High precision numerical accuracy in Physics research

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

ACAT 2005

ECOLE NORMALE SUPERIEURE DE LYON

Introduction: double-precision not enough ?

Double precision ?

50 years of computing heritage lead to rather silly names...

- `float`, `REAL` or *single precision* means
  - a floating-point format
  - with 24 bits of mantissa,
    (and 7 bits of exponent, and a sign bit, total 4 bytes)
  - or roughly 7 significant decimal digits
  - Not enough to compute $\boxed{\sum_{i=0}^{10.000.000} x_i}$

Double precision ?

50 years of computing heritage lead to rather silly names...

- `float`, `REAL` or *single precision* means
  - a floating-point format
  - with 24 bits of mantissa,
    (and 7 bits of exponent, and a sign bit, total 4 bytes)
  - or roughly 7 significant decimal digits
  - Not enough to compute $\boxed{\sum_{i=0}^{10.000.000} x_i}$
- `double`, `REAL(8)` or *double precision* means roughly 15 digits

Double precision ?

50 years of computing heritage lead to rather silly names...

- `float`, `REAL` or *single precision* means
  - a floating-point format
  - with 24 bits of mantissa,
    (and 7 bits of exponent, and a sign bit, total 4 bytes)
  - or roughly 7 significant decimal digits
  - Not enough to compute $\boxed{\sum_{i=0}^{10.000.000} x_i}$
- `double`, `REAL(8)` or *double precision* means roughly 15 digits
- `quad`, `REAL*16`, `REAL(16)` or *quadruple precision* will mean roughly 30 digits.

Double precision ?

50 years of computing heritage lead to rather silly names...

- `float`, `REAL` or *single precision* means
  - a floating-point format
  - with 24 bits of mantissa,
    (and 7 bits of exponent, and a sign bit, total 4 bytes)
  - or roughly 7 significant decimal digits
  - Not enough to compute $\boxed{\sum_{i=0}^{10.000.000} x_i}$
- `double`, `REAL(8)` or *double precision* means roughly 15 digits
- `quad`, `REAL*16`, `REAL(16)` or *quadruple precision* will mean roughly 30 digits.

There is a standard (IEEE-754, IEC 60559) that defines all this (number representations, operation behaviour, and more).

Floating point hardware in 2005

- All recent processors have double-precision hardware operators for $+$, $-$, $\times$
- They all do their best to implement the IEEE-754 standard
- Peak throughput of 2 to 4 double-precision FP op per cycle.
- Power/PowerPC and Itanium hardware is based on fused multiply-and-add: $a \times b + c$ in one instruction
    - more efficient (two operations in one instruction)
    - more accurate (only one rounding)

Less relevant to this talk:

- IA32 (Intel/AMD) and IA64 (HP/Intel) hardware is actually double-extended precision (18 digits)
- Division is always much slower than the others, and is sometimes implemented in software.

(More details)

Double-precision not enough ?

From the web page of a quad-like library[1]:

This code has been used for:

- *Studies of Feigenbaum scaling in discrete dynamical systems.*
- *Two-loop integral for radiative corrections in muon decay.*
- *Number theory research, e.g. in LLL algorithms.*
- *Coefficient generation and checking of double-precision algorithms for transcendental functions.*
- *Testing sensitivity to rounding errors of existing double-precision code.*
- *Linear programming problems which arise in the study of linear codes.*

[1] K. Briggs' doubledouble

Double-precision not enough ?

From the web page of a quad-like library[1]:
This code has been used for:

- *Studies of Feigenbaum scaling in discrete dynamical systems.*
- *Two-loop integral for radiative corrections in muon decay.*
- *Number theory research, e.g. in LLL algorithms.*
- *Coefficient generation and checking of double-precision algorithms for transcendental functions.*
- *Testing sensitivity to rounding errors of existing double-precision code.*
- *Linear programming problems which arise in the study of linear codes.*

Quadruple precision considered in the revision of the IEEE 754 Standard for Floating-Point Arithmetic

[1]K. Briggs' doubledouble

Purpose of this talk

- What are the needs for quadruple (and more) precision ?
- Can we hope for quadruple precision in hardware soon ?
- What are the software alternatives, and their cost ?
- Can algorithm changes reduce the need for quadruple precision ?

