Numerical validation using stochastic arithmetic

Fabienne Jézéquel

LIP6

Master 2 SFPN & MAIN5

`http://www-pequan.lip6.fr/~jezequel/AFAE.html`

Overview

- Floating-point arithmetic: the IEEE 754 standard
- The CESTAC method and the stochastic arithmetic
- The CADNA software
- Contributions of CADNA in numerical methods

Representation of real numbers

In a floating-point arithmetic using the radix b,

$$X = \varepsilon\, M\, b^E$$

is represented by:

- its sign ε, encoded on one digit (0 if X is positive, 1 if X is negative),
- its exponent E, a k digit integer,
- its mantissa M, encoded on p digits.

  $M = \sum_{i=0}^{p-1} a_i\, b^{-i}$ and $a_i \in \{0, ..., b-1\}$.

  Floating-point numbers are usually normalized:
  $a_0 \neq 0$, $M \in [1, b)$ and zero has a special representation.

The IEEE 754 - 2008 standard

Formats using the radix 2:

- *binary16* (half precision)
- *binary32* (single precision)
- *binary64* (double precision)
- *binary128* (quadruple precision)
- *binary256* (octuple precision)

Formats using the radix 10 (emulate decimal rounding exactly):

- *decimal32* (storage on 32 bits)
- *decimal64* (storage on 64 bits)
- *decimal128* (storage on 128 bits)

Single / double precision

IEEE 754 single precision:

1	2 ... 9	10	....	32
s	$E + 2^7 - 1$	a_1	....	a_{23}

$\Rightarrow$ range: $10^{\pm 38}$, accuracy: $u = 2^{-24} \approx 6\ 10^{-8}$

IEEE 754 double precision:

1	2 ... 12	13	....	64
s	$E + 2^{10} - 1$	a_1	....	a_{52}

$\Rightarrow$ range: $10^{\pm 308}$, accuracy: $u = 2^{-53} \approx 1\ 10^{-16}$

Half precision

- **binary16 (fp16):**
    - 11 bits for the mantissa, 5 for the exponent
      $\Rightarrow$ range: $10^{\pm 5}$, accuracy: $u = 2^{-11} \approx 5\ 10^{-4}$
    - used by NVIDIA GPUs, AMD Radeon Instinct MI25 GPU, ARM NEON, Fujitsu A64FX ARM
- **bfloat16:**
    - 8 bits for the mantissa, also 8 for the exponent
      $\Rightarrow$ range: $10^{\pm 38}$, accuracy: $u = 2^{-8} \approx 4\ 10^{-3}$
    - used by Google TPU, NVIDIA GPUs, Arm, Intel.

Rounding mode

$\mathbb{F}$: set of real numbers which can be coded exactly on a computer (set of floating point numbers)

Every real number $x \notin \mathbb{F}$ is approximated by a number $X \in \mathbb{F}$.

Let X_{min} (resp. X_{max}) be the smallest (resp. the greatest) floating point number:

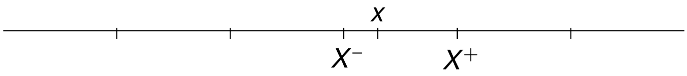
$$\forall x \in \left]X_{min}, X_{max}\right[,\ \exists \left\{X^-, X^+\right\} \in \mathbb{F}^2$$

such that

$$X^- < x < X^+ \text{ and } \left]X^-, X^+\right[\cap \mathbb{F} = \emptyset$$

The rounding mode is the algorithm that, according to x, gives X^- or X^+.

The 4 rounding modes of the IEEE 754 standard



Rounding to zero: x is represented by the floating point number the nearest to x between x and 0.

Rounding to nearest: x is represented by the floating point number the nearest to x.

Rounding to plus infinity: x is represented by X^+.

Rounding to minus infinity: x is represented by X^-.

The rounding operation is performed after each assignment and after every elementary arithmetic operation.

A significant example - I

$$0.3 * x^2 \quad + \quad 2.1 * x \quad + 3.675 \quad = \quad 0$$

- **Rounding to nearest**
  d = -3.81470E-06
  There are two conjugate complex roots.
  z1 = -.3500000E+01 + i * 0.9765625E-03
  z2 = -.3500000E+01 + i * -.9765625E-03

- **Rounding to zero**
  d = 0.
  The discriminant is null.
  The double real root is -.3500000E+01

A significant example - II

$$0.3 * x^2 \quad + \quad 2.1 * x \quad + 3.675 \quad = \quad 0$$

- **Rounding to plus infinity**
  d = 3.81470E-06
  There are two different real roots.
  x1 = -.3500977E+01
  x2 = -.3499024E+01

- **Rounding to minus infinity**
  d = 0.
  The discriminant is null.
  The double real root is -.3500000E+01

Inconsistency of the floating point arithmetic

On a computer, arithmetic operators are only approximations.

- commutativity: $X \circ Y = Y \circ X$
- no associativity: $(X \circ Y) \circ Z \neq X \circ (Y \circ Z)$
- no distributivity: $X \otimes (Y \oplus Z) \neq (X \otimes Y) \oplus (X \otimes Z)$

On a computer, order relationships are used as in mathematics

$\Longrightarrow$ it leads to a global inconsistent behaviour.

Let x, y be exact results and X, Y the associated floating-point numbers:

$$X = Y \quad \not\Rightarrow \quad x = y \quad \text{and} \quad x = y \quad \not\Rightarrow \quad X = Y.$$

$$X \geq Y \quad \not\Rightarrow \quad x \geq y \quad \text{and} \quad x \geq y \quad \not\Rightarrow \quad X \geq Y.$$

Round-off error model

$r \in \mathbb{R}$: exact result of a computation of n elementary arithmetic operations.

On a computer, one obtains $R \in \mathbb{F}$ which is affected by round-off errors.

R can be modeled, at the first order with respect to 2^{-p}, by

$$R \approx r + \sum_{i=1}^{n} g_i(d)\, 2^{-p}\, \alpha_i$$

- p is the number of bits including the hidden bit
- $g_i(d)$ are coefficients depending on data and on the algorithm
- α_i are the round-off errors.

Remarks:

- the number ot terms may be $> n$ (ex: for $n = 1$, we have 3 terms if data are not exactly encoded)
- we have assumed that exponents and signs of intermediate results do not depend on α_i.

