Interval arithmetic and numerical reproducibility issues

Nathalie Revol[1] Philippe Théveny[2]

[1]INRIA – LIP (UMR 5668 CNRS - ENS Lyon - INRIA - UCBL)

[2]ENS Lyon – LIP (UMR 5668 CNRS - ENS Lyon - INRIA - UCBL)

SWIM – June 2013

Schedule

1. the Bad
2. the Good
3. the Ugly

Floating-point computations

Numerical results may be different when the **same** computation is performed twice.

- on the same machine
- on different machines

round-off errors or bug?

Software engineering problem

Q: how to verify program?

A: require the same bit-to-bit result (*reproducibility*)

Why is reproducibility not guaranteed?

Lack of specification in programming languages

```
float a, b, c, d, x;
x = a + b + c + d;
```

C does not specify the precision of intermediate calculations

Why is reproducibility not guaranteed?

Lack of specification in programming languages

```
real :: a, b, c, d, x;
x = a + b  + c + d;
```

FORTRAN does not specify the order of evaluation

Multithreaded programs

Floating-point addition/multiplication are non-associative

$+$

Non-deterministic scheduling

$=$

multithreaded reductions $(+/*)$ may yield different results

How to enforce numerical reproducibility?

solution 1 Require correct rounding

- provided by IEEE-754 compliant processors for arithmetic operations
- hard to obtain and expensive for compound expressions

How to enforce numerical reproducibility?

solution 2 Use specific algorithms

Example: reproducible sums (Demmel and Nguyen 2013), sum the following data in 6 decimal digits precision

1.23456×10^7		1234	5600	0000
$+9.87654 \times 10^1$		$+0000$	$+0098$	$+7654$
-1.00001×10^5		-0010	-0001	-0000
$+1.21215 \times 10^7$		$+1212$	$+1500$	$+0000$
-4.44444×10^2		-0000	-0444	-4440
$+3.33333 \times 10^5$		$+0033$	$+3333$	$+0000$
Σ		2479	10086	3210

We have $\Sigma \approx 2479 \times 10^4$ with no guarantee about accuracy

How to enforce numerical reproducibility?

solution 3 Serialize reductions Intel MKL CNR

- calls to Intel MKL occur in a single executable
- input and output arrays in function calls are properly aligned
- the number of computational threads used by the library does not change in the run

cost: run-time $+100\%$

Verified computing

Interval computations

- take round-off errors into account
- are subject to overestimation

Software engineering problem

Q: how to verify program?

A: compute an interval result
the result must

1. intersect the expected result
2. have a small enough width

Certified results

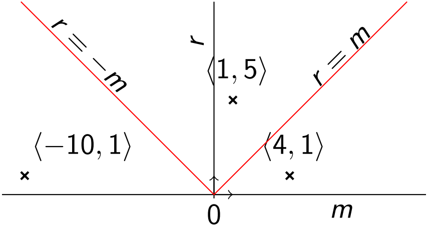
we compute $\langle \hat{m}, \hat{r} \rangle$
we know that

$$x \in \langle \hat{m}, \hat{r} \rangle$$

or

$$\mathbf{y} \subset \langle \hat{m}, \hat{r} \rangle$$

Is $x > 0$? or $\mathbf{y} > 0$?

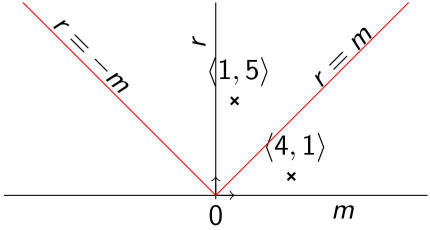


Different results on different runs

Compatible results

- run1 says "I don't know"
- run2 says "Yes!"

Why?



Different intermediate precisions

machine 2 uses more precision for intermediate calculation

try to add some iterative refinement steps

Different order of operations

SIMD identical alignment and vector length

multithread indeterminism

- reductions depend on scheduling
- list insertions depend on timing

Solution : log and replay

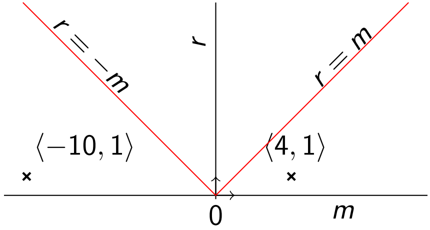
Different results on different runs

Incompatible results

run1 says "No!

run2 says "Yes!"

bug!



Inclusion property is not satisfied

Compilers do not respect rounding modes other than default
GCC Bug #34678 (2008)

```
void
interval_div (double *left, double *right,
              double x, double y) {
  #pragma STDC FENV_ACCESS ON
  fesetround (FE_DOWNWARD);
  *left = x / y;
  fesetround (FE_UPWARD);
  *right = x / y;
}
```

Inclusion property is not satisfied

math libraries do not respect rounding modes other than default
example (Rump 1999):

Input: $\mathbf{A} = [\underline{A}, \overline{A}], \mathbf{B} = [\underline{B}, \overline{B}]$

Output: $\mathbf{C} \supseteq \mathbf{A} \cdot \mathbf{B}$

$\langle M_{\mathbf{A}}, R_{\mathbf{A}} \rangle \leftarrow \text{InfsupToMidrad}(\mathbf{A})$

$\langle M_{\mathbf{B}}, R_{\mathbf{B}} \rangle \leftarrow \text{InfsupToMidrad}(\mathbf{B})$

$R_{\mathbf{C}} \leftarrow \mathrm{RU}(|M_{\mathbf{A}}| \cdot R_{\mathbf{B}} + R_{\mathbf{A}} \cdot (|M_{\mathbf{B}}| + R_{\mathbf{B}}))$

$\overline{C} \leftarrow \mathrm{RU}(M_{\mathbf{A}} \cdot M_{\mathbf{B}} + R_{\mathbf{C}})$

$\underline{C} \leftarrow \mathrm{RD}(M_{\mathbf{A}} \cdot M_{\mathbf{B}} - R_{\mathbf{C}})$

return $[\underline{C}, \overline{C}]$

Inclusion property is not satisfied

thread managers do not respect rounding modes
from OpenMP API Version 4.0 - RC 1 - November 2012:
"This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003. The following features are not supported:

- IEEE Arithmetic issues covered in Fortran 2003 Section 14
- ..."

Order of operation matters

Theorem (Rump 2012)

Let $A \in \mathbb{F}^{m\times k}$ and $B \in \mathbb{F}^{k\times n}$ with $2(k+2)\mathbf{u} \le 1$ be given, and let $C = \mathrm{RN}(A \times B)$ and $\Gamma = \mathrm{RN}(|A| \times |B|)$. Here C may be computed in any order, and we assume that Γ is computed in the same order. Then

$$|\mathrm{RN}(A \times B) - A \times B| \le \mathrm{RN}\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \frac{1}{2}\mathbf{u}^{-1}\eta\right)$$

Conclusion

Any good reason to require bit-to-bit identity with a who-knows-to-what-accuracy approximation?
Any real difficulty in implementing a compiler that respect the changes of rounding mode?