The Arénaire project @ École Normale Supérieure de Lyon

- Computer Arithmetic at large:
    - designing new operators
    - cleverly using existing operators
    - performance and accuracy
- Current research domains:
    - Number representation and algorithms
    - Operations in fixed and floating point, cryptography, elementary functions
    - Hardware (ASIC and FPGA), hardware-oriented algorithms
    - Software: multiple precision, intervals, elementary functions
    - Machine-checked proofs of all the above

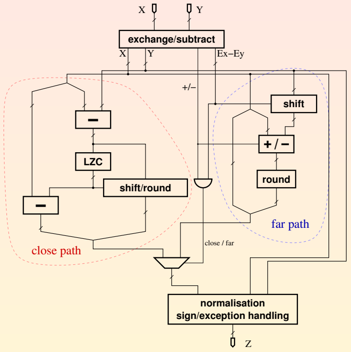
The Arénaire project @ École Normale Supérieure de Lyon

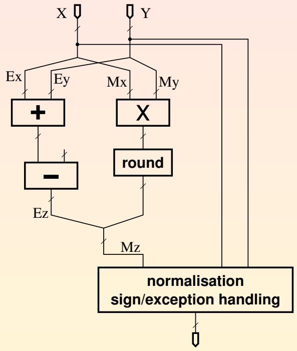
- Computer Arithmetic at large:
  - designing new operators
  - cleverly using existing operators
  - performance and accuracy
- Current research domains:
  - Number representation and algorithms
  - Operations in fixed and floating point, cryptography, elementary functions
  - Hardware (ASIC and FPGA), hardware-oriented algorithms
  - Software: multiple precision, intervals, elementary functions
  - Machine-checked proofs of all the above

The speaker is not a physicist...

Hardware solutions ?

I have never built a workstation FPU

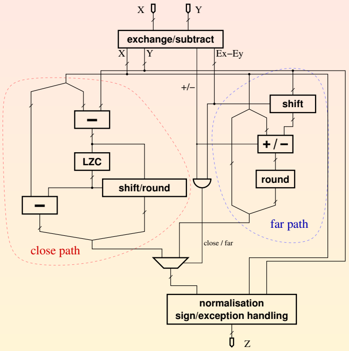


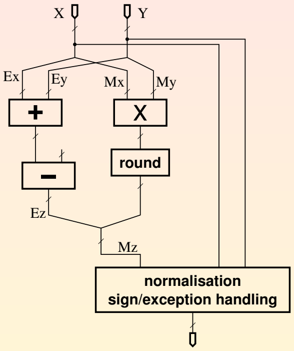


Building a quad-precision FPU:

- Main blocks are of size n^2 and $n \log(n)$

I have never built a workstation FPU

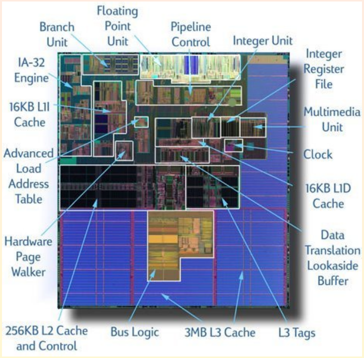


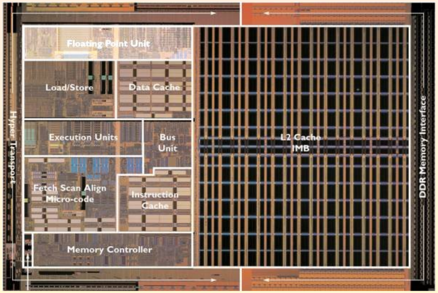


Building a quad-precision FPU:

