# Benefits of stochastic arithmetic in high performance simulations and arbitrary precision codes

Fabienne Jézéquel
LIP6, Sorbonne Université, France
http://www.lip6.fr/Fabienne.Jezequel

SCAN 2020 (19th International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations)
13-15 Sept. 2021



# Jean Vignes
1933-2021

Jean Vignes passed away in August. He proposed the CESTAC method and stochastic arithmetic for rounding error estimation.

Jean Vignes is at the origine of most results presented in this talk.

# Rounding error analysis

## Several approaches

- Interval arithmetic
  - guaranteed bounds for each computed result
  - the error may be overestimated
  - specific algorithms
  - ex: INTLAB [Rump'99]
- Static analysis
  - no execution, rigorous analysis, all possible input values taken into account
  - not suited to large programs
  - ex: FLUCTUAT [Goubault & al.'06], FLDLib [Jacquemin & al.'19]
- Probabilistic approach
  - estimates the number of correct digits of any computed result
  - can be used in HPC programs
  - requires no algorithm modification
  - ex: CADNA [Chesneaux'90], VERIFICARLO [Denis & al.'16], VERROU [Févotte & al.'17]

# Outline

1. Discrete Stochastic Arithmetic (DSA) and related tools
2. Benefits of DSA for the computation of multiple polynomial roots
3. Fast numerical validation of HPC codes

# The CESTAC method [Vignes & al'74], [Chesneaux & al'88]

Classic arithmetic

$A \oplus B \longrightarrow R$

$R$ =3.14237654356891

CESTAC method

Random rounding

$A_1 \oplus B_1$  $\longrightarrow R_1$

$A_2 \oplus B_2$  $\longrightarrow R_2$

$A_3 \oplus B_3$  $\longrightarrow R_3$

$R_1$ =**3.14**1354786390989
$R_2$ =**3.14**3689456834534
$R_3$ =**3.14**2579087356598

- each operation executed $N = 3$ times with a random rounding mode

# The CESTAC method [Vignes & al'74], [Chesneaux & al'88]

Classic arithmetic

$A \oplus B \longrightarrow R$

$R$ =3.14237654356891

CESTAC method

Random rounding

$A_1 \oplus B_1$  $\longrightarrow R_1$

$A_2 \oplus B_2$  $\longrightarrow R_2$

$A_3 \oplus B_3$  $\longrightarrow R_3$

$R_1$ =**3.14**1354786390989
$R_2$ =**3.14**3689456834534
$R_3$ =**3.14**2579087356598

- each operation executed $N = 3$ times with a random rounding mode
- number of correct digits in the result estimated using Student's test with the confidence level 95%

# How to optimize 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);
}
```

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

# How to optimize 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);
}
```

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

It would be optimal to stop when $X - Y$ is numerical noise.

Such a stopping criterion

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

# The concept of computational zero [Vignes'86]

## Definition

A result $R$ computed using the CESTAC method ($N$-sample) is a computational zero, denoted by @.0, if

- $\forall i, R_i = 0$ *(mathematical zero)*
- or $R$ has no correct digits *(numerical noise)*.

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

# The stochastic definitions [Chesneaux'92]

An equality concept and order relations that take into account rounding errors have been introduced:

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

- $X$ is stochastically equal to $Y$, noted $X \stackrel{s}{=} Y$, iff

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

- $X$ is stochastically strictly greater than $Y$, noted $X \stackrel{s}{>} Y$, iff

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

- $X$ is stochastically greater than or equal to $Y$, noted $X \stackrel{s}{\geq} Y$, iff

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

Ex: if $X - Y$ is numerical noise, $X \stackrel{s}{>} Y$ is false, but $X \stackrel{s}{\geq} Y$ is true.

# Discrete Stochastic Arithmetic

**Discrete Stochastic Arithmetic** (DSA) is the joint use of

- the CESTAC method
- the computational zero
- the stochastic relation definitions.

[Chesneaux & al'92], [Vignes'93], [Vignes'04]

# Discrete Stochastic Arithmetic

**Discrete Stochastic Arithmetic** (DSA) is the joint use of

- the CESTAC method
- the computational zero
- the stochastic relation definitions.