A theorem on numerical accuracy

The number of significant bits in common between R and r is defined by

$$C_R \approx -\log_2 \left| \frac{R-r}{r} \right| = p - \log_2 \left| \sum_{i=1}^{n} g_i(d).\frac{\alpha_i}{r} \right|$$

The last part corresponds to the accuracy which has been lost in the computation of R, we can note that it is independent of p.

Theorem

The loss of accuracy during a numerical computation is independent of the precision used for the representation of floating point numbers.

Round-off error analysis

Several approaches

- Inverse analysis
  based on the " Wilkinson principle": the computed solution is assumed to be the exact solution of a nearby problem
  - provides error bounds for the computed results
- Interval arithmetic
  The result of an operation between two intervals contains all values that can be obtained by performing this operation on elements from each interval.
  - guaranteed bounds for each computed result
  - the error may be overestimated
  - specific algorithms

- Static analysis
  - no execution, rigorous analysis, all possible input values taken into account
  - not suited to large programs
- Methods for accurate computations
  - Compensated methods (based on error-free transformations)
  - Multiple precision arithmetic
- Probabilistic approach
  - uses a random rounding mode
  - estimates the number of exact significant digits of any computed result

The CESTAC method

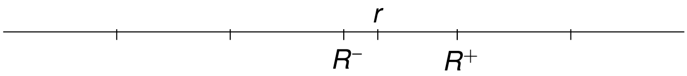
The CESTAC method (Contrôle et Estimation Stochastique des Arrondis de Calculs) was proposed by M. La Porte and J. Vignes in 1974.

It consists in performing the same code several times with different round-off error propagations. Then, different results are obtained.

Briefly, the part that is common to the different results is assumed to be in common also with the exact result and the part that is different is affected by round-off errors.

The random rounding mode

Let r be the exact result of an arithmetic operation: $R^- \leq r \leq R^+$.



The random rounding mode consists in rounding r to $-\infty$ or $+\infty$ with the probability 0.5.

If round-off errors affect the result, even slightly, one obtains for N different runs, N different results on which a statistical test may be applied.

By running N times the code with the random arithmetic, one obtains a N-sample of the random variable modeled by

$$R \approx r + \sum_{i=1}^{n} g_i(d).2^{-p}.\alpha_i$$

where the α_i's are modeled by independent identically distributed random variables. The common distribution of the α_i is uniform on $[-1, +1]$.

$\Rightarrow$ the mathematical expectation of R is the mathematical result r,

$\Rightarrow$ the distribution of R is a quasi-Gaussian distribution.

We use the classical Student's test which provides a confidence interval of the expectation of a Gaussian distribution from a sample.

$\forall \beta \in [0,1]$, $\exists \tau_\beta \in \mathbb{R}$ such that

$$P\left(r \in \left[\overline{R} - \frac{\tau_\beta \, \sigma}{\sqrt{N}}, \overline{R} + \frac{\tau_\beta \, \sigma}{\sqrt{N}}\right]\right) = P\left(\left|\overline{R} - r\right| \leq \frac{\tau_\beta \, \sigma}{\sqrt{N}}\right) = \beta$$

with

$$\overline{R} = \frac{1}{N}.\sum_{i=1}^{N} R_i \quad \text{and} \quad \sigma^2 = \frac{1}{N-1}.\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2 .$$

The relative error on $\overline{R}$ is $\left|\frac{\overline{R}-r}{r}\right| = 10^{-C_{\overline{R}}}$

With a probability β, the number of exact significant digits of $\overline{R}$

$$C_{\overline{R}} \approx \log_{10}\left|\frac{\overline{R}}{\overline{R}-r}\right|$$

is undervalued by

$$C_{\overline{R}} \approx \log_{10}\left(\frac{\sqrt{N}\left|\overline{R}\right|}{\sigma \tau_\beta}\right).$$

Implementation of the CESTAC method

The implementation of the CESTAC method in a code providing a result R consists in:

- performing N times this code with the random rounding mode to obtain N samples R_i of R,
- choosing as the computed result the mean value $\overline{R}$ of R_i, $i = 1, ..., N$,
- estimating the number of exact significant decimal digits of $\overline{R}$ with

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}\left|\overline{R}\right|}{\sigma\tau_\beta}\right)$$

where

$$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \quad \text{and} \quad \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2 .$$

τ_β is the value of Student's distribution for $N-1$ degrees of freedom and a probability level β.

On the number of runs

2 or 3 runs are enough. To increase the number of runs is not necessary

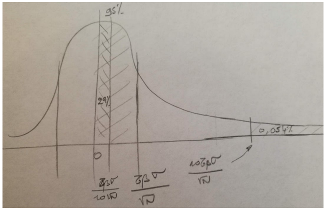
From the model, to increase by 1 the number of exact significant digits given by $C_{\overline{R}}$, we need to multiply the size of the sample by 100.

Such an increase of N will only point out the limit of the model and its error without really improving the quality of the estimation.

It has been shown that $N = 3$ is the optimal value.

On the probability of the confidence interval

Up to one digit, $\dfrac{\sigma\tau_\beta}{10\sqrt{N}} \leq \left|\overline{R} - r\right| \leq \dfrac{10\sigma\tau_\beta}{\sqrt{N}}$



With $\beta = 0.95$ and $N = 3$,

the probability of overestimating the number of exact significant digits of at least 1 is 0.054%

the probability of underestimating the number of exact significant digits of at least 1 is 29% .

By choosing a confidence interval at 95%, we prefer to guarantee a minimal number of exact significant digits with a high probability (0.99946), even if we are often pessimistic by 1 digit.

Self-validation of the CESTAC method

The CESTAC method is based on a 1st order model.

- A multiplication of two insignificant results
- or a division by an insignificant result

may invalidate the 1st order approximation.

Therefore the CESTAC method requires a dynamical control of multiplications and divisions, during the execution of the code.

The problem of stopping criteria

Let us consider a general iterative algorithm: $U_{n+1} = F(U_n)$.

```
while (fabs(X-Y) > EPSILON) {
    X = Y;
    Y = F(X);
}
```

ε too low $\Longrightarrow$ risk of infinite loop
ε too high $\Longrightarrow$ too early termination.

The problem of stopping criteria

Let us consider a general iterative algorithm: $U_{n+1} = F(U_n)$.

```
while (fabs(X-Y) > EPSILON) {
    X = Y;
    Y = F(X);
}
```

ε too low $\Longrightarrow$ risk of infinite loop
ε too high $\Longrightarrow$ too early termination.

