Can we avoid rounding-error estimation in HPC codes and still get trustworthy results?

Fabienne Jézéquel[1], Stef Graillat[1], Daichi Mukunoki[2], Toshiyuki Imamura[2], Roman Iakymchuk[1]

[1]LIP6, Sorbonne Université, CNRS, Paris, France

[2]RIKEN Center for Computational Science, Kobe, Japan

13th International Workshop on Numerical Software Verification 2020
20-21 July 2020

Introduction

- Increasing power of current computers
  → accelerators: GPUs, TPUs, FPGAs, etc.
- Enable to solve more complex problems
  → Quantum field theory, supernova simulation, etc.
- A high number of floating-point operations performed
  → Each of them can lead to a rounding error

Introduction

- Increasing power of current computers
  → accelerators: GPUs, TPUs, FPGAs, etc.
- Enable to solve more complex problems
  → Quantum field theory, supernova simulation, etc.
- A high number of floating-point operations performed
  → Each of them can lead to a rounding error

⇒ Numerical validation is crucial

Introduction

- Increasing power of current computers
  → accelerators: GPUs, TPUs, FPGAs, etc.
- Enable to solve more complex problems
  → Quantum field theory, supernova simulation, etc.
- A high number of floating-point operations performed
  → Each of them can lead to a rounding error

⇒ Numerical validation is crucial ...but costful ☹

- execution time overhead
- development cost induced by the application of numerical validation methods to HPC codes

Introduction

- Increasing power of current computers
  → accelerators: GPUs, TPUs, FPGAs, etc.
- Enable to solve more complex problems
  → Quantum field theory, supernova simulation, etc.
- A high number of floating-point operations performed
  → Each of them can lead to a rounding error

⇒ Numerical validation is crucial ...but costful ☹

- execution time overhead
- development cost induced by the application of numerical validation methods to HPC codes

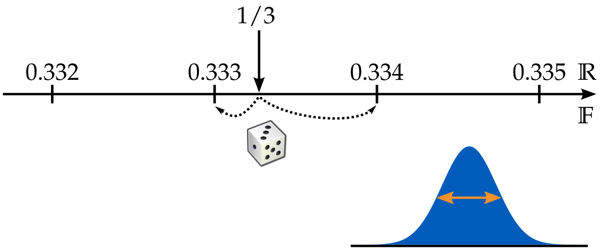
Can we address this cost problem
...and still get trustworthy results?

Yes, when the input data is affected by rounding and/or measurement errors.

Overview

1. Estimation of rounding errors:
   Discrete Stochastic Arithmetic (DSA) and the CADNA library
2. Error induced by perturbed data
3. Our approach: combining DSA and standard floating-point arithmetic
4. Numerical experiments
5. Pros and cons of our approach

Probabilistic approach for numerical validation



- operations are performed several times with random perturbations
  → accuracy estimation
- analysis of the user code
  → no specific numerical algorithms

Several tools:

CADNA [Chesneaux, 1990], MCAlib [Frechling et al., 2015], SAM [S. Graillat et al., 2011], VerifiCarlo [Denis et al., 2016], Verrou [Févotte et al., 2017]

Discrete Stochastic Arithmetic (DSA) [J. Vignes, 2004]

Principles

- each operation is executed 3 times with a random rounding mode: $R \to (R_1, R_2, R_3)$ where each result R_i is rounded up or down with the same probability
- the number of correct digits in the results is estimated using Student's test with the confidence level 95%
- operations are executed synchronously
  - ⇒ detection of numerical instabilities
    Ex: `if (A>B)` with `A-B` numerical noise
  - ⇒ optimization of stopping criteria
    Ex: stop when $x_n - x_{n+1}$ is numerical noise

Discrete Stochastic Arithmetic (DSA) [J. Vignes, 2004]

Principles

- each operation is executed 3 times with a random rounding mode: $R \to (R_1, R_2, R_3)$ where each result R_i is rounded up or down with the same probability
- the number of correct digits in the results is estimated using Student's test with the confidence level 95%
- operations are executed synchronously
  - ⇒ detection of numerical instabilities
    Ex: `if (A>B)` with `A-B` numerical noise
  - ⇒ optimization of stopping criteria
    Ex: stop when $x_n - x_{n+1}$ is numerical noise

Implementations of DSA

- CADNA: for programs in double, single and/or half precision
  `http://cadna.lip6.fr`
- SAM: for arbitrary precision programs (based on MPFR)
  `http://www-pequan.lip6.fr/~jezequel/SAM`

The CADNA library http://cadna.lip6.fr

CADNA allows one to estimate rounding error propagation in any scientific program written in C, C++ or Fortran.

