# Validation for scientific computations
# Multiple precision arithmetic

## Cours de recherche master informatique

Nathalie Revol

Nathalie.Revol@ens-lyon.fr

24 November 2006

# References for today's lecture

- J.-C. Bajard et N. Revol: *Arithmétique multi-précision*, coordonné par M. Daumas et J.-M. Muller, Masson, 1998.
- R.P. Brent: *The Complexity of Multiple-Precision Arithmetic*, The Complexity of Computation Problem Solving, ed. by R.S. Anderssen and R.P. Brent, U. of Queensland Press, 1976. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb032.pdf
- R.P. Brent: *Fast Multiple-Precision Evaluation of Elementary Functions*, Journal of the ACM, vol 23, no 2, pp 242-251, 1976. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb034
pdf

- L. Imbert: *Arithmétique multi-précision*, in *Calcul et arithmétique des ordinateurs*, édité par J.-C. Bajard et J.-M. Muller, Traité IC2, série Informatique et systèmes d'information, Hermès, 2004.
- V. Kreinovich and S.M. Rump: *Towards Optimal Use of Multi-Precision Arithmetic*, Reliable Computing, vol 12, no 5, pp 365-369, 2006.
- J.R. Shewchuk: *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, Discrete and Computational Geometry, vol 18, no 3, pp 305-363, 1997.
- P. Zimmermann: *Arithmétique en précision arbitraire*, Réseaux et systèmes répartis, vol 13, no 4-5, pp 357-386, 2001.

# Agenda

- A remark on the number of correct digits of a computed result
- Introduction to multiple precision arithmetic
- Multiple precision arithmetic implemented using integers
- Multiple precision arithmetic implemented using floating-point numbers: Shewchuk's expansions
- Optimal adaptation of the computing precision (Kreinovich & Rump)

# The number of correct digits of a computed result

**"Rule of thumb":**

$$\text{forward error} \simeq \text{condition number} \times \text{backward error}$$

or in other words

$$\text{number of correct digits} = \text{computing precision} - \text{constant quantity}$$

# The number of correct digits of a computed result

**Counter-example:** determination of a multiple root $x^*$, of multiplicity $m$, of a polynomial $p(x) = a_n x^n + \cdots + a_1 x + a_0$.
A relative error $u$ on $a_i$ perturbs the root $x^*$ to

$$x^*(u) - x^* = u^{1/m} \left[ -\frac{m! a_i x^{*i}}{p^{(m)}(x^*)} \right]^{1/m}$$

multiple roots: always ill-conditioned, forward error: of the order of $u^{1/m}$.
In other words,
number of correct digits = computing precision / constant quantity

# Introduction to multiple precision arithmetic

Vocabulary: multiple precision vs arbitrary precision.

**arbitrary precision:**
used for integer or rational arithmetic, where the representation sizes of the operands vary arbitrarily and can be arbitrarily large.

**multiple precision** (aka multi-precision):
used for floating-point arithmetic, where the lengths of the mantissas and exponents are fixed but can be arbitrarily large.

# Introduction to multiple precision arithmetic

**Exact arithmetic** will not be covered here (lack of time)
but some applications that use it (as a last resort), in computational geometry, will be studied in the exam papers.

Lecture by C.-P. Jeannerod (2nd semester, M1):
mainly exact arithmetic (but also floating-point arithmetic)
mainly principles of the main algorithms in the field (and not validation aspects).

# Introduction to multiple precision arithmetic
# Applications

Either **a bit more accuracy** than floating-point computations, and thus a bit more computing precision (several hundreds of bits)

or **extreme computations**, such as the computation of the largest number of digits of $\pi$: 1,241,100,000,000 first decimals of $\pi$ (Kanada, Tokyo)
or checking some special cases to prove theorems
or determining a counter-example to a conjecture.

# Introduction to multiple precision arithmetic
# Representation

A multiple-precision floating-point number is a number of the form $s.m.\beta^e$.

**representation using integers:**
the mantissa (of arbitrary length) is represented as an exact integer.
Exact integers may be represented as a sequence of machine integers (cf. GMP):

$$m = \sum_{i=0}^{n} m_i B^i$$

where $m_i$ are machine integers and $B$ is the length of a machine word.
(This is more or less true, cf. later).

# Introduction to multiple precision arithmetic
# Representation

**representation using floating-point numbers:**

$$\sum_{i=0}^{n} f_i$$

where the $f_i$ are floating-point numbers, if possible with exponents sufficiently wide apart so that the mantissas do not overlap.
(This is more or less true, cf. later).