It would be optimal to stop when $X - Y$ is an **insignificant value**.

Such a stopping criterion

- would enable one to develop new numerical algorithms
- is possible thanks to the concept of *computed zero*.

The concept of computed zero

J. Vignes, 1986

Definition

Using the CESTAC method, a result R is a computed zero, denoted by @.0, if

$$\forall i, R_i = 0 \text{ or } C_{\overline{R}} \leq 0.$$

This means that 0 belongs to the confidence interval.

It means that R is a computed result which, because of round-off errors, cannot be distinguished from 0.

The stochastic definitions

Definition

Let X and Y be two results computed using the CESTAC method (N-samples).

- X is stochastically equal to Y, noted $X \; s\!= \; Y$, iff

$$X - Y = @.0.$$

- X is stochastically strictly greater than Y, noted $X \; s\!> \; Y$, iff

$$\overline{X} > \overline{Y} \quad \text{and} \quad X \; s\!\neq \; Y$$

- X is stochastically greater than or equal to Y, noted $X \; s\!\geq \; Y$, iff

$$\overline{X} \geq \overline{Y} \quad \text{or} \quad X \; s\!= \; Y$$

DSA **Discrete Stochastic Arithmetic** is defined as the joint use of the CESTAC method, the computed zero and the relation definitions.

A few properties

- $x = 0 \implies X = @.0$.
- $X \; s\neq \; Y \implies x \neq y$.
- $X \; s> \; Y \implies x > y$.
- $x \geq y \implies X \; s\geq \; Y$.
- The relation $s>$ is transitive.
- The relation $s=$ is reflexive, symmetric but not transitive.
- The relation $s\geq$ is reflexive, antisymmetric but not transitive.

The CADNA library - I

The CADNA library allows one to estimate round-off error propagation in any scientific program.

CADNA enables one to:

- estimate the numerical quality of any result
- control branching statements
- perform a dynamic numerical debugging
- take into account uncertainty on data.

CADNA is a library which can be used with Fortran, C, or C++ programs and also with parallel programs (using MPI, OpenMP, CUDA).

CADNA can be downloaded from `http://cadna.lip6.fr`

The CADNA library - II

CADNA implements Discrete Stochastic Arithmetic

CADNA provides new numerical types, the stochastic types (3 floating point variables `x,y,z` and an integer variable `accurracy`):

- `half_st` for stochastic variables in half precision
- `float_st` for stochastic variables in single precision
- `double_st` for stochastic variables in double precision

All operators and mathematical functions are redefined for these types.

The cost of CADNA is about:

- 4 for memory
- 10 for run time.

Numerical debugging

The following instabilities can be detected:

- **unstable division**: the divisor is insignificant
- **unstable power function**: one operand of the pow function is insignificant
- **unstable multiplication**: both operands are insignificant

Numerical debugging(2)

- **unstable branching**: the difference between the two operands is insignificant (a stochastic zero). The chosen branching statement is associated with the equality
- **unstable mathematical function**: in the log, sqrt or exp function, the argument is insignificant.
- **unstable intrinsic function**:
  - inherited from Fortran
  - in the floor or ceil function: the floor (or ceil) function returns different values for each component.
  - in the abs function: different components have different signs.
- **unstable cancellation**: for addition (and subtraction)

$$\min(accuracy(a), accuracy(b)) - accuracy(a+b) > \texttt{CANCEL\_LEVEL}$$

How to implement CADNA

The use of the CADNA library involves at most 6 steps:

- inclusion of the CADNA header for the compiler,
- initialization of the CADNA library,
- substitution of the classic floating-point types by stochastic types in variable declarations,
- possible changes in the input data if perturbation is desired, to take into account uncertainty in initial values,
- change of output statements to print stochastic results with their accuracy,
- termination of the CADNA library.

Declaration of the CADNA library

The `#include <cadna.h>` preprocessor directive must take place before any declaration of stochastic variables, for stochastic types and overloaded or new functions to be found by the compiler.

Initialization of the CADNA library (1)

The call to the `cadna_init` function must be added just after the main function declaration statements to initialize the library.

```
cadna_init(numb_instability, cadna_instability,
           cancel_level, init_random)
```

- `numb_instability` = -1: all the instabilities will be detected
- `numb_instability` = 0: no instability will be detected
- `numb_instability` = M (strictly positive M): the first M instabilities will be detected.

The other arguments are optional.

Initialization of the CADNA library (2)

`cadna_instability`: describes the instabilities which are disabled

- `CADNA_DIV`, `CADNA_MUL`, `CADNA_POWER`
- `CADNA_BRANCHING`
- `CADNA_CANCEL`
- `CADNA_MATH`, `CADNA_INTRINSIC`
- `CADNA_ALL`

`cancel_level`: a cancellation is detected if the difference in terms of accuracy between the two operands and the result is larger than `cancel_level`.
Default: 4.

`init_random`: seed of the random generator used by CADNA.

Termination

The call to the `cadna_end` function should be the last statement.

The `cadna_end` function writes on the standard output a report on the numerical stability of the run.

Changes in the type of variables

To control the numerical quality of a variable, just replace its standard type by the associated stochastic type.

`half`	⇒	`half_st`
`float`	⇒	`float_st`
`double`	⇒	`double_st`

Example:

```
float_st a,b,c;
double_st e,f,g;
float_st d[6];
```

Changes in printing statements

Before printing each stochastic variable, it must be transformed into a string by the `strp` function. This function returns a `char *`, therefore formats in print functions should be modified.

Initial C/C++ code	Modified statements for CADNA
	`#include <cadna.h>`
`float x;`	**`float_st`** `x;`
	`cadna_init(-1);`
...	...
`printf("%f8.3\n", x);`	`printf("`**`%s`**`\n", `**`strp(x)`**`);`

Changes in printing statements(2)

With the `strp` function, only the exact significant digits are printed.
If a result has no exact significant digit, `@.0` is printed.

Example:

```
U(3) = 0.559016393344262E+001
U(4) = 0.563343108504E+001
U(5) = 0.56746486205E+001
U(6) = 0.5713329052E+001
U(7) = 0.574912092E+001
U(8) = 0.57818109E+001
U(9) = 0.581131E+001
U(10) = 0.58376E+001
U(11) = 0.5861E+001
U(12) = 0.588E+001
U(13) = 0.5E+001
U(14) = @.0
```

Changes in reading statements

The reading functions are adapted to classical floating-point variables, which must be transformed into stochastic variables.