CADNA enables one to estimate the numerical quality of any result and detect numerical instabilities.

The CADNA library http://cadna.lip6.fr

CADNA allows one to estimate rounding error propagation in any scientific program written in C, C++ or Fortran.

CADNA enables one to estimate the numerical quality of any result and detect numerical instabilities.

CADNA provides new numerical types, the stochastic types, which consist of:

- 3 floating point variables
- an integer variable to store the accuracy.

All operators and mathematical functions are redefined for these types.
$\Rightarrow$ CADNA requires only a few modifications in user programs.

Performance overhead: $\times 4$ memory, $\approx \times 10$ execution time

An example without/with CADNA

Computation of $P(x, y) = 9x^4 - y^4 + 2y^2$ [S.M. Rump, 1983]

```
#include <stdio.h>

double rump(double x, double y) {
  return 9.0*x*x*x*x - y*y*y*y + 2.0*y*y;
}
int main(int argc, char **argv) {
  double x, y;
  x = 10864.0;
  y = 18817.0;
  printf("P1=%.14e\n", rump(x, y));
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("P2=%.14e\n", rump(x, y));
  return 0;
}
```

An example without/with CADNA

Computation of $P(x,y) = 9x^4 - y^4 + 2y^2$ [S.M. Rump, 1983]

```
#include <stdio.h>

double rump(double x, double y) {
  return 9.0*x*x*x*x - y*y*y*y + 2.0*y*y;
}
int main(int argc, char **argv) {
  double x, y;
  x = 10864.0;
  y = 18817.0;
  printf("P1=%.14e\n", rump(x, y));
  x = 1.0/3.0;
  y = 2.0/3.0;
  printf("P2=%.14e\n", rump(x, y));
  return 0;
}
```

P1=2.00000000000000e+00

P2=8.02469135802469e–01

```
#include <stdio.h>

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

  double   x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"

  return 0;
}
```

