The Floating Point Environment

Florent de Dinechin, Arénaire Project, ENS-Lyon

Нижний Новгород, 30/03/2010.99999

Floating-point as it should be

Standard compliance

Floating point in current processors

Floating point in current software (OS, languages and compilers)

Conclusion: educating the weakest link

Floating-point as it should be: The IEEE-754-85 standard

Floating-point as it should be

Standard compliance

Floating point in current processors

Floating point in current software (OS, languages and compilers)

Conclusion: educating the weakest link

The dark ages of anarchy

In the ancient times (before 1985), there were
as many implementations of floating-point as there were machines

- 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

I hope you will tell me a few horror stories from Russia, too.

Rationale behind the IEEE-754-85 standard

- Enable data exchange
- 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 $x == y \Leftrightarrow x - y == 0$
  - People will assume that $\frac{x}{\sqrt{x^2+y^2}} \leq 1$
  - ...
- These benefits should not come at a significant performance cost

Obviously, need to specify not only the number formats
but also the operations on these numbers.

Normal numbers

Desirable properties :

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

Normal numbers

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

For unicity of representation, we impose $d_0 \neq 0$.
(In binary, $d_0 \neq 0 \Longrightarrow d_0 = 1$: It needn't be stored.)

- single precision: 32 bits
  - 23+1-bit significand, 8-bit exponent, sign bit
- double precision: 64 bits
  - 52+1- bit significand, 12-bit exponent, sign bit
- double-extended: anything better than double
  - IA32: 80 bits
  - IA64: 80 or 82 bits

Exceptional numbers

Desirable properties :

- representation of 0
- 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

Choice of binary representation

Desirable property: the order of FP numbers is the lexicographical order of their binary representation

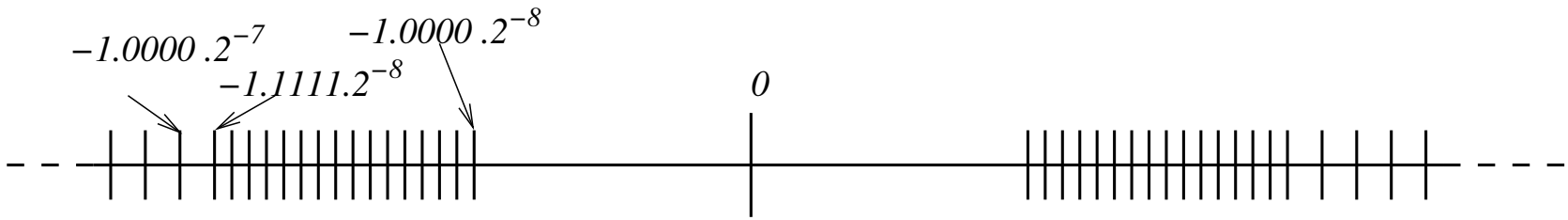
Binary encoding of positive numbers

- place exponent at the MSB (left of significand)
- infinity is larger than any normal number:
  code it with the largest exponent 111...1
- zero is smaller than any normal number:
  code it with the smallest exponent 000...0
- for normal exponents: biased representation
  - assume w_E bits of exponent
  - exponent field $E \in \{0 \ldots 2^{w_E} - 1\}$ codes for exponent $e = E - \text{bias}$
  - In IEEE-754, bias for significand in $[1, 2)$ is $\text{bias} = 2^{w_E - 1} - 1 \; = \; 0111 \ldots 1$

How to code NaNs? Significand of infinity? Significand of 0? ...

Subnormal numbers

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



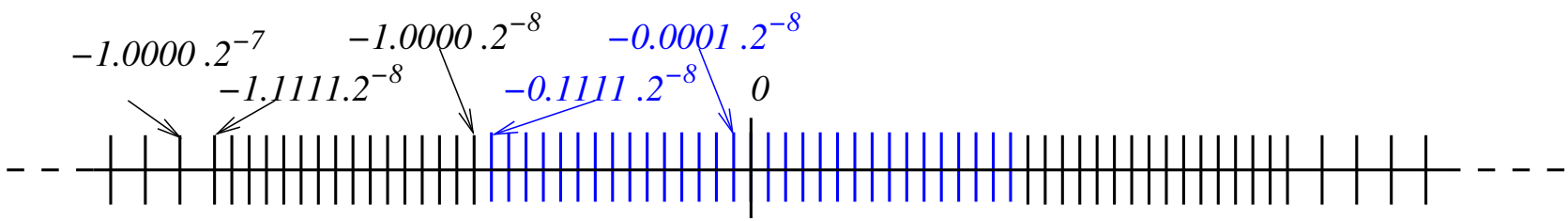
Desirable properties :

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

Subnormal numbers

if $E = 00...0$, the implicit d_0 is equal to 0:

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



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 significand $\Longrightarrow$ no bias)

Three other rounding modes: to $+\infty$, to $-\infty$, to 0, with similar correct rounding requirement (and no tie problem).