Example:

Initial C/C++ code	Modified statements for CADNA
	#include <cadna.h>
float x;	**float_st** x;
	float xaux;
	cadna_init(-1);
.....	.....
scanf("%f",&x);	scanf("%f",&**xaux**);
	x=xaux;

An example proposed by S. Rump

Computation of $f(10864, 18817)$ and $f(\frac{1}{3}, \frac{2}{3})$ with $f(x, y) = 9x^4 - y^4 + 2y^2$

```
#include <stdio.h>

double rump(double x, double y) {
  double a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  double x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n", rump(x, y));
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n", rump(x, y));
  return 0;
}
```

An example proposed by S. Rump (2)

The results:

```
P(10864,18817) =    2.00000000000000
P(1/3,2/3) =    0.802469135802469E+00
```

```
#include <stdio.h>

double   rump(double   x, double   y) {
  double   a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {

  double   x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",      rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  double   a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {

  double   x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",      rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  double   a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double   x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",      rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  double   a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double   x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",      rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  double   a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double   x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",       rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",       rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  double_st a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",      rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  double_st a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%f\n",      rump(x, y) );"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%f\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  double_st a, b, c;
  a = 9.0*x*x*x*x;
  b = y*y*y*y;
  c = 2.0*y*y;
  return a-b+c;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x = 10864.0;
  y = 18817.0;
  printf("%s\n", strp(rump(x, y)));"
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("%s\n", strp(rump(x, y)));"
  cadna_end();
  return 0;
}
```

The run with CADNA

```
----------------------------------------------------------------
CADNA software
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
----------------------------------------------------------------
P(10864,18817) = @.0
P(1/3,2/3) = 0.802469135802469E+000
----------------------------------------------------------------
There are 2 numerical instabilities
0 UNSTABLE DIVISION(S)
0 UNSTABLE POWER FUNCTION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE BRANCHING(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE INTRINSIC FUNCTION(S)
2 UNSTABLE CANCELLATION(S)
```

Explanation

The run without CADNA:

```
9*x*x*x*x →                           1.25372283822342144E+017
y*y*y*y →                             1.25372284530501120E+017
9*x*x*x*x - y*y*y*y →                          -708158976.00000000
2*y*y →                                         708158978.00000000
9*x*x*x*x - y*y*y*y +2y*y →                               2.0000000000000000
```

The run with CADNA:

```
9*x*x*x*x →                         0.125372283822342E+018
y*y*y*y →                           0.125372284530501E+018
9*x*x*x*x - y*y*y*y →                           -0.7081589E+009
2*y*y →                                          0.708158977999999E+009
9*x*x*x*x - y*y*y*y +2y*y →                                @.0
```

Several versions of the program

	without CADNA	with CADNA

a = 9.*x*x*x*x
b = y*y*y*y
c = 2.*y*y
rump = a-b+c
	2	@.0

a = 9.*pow(x,4)
b = pow(y,4)
c = 2.*pow(y,2)
rump = a-b+c
	2	2

rump = 9.*x*x*x*x- y*y*y*y + 2.*y*y	2	@.0
rump = 9.*pow(x,4)- pow(y,4) + 2.*pow(y,2)	2	2

CADNA requires a correct rounding toward $+\infty$ and $-\infty$.

Correct rounding for mathematical functions?

Different mathematical libraries may provide different results: the last bit in the results may differ.

Correct rounding for mathematical functions: an open problem
Extra bits are sometimes required to obtain a correct rounding.

The gnu mathematical library on 64-bit processors:

- provides correct results with rounding to the nearest
- severe bugs may occur with the other rounding modes.

$\Rightarrow$ Taken into account in CADNA

the `data_st` function

It allows us to take into account errors on data by perturbing the samples of a stochastic variable.

- `data_st(X)` : perturbation of the last bit of the mantissa.
- `data_st(X,ERX,0)` : relative error

$$X_i = X_i * (1 + ERX * ALEA)$$

- `data_st(X,ERX,1)` : absolute error

$$X_i = X_i + (ERX * ALEA)$$

Example :

```
float_st b;
b=-2.1;
data_st(b,0.1,0);
```

The 3 samples become:

```
 -2.309487      -1.980967      -2.100000
```

Contributions of CADNA

- In direct methods:
  - estimate the numerical quality of the results
  - control branching statements
- In iterative methods:
  - optimize the number of iterations
  - check if the computed solution is satisfactory
- In approximation methods:
  - optimize the integration step

In direct methods - Example

$$0.3x^2 - 2.1x + 3.675 = 0$$

Without CADNA, in single precision with rounding to the nearest:

```
d = -3.8146972E-06
Two complex roots
z1 = 0.3499999E+01 + i * 0.9765625E-03
z2 = 0.3499999E+01 + i * -.9765625E-03
```

With CADNA:

```
d = @.0
The discriminant is null
The double real root is 0.3500000E+01
```

Contribution of CADNA in iterative methods

$$U_{n+1} = F(U_n)$$

Without / with CADNA

```
while (fabs(X-Y) > EPSILON)
{
    X = Y;
    Y = F(X);
}
```

With CADNA

```
while (X != Y) {
    X = Y;
    Y = F(X);
}
```

☺ optimal stopping criterion

Iterative methods: example

$$S_n(x) = \sum_{i=1}^{i=n} \frac{x^i}{i!}$$

Stopping criterion

- IEEE: $|S_n - S_{n-1}| < 10^{-15}|S_n|$
- CADNA: $S_n == S_{n-1}$

	IEEE		CADNA	
x	iter	$S_n(x)$	iter	$S_n(x)$
-5.	37	6.737946999084039E-003	38	0.673794699909E-002
-10.	57	4.539992962303130E-005	58	0.45399929E-004
-15.	76	3.059094197302006E-007	77	0.306E-006
-20.	94	5.621884472130416E-009	95	@.0
-25.	105	-7.129780403672074E-007	106	@.0

Approximation methods

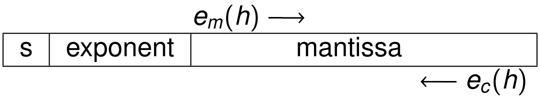
Approximation of a limit $L = \lim_{h \to 0} L(h)$

Two kind of errors:

- $e_m(h)$: truncation error (*mathematical* error)
- $e_c(h)$: rounding error (*computation* error)

If h decreases, $e_m(h)$ decreases, but $e_c(h)$ increases.

