Numerical reproducibility in HPC: issues in floating-point arithmetic and in interval arithmetic

Nathalie Revol et Philippe Théveny
INRIA et ENS de Lyon
LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL) - ENS de Lyon
Université de Lyon

RAIM
November 20, 2013

No numerical reproducibility

(Diethelm 2012)

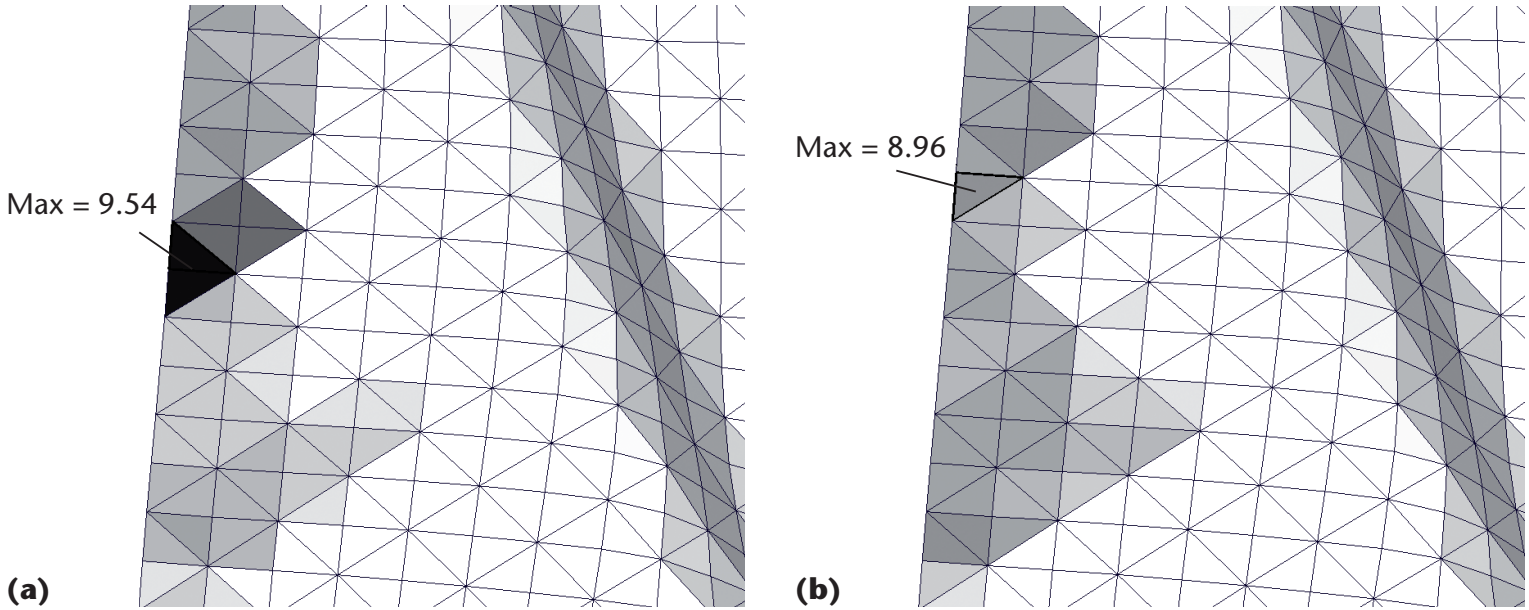


Figure 4. Location of the computed maxima of the sheet thickness change. (a) The simulation with one processor. (b) The second run of the simulation with four processors. The darker the element is colored, the larger the corresponding sheet-thickness change. Elements colored in white have a sheet thickness change of less than 8.5 percent.

No numerical reproducibility

(He and Ding 2001)

Table 1. Results of the summation in different natural orders with different methods in double precision

Order	Result
Longitude first	34.4147682189941 41
Reverse longitude first	32.302734375
Latitude first	0.67326545715332031
Reverse latitude first	0.734375
Longitude first SCS	0.3823695182800293
Longitude first DCS	0.3882288932800293
Latitude first SCS	0.37443733215332031
Latitude first DCS	0.32560920715332031

Agenda

Agenda

No numerical reproducibility: why?