Complete binary representation (positive numbers)

3 bits of exponent, 4 bits of fraction (4+1 bits of significand)

exp fraction	value	comment
000 0000	0	Zero
000 0001	$0.0001 \cdot 2^{e_{\min}}$	smallest positive (subnormal)
...	...	
000 1111	$0.1111 \cdot 2^{e_{\min}}$	largest subnormal
001 0000	$1.0000 \cdot 2^{e_{\min}}$	smallest normal
...	...	
110 1111	$1.1111 \cdot 2^{e_{\max}}$	largest normal
111 0000	$+\infty$	
111 0001	NaN	
...	...	
111 1111	NaN	

NextAfter obtained by adding 1 to the binary representation

from 0 to $+\infty$

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 if $x \geq y$ it may be computed as

$$r := y \ominus ((x \oplus y) \ominus x);$$

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

Here I should try to prove Sterbenz lemma

Floating-point format in radix β with p digits of significand
Suppose x and y are positive.
Notation using integral significands:

$$x = M_x \times \beta^{e_x - p + 1},$$

$$y = M_y \times \beta^{e_y - p + 1},$$

with

$$\begin{cases} e_{\min} \leq e_x \leq e_{\max} \\ e_{\min} \leq e_y \leq e_{\max} \\ 0 \leq M_x \leq \beta^p - 1 \\ 0 \leq M_y \leq \beta^p - 1. \end{cases}$$

$y \leq x$ therefore $e_y \leq e_x$: define $\delta = e_x - e_y$

$$x - y = \left(M_x \beta^\delta - M_y\right) \times \beta^{e_y - p + 1}.$$

Define $M = M_x \beta^\delta - M_y$

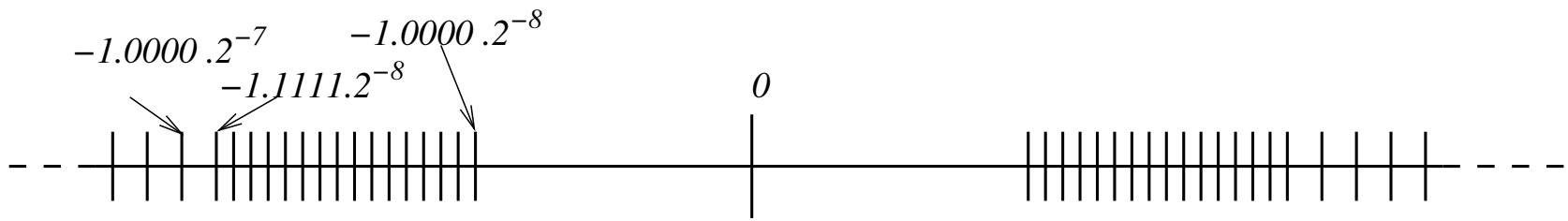
- $x \geq y$ implies $M \geq 0$;
- $x \leq 2y$ implies $x - y \leq y$, hence $M\beta^{e_y - p + 1} \leq M_y \beta^{e_y - p + 1}$; therefore,

$$M \leq M_y \leq \beta^p - 1.$$

So $x - y$ is equal to $M \times \beta^{e - p + 1}$ with $e_{\min} \leq e \leq e_{\max}$ and $|M| \leq \beta^p - 1$. This shows that $x - y$ is a floating-point number, which implies that it is exactly computed.

Remarks on this proof

- We haven't used the rounding mode ?!?
  - We just proved that the mathematical result is representable
  - Any rounding mode $\circ$ verifies: if Z is representable, then $\circ(Z) = Z$
  - Sterbenz lemma is true for any rounding mode.
- We need subnormals, of course.