If h decreases, $L(h)$:



How to estimate the optimal step?

If $e_c(h) < e_m(h)$, decreasing h brings reliable information.

Computation should stop when $e_c(h) \approx e_m(h)$

Dynamical control of approximation methods

Theorem

Let us consider a numerical method which provides an approximation $L(h)$ of order p to an exact value L:

$$L(h) - L = Kh^p + \mathcal{O}(h^q) \text{ with } 1 \leq p < q,\ K \in \mathbb{R}.$$

If L_n is the approximation computed with the step $\frac{h_0}{2^n}$, then

$$C_{L_n,L_{n+1}} = C_{L_n,L} + \log_{10}\left(\frac{2^p}{2^p - 1}\right) + \mathcal{O}\left(2^{n(p-q)}\right).$$

$\log_{10}\left(\frac{2^p}{2^p-1}\right) \leq \log_{10}\left(\frac{2}{2-1}\right) = \log_{10}(2) \approx 0.3$

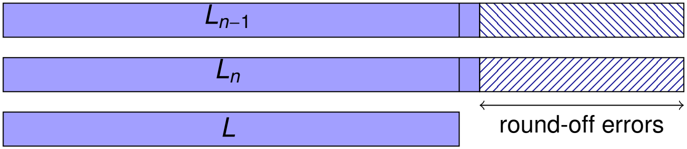
If the convergence zone is reached, the digits common to two successive iterates are also common to the exact result, up to one.

Approximation methods with the CADNA library

The technique of “step halving” is applied and iterations are stopped when $L_n - L_{n-1} = @.0$

You are sure that the result L_n is optimal.

Furthermore its significant digits which are not affected by round-off errors are in common with the exact result L, up to one.



Approximation methods with the CADNA library

Approximations are computed using Simpson's method.

```
n= 1 In= 0.532202672142964E+002 err= 0.459035794670113E+002
n= 2 In=-0.233434428466744E+002 err= 0.306601305939595E+002
n= 3 In=-0.235451792663099E+002 err= 0.308618670135950E+002
n= 4 In= 0.106117380632568E+002 err= 0.329505031597175E+001
n= 5 In= 0.742028156692706E+001 err= 0.103593819641 9E+000
n= 6 In= 0.732233719854278E+001 err= 0.564945125770E-002
n= 7 In= 0.731702967403266E+001 err= 0.34192674758E-003
n= 8 In= 0.731670894914430E+001 err= 0.2120185922E-004
n= 9 In= 0.731668906978969E+001 err= 0.13225046E-005
n=10 In= 0.731668782990089E+001 err= 0.8261581E-007
n=11 In= 0.731668775244794E+001 err= 0.516286E-008
n=12 In= 0.7316687747 6078E+001 err= 0.3227E-009
n=13 In= 0.731668774730 53E+001 err= 0.202E-010
n=14 In= 0.731668774728 64E+001 err= 0.1E-011
n=15 In= 0.731668774728 52E+001 err= 0.1E-012
n=16 In= 0.731668774728 51E+001 err=@.0
```

The exact solution is: `7.316687747285081429939`.

Dynamical control of approximation methods

Theorem also valid for the trapezoidal method, Gauss-Legendre method,...
Similar theoretical result for Romberg's method
$\Rightarrow$ same strategy: in the result obtained, the digits which are not affected by round-off errors are those of the exact result, up to one.

Also theoretical results for combined sequences
$\Rightarrow$ dynamical control of infinite integrals, multidimensional integrals

Recent developments related to stochastic arithmetic

- Improvement of CADNA
  - no more explicit change of the rounding mode thanks to
    $a \oplus_{+\infty} b = -(-a \oplus_{-\infty} -b)$ (similarly for $\ominus$)
    $a \otimes_{+\infty} b = -(a \otimes_{-\infty} -b)$ (similarly for $\oslash$)
    $\bigcirc_{+\infty}$ (resp. $\bigcirc_{-\infty}$): floating-point operation rounded towards $+\infty$ (resp. $-\infty$)
  - inlining of operators
  - quadruple precision
  - half precision
- CADNA for parallel programs
  - GPU
  - MPI
    - new MPI types for stochastic variables
    - works as for sequential codes
  - OpenMP
    - management of the rounding mode with the threads
    - instability detection
- CADNA for vectorised programs

Tools related to CADNA

available on `cadna.lip6.fr`

- CADNAIZER
  - automatically transforms C codes to be used with CADNA
- CADTRACE
  - identifies the instructions responsible for numerical instabilities

    Example:

```
There are 12 numerical instabilities.
 10 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S).
     5 in <ex> file "ex.f90" line 58
     5 in <ex> file "ex.f90" line 59
 1 INSTABILITY IN ABS FUNCTION.
     1 in <ex> file "ex.f90" line 37
 1 UNSTABLE BRANCHING.
     1 in <ex> file "ex.f90" line 37
```

The SAM library

www-pequan.lip6.fr/~jezequel/SAM

SAM (Stochastic Arithmetic in Multiprecision) [Graillat & al.'11]

- implements stochastic arithmetic in arbitrary precision (based on MPFR[1])
  `mp_st` stochastic type
- operator overloading ⇒ few modifications in user C/C++ programs

[1] www.mpfr.org

The SAM library

www-pequan.lip6.fr/~jezequel/SAM

SAM (Stochastic Arithmetic in Multiprecision) `[Graillat & al.'11]`

- implements stochastic arithmetic in arbitrary precision (based on MPFR[1])
  `mp_st` stochastic type
- operator overloading ⇒ few modifications in user C/C++ programs

Recent improvement: control of operations mixing different precisions

Ex: mp_st<23> A; mp_st<47>B; mp_st<35> C;

$$\begin{tikzcd} C \arrow[d] & = & A \arrow[d] & \oplus & B \arrow[d] \\ \text{35 bits} & & \text{23 bits} & & \text{47 bits} \end{tikzcd}$$

⇒ accuracy estimation on FPGA

[1] www.mpfr.org

Accuracy analysis... and then?

accurate results?