# Introduction to multiple precision arithmetic
# Representation: performance issues (time & memory)

**discussion of the number of used bits in each machine word:**
(either integer or floating-point)
if one uses all the bits of the machine word:

- optimal storage use, less steps in algorithms
- handling of overflows (such as carries in addition) is more complex

# Introduction to multiple precision arithmetic
# Representation: performance issues (time & memory)

**discussion of the number of used bits in each machine word:**
if one uses less than all the bits of the machine word (e.g. basis $B < 2^{32}$):

- wasted storage use, more steps in algorithms
- the addition of m products of "digits" must be representable in one word, i.e. $nB^2$ is representable as a digit.
  Useful for multiplication: if $X = \sum_{i=0}^{n} x_i B^i$ and $Y = \sum_{i=0}^{n} y_i B^i$,

$$\text{then } Z = X \times Y = \sum_{i=0}^{2n} z_i B^i \text{ where } z_i = \sum_{j+k=i} x_j \cdot y_k.$$

# Implementation using machine integers addition and subtraction

**Algorithms for the addition or subtraction** = methods learnt at school:

- align the mantissas
- from right to left
- add or subtract the corresponding digits and propagate the carry.

# Implementation using machine integers addition and subtraction

## Algorithms for the addition or subtraction

- align the mantissas
- naive method = add or subtract the corresponding digits
  assumption: the sum or difference of two digits fits in a machine word
- normalize the computed result,
  i.e. get a representation with digits between $0$ and $B-1$.

# Implementation using machine integers normalization

$$\text{Go from } X = \sum_{i=0}^{n} \hat{x}_i B^i \text{ to } X = \sum_{i=0}^{n} x_i B^i \text{ with } 0 \leq x_i < B.$$

$t_0 = \hat{x_0}$
$\texttt{for}\ i = 0 \ldots n-1\ \texttt{do}$
$\qquad x_i = t_i \ \texttt{mod}\ B$
$\qquad t_{i+1} = t_i \ \texttt{div}\ B + \hat{x}_{i+1}$
$x_n = t_n$

# Implementation using machine integers multiplication

**Naive algorithm = school algorithm.**
if $X = \sum_{i=0}^{n} x_i B^i$ and $Y = \sum_{i=0}^{n} y_i B^i$,

$$\text{then } Z = X \times Y = \sum_{i=0}^{2n} z_i B^i \text{ where } z_i = \sum_{j+k=i} x_j \cdot y_k.$$

Of course, this representation of $Z$ must be normalized, i.e. carries must be handled.

# Implementation using machine integers complexity of the naive multiplication

- each digit of $X$ is multiplied by each digit of $Y$ : $n^2$ products
- each digit of $Z$ is the sum of $l$ such products: $O(n^2)$ additions

$\Rightarrow$ **overall complexity** $= O(n^2)$

In practice, difference between school method and algorithm: the sum of the partial result and of the product of $X$ by one digit of $Y$ is done before $X$ is multiplied by the next digit of $Y$ (better storage use).

For multiple precision, only the $n$ first digits are needed. . . but most often the $2n$ digits are computed.

# Implementation using machine integers faster multiplication: Karatsuba (Knuth version)

Let's assume $n$ is even and let's decompose

$$X = X_H \cdot B^{n/2} + X_L \text{ and } Y = Y_H \cdot B^{n/2} + Y_L.$$

$$\begin{aligned} Z &= X \cdot Y \\ &= X_H \cdot Y_H \cdot B^n \\ &\quad + [X_H \cdot Y_H - (X_H - X_L) \cdot (Y_H - Y_L) + X_L \cdot Y_L] \cdot B^{n/2} \\ &\quad + X_L \cdot Y_L. \end{aligned}$$

Only **3** multiplications of numbers of length $n/2$.
Recursively, one gets a complexity $O(n^{\log_2 3}) = O(n^{1.585})$.

# Implementation using machine integers even faster multiplication: Toom-Cook

- split $X$ and $Y$ into $k$ parts
- compute $Z$ using $2k-1$ multiplications
- get a complexity $O(n^{\log_k(2k-1)})$.