Consider the following program, whatever the language

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

Two questions:

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

No numerical reproducibility: why?

Consider the following program, whatever the language

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

Two questions:

- What precision will be used for the intermediate results?
- In which order will the three additions be executed?
  Here we should remind that FP addition is not associative: consider $1 + 2^{100} - 2^{100}$.

No numerical reproducibility: why?

Consider the following program, whatever the language

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

Two questions:

- What precision will be used for the intermediate results?
- In which order will the three additions be executed?
  Here we should remind that FP addition is not associative: consider $1 + 2^{100} - 2^{100}$.

Fortran, C and Java have completely different answers.

No numerical reproducibility: change of precision

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

Two questions:

- What precision will be used for the intermediate results?
  - Bottom up precision: (here all float)
  - Use the maximum precision available which is no slower
  - Is the precision fixed by the language, or is the compiler free to choose?
- In which order will the three additions be executed?

No numerical reproducibility: change of order

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

Two questions:

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

Example of the summation

The floating-point addition is not associative.

Problem of numerical reproducibility with multicore or HPC computations:
as the summation $\sum_{i=1}^{n} a_i$ on a multicore is not done in a deterministic order,

- depending on the number of threads,
- depending on the state of the execution environment (various loads imply various schedulings),
- depending on the execution order,

the result varies from on execution to the other.

Example of the summation: HPC issues

Problems:

- **order of execution:** depending on the number of threads, on the state of the execution environment;
- **computing precision:** on heterogeneous targets, various precisions for the registers...

"Solution" for the summation

(He and Ding 2001, Bailey 2012)

To increase the accuracy on the result, whatever the execution: **increase the computing precision.**

- **(He and Ding 2001)**: self-compensated summation and double-double arithmetic;
- **(Bailey 2012)**: double-double arithmetic.

Accuracy, stability are improved...

"Solution" for the summation

(He and Ding 2001, Bailey 2012)

To increase the accuracy on the result, whatever the execution:
increase the computing precision.

- **(He and Ding 2001)**: self-compensated summation and double-double arithmetic;
- **(Bailey 2012)**: double-double arithmetic.

Accuracy, stability are improved...
Reproducibility is still not guaranteed.

Solution for the summation

(Rump, Ogita and Oishi 2008), (Nguyen and Demmel 2013)

Rump, Ogita and Oishi 2008:

Provide the correct rounding of the exact result: reproducible result.

Solution for the summation

(Rump, Ogita and Oishi 2008), (Nguyen and Demmel 2013)

Rump, Ogita and Oishi 2008:
Provide the correct rounding of the exact result: reproducible result.

Nguyen and Demmel 2013:
The result is reproducible: bit-to-bit reproducibility whatever the execution.

The accuracy is variable. The result is not necessarily the correctly rounded sum.

Tradeoff between the accuracy of the result and the execution time.

Reproducibility in the MKL of Intel: CNR

MKL: Math Kernel Library, includes the BLAS.
On multicores, the MKL does not produce reproducible results.

.

Thus Intel got bug reports and requests for reproducibility.

CNR: conditional numerical reproducibility by MKL 11.0:
if the processors, the OS, the number of threads and the memory alignment are preserved, then MKL guarantees numerical reproducibility.
Non-efficient, non-user-friendly, non-portable solution.

Numerical reproducibility?

Definition?

- Numerical reproducibility = best possible result = correct rounding of the exact result?
- Numerical reproducibility = getting the same string of bits whatever the run?

New light on numerical reproducibility:

- reproducibility and correct rounding are separate notions
- a hierarchy of reproducibility levels exists: accuracy vs execution time.

Cf. Dongarra: get numerical quality rather than bit-to-bit, use the least needed computing precision.