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

  double   x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double   x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double   x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double   rump(double   x, double   y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double   x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",       rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",       rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",       rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",       rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  printf("P1=%s\n", strp(rump(x, y)));"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%s\n", strp(rump(x, y)));"
  cadna_end();
  return 0;
}
```

Results with CADNA

only correct digits are displayed

Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON

P1= @.0 (no correct digits)
P2= 0.802469135802469E+000

There are 2 numerical instabilities
2 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)

A closer look at the floating-point values in P1 and P2:

P1=	P2=
-1.400000000000000e+01	0.802469135802469e+00
-1.400000000000000e+01	0.802469135802469e+00
2.000000000000000e+00	0.802469135802469e+00

Outline

1. Discrete Stochastic Arithmetic (DSA) and the CADNA library
2. Error induced by perturbed data
3. Our approach: combining DSA and standard floating-point arithmetic
4. Numerical experiments
5. Pros and cons of our approach

Error induced by perturbed data

Definitions

Let $y = f(x)$ be an exact result and $\hat{y} = \hat{f}(x)$ be the associated computed result.

- The forward error is the difference between y and $\hat{y}$.
- The backward analysis tries to seek for Δx s.t. $\hat{y} = f(x + \Delta x)$.

  Δx is the backward error associated with $\hat{y}$.

  It measures the distance between the problem that is solved and the initial one.
- The condition number C of the problem is defined as:

$$C := \lim_{\varepsilon \to 0^+} \sup_{|\Delta x| \leq \varepsilon} \left[\frac{|f(x+\Delta x) - f(x)|}{|f(x)|} / \frac{|\Delta x|}{|x|} \right].$$

  It measures the effect on the result of data perturbation.

Error induced by perturbed data

The relative rounding error is denoted by $\mathbf{u}$.

- *binary64* format (double precision): $\mathbf{u} = 2^{-53}$
- *binary32* format (single precision): $\mathbf{u} = 2^{-24}$.

If the algorithm is backward-stable (*i.e.* the backward error is of the order of $\mathbf{u}$)

$$|f(x) - \hat{f}(x)|/|f(x)| \lesssim C\mathbf{u}.$$

If the input data are perturbed, *i.e.* the input data are not x but $\hat{x} = x(1+\delta)$, then one computes $\hat{f}(\hat{x})$ with

$$|f(x) - \hat{f}(\hat{x})|/|f(x)| \lesssim C(\mathbf{u} + |\delta|).$$

If $|\delta| \gg \mathbf{u}$, the rounding error generated by $\hat{f}$ is negligible w.r.t. $C|\delta|$.

Error induced by perturbed data

The relative rounding error is denoted by $\mathbf{u}$.

- *binary64* format (double precision): $\mathbf{u} = 2^{-53}$
- *binary32* format (single precision): $\mathbf{u} = 2^{-24}$.

If the algorithm is backward-stable (*i.e.* the backward error is of the order of $\mathbf{u}$)

$$|f(x) - \hat{f}(x)|/|f(x)| \lesssim C\mathbf{u}.$$