- Main blocks are of size n^2 and $n \log(n)$
- Expect more than doubling the area of current FPU
- Expect less than doubling the pipeline depth (3-6 levels currently)

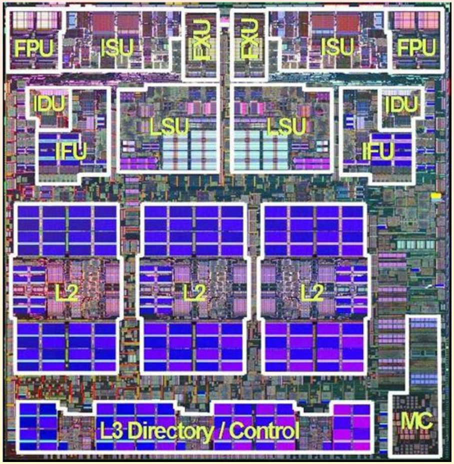
Raw silicon cost



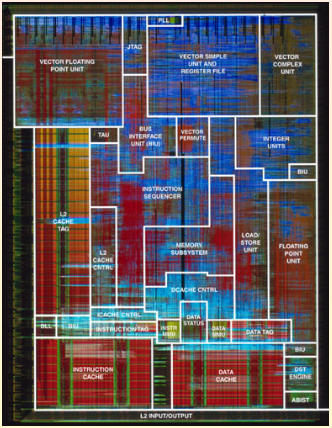


Itanium 2 Opteron

Where the mass market is



Power 5



PowerPC 7400 (aka G4)

The economics of the market

Does the market (you) justify hardware quad ?

An older question:

- Does the market justify hardware division ?
- In Itanium,
  - FP division has been delegated to software,
  - with specific instructions (`frcpa`) to accelerate it.
  - It makes sense in a global silicon budget.
- Let us study the software quad alternatives
  - instructions to accelerate it ?

The economics of your code

The most probable approach:
SSE2-like FPU with quad capabilities.

- FPadd: each cycle, 2 // DP or 1 QP.
- FPmul/FMA:
  - 2 // DP each cycle, or 1 QP *each other cycle*[2]
  - more silicon: 4 // DP each cycle, or 1 QP each cycle

Conclusion:

- Quad will mean a 2x-4x reduction in peak performance compared to double.
- Even with hardware quad support, it will make sense to keep using double-precision as often as possible
- (at least it will make sense for a while)

[2]Akkaș and Schulte, DSD'03

Software solutions

Double-double versus Quad

Two options for reaching about 30 significant digits:

- The double-double approach:
  represent 1.2345678 as 1.234 + 5.678E-4
    - Some Fortran `REAL*16` implementations use this
    - $\oplus$ Mixing double and double-double easy
    - $\ominus$ Exponent range is that of double-precision
- quadruple precision is a 128-bit format (e15,m112) [3]
    - Supported at least by Sun, HP, and Intel (on Itanium)
    - Revision of IEEE-754 should include it
    - $\oplus$ A real floating-point format
    - $\oplus$ Larger exponent
    - $\ominus$ Conversions to/from double expensive, mixing double/quad expensive

[3] IEEE Standard for Shared Data Formats 1596.5-1993

Performance cost of double-double arithmetic

- Double-double operations [4]
  - Addition: 8 FP add/sub, and one comparison
  - Multiplication: 7 FMA, or (without an FMA) 9+15 mul+add

[4] Sterbenz (74), Dekker(1971), Knuth (1973), Linnainmaa and Seppo (1981), Goldberg(1991). See also a Fortran90 implementation by Miller.

[5] Cornea et al (2002), Markstein (2003)

Performance cost of double-double arithmetic

- Double-double operations [4]
  - Addition: 8 FP add/sub, and one comparison
  - Multiplication: 7 FMA, or (without an FMA) 9+15 mul+add
  - dependent operations, poor pipeline usage

[4] Sterbenz (74), Dekker(1971), Knuth (1973), Linnainmaa and Seppo (1981), Goldberg(1991). See also a Fortran90 implementation by Miller.