(Normal numbers have an integral significand such that $\beta^{p-1} \leq M \leq \beta^p - 1$ and we couldn't prove the left inequality)

- We don't care about the binary encoding (only that there is an $e_{\min}$)

The conclusion so far

- We have a standard for FP, and it seems well thought out.
- (all we have seen was already in the 1985 version – more on the 2008 revision later)

Let us try to use it.

Standard compliance

Floating-point as it should be

Standard compliance

Floating point in current processors

Floating point in current software (OS, languages and compilers)

Conclusion: educating the weakest link

A frightening introductory example

Let us compile the following program with gcc under Linux for an IA32 processor (Intel Pentium, AMD Athlon, ...)

```
1   double ref, index;
2
3   ref = 169.0 / 170.0;
4
5   for (i = 0; i < 250; i++) {
6     index = i;
7     if (ref == (index / (index + 1))  )  break;
8   }
9
10  printf("i=%d\n",i);
```

First conclusion

Equality test between FP variables is dangerous.

Or,

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

A physical point of view:
Given two coordinates (x, y) *on a snooker table,*
the probability that the ball stops at position (x, y) *is always zero.*

Yes but which epsilon in the previous program?
Besides, obviously, on this expensive laptop, FP computing is not deterministic, even within such a small program.

Go fetch me the person in charge

Who is in charge of ensuring the standard?

- The processor
  - has internal FP registers,
  - performs basic FP operations,
  - raises exceptions,
  - writes results to memory.

Who is in charge of ensuring the standard?

- The processor
- The operating system
  - handles exceptions
  - computes functions/operations not handled directly in hardware
    - most elementary functions (sine/cosine, exp, log, ...),
    - divisions and square roots on Itanium
    - subnormal numbers on Alpha
  - handles floating-point status: precision, rounding mode, ...

Who is in charge of ensuring the standard?

- The processor
- The operating system
- The programming language
  - should have a well-defined semantic,
  - ... (detailed in some arcane 1000-pages document)

Who is in charge of ensuring the standard?

- The processor
- The operating system
- The programming language
- The compiler
  - has hundreds of options
  - some of which to preserve the well-defined semantic of the language
  - but probably not by default
    - Marketting says: default should be *optimize for speed*!

Who is in charge of ensuring the standard?

- The processor
- The operating system
- The programming language
- The compiler
- The programmer
  - ... is in charge in the end.

I knew it would all be my fault in the end, says the Programmer.

The conclusion so far

- We have a standard for FP, and it is a good one
- It doesn't seem enabled by default
- Enabling it will require cooperation of many entities

Floating point in current processors

Floating-point as it should be

Standard compliance

Floating point in current processors

Floating point in current software (OS, languages and compilers)

Conclusion: educating the weakest link

Let us review a few processors

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

The legacy FPU of IA32 instruction set

Implemented in processors by Intel, AMD, Via/Cyrix, Transmeta... since the Intel 8087 coprocessor in 1985

- internal double-extended format on 80 bits: significand 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 significand: 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 probably is a rationale for all this, but... ask Intel people.

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 significand 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 significand 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
  - OK for 169/170, right ?

The broken program, uglily fixed

```
11 #include<stdio.h>
12 #include <fpu_control.h> /* Non portable, non standard */
13
14 int main() {
15   int i;
16    double ref, index;
17
18   unsigned short cw;
19   /* Set FPU flags to use double, not double extended,
20      with rounding to nearest */
21   cw = (_FPU_DEFAULT & ~_FPU_EXTENDED)|_FPU_DOUBLE;
22   _FPU_SETCW(cw);
23
24
25
26   ref = 169.0 / 170.0;
27
28   for (i = 0; i < 250; i++) {
29     index = i;
30   if (ref == index / (index + 1.0))
31     break;
32   }
33
34   printf("i=%d\n",i);
35 }
```

Great, the FP computation seems deterministic (i.e. portable) now, but with this non-portable mess around it.

Wait a minute

Obviously, the problem was that the processor was rounding one 169/170 to double-extended, and the other to double.

But but but, all my variables were declared as double !?!

Is it a gcc bug? A linux bug? A programmer bug?

...

A light at the end of the IA32 tunnel?

It is called SSE2.