No ☹

- increase precision: `single` → `double` → `quad` → arbitrary precision
- compensated algorithms
  `[Kahan'87], [Priest'92], [Ogita & al.'05], [Graillat & al.'09]`
  - for sum, dot product, polynomial evaluation,...
  - results ≈ as accurate as with twice the working precision
- accurate and reproducible BLAS
  - ExBLAS `[Collange & al.'15]`
  - RARE-BLAS `[Chohra & al.'16]`
  - OzBLAS `[Mukunoki & al.'19]`
- symbolic computation

Accuracy analysis... and then?

accurate results?

No ☹

- increase precision: `single` → `double` → `quad` → arbitrary precision
- compensated algorithms
  `[Kahan'87], [Priest'92], [Ogita & al.'05], [Graillat & al.'09]`
  - for sum, dot product, polynomial evaluation,...
  - results ≈ as accurate as with twice the working precision
- accurate and reproducible BLAS
  - ExBLAS `[Collange & al.'15]`
  - RARE-BLAS `[Chohra & al.'16]`
  - OzBLAS `[Mukunoki & al.'19]`
- symbolic computation

Yes ☺

performance improvement thanks to mixed precision?

Precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
  - Precimonious `[Rubio-Gonzàlez & al.'13]`
    - source modification with LLVM
  - CRAFT `[Lam & al.'13]`
    - binary modifications on the operations
  - ADAPT `[Menon & al.'18]`
    - based on algorithmic differentiation
  - CRAFT & ADAPT now combined in FloatSmith `[Lam & al.'19]`

Precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
  - Precimonious [Rubio-Gonzàlez & al.'13]
    - source modification with LLVM
  - CRAFT [Lam & al.'13]
    - binary modifications on the operations
  - ADAPT [Menon & al.'18]
    - based on algorithmic differentiation
  - CRAFT & ADAPT now combined in FloatSmith [Lam & al.'19]

They rely on comparisons with the highest precision result

Precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
  - Precimonious [Rubio-Gonzàlez & al.'13]
    - source modification with LLVM
  - CRAFT [Lam & al.'13]
    - binary modifications on the operations
  - ADAPT [Menon & al.'18]
    - based on algorithmic differentiation
  - CRAFT & ADAPT now combined in FloatSmith [Lam & al.'19]

They rely on comparisons with the highest precision result

[Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$
float: $P =$2.571784e+29

Precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
  - Precimonious [Rubio-Gonzàlez & al.'13]
    - source modification with LLVM
  - CRAFT [Lam & al.'13]
    - binary modifications on the operations
  - ADAPT [Menon & al.'18]
    - based on algorithmic differentiation
  - CRAFT & ADAPT now combined in FloatSmith [Lam & al.'19]

They rely on comparisons with the highest precision result

[Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$

float:	$P =$2.571784e+29
double:	$P =$1.17260394005318

Precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
  - Precimonious [Rubio-Gonzàlez & al.'13]
    - source modification with LLVM
  - CRAFT [Lam & al.'13]
    - binary modifications on the operations
  - ADAPT [Menon & al.'18]
    - based on algorithmic differentiation
  - CRAFT & ADAPT now combined in FloatSmith [Lam & al.'19]

They rely on comparisons with the highest precision result

[Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$

float:	P =2.571784e+29
double:	P =1.17260394005318
quad:	P =1.17260394005317863185883490452018

Precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
  - Precimonious [Rubio-Gonzàlez & al.'13]
    - source modification with LLVM
  - CRAFT [Lam & al.'13]
    - binary modifications on the operations
  - ADAPT [Menon & al.'18]
    - based on algorithmic differentiation
  - CRAFT & ADAPT now combined in FloatSmith [Lam & al.'19]

They rely on comparisons with the highest precision result

[Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$

float:	P =2.571784e+29
double:	P =1.17260394005318
quad:	P =1.17260394005317863185883490452018
exact:	P ≈-0.827396059946821368141165095479816292

PROMISE (PRecision OptiMISE) [Graillat & al.'19]

promise.lip6.fr

- provides a mixed precision code (half, single, double, quad) taking into account a required accuracy
- uses CADNA to validate a type configuration
- uses the Delta Debug algorithm `[Zeller'09]` to search for a valid type configuration with a mean complexity of $O(n \log(n))$ for n variables.

Recent improvements:

- complete rewriting (more user friendly, performance improved)
- half precision

Searching for a valid type configuration

PROMISE with 2 types (ex: double & single precision)

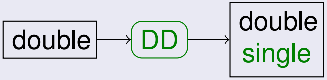
From a code in double, the Delta Debug (DD) algorithm finds which variables can be relaxed to single precision.

$$\begin{tikzcd}
\boxed{\text{double}} \arrow[r] & \text{DD} \arrow[r] & \boxed{\begin{array}{c}\text{double}\\ \text{single}\end{array}}
\end{tikzcd}$$

Searching for a valid type configuration

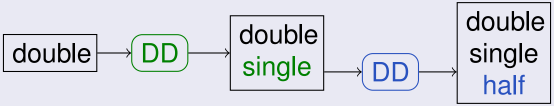
PROMISE with 2 types (ex: double & single precision)

From a code in double, the Delta Debug (DD) algorithm finds which variables can be relaxed to single precision.



PROMISE with 3 types (ex: double, single & half precision)

The Delta Debug algorithm is applied twice.



Precision auto-tuning using PROMISE

MICADO: simulation of nuclear cores (EDF)

- neutron transport iterative solver
- 11,000 C++ code lines

# Digits	# double - # float	Speed up	memory gain
10	19-32	1.01	1.00
8	18-33	1.01	1.01
6	13-38	1.20	1.44
5 4	0-51	1.32	1.62

- Speedup, memory gain: w.r.t. the initial configuration (in double precision).
- Speed-up up to 1.32 and memory gain 1.62
- Mixed precision approach successful: speed-up 1.20 and memory gain 1.44

Related works

Other numerical validation tools based on result pertubation

- MCAlib `[Frechling et al., 2015]`
- VerifiCarlo `[Denis et al., 2016]`
  based on LLVM
- Verrou `[Févotte et al., 2017]`
  based on Valgrind, no source code modification ☺

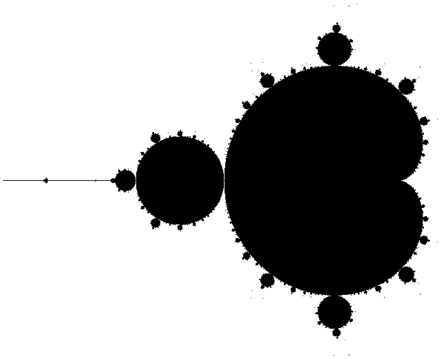
- asynchronous approach: 1 complete run → 1 result, no accuracy analysis during the run
- if branches in the user code:
  several executions → possibly several branches
  (require more samples than CADNA)
- no support for GPU codes.