[5] Cornea et al (2002), Markstein (2003)

Performance cost of double-double arithmetic

- Double-double operations [4]
  - Addition: 8 FP add/sub, and one comparison
  - Multiplication: 7 FMA, or (without an FMA) 9+15 mul+add
  - dependent operations, poor pipeline usage
  - In many cases (if you know what you do) you can save a lot of operations
    - Kahan summation algorithm
    - double-doubles for elementary functions
    - ... (illustrations of “Mixing double and double-double easy”)

[4] Sterbenz (74), Dekker(1971), Knuth (1973), Linnainmaa and Seppo (1981), Goldberg(1991). See also a Fortran90 implementation by Miller.

[5] Cornea et al (2002), Markstein (2003)

Performance cost of double-double arithmetic

- Double-double operations [4]
  - Addition: 8 FP add/sub, and one comparison
  - Multiplication: 7 FMA, or (without an FMA) 9+15 mul+add
  - dependent operations, poor pipeline usage
  - In many cases (if you know what you do) you can save a lot of operations
    - Kahan summation algorithm
    - double-doubles for elementary functions
    - ... (illustrations of "Mixing double and double-double easy")
- Division, square root, elementary functions: in a factor 2-10 [5]

Conclusion: With an FMA, double-double slowdown less than 10

[4] Sterbenz (74), Dekker(1971), Knuth (1973), Linnainmaa and Seppo (1981), Goldberg(1991). See also a Fortran90 implementation by Miller.

[5] Cornea et al (2002), Markstein (2003)

Performance cost of quad

Operations:

- Unpack numbers
- Integer operations on 112-bit mantissa and 15-bit exponent
- Normalisation of the result
- Pack again

Performance cost of quad

Operations:

- Unpack numbers
- Integer operations on 112-bit mantissa and 15-bit exponent
- Normalisation of the result
- Pack again

Possible optimisations:

- Use of 128-bit wide multimedia extensions
- New specific processor instructions to help round to 112 bits ?

Performance cost of quad

Operations:

- Unpack numbers
- Integer operations on 112-bit mantissa and 15-bit exponent
- Normalisation of the result
- Pack again

Possible optimisations:

- Use of 128-bit wide multimedia extensions
- New specific processor instructions to help round to 112 bits ?

Remarks

- Software quad mostly relies on integer arithmetic
- Quad elementary functions in Intel and HP libraries use double-double-extended arithmetic internally.

Important general remark

The price of a function call is several tens of cycles these days !

- Using a library will be slow
- `REAL*16` handled by the compiler will be much faster
  - Inlined operations, no function call
  - Possible global optimisation of register and pipeline usage
  - For quad, most packing/unpacking disappears (internal format)
- Alternative: C/C++ macros [6]

[6] See for example `crlibm` at `http://lipforge.ens-lyon.fr/projects/crlibm/`

Better than Quad

If you really have to add more than 10^{30} numbers...

- Quad-double[7] and floating-point expansions[8]
- Integer-based multiple precision: GMP and MPFR (best for arbitrary precision)
- Software Carry Save[9] (best for fixed precision)

[7] Hida, Li, Bailey (2001)

[8] Priest (91), Daumas, Moreau-Fineau(98-99)

[9] Brent (78), Defour, de Dinechin (2002)

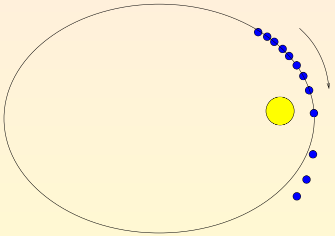
Mathematical solutions ?

Rounding 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

Rounding 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.



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

The following only deals with roundoff errors.

How do you know that double is not enough ?

Kahan: "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?"

How do you know that double is not enough ?

Kahan: “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?”

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.

How do you know that double is not enough ?

Kahan: "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?"

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.
- Repeat the computation with same precision but different/variable (IEEE-754) rounding modes, and compare the results.

How do you know that double is not enough ?