- Available for all recent IA32 and EMT64 processors (AMD and Intel)
- An additional FP unit able of
  - 2 identical double-precision FP operations in parallel, or
  - 4 identical single-precision FP operations in parallel.
- Real double precision (12-bit exponents)
- ... and usually faster than the standard FPU

Let us see if this PC is recent enough:

```
gcc -msse2 -mfpmath=sse retrouve.c
```

Quickly, the old Macs

Power and PowerPC processors

- No double-extended hardware
- But one or two FMA: Fused Multiply-and-Add
  - Compute round($a \times b + c$):
    - faster: Roughly in the time of a FP multiplication
    - more accurate: Only one rounding instead of 2
  - but breaks some expected mathematical properties:
    - Loss of symmetry in $\sqrt{a^2 + b^2}$
    - Worse: `if` $b^2 \geq 4ac$ `then` (...) $\sqrt{b^2 - 4ac}$
- By default, gcc on MacOS X disables the use of FMA altogether
  - last time I checked. Your mileage may vary!
- IEEE-754-1985 compliance costs a factor 2 in performance...
  - Addition: round($a \times 1 + c$)
  - Multiplication: round($a \times b + 0$)
- Fixed in IEEE-754-2008

Quickly, IA64 (aka Itanium)

They don't sell that many of them, 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
- All processors can do better than the standard,
  in different ways
- To get standard behaviour,
  you will lose performance and/or accuracy

Unfortunately, the same holds for the software...

Floating point in current software (OS, languages and compilers)

Evaluation of an expression

Consider the following program, whatever the language

```
36    float a,b,c,x;
37
38    x = a+b+c+d;
39  }
```

Two questions:

- In which order will the three addition be executed?
- What precision will be used for the intermediate results?

Fortran, C and Java have completely different answers.

Evaluation of an expression

```
40    float a,b,c,x; /* Simple precision */
41
42    x = a+b+c+d;
43  }
```

- In which order will the three addition be executed?
  - With two FPUs (dual FMA, or SSE2, ...),
    $(a + b) + (c + d)$ faster than $((a + b) + c) + d$
  - If a, c, d are constants, $(a + c + d) + b$ faster.
  - Is the order fixed by the language, or is the compiler free to choose?
  - Similar issue: should multiply-additions be fused in FMA?

Evaluation of an expression

```
44    float a,b,c,x;
45
46    x = a+b+c+d;
47  }
```

- In which order will the three addition be executed?
- What precision will be used for the intermediate results?
  - *Bottom up* precision: (here all `float`)
    - elegant (context-independent)
    - portable
    - sometimes dangerous: compare `C=(F-32)*(5/9)` and `C=(F-32)*5/9`
  - Use the maximum precision available which is no slower
    - in C, variable types refer to memory locations
    - more accurate result
  - Is the precision fixed by the language, or is the compiler free to choose?

Fortran's philosophy (1)

Citations are from the Fortran 2000 language standard: *International Standard ISO/IEC1539-1:2004. Programming languages – Fortran – Part 1: Base language*

The FORmula TRANslator translates mathematical formula into computations.

Any difference between the values of the expressions `(1./3.)*3.` and `1.` is a computational difference, not a mathematical difference. The difference between the values of the expressions `5/2` and `5./2.` is a mathematical difference, not a computational difference.

La philosophie de Fortran (2)

Fortran respects mathematics, and only mathematics.

(...) the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated. Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results.

Remark: This philosophy applies to both order and precision.

Fortran in details

X,Y,Z of any numerical type, A,B,C of type real or complex, I, J of integer type.

Expression	Allowable alternative form
X+Y	Y+X
X*Y	Y*X
-X + Y	Y-X
X+Y+Z	X + (Y + Z)
X-Y+Z	X - (Y - Z)
X*A/Z	X * (A / Z)
X*Y-X*Z	X * (Y - Z)
A/B/C	A / (B * C)
A / 5.0	0.2 * A

Last line is valid if you replace 5 by 4, but not by 3. Why?

Fortran in details (2)

X,Y,Z of any numerical type, A,B,C of type real or complex, I, J of integer type.

Expression	Forbidden alternative form
I/2	0.5 * I
X*I/J	X * (I / J)
I/J/A	I / (J * A)
(X + Y) + Z	X + (Y + Z)
(X * Y) - (X * Z)	X * (Y - Z)
X * (Y - Z)	X*Y-X*Z

Fortran in details (3)

Fortunately, Fortran respects your parentheses.

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression.