Agenda

A brief introduction

Interval arithmetic: replace numbers by intervals and compute.

Fundamental theorem of interval arithmetic:
(or "Thou shalt not lie"):
the exact result (number or set) is contained in the computed interval.

No result is lost, the computed interval is guaranteed to contain every possible result.

A brief introduction

Interval arithmetic: replace numbers by intervals and compute.
Initially: introduced to take into account roundoff errors (Moore 1966)
and also uncertainties (on the physical data...).
Later: computations "in the large", computations with sets.

Interval analysis: develop algorithms for **reliable (or verified, or guaranteed, or certified) computing**,
that are suited for interval arithmetic,
i.e. different from the algorithms from classical numerical analysis.

Definitions: operations

$\mathbf{x} \diamond \mathbf{y} = \mathbf{Hull}\{x \diamond y \ : \ x \in \mathbf{x}, y \in \mathbf{y}\}$

Arithmetic and algebraic operations: use the monotonicity

$$
\begin{array}{lcl}
[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] & = & [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
[\underline{x}, \overline{x}] - [\underline{y}, \overline{y}] & = & [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] & = & [\min(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y}), \max(\text{ibid.})] \\
[\underline{x}, \overline{x}]^2 & = & [\min(\underline{x}^2, \overline{x}^2), \max(\underline{x}^2, \overline{x}^2)] \ \text{ if } 0 \notin [\underline{x}, \overline{x}] \\
 & & [0, \max(\underline{x}^2, \overline{x}^2)] \ \text{ otherwise}
\end{array}
$$

Interval arithmetic: implementation using floating-point arithmetic

Implementation using floating-point arithmetic:
use directed rounding modes (cf. IEEE 754 standard)

$$\sqrt{[2,3]} = [\nabla\sqrt{2}, \triangle\sqrt{3}]$$

Advantage: every result is guaranteed, in the sense that the exact, unknown result, belongs to t he computed interval result.

Operations

Algebraic properties: associativity, commutativity hold, some are lost:

- subtraction is not the inverse of addition, in particular $x - x \neq [0]$
- division is not the inverse of multiplication
- squaring is tighter than multiplication by oneself
- multiplication is only sub-distributive wrt addition
- with floating-point implementation, operations are not associative either

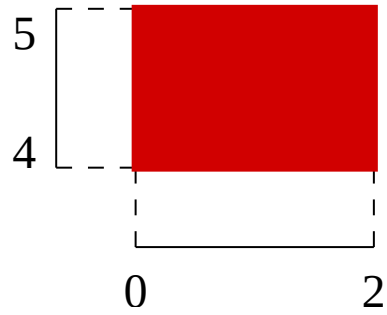
Definitions: intervals, vectors, matrices

Objects:

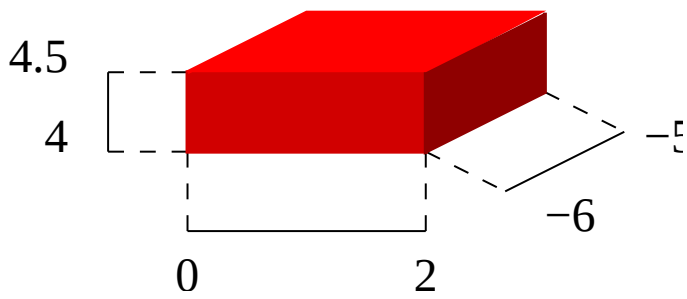
- intervals of real numbers = closed connected sets of $\mathbb{R}$
  - interval for π: $[3.14159, 3.14160]$
  - data d measured with an absolute error less than $\pm\varepsilon$: $[d-\varepsilon, d+\varepsilon]$
- interval vector: components = intervals; also called *box*

$[0\ ;\ 2]$

$\begin{pmatrix}[0\ ;\ 2]\\ [4\ ;\ 5]\end{pmatrix}$



$\begin{pmatrix}[0;2]\\ [4\ ;\ 4.5]\\ [-6\ ;\ -5]\end{pmatrix}$



- interval matrix: components = intervals.