Numerical applications

Example: Mandelbrot set computed on GPU

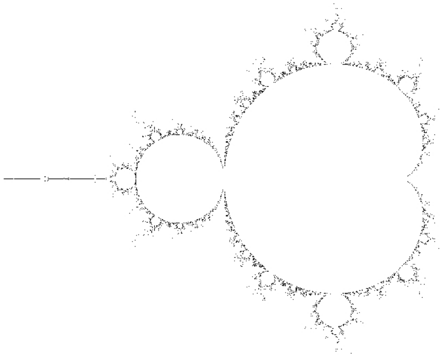
- We map a 2D image on a part of the complex plane
- for each pixel we iterate at most N times:

  $z_{n+1} = z_n^2 + c$, with $z_0 = 0$ and $c \in \mathbb{C}$ the pixel center coordinates.
  - If $\exists n$ s.t. $|z_n| > 2$, the sequence will diverge and c is not in the set.
  - Otherwise, c is in the set.



Mandelbrot set computed on GPU with CADNA

Pixels with unstable tests:



unstable test $|z_n| > 2$ $\Rightarrow$ complete loss of accuracy in z_n

Should these points be in the set ?

Reproducibility failures in a wave propagation code

For oil exploration, the 3D acoustic wave equation

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \sum_{b \in x,y,z} \frac{\partial^2}{\partial b^2} u = 0$$

where u is the acoustic pressure, c is the wave velocity and t is the time

is solved using a finite difference scheme

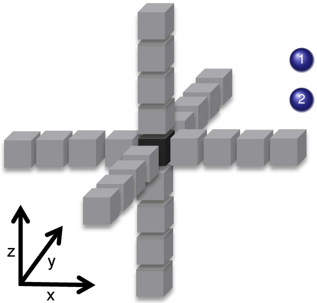
- time: order 2
- space: order p (in our case $p = 8$).

2 implementations of the finite difference scheme

1. $$u_{ijk}^{n+1} = 2u_{ijk}^{n} - u_{ijk}^{n-1} + \frac{c^2\Delta t^2}{\Delta h^2} \sum_{l=-p/2}^{p/2} a_l \left(u_{i+ljk}^{n} + u_{ij+lk}^{n} + u_{ijk+l}^{n}\right) + c^2\Delta t^2 f_{ijk}^{n}$$

2. $$u_{ijk}^{n+1} = 2u_{ijk}^{n} - u_{ijk}^{n-1} + \frac{c^2\Delta t^2}{\Delta h^2} \left(\sum_{l=-p/2}^{p/2} a_l u_{i+ljk}^{n} + \sum_{l=-p/2}^{p/2} a_l u_{ij+lk}^{n} + \sum_{l=-p/2}^{p/2} a_l u_{ijk+l}^{n} \right) + c^2\Delta t^2 f_{ijk}^{n}$$

where u_{ijk}^{n} (resp. f_{ik}^{n}) is the wave (resp. source) field in (i, j, k) coordinates and n^{th} time step and $a_{l\in-p/2,p/2}$ are the finite difference coefficients



1. nearest neighbours first
2. dimension 1, 2 then 3

Reproducibility problems

Results depend on :

- the implementation of the finite difference scheme
- the compiler / architecture (various CPUs and GPUs used)

In *binary32*, for $64 \times 64 \times 64$ space steps and 1000 time iterations:

- any two results at the same space coordinates have 0 to 7 common digits
- the average number of common digits is about 4.

Results computed at 3 different points

scheme	point in the space domain		
	$p_1 = (0, 19, 62)$	$p_2 = (50, 12, 2)$	$p_3 = (20, 1, 46)$
AMD Opteron CPU with gcc			
1	**-1.11**0479E+0	**5.454**238E+1	**6.1410**38E+2
2	**-1.11**0426E+0	**5.454**199E+1	**6.1410**35E+2
NVIDIA C2050 GPU with CUDA			
1	**-1.11**0204E+0	**5.454**224E+1	**6.1410**46E+2
2	**-1.10**9869E+0	**5.454**244E+1	**6.1410**47E+2
NVIDIA K20c GPU with OpenCL			
1	**-1.10**9953E+0	**5.454**218E+1	**6.1410**44E+2
2	**-1.11**1517E+0	**5.454**185E+1	**6.1410**24E+2
AMD Radeon GPU with OpenCL			
1	**-1.10**9940E+0	**5.454**317E+1	**6.1410**38E+2
2	**-1.11**0111E+0	**5.454**170E+1	**6.1410**44E+2
AMD Trinity APU with OpenCL			
1	**-1.11**0023E+0	**5.454**169E+1	**6.1410**62E+2
2	**-1.11**0113E+0	**5.454**261E+1	**6.1410**49E+2

Results computed at 3 different points

scheme	point in the space domain		
	$p_1 = (0, 19, 62)$	$p_2 = (50, 12, 2)$	$p_3 = (20, 1, 46)$
AMD Opteron CPU with gcc			
1	**-1.11**0479E+0	**5.454**238E+1	**6.1410**38E+2
2	**-1.11**0426E+0	**5.454**199E+1	**6.1410**35E+2
NVIDIA C2050 GPU with CUDA			
1	**-1.11**0204E+0	**5.454**224E+1	**6.1410**46E+2
2	**-1.10**9869E+0	**5.454**244E+1	**6.1410**47E+2
NVIDIA K20c GPU with OpenCL			
1	**-1.10**9953E+0	**5.454**218E+1	**6.1410**44E+2
2	**-1.11**1517E+0	**5.454**185E+1	**6.1410**24E+2
AMD Radeon GPU with OpenCL			
1	**-1.10**9940E+0	**5.454**317E+1	**6.1410**38E+2
2	**-1.11**0111E+0	**5.454**170E+1	**6.1410**44E+2
AMD Trinity APU with OpenCL			
1	**-1.11**0023E+0	**5.454**169E+1	**6.1410**62E+2
2	**-1.11**0113E+0	**5.454**261E+1	**6.1410**49E+2

How to estimate the impact of rounding errors?