If the input data are perturbed, *i.e.* the input data are not x but $\hat{x} = x(1+\delta)$, then one computes $\hat{f}(\hat{x})$ with

$$|f(x) - \hat{f}(\hat{x})|/|f(x)| \lesssim C(\mathbf{u} + |\delta|).$$

If $|\delta| \gg \mathbf{u}$, the rounding error generated by $\hat{f}$ is negligible w.r.t. $C|\delta|$.

$\Rightarrow$ Estimating this rounding error may be avoided.

Outline

Combining DSA and standard floating-point arithmetic

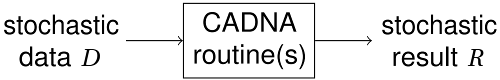
Computation routines are executed in a code that is controlled using DSA.

Their input data are affected by errors (rounding errors and/or measurement errors).

We compare 2 kinds of computation:

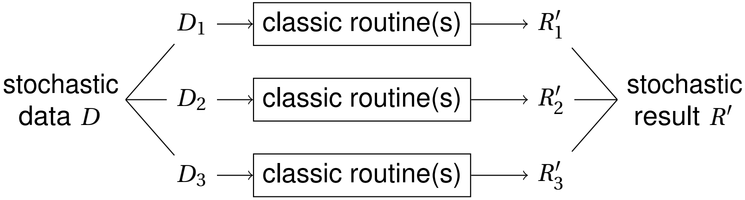
- with a call to CADNA routines
- with 3 calls to classic routines.

Computation with a call to CADNA routines



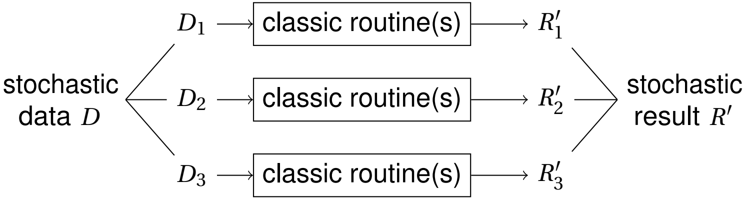
- D and R consist in stochastic arrays (each element is a triplet).
- Every arithmetic operation is performed 3 times with the random rounding mode.

Our approach: computation with 3 calls to classic routines



- input data: 3 classic floating-point arrays D_1, D_2, D_3 created from the triplets of D
- We get 3 classic floating-point arrays R'_1, R'_2, R'_3.
- A stochastic array R' created from R'_1, R'_2, R'_3 can be used in the next parts of the code.

Our approach: computation with 3 calls to classic routines



- input data: 3 classic floating-point arrays D_1, D_2, D_3 created from the triplets of D
- We get 3 classic floating-point arrays R'_1, R'_2, R'_3.
- A stochastic array R' created from R'_1, R'_2, R'_3 can be used in the next parts of the code.

$\Rightarrow$ we compare the number of correct digits (estimated by CADNA) in R and R'

Outline

Accuracy comparison

Data initialization

Each stochastic value is initialized as $\alpha 10^e$

- α: random variable uniformly distributed in $[-1,1]$
- e: integer randomly generated in $\{0,...,E\}$ (DP: $E=20$, SP: $E=3$)

⇒ generation of random data with **different orders of magnitude**.

Data perturbation

Each input value is perturbed with a **relative error** δ using a CADNA function

Accuracy comparison

Data initialization

Each stochastic value is initialized as $\alpha 10^e$

- α: random variable uniformly distributed in $[-1,1]$
- e: integer randomly generated in $\{0,...,E\}$ (DP: $E=20$, SP: $E=3$)

$\Rightarrow$ generation of random data with **different orders of magnitude**.

Data perturbation

Each input value is perturbed with a **relative error** δ using a CADNA function

Accuracy analysis

For $i=1,\dots,n^2$ (matrix mult.) or for $i=1,\dots,n$ (matrix-vector mult.) we analyze:

- the accuracy C_{R^i} of the element R^i of R
- the accuracy $C_{R'^i}$ of the element R'^i of R'
- $\Delta^i = \left|C_{R^i} - C_{R'^i}\right|$

Accuracy comparison for matrix multiplication

Multiplication of square random matrices of size 500:

δ	accuracy of R		accuracy difference between R & R'	
	mean	min-max	mean	max
double precision				
1.e-14	13.9	9-15	2.5e-02	2
1.e-13	12.8	8-15	5.8e-03	1
1.e-12	11.9	7-14	4.2e-04	1
1.e-11	10.9	6-13	2.4e-05	1
single precision				
1.e-6	5.6	1-7	2.3e-1	2
1.e-5	4.8	0-7	1.9e-2	2
1.e-4	3.7	0-6	2.8e-3	1
1.e-3	2.8	0-5	2.8e-4	1

- As the order of magnitude of $\delta \nearrow$ the mean accuracy $\searrow$ by 1 digit