(this was the solution to the last FP bug of LHC@Home at CERN)

Fortran in details (4)

You have been warned.

The inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions `A*I/J` *and* `A*(I/J)` *may have different mathematical values if I and J are of type integer.*

That was the difference between `C=(F-32)*(5/9)` and `C=(F-32)*5/9`.

Enough standard, the rest is in the manual

(yes, you should read the manual of your favorite language and also that of your favorite compiler)

Advertising: the Lahey compiler seems to make its best with respect to portability.

The C philosophy

The "C99" standard:
International Standard ISO/IEC 9899:1999(E).
Programming languages – C

- Contrary to Fortran, the standard imposes an order of evaluation
  - Parentheses are always respected,
  - Otherwise, left to right order with usual priorities
  - If you write $x = a/b/c/d$ (all FP), you get 3 (slow) divisions.
- Consequence: little expressions rewriting
  - Only if the compiler is able to prove that the two expressions always return the same FP number, including in exceptional cases

C in the gory details

Morceaux choisis from appendix F.8.2 of the C99 standard:

- Commutativities are OK
- `x/2` may be replaced with `0.5*x`,
  because both operations are always exact in IEEE-754.
- `x*1` and `x/1` may be replaced with `x`
- `x-x` may not be replaced with `0`
  unless the compiler is able to prove that `x` will never be ∞ nor NaN
- Worse: `x+0` may not be replaced with `x`
  unless the compiler is able to prove that `x` will never be -0
  because $(-0) + (+0) = (+0)$ and not (-0)
- On the other hand `x-0` may be replaced with `x`
  if the compiler is sure that rounding mode will be to nearest.
- `x == x` may not be replaced with `true`
  unless the compiler is able to prove that `x` will never be NaN.

Obvious impact on *performance*

Therefore, default behaviour of commercial compiler tend to ignore this part of the standard...
But there is always an option to enable it.

The C philosophy (2)

- So, perfect determinism wrt order
- Strangely, precision is not determined by the standard: it defines a bottom-up minimum precision, but invites the compiler to take the largest precision which is larger than this minimum, and no slower
- Idea:
  - If you wrote `float` somewhere, you probably did so because you thought it would be faster than `double`.
  - If the compiler gives you `long double` you won't complain.

If you're not yet fully asleep, you now realize that my example buggy program fully respects C99 and IEEE-754.

Of registers and memory

```
48    double ref, index;
49
50    ref = 169.0 / 170.0;
51
52    for (i = 0; i < 250; i++) {
53      index = i;
54      if (ref == (index / (index + 1))  )  break;
55    }
56
57    printf("i=%d\n",i);
```

Drawbacks of C philosophy

- Small drawback
  - Before SSE, `float` was almost always double or double-extended
  - With SSE, `float` should be single precision (2-4× faster)
  - Or, on a newer PC, the same computation gets much less accurate!
- Big drawbacks
  - Storing a `float` variable in 64 or 80 bits of memory instead of 32 is usually slower, therefore in the C philosophy it should be avoided.
  - The compiler is free to choose which variables stay in registers, and which go to memory (register allocation/spilling)
  - It does so almost randomly (it totally depends on the context)
  - Thus, sometimes a value is rounded twice, which may be even less accurate than the target precision
  - And sometimes, the same computation may give different results at different points of the program.

A funny horror story

(it's a real story, told by somebody at CERN)

- Use the (robust and tested) standard sort function of the STL C++ library
- to sort objects by their radius: according to `x*x+y*y`.
- Sometimes (rarely) segfault, infinite loop...
- Why?
  - the sort algorithm works under the naive asumption that if $A \not< B$, then $A \geq B$
  - (it is difficult to write a sort algorithm without this asumption)
  - Sometimes, `x*x+y*y` compiled differently at two points of the programme, which breaks the asumption.

It's a bug, for sure, but there was no programming mistake.
And it is very difficult to fix.
(let us try to fix `retrouve.c`)

Still not there

```
58    long double ref, index;
59
60    ref = 169.0 / 170.0;
61
62    for (i = 0; i < 250; i++) {
63      index = i;
64      if (ref == index / (index + 1)) break;
65    }
66
67    printf("i=%d\n",i);
```

It takes what it takes

```
68    long double ref, index;
69
70    ref = (long double) 169.0 / 170.0;
71
72    for (i = 0; i < 250; i++) {
73      index = i;
74      if (ref == index / (index + 1)) break;
75    }
76
77    printf("i=%d\n",i);
```

Don't worry, this program also works on processors without double-extended:
double will be used instead.