The acoustic wave propagation code examined with CADNA

The code is run on:

- an AMD Opteron 6168 CPU with gcc
- an NVIDIA C2050 GPU with CUDA.

With both implementations of the finite difference scheme, the number of exact digits varies from 0 to 7 (single precision).

Its mean value is:

- 4.06 with both schemes on CPU
- 3.43 with scheme 1 and 3.49 with scheme 2 on GPU.

$\Rightarrow$ consistent with our previous observations

Instabilities detected: > 270 000 cancellations

The acoustic wave propagation code examined with CADNA

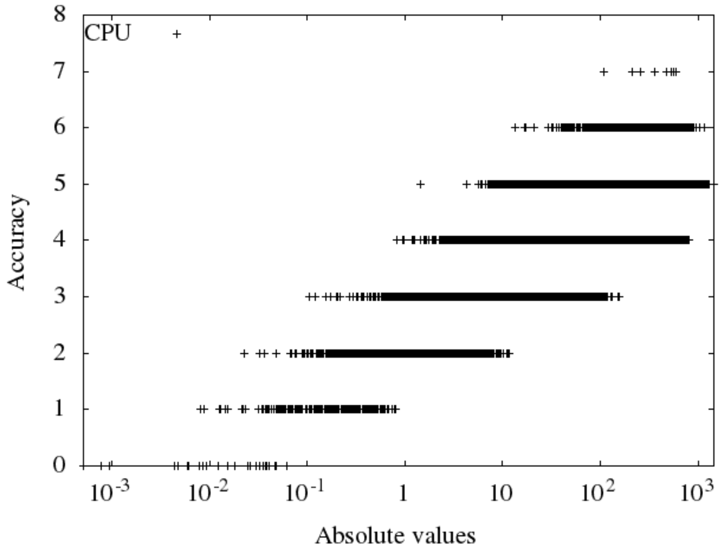
Results computed at 3 different points using scheme 1:

	Point in the space domain		
	$p_1 = (0, 19, 62)$	$p_2 = (50, 12, 2)$	$p_3 = (20, 1, 46)$
IEEE CPU	-1.110479E+0	5.454238E+1	6.141038E+2
IEEE GPU	-1.110204E+0	5.454224E+1	6.141046E+2
CADNA CPU	-1.1E+0	5.454E+1	6.14104E+2
CADNA GPU	-1.11E+0	5.45E+1	6.1410E+2
Reference	-1.108603879E+0	5.454034021E+1	6.141041156E+2

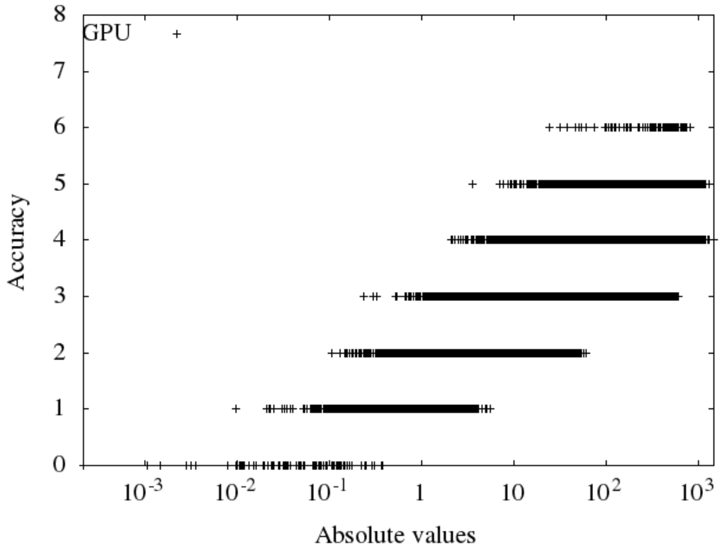
Despite differences in the estimated accuracy, the same trend can be observed on CPU and on GPU.

- Highest round-off errors impact negligible results.
- Highest results impacted by low round-off errors.

Accuracy distribution on CPU

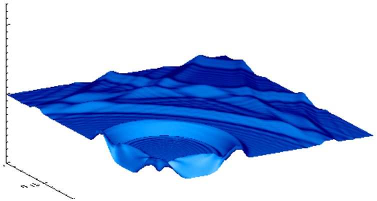


Accuracy distribution on GPU



Numerical validation of a shallow-water (SW) simulation on GPU

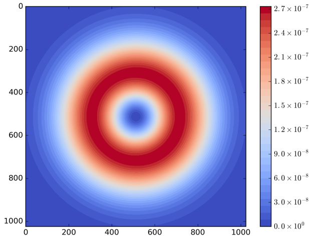
- Numerical model (combination of finite difference stencils) simulating the evolution of water height and velocities in a 2D oceanic basin



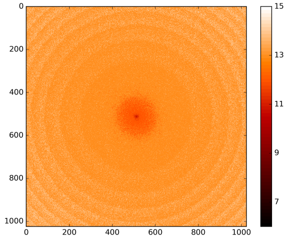
- Focusing on an eddy evolution:
  - 20 time steps (12 hours of simulated time) on a 1024 × 1024 grid
  - CUDA GPU deployment
  - in double precision

SW eddy simulation with CADNA-GPU

At the end of the simulation:



Square of water velocity in $m^2.s^{-2}$



Number of exact significant digits estimated by CADNA-GPU

- at eddy center: great accuracy loss
  equilibrium between several forces (pressure, Coriolis)
  $\Rightarrow$ **possible cancellations**
- point at the very center: 9 exact significant digits lost
  $\Rightarrow$ **no correct digits in SP**
- fortunately, velocity values close to zero at eddy center
  $\rightarrow$ negligible impact on the output
  $\rightarrow$ **satisfactory overall accuracy**

Conclusion

Stochastic arithmetic can estimate which digits are affected by round-off errors and possibly explain reproducibility failures.

- Relatively low overhead
- Support for wide range of codes (GPU, vectorised, MPI, OpenMP)
- Numerical instabilities sometimes difficult to understand in a large code
- Easily applied to real life applications

CADNA has been successfully used for the numerical validation of academic and industrial simulation codes in various domains such as astrophysics, atomic physics, chemistry, climate science, fluid dynamics, geophysics.