- High perturbation in single precision $\Rightarrow$ low accuracy on the results
- Low difference between the accuracy of R & R'

Accuracy comparison for matrix-vector multiplication

Multiplication of a square random matrix of size 1000 with a vector:

δ	accuracy of R		accuracy difference between R & R'	
	mean	min-max	mean	max
double precision				
1.e-14	13.9	12-15	4.6e-02	1
1.e-13	12.7	11-14	7.0e-03	1
1.e-12	11.8	10-13	0	0
1.e-11	10.9	9-12	0	0
single precision				
1.e-6	5.5	3-7	3.2e-1	2
1.e-5	4.8	2-6	2.4e-2	1
1.e-4	3.7	1-5	7.0e-3	1
1.e-3	2.8	0-4	1.0e-3	1

- As the order of magnitude of $\delta \nearrow$ the mean accuracy $\searrow$ by 1 digit
- High perturbation in single precision $\Rightarrow$ low accuracy on the results
- The accuracy difference between R & R' remains low
  (in double precision, all the results have the same accuracy if $\delta \geq 10^{-12}$)

Performance comparison

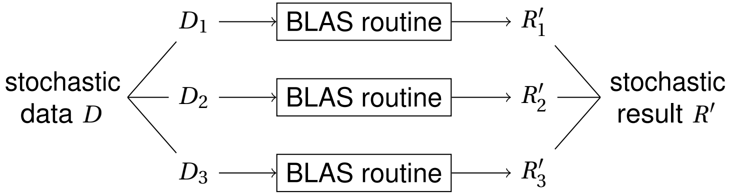
Matrix and matrix-vector multiplication

We analyze the performance of various double precision codes.

- **“CADNA"**:
  naive sequential multiplication with CADNA
- **“naive seq"**:
  our approach using a sequential naive multiplication
- **“naive OMP"**:
  our approach using a naive parallel (OpenMP, 4 cores) multiplication
- **“MKL seq"**:
  our approach using a sequential BLAS routine from the Intel MKL library
- **“MKL OMP"**:
  our approach using a parallel (OpenMP, 4 cores) MKL BLAS routine

Array copies except with CADNA

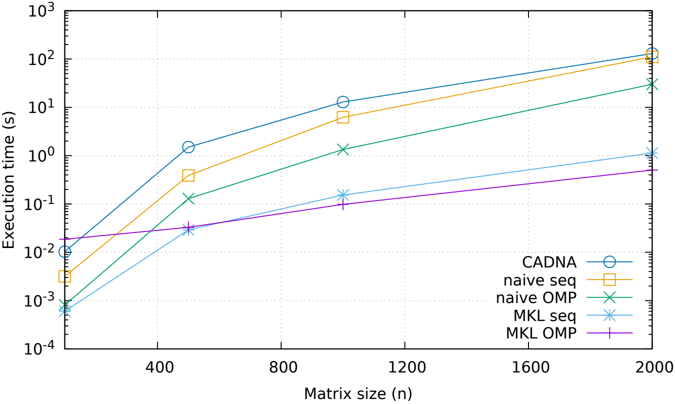
Array copies in our experiments



- Conversions: array-of-structures ↔ structure-of-arrays
  - before the BLAS routine: stochastic array → 3 classic arrays
  - after the BLAS routine: 3 classic arrays → stochastic array
- Worst case (maximum array copy cost in total execution time)
  BLAS routines continuously used
  ⇒ array copies only before and after them
- Both computation and array copies parallelized in the OpenMP codes

Performance for matrix multiplication

Execution time including matrix multiplications and array copies:



- Despite memory copies, the codes using 3 classic matrix multiplications perform better than the CADNA routine.
- For matrices of size 2000, the MKL OpenMP implementation outperforms the CADNA routine by a factor 294.

Performance for matrix multiplication

- Most of the execution time is spent in matrix multiplication.

Performance for matrix multiplication

CADNA vs our approach with MKL OMP

Core i7-8650U (1.9 GHz, 4 cores), n=2000:

	CADNA	Proposed w/ MKL OMP	Speedup
Comp	130	0.393	331x
Copy	–	0.0495	–
Total	130	0.4425	294x

Dual-socket Xeon Gold 6126 (2.6 GHz, 12 cores×2), n=5000:

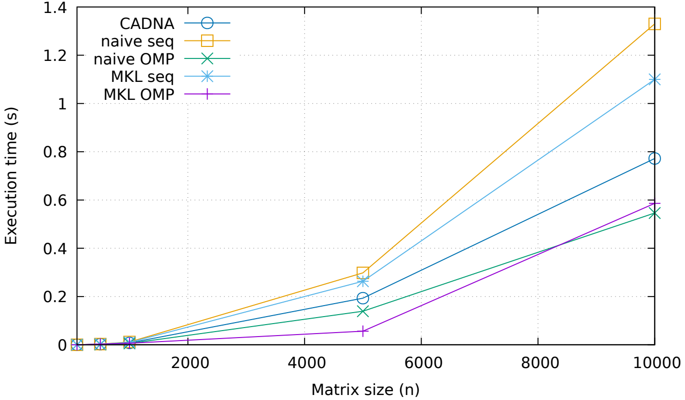
	CADNA	Proposed w/ MKL OMP	Speedup
Comp	2520	0.563	4476x
Copy	–	0.0889	–
Total	2520	0.652	3865x

On large scale:

- the performance gain increases
- the array copy cost becomes visible

Performance for matrix-vector multiplication

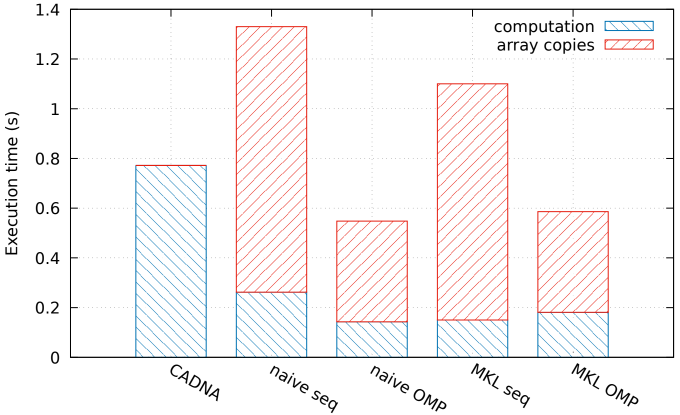
Execution time including matrix-vector multiplications and array copies:



- The CADNA routine performs better than the other sequential codes.
- From a certain matrix size, the OpenMP codes that use classic floating-point arithmetic perform better than the CADNA code.

Performance for matrix-vector multiplication