Quickly, Java

- Integrist approach to determinism: *compile once, run everywhere*
  - `float` and `double` only.
  - Evaluation semantics with fixed order and precision.
  - ⊕ No sort bug.
  - ⊖ Performance impact, but... only on PCs (Sun also sells SPARCs)
  - ⊖ You've paid for double-extended processor, and you can't use it (because it doesn't *run anywhere*)

The great Kahan doesn't like it.

- Many numerical unstabilities are solved by using a larger precision
- Look up *Why Java hurts everybody everywhere* on the Internet

I tend to disagree with him here.

Quickly, Python

Floating point numbers

These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow.

You have been warned.

Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

Quickly, the operating systems

Default flags are typically round to nearest.
WRT rounding precision on IA32 hardware,

- Linux will use double-extended by default,
  - (but on Debian AMD64, SSE2 is used by default, so most FP is double anyway)
- MacOS X on intel uses only SSE2, too.
- Solaris will work in double by default,
  - Sun sells Solaris on SPARC and x86.
  - Sun also sells Java.
- Microsoft OSs work in double by default
  - since Bill Gates decided that "the 8087 *socket* will mostly stay empty anyway"
- These defaults may be overriden by languages and programs.

Conclusion: educating the weakest link

It shouldn't be so messy, should it?

Don't worry, things are improving

- SSE2 makes life a lot simpler for most of us
- The 2008 revision of IEEE-754 addresses the issues of
  - reproducibility versus performance
  - precision of intermediate computations
  - etc
- but it will take a while to percolate to your programming environment

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

(... well, now they are *piloted* using floating-point arithmetic...)

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

Let us quickly review how we can improve our confidence in floating-point.

Arithmetic is not always the culprit

- Ask first-year students to write a simulation of one planet around a sun

$$\begin{cases} \mathbf{x}(t) & := & \mathbf{v}(t)\delta t \\ \mathbf{v}(t) & := & \mathbf{a}(t)\delta t \\ \mathbf{a}(t) & := & \dfrac{K}{||\mathbf{x}(t)||^2} \end{cases}$$

- You always get rotating ellipses
- Analysing the simulation shows that it creates energy.

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

Avoiding cancellations in practice

Computing the area of a triangle

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 $A :=$
$\sqrt{((x+(y+z)) \times (z-(x-y)) \times (z+(x-y)) \times (x+(y-z)))/4}$

Exercise: Solving the quadratic equation by $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$

Beware of additions

In floating-point:

$$BigNumber + SmallNumber = BigNumber$$

if *BigNumber* is big enough.

Solutions:

- If you have to add terms of known different magnitude, it may be a good idea to sort them (see triangle example)
- Compensated summation algorithms:
  - compute the rounding error made by each operation
  - and add it back at a later stage to correct the final result
  - works even for large sums-of-products (matrix operations etc)
  - Look up recent papers by Rump and Ojita

Speaking of which

When you write a compensated summation algorithm, it is full of
`r := b - ((a + b)- a);`
That's why it's a good thing that even Fortran respects your parentheses.

Beware of flushing to zero/infinity

Typical examples:

- You compute $\frac{x^2}{\sqrt{x^3+1}}$ 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
- $g(x) = \left(\left((x^2)^2\right)...\right)^2$ 128 times
- Compute and plot $g(f(x))$ for $x \in [0, 2]$

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

Trust your math

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 very convincingly to 100

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

Error analysis

- Proving the absence of over/underflow may be relatively easy
  - when you compute energies, not when you compute areas
- 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 ?
- Notion of 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...

More details (and even tutorials) to follow.

"Mindless" schemes to improve confidence

- 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

Practical examples later (course on Gappa-assisted error analysis)

Conclusion: to be continued

- We have a standard for FP, and eventually your PC will comply
- 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 enables serious mathematics with floating-point

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...

Tentative outline of next lecture

Becoming a binary floating-point expert

- Representation tips and tricks
- Rounding to an integer
- Sterbenz Lemma
- Exact addition
- Exact multiplication
- Evaluating a polynomial accurately to the last bit
- Compensated sums
- Compensated Horner evaluation