations
    - compute a good polynomial with coefficients representable as doubles or double-doubles
    - compute bound on approximation error
    - compute bounds on cumulated rounding errors in Horner evaluation (both absolute and relative)
- Function-specific work
  - special cases
  - argument reduction
  - specific tricks (multiplication by a constant, ...)
- A Maple script produces the C header file with all the constants (poly coeffs etc) and implements the error analysis
  - will be part of the proof
  - allows secure exploration of various tradeoffs

Long-term goal of this work

Correctly-rounded elementary functions as standard

Proposal: several levels of quality for elementary functions

- Level 0: current situation (accurate-faithful)
  - plus well-defined behaviour in exceptional cases
  - correct rounding may conflict with the preservation of useful mathematical properties, e.g. $\arctan(x) < \pi/2$
- Level 1: accurate-faithful, with correct rounding on well-defined, sensible intervals
  - sine function: on $[-2^{64}, 2^{64}]$ (otherwise it's noise)
  - or even on $[-\pi, \pi]$
- Level 2: correct rounding everywhere
  - currently feasible for single precision
  - in double precision, currently feasible for e^x, log, 2^x and log_2 thanks to Muller/Lefèvre
  - trigonometric functions will require theoretical advances
  - double-extended precision, too

One important question:

What price are you, the users, ready to pay for correct rounding ?

Performance results

log timings:

Pentium 4 Xeon / Linux Debian sarge / gcc 3.3		
	avg time	max time
`mpfr`	61325	307628
`libultim`	521	388196
`crlibm`	534	51608
libm (accurate faithful)	*191*	*6540*

PowerPC G4 / MacOS X / gcc2.95		
	avg time	max time
`mpfr`	4895	8620
`libultim`	22	19890
`crlibm` (without FMA)	32	1241
`crlibm` (using FMA)	24	1144
libm (accurate faithful)	*15*	*16*

Relaxing portability constraint

An exponential optimized for the Itanium-1 processor, with a little help of Intel (gratefully acknowledged)

- use double-extended arithmetic for the first step
- use double-double-extended arithmetic for the second step
- use fused multiply-and-add everywhere
- allow 8KB of tables (Itanii have huge caches)

(timings in cycles, including 37 cycles for a function call)

exp Itanium-1	avg time	max time
`libultim`	193	2439385
`mpfr`	24540	115152
`crlibm` portable	295	5633
`crlibm` using DE, two steps	100	162
`crlibm-DE`, second step alone	124	126
`libm` (accurate faithful)	*89*	*89*

Overhead of correct rounding is getting negligible

Conclusions on our work on `crlibm`

- `crlibm` is a good framework for implementing correctly rounded functions
  - 100 pages of documentation/proof
  - The Mean Implementation Time per Function decreases (currently down to 2 student$\times$month). Still, the real cost of implementing a correctly rounded function is coffee consumption, not performance
  - Reasonable confidence in the code
  - Reasonable confidence that we can locate remaining bugs
  - However the proof is a mixture of C, LaTeX and Maple
- Discipline is good
  - Sun published a correctly rounded library in December 2004, we found errors in the trigonometric functions in a few hours.
  - The discipline we set up to manage correctness helps a lot for performance tuning (including future-proofness ?)
- Relaxing portability allows negligible performance cost
  - I'm off to Intel to sell them this idea.
- Correctly rounded elementary functions for the masses are around the corner.

Conclusion

It's been said already

- We have a standard for FP, and it is a good one
- But it is difficult to trust the machine compliance
- Anyway even if with perfect compliance, the standard doesn't guarantee that the result of your program is close at all to the mathematical result it is supposed to compute.
- But at least it makes it possible to do serious mathematics on it, and also to try various recipes
- It also makes it possible to implement correctly rounded elementary functions
  - otherwise it's mostly useless to you, the users.

So, do you trust your computer now ?

"It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic."

A. Householder

Feel nervous, but feel in control. It's not dark magic, it's science.

Any questions ?