Inclusion property

Interval arithmetic: replace numbers by intervals and compute.

Fundamental theorem of interval arithmetic:
aka "Inclusion property"
(or "Thou shalt not lie"):
the exact result (number or set) is contained in the computed interval.

No result is lost, the computed interval is guaranteed to contain every possible result.

Optimistic point of view

Whatever the result, the inclusion property is satisfied:
interval arithmetic is not perturbed by numerical reproducibility issues.

Optimistic point of view

Whatever the result, the inclusion property is satisfied:
interval arithmetic is not perturbed by numerical reproducibility issues.

Even better, as each different result encloses the exact result, a more accurate result can be obtained by intersecting all computed results:
interval arithmetic benefits from the lack of numerical reproducibility.

Pragmatic point of view

Reproducibility is important for

- debugging purposes,
- testing purposes.

Pragmatic point of view

Reproducibility is important for

- debugging purposes,
- testing purposes.

What may hinder reproducibility?

- computing precision,
- order of the operations, expressions,
- rounding modes.

Influence of the computing precision

Influence on an interval computation: in practice,

- use the midpoint-radius representation for thin intervals: the radius accounts for roundoff errors,
- use iterative refinement to reduce the width,
- use higher precision for critical intermediate computations (residual) to hide the effect of the computing precision,

and get $\mathrm{w}(\hat{\mathbf{x}}) - \mathrm{w}(\mathbf{x}) \simeq 2^{-p}|\mathbf{x}|$, i.e. the best possible result.

Examples: linear systems solving, Newton iteration.

Influence of the expression

Using floating-point arithmetic: the problem comes from the non-associativity of the operations

$$(a_1 + a_2) + (a_3 + a_4) \neq ((a_1 + a_2) + a_3) + a_4.$$

Using interval arithmetic: the expression influences the result because operations are neither distributive nor reciprocal ($+$ of $-$, $\times$ of $/$).

Using interval arithmetic implemented with floating-point arithmetic: because operations are neither distributive nor reciprocal ($+$ of $-$, $\times$ of $/$) nor associative: problems cumulate.

Influence of the expression: example

$$[1,1] + [2^{100}, 2^{100}] - [2^{100}, 2^{100}]?$$

With these parentheses:

$$([1,1]+[2^{100}, 2^{100}])-[2^{100}, 2^{100}] = [2^{100}, \mathrm{succ}(2^{100})]-[2^{100}, 2^{100}] = [0, \mathrm{ulp}(2^{100})].$$

With those parentheses:

$$[1,1] + ([2^{100}, 2^{100}] - [2^{100}, 2^{100}]) = [1,1] + [0,0] = [1,1].$$

Both include the results, one is more accurate than the other...

Moral lesson: interval results are always guaranteed to include the exact result, whatever the chosen expression. However their accuracy strongly depends on the chosen expression, on the order of operations.

More on the influence of the order of the operations

Beware "hidden" assumptions on the order of the operations.

Example: interval matrix product.
In order to save 1 or 2 calls to `gemm` (BLAS matrix product), Rump's algorithm (2012) assumes that $\mathbf{A}_m \cdot \mathbf{B}_m$ and $|\mathbf{A}_m| \cdot |\mathbf{B}_m|$ are computed in the same order.

BLAS do not guarantee anything on the order of operations nor on the reproducibility of this order from one product to the next.

Moral lesson: interval results could depend on the order of operations,
interval results could be wrong if they relied too much on the order of operations.

Circumventing these difficulties

(Théveny 2013)

To ensure that $\mathbf{A}_m \cdot \mathbf{B}_m$ and $|\mathbf{A}_m| \cdot |\mathbf{B}_m|$ are computed in the same order:

- do not use `gemm`;
- compute simultaneously $\mathbf{A}_m \cdot \mathbf{B}_m$ and $|\mathbf{A}_m| \cdot |\mathbf{B}_m|$: reduce the memory transfers;
- to get performances:
  - optimize the use of the cache (L1);
  - manually vectorize with SSE2 instructions.