# Implementation using machine integers fastest known multiplication

- algorithm due to Schönhage and Strassen (1971)
- inspired from FFT: Fast Fourier Transform
- complexity: $O(n \log n)$

# Implementation using machine integers division and square root: Newton's iteration

## Newton's iteration:

to solve $f(x) = 0$, compute the sequence $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

## Advantages of Newton's iteration:

- quadratic convergence: the number of correct digits roughly doubles between $x_n$ and $x_{n+1}$;
- auto-correction: computing errors made on $x_n$ do not modify the limit of the sequence.

Consequence: double the computing precision at each iteration.

**Complexity:** complexity of the last iteration.

# Implementation using machine integers division: Newton's iteration

**Division:**

solve $f(x) = 1/x - A$ to compute the inverse of $A$.
The iteration is

$$x_{n+1} = x_n(2 - Ax_n).$$

Starting point: machine precision approximate inverse.

# Implementation using machine integers square root: Newton's iteration

**Square root:**
solve $f(x) = x^2 - A$ to compute $\sqrt{A}$.
The iteration is

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right).$$

Better idea:
solve $f(x) = 1/x^2 - A$ to compute $1/\sqrt{A}$ and post-multiply by $A$.
The iteration is

$$x_{n+1} = \frac{1}{2}x_n(3 - Ax_n^2).$$

# Implementation using machine integers elementary functions

**Polynomial approximations:**

domain reduction + Taylor expansions + reconstruction.

# Implementation using machine integers elementary functions

**Example: exponential $\exp x$**

- domain reduction: determine $t$ and $n$ such that $n$ is an integer and $t = \frac{x - n\ln 2}{256}$ belongs to $[-\frac{\ln 2}{512}, \frac{\ln 2}{512}]$;
- Taylor expansion: $\exp t = \sum_{i=0}^{+\infty} \frac{t^i}{i!}$
- reconstruction: $\exp x = (\exp t)^{256} \cdot 2^n$, where $(\exp t)^2 56$ is obtained through 8 successive squarings.

The logarithm is then obtained by Newton's iteration.

# Implementation using machine integers elementary functions

**Trigonometric functions:**
use of the periodicity and of trigonometric identities to work on the domain $[-\frac{\pi}{32}, \frac{\pi}{32}]$.

**Inverse trigonometric functions:**
Newton's iteration applied to the trigonometric functions.

# Implementation using machine integers elementary functions

## Arithmetic-geometric mean:

$$\begin{cases} a_0 & = & a \\ b_0 & = & b \\ a_{i+1} & = & \frac{a_i+b_i}{2} \\ b_{i+1} & = & \sqrt{a_i b_i} \end{cases}$$

Historical note: close to the method employed in Antiquity to compute $\pi$: compute the length of regular polygons with $2^n$ sides inscribed and circonscribed to the unit circle.

# Implementation using machine integers elementary functions

## Example: logarithm $\ln x$

- domain reduction: determine $s$ and $m$ such that $m$ is an integer and $s = x \cdot 2^m > 2^{n/2}$ where $n$ is the precision;
- arithmetic-geometric mean of 1 and $4/s$:

$$\ln x \simeq \frac{\pi}{2AG(1, 4/s)} - m \ln 2$$

  where $\pi$ and $\ln 2$ are also computed using AGMs;

The exponential is then obtained by Newton's iteration.

# Implementation using machine integers complexity of evaluating elementary functions

Using the AGM, the complexity is the complexity of the multiplication times a logarithmic factor.

# Implementation using machine integers algorithms in MPFR to evaluate elementary functions with correct rounding

# Implementation using machine floating-point numbers Shewchuk's expansions

Cf. Section 2 of Shewchuk's paper (ref. on the Web page of this class).

# Automatic adaptation of the computing precision

Computations done with precision $p_0$ and computational time $t_0$:
if the accuracy of the result is not sufficient, restart with precision $p_1$ and computational time $t_1 = f(t_0)$;
if the accuracy of the result is not sufficient, restart with precision $p_2$ and computational time $t_2 = f(t_1)\ldots$
stop when the precision $p_{final}$ satisfies $p_{final-1} < p_{opt} \leq p_{final}$.

**What is the best strategy to choose $p_i$?**
**What is the best function $f$?**

# Automatic adaptation of the computing precision

**Overhead:**
ratio between the time spent: $t_0 + t_1 + \cdots + t_{final}$
and the optimal time $t_{opt}$.

**Optimal strategy:** choose $p_{i+1}$ such that $t_{i+1} = 2t_i$
**Optimal overhead:** ratio $= 4$.

**Comments, limits:**
the implicit assumption is that no previous computation can be used to improve/speed up the next one.