Kahan: "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?"

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.
- Repeat the computation with same precision but different/variable (IEEE-754) rounding modes, and compare the results.
- Repeat the computation a few times with same precision but slightly different inputs, and compare the results.

How do you know that double is not enough ?

Kahan: "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?"

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.
- Repeat the computation with same precision but different/variable (IEEE-754) rounding modes, and compare the results.
- Repeat the computation a few times with same precision but slightly different inputs, and compare the results.
- Use interval arithmetic

How do you know that double is not enough ?

Kahan: "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?"

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.
- Repeat the computation with same precision but different/variable (IEEE-754) rounding modes, and compare the results.
- Repeat the computation a few times with same precision but slightly different inputs, and compare the results.
- Use interval arithmetic

Of these schemes, only interval arithmetic provides a guarantee
(often " Your result is in $[-\infty, +\infty]$, guaranteed")

Analysis of rounding errors propagation

- 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 ?
  - See Higham. Several attempts to automate them (see Langlois' habilitation thesis @ ENS-Lyon)

Analysis of rounding errors propagation

- 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 ?
  - See Higham. Several attempts to automate them (see Langlois' habilitation thesis @ ENS-Lyon)
- Measure of the behaviour of the problem: Conditioning:

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

  - The greater *Cond*, the more your problem can be sensitive to rounding (ill-conditionned).
  - $\textit{forward error} < \textit{Cond} \times \textit{backward error}$
  - Possibly pessimistic bound

Solutions

- Analyse the algorithm and localise the procedures where precision is lost
  - Inversion of an ill-conditionned matrix[10]
  - Polynomial roots close one to another [11]
  - Flat triangles [12]
  - ...

[10] Hasegawa (2003)

[11] Kahan, Langlois (2002)

[12] Shewshuck (1997), Kahan

Solutions

- Analyse the algorithm and localise the procedures where precision is lost
  - Inversion of an ill-conditionned matrix[10]
  - Polynomial roots close one to another [11]
  - Flat triangles [12]
  - ...
  - Was that part of your training ?

[10] Hasegawa (2003)

[11] Kahan, Langlois (2002)

[12] Shewshuck (1997), Kahan

Solutions

- Analyse the algorithm and localise the procedures where precision is lost
  - Inversion of an ill-conditionned matrix[10]
  - Polynomial roots close one to another [11]
  - Flat triangles [12]
  - ...
  - Was that part of your training ?
- Improve this procedure
  - Find another algorithm
  - Keep the algorithm, but change all the `REAL*8` to `REAL*16`
  - Change only some

[10] Hasegawa (2003)

[11] Kahan, Langlois (2002)

[12] Shewshuck (1997), Kahan

Solutions

- Analyse the algorithm and localise the procedures where precision is lost
  - Inversion of an ill-conditionned matrix[10]
  - Polynomial roots close one to another [11]
  - Flat triangles [12]
  - ...
  - Was that part of your training ?
- Improve this procedure
  - Find another algorithm
  - Keep the algorithm, but change all the `REAL*8` to `REAL*16`
  - Change only some

Hasegawa compares the performance of the two first options on a range of systems.

[10] Hasegawa (2003)

[11] Kahan, Langlois (2002)

[12] Shewshuck (1997), Kahan

Conclusion

Near future

- If you add together more than 10^{15} numbers of similar magnitude, then you do need quad precision.
- Don't expect 100% hardware quad too soon (especially for Grid@home-like computing)
- Software quad or double-double can be reasonably fast if managed by the compiler, or by yourself...
  - We would love to collaborate on that.
- There is a lot of science to do in mathematic and algorithmic approaches.
  - We would love to collaborate on that, too.

Any questions ?