Influence of the order of operations

Other important operations in interval arithmetic may also be sensitive to the order:

Influence of the order of operations

Other important operations in interval arithmetic may also be sensitive to the order:

bisection, working list of intervals to process later
in global optimization.

Rounding modes

Implementation using floating-point arithmetic:
use directed rounding modes (cf. IEEE 754 standard)

$$\sqrt{[2,3]} = [\mathrm{RD}(\sqrt{2}), \mathrm{RU}(\sqrt{3})].$$

The implementation of interval arithmetic using floating-point arithmetic is based on setting properly the rounding modes.

Rounding modes

Are rounding modes respected?

- by the compiler?
- by the libraries?
  - Undocumented for the classical BLAS,
  - experimentally: no **(Lauter and Ménissier-Morain 2012)**,
  - for fast methods such as Strassen's matrix multiplication: no,
  - for specific libraries such as xBLAS (extended BLAS that are based on error free transforms) **(Li, Demmel, Bailey et al. 2008)**: impossible;
- by the execution environment? Undocumented, or explicitely documented as non supported (for OpenMP).

Solution

(Revol, Makino and Berz 2003, Rump 2012)

Bound each rounding error by a quantity that is computable using floating-point arithmetic:

- multiply by $(1 + 2u)$ in rounding-to-nearest;
- multiply by $(1 + 4u)$ in directed rounding modes or in unknown rounding modes;

when adding or multiplying nonnegative quantities.

Limits:

this does not work with fast algorithms, e.g. for fast matrix multiplications (different numbers of operations, varying monotony).

HPC issues

Specific issues:

- order of operations: no specified order in parallel evaluations
- computing precision: problem on distributed, heterogeneous environments (not – yet – our problem)
- rounding modes:
  - is the rounding mode local to each thread or global?
  - is the rounding mode respected by the thread or set to a default value?
  - are rounding modes saved and restored at context switches during a multithreaded computation?

Agenda

Conclusion

New light on numerical reproducibility:

- reproducibility and correct rounding are separate notions
- a hierarchy of reproducibility levels exists: accuracy vs execution time.

Interval equivalent of the numerical reproducibility?

- the inclusion property (the guarantee that the computed result contains the exact result) must be preserved,
- preserved inclusion property and correct rounding of the exact result are separate notions,
- to guarantee the inclusion property, brute-force bounds on roundoffs errors can be used,
- a hierarchy of guarantee levels exists: accuracy vs execution time.

Conclusion: intervals and reproducibility

Interval computations are:

- apparently, safe against the lack of reproducibility;
- however, sensitive to the respect of the rounding mode and to floating-point reproducibility;
- adopted methodology:
  - firstly, develop interval algorithms that are based on well-established numerical bricks;
  - second, convince developers and vendors of these bricks to clearly specify their behavior (rounding modes)
  - if the second step fails,
    - either replicate the work done for the optimization of the considered numerical bricks;
    - or use brute-force methods to compute an upper bound of the roundoff errors.

References on interval arithmetic

- R. Moore: *Interval Analysis*, Prentice Hall, Englewood Cliffs, 1966.
- A. Neumaier: *Interval methods for systems of equations*, CUP, 1990.
- R. Moore, R.B. Kearfott, M.J. Cloud: *Introduction to interval analysis*, SIAM, 2009.
- W. Tucker: *Validated Numerics: A Short Introduction to Rigorous Computations*, Princeton University Press, 2011.
- S.M. Rump: *Computer-assisted proofs and Self-Validating Methods*, pp. 195-240. Handbook on Accuracy and Reliability in Scientific Computation (B. Einarsson ed.), SIAM, 2005.
- S.M. Rump: *Verification methods: Rigorous results using floating-point arithmetic*, Acta Numerica, vol. 19, pp. 287-449, 2010.