Execution time for matrices of size 10000:



- In the sequential codes that use classic floating-point arithmetic the main part of the execution time is spent in array copies.

Outline

Pros and cons

Pros

- performance gain:
  - DSA operations are avoided
  - use of vendor optimized libraries
- applicability:
  - no code translation to a CADNA version

Cons

we lose CADNA features:

- instability detection
- accuracy improvement

Instability detection

Without CADNA:

- numerical instabilities are not detected ☹
- results with no correct digits appear as numerical noise ☺

Example: matrix multiplication with catastrophic cancellations

Input data: square matrices A & B of size 10 in double precision

- 1st line of A: $[1,...,1,-1,...,-1]$ (1st half: 1, 2nd half: -1)
- each element of B set to 1
- A and B pertubed with a relative error $\delta = 10^{-12}$

Results: $C = A * B$ with CADNA, $C' = A * B$ without CADNA

- 1st line of C and C': @.0 (numerical noise, triplet with no common digits)

With CADNA:

- 10 catastrophic cancellations are detected.

Accuracy improvement with CADNA

Example: Gauss algorithm with pivoting

Input data:

We solve in single precision the system $Ax = b$ with

$$A = \begin{pmatrix} 21 & 130 & 0 & 2.1 \\ 13 & 80 & 4.74\ 10^8 & 752 \\ 0 & -0.4 & 3.9816\ 10^8 & 4.2 \\ 0 & 0 & 1.7 & 9\ 10^{-9} \end{pmatrix} \quad b = \begin{pmatrix} 153.1 \\ 849.74 \\ 7.7816 \\ 2.6\ 10^{-8} \end{pmatrix}$$

A and b perturbed with a relative error $\delta = 10^{-6}$

Results: x with CADNA, x' without CADNA

$$x = \begin{pmatrix} \text{0.100E+001} \\ \text{0.999E+000} \\ \text{0.999999E-008} \\ \text{0.999999E+000} \end{pmatrix} \quad x' = \begin{pmatrix} \text{@.0} \\ \text{@.0} \\ \text{@.0} \\ \text{0.999999E+000} \end{pmatrix} \quad x_{exact} = \begin{pmatrix} 1 \\ 1 \\ 10^{-8} \\ 1 \end{pmatrix}$$

Accuracy improvement with CADNA

Example: Gauss algorithm with pivoting

Results: x with CADNA, x' without CADNA

$$x = \begin{pmatrix} \text{0.100E+001} \\ \text{0.999E+000} \\ \text{0.999999E-008} \\ \text{0.999999E+000} \end{pmatrix} \quad x' = \begin{pmatrix} \text{@.0} \\ \text{@.0} \\ \text{@.0} \\ \text{0.999999E+000} \end{pmatrix} \quad x_{exact} = \begin{pmatrix} 1 \\ 1 \\ 10^{-8} \\ 1 \end{pmatrix}$$

Test for pivoting: if $(|A_{i,j}| > p_{max})$ ...

With CADNA a non-significant element is not chosen as a pivot.

Instabilities detected by CADNA:

```
There are 3 numerical instabilities
1 UNSTABLE BRANCHING(S)
1 UNSTABLE INTRINSIC FUNCTION(S)
1 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)
```

Conclusions/Perspectives

- In a code controlled using CADNA, if computation-intensive routines are run with perturbed data,
  - classic BLAS routines can be executed 3 times instead of the CADNA routines with almost no accuracy difference on the results
  - the performance gain can be high with BLAS routines from an optimized library
  - but we lose the instability detection.

- The same conclusions would be valid with an HPC code using MPI.

  In the same conditions (computation-intensive routines & perturbed data) CADNA-MPI routine $\Rightarrow$ optimized floating-point MPI routines.

- Application of our approach to real-life examples with realistic data sets.

Thanks for your attention!