(I have asked all mine already)

Floating point hardware in 2005

- IA32 instruction set (AMD and Intel x86 processors)
  - single, double (e11,m53) and double-extended (e15,m64)
  - one FP adder, one FP mult
  - IEEE-754 compliance difficult (and cornercases slow on Intel)
  - SSE2 adds two IEEE-754 compliant, double-precision FPUs
- Power instruction set (Power/PowerPC processors)
  - single and double precision only
  - one or two FMA *(Fused Multiply and Add)*
  - IEEE-754 compliance easy (but downgrade FMA)
- SPARC instruction set (processors by Sun, Fujitsu, ...)
  - single and double precision only
  - up to two adders and two multipliers (Fujitsu SPARC64 V)
  - IEEE-754 compliance easy
- IA64 instruction set
  - single, double and double-extended (e=17, m=64)
  - Two FMA
  - intruction-wise precision and rounding control
  - IEEE-754 compliance easy

Some recent processors

Design	Cycle Time (ns)	Integer units			FP units			
		Nb	$a \pm b$ L/T	$a \times b$ L/T	Nb	$a \pm b$ L/T	$a \times b$ L/T	$a \div b$ L/T
Alpha 21264	1.6	4	1/1	7/1	2	4/1	4/1	15/12
Athlon K6-III	0.71	6	1/1	1/1	3	3/1	3/2	20/17
Pentium III	1.0	3	1/1	4/1	1	3/1	5/2	32/32
Pentium IV	0.4	3	.5/.5	14/3	2	5/1	7/2	38/38
Itanium 2	1.25	4	1/1	18/1	4	4/1	4/1	soft
PowerPC 750	2.5	2	1/1	2-5/1	1	3/1	4/2	31/31
Sun UltraSparc III	0.95	4	3/1	4/1	2	4/1	4/1	24/17

L : *latency*, T : *throughput*, in cycles.

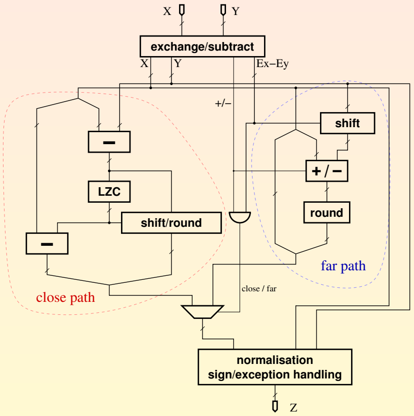
The IEEE-754 standard for floating-point arithmetic

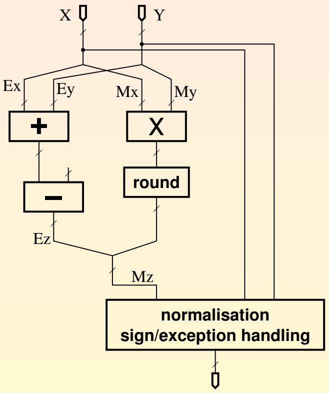
- There exists a standard for FP, and it is a good one
- It enables portability and provability of FP programs
- It requires cooperation of processor, OS, language, compiler, ...
- Standard compliance conflicts with performance, and is probably disabled by default on your system

One drawback of the standard:

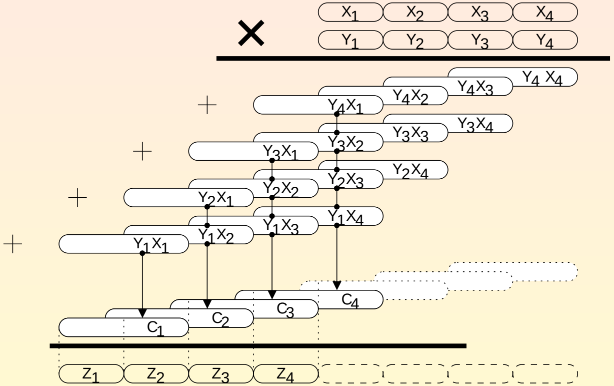
- In the 70s, when people ran the same program on different machines, they got widely different results.
  - They had to think about it and find what was wrong.
- Now they get the same result, and therefore trust it.
  - We (computer scientists) have to educate them...

Architecture of FP adder and multiplier





Software Carry-Save



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

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

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")

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).

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