References on interval arithmetic

- J. Rohn: *A Handbook of Results on Interval Linear Problems*, http://www.cs.cas.cz/rohn/handbook 2006.
- E. Hansen and W. Walster: *Global optimization using interval analysis*, MIT Press, 2004.
- R.B. Kearfott: *Rigorous global search: continuous problems*, Kluwer, 1996.
- V. Kreinovich, A. Lakeyev, J. Rohn, P. Kahl: *Computational Complexity and Feasibility of Data Processing and Interval Computations*, Dordrecht, 1997.
- L.H. Figueiredo, J. Stolfi: *Affine arithmetic* http://www.ic.unicamp.br/~stolfi/EXPORT/projects/affine-arith/.
- *Taylor models arith.:* M. Berz and K. Makino, N. Nedialkov, M. Neher.

Further readings

Numerical Reproducibility and Parallel Computations: Issues for Interval Algorithms

Nathalie Revol and Philippe Théveny

July 2013

http://hal.inria.fr/hal-00845839/

Repeatability and reproducibility

Repeatability:
getting the same result (the same bits) from run to run, on the same machine.

Reproducibility:
getting the same result (the same bits) from run to run, whatever the machine.

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.

Fortran's philosophy (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 (2)

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)

The C philosophy

The "C11" standard:
International Standard ISO/IEC ISO/IEC 9899:2011.

- 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

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 of evaluation**
- Strangely, **intermediate 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` for the same price, you won't complain.

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 became much less accurate!
- Big drawbacks
  - 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)
  - But... storing a `float` variable in 64 or 80 bits of memory instead of 32 is usually slower, therefore (C philosophy) it should be avoided.
  - 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.

  The sort bug explained (because `double` promoted to 80 bits)

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 sold 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. We can't allow the sort bug.

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.

Solution for the summation

(Rump, Ogita and Oishi 2008)

Provide the correct rounding of the exact result: reproducible result.

Principle:

- align mantissas (as if in fixed-point arithmetic);
- split mantissas into vertical slices, of width small enough to allow exact summation of a slice into a floating-point number;
- add the leftmost slice;
- as long as it is not possible to round, add one more slice from the right.

Solutions for the summation

(Nguyen and Demmel 2013)

- Fix in advance the number of slices;
- sum the slices.

The result is reproducible: bit-to-bit reproducibility whatever the execution.

The accuracy is variable and is determined by the number of slices. The result is not necessarily the correctly rounded sum.

Tradeoff between the accuracy of the result and the execution time.

Reproducibility in the MKL of Intel: CNR

(From Intel Developer Zone web page)

Intel®MKL 11.0 introduces a feature called Conditional Numerical Reproducibility (CNR) which provides functions for obtaining reproducible floating-point results when calling library functions from their application. When using these new features, Intel MKL functions are designed to return the same floating-point results from run-to-run, subject to the following limitations:

- *calls to Intel®MKL occur in a single executable*
- *input and output arrays in function calls must be aligned on 16, 32, or 64 byte boundaries on systems with SSE / AVX1 / AVX2 instructions support (resp.)*
- *the number of computational threads used by the library remains constant throughout the run*

Reproducibility in the MKL of Intel: CNR

(From Intel Developer Zone web page)

Intel®MKL 11.0 introduces a feature called Conditional Numerical Reproducibility (CNR) which provides functions for obtaining reproducible floating-point results when calling library functions from their application. When using these new features, Intel MKL functions are designed to return the same floating-point results from run-to-run, subject to the following limitations:

- *calls to Intel®MKL occur in a single executable*
- *input and output arrays in function calls must be aligned on 16, 32, or 64 byte boundaries on systems with SSE / AVX1 / AVX2 instructions support (resp.)*
- *the number of computational threads used by the library remains constant throughout the run*

Not what I call convenient.