[Chesneaux & al'92], [Vignes'93], [Vignes'04]

With DSA, operations are executed **synchronously**

⇒ detection of numerical instabilities
ex: `if (A>B)` with `A-B` numerical noise

⇒ optimization of stopping criteria

# The CADNA library

cadna.lip6.fr


- implements stochastic arithmetic for C/C++ or Fortran codes
- provides stochastic types (3 floating-point variables and an integer)

  `half_st` `float_st` `double_st` `quad_st`
- all operators and mathematical functions overloaded
  ⇒ few modifications in user programs
- overhead: $4\times$ memory, $\approx 10\times$ time
- efforts to $\searrow$ time overhead (ex: implicit rounding mode change)
- support for MPI, OpenMP, GPU, vectorised codes

[Chesneaux'90], [Jézéquel & al'08], [Lamotte & al'10], [Eberhart & al'18],...

# 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

[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

- recent improvement: control of operations mixing different precisions

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

$$C = A \oplus B$$

35 bits 23 bits 47 bits

⇒ accuracy estimation on FPGA

[1] www.mpfr.org

# An example without/with CADNA

Computation of $P(x,y) = 9x^4 - y^4 + 2y^2$ [Rump'83]

```
#include <iostream>
using namespace std;
double rump(double x, double y) {
  return 9.0*x*x*x*x - y*y*y*y + 2.0*y*y;
}
int main() {
  cout.precision(15);
  cout.setf(ios::scientific,ios::floatfield);
  double x, y;
  x = 10864.0;
  y = 18817.0;
  cout<<"P1="<<rump(x, y)<< endl;
  x = 1.0/3.0;
  y = 2.0/3.0;
  cout<<"P2="<<rump(x, y)<< endl;
  return 0;
}
```

# An example without/with CADNA

Computation of $P(x,y) = 9x^4 - y^4 + 2y^2$ [Rump'83]

```
#include <iostream>
using namespace std;
double rump(double x, double y) {
  return 9.0*x*x*x*x - y*y*y*y + 2.0*y*y;
}
int main() {
  cout.precision(15);
  cout.setf(ios::scientific,ios::floatfield);
  double x, y;
  x = 10864.0;
  y = 18817.0;
  cout<<"P1="<<rump(x, y)<< endl;
  x = 1.0/3.0;
  y = 2.0/3.0;
  cout<<"P2="<<rump(x, y)<< endl;
  return 0;
}
```

P1=2.000000000000000e+00

P2=8.024691358024690e–01

```
#include <iostream>

using namespace std;
double   rump(double   x, double   y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main() {
  cout.precision(15);
  cout.setf(ios::scientific,ios::floatfield);

  double   x, y;
  x=10864.0; y=18817.0;
  cout«"P1="«rump(x, y)«endl;
  x=1.0/3.0; y=2.0/3.0;
  cout«"P2="«rump(x, y)«endl;

  return 0;
}
```

```
#include <iostream>
#include <cadna.h>
using namespace std;
double   rump(double   x, double   y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main() {
  cout.precision(15);
  cout.setf(ios::scientific,ios::floatfield);
  cadna_init(-1);
  double   x, y;
  x=10864.0; y=18817.0;
  cout«"P1="«rump(x, y)«endl;
  x=1.0/3.0; y=2.0/3.0;
  cout«"P2="«rump(x, y)«endl;
  cadna_end();
  return 0;
}
```

```
#include <iostream>
#include <cadna.h>
using namespace std;
double   rump(double   x, double   y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main() {
  cout.precision(15);
  cout.setf(ios::scientific,ios::floatfield);
  cadna_init(-1);
  double   x, y;
  x=10864.0; y=18817.0;
  cout«"P1="«rump(x, y)«endl;
  x=1.0/3.0; y=2.0/3.0;
  cout«"P2="«rump(x, y)«endl;
  cadna_end();
  return 0;
}
```

```
#include <iostream>
#include <cadna.h>
using namespace std;
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() {
  cout.precision(15);
  cout.setf(ios::scientific,ios::floatfield);
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  cout«"P1="«rump(x, y)«endl;
  x=1.0/3.0; y=2.0/3.0;
  cout«"P2="«rump(x, y)«endl;
  cadna_end();
  return 0;
}
```

# Results with CADNA

only correct digits are displayed

CADNA_C software
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= 0.80246913580246 9E+000

---

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

# PROMISE (PRecision OptiMISE)

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.

[Graillat & al.'19], [Jézéquel & al.'21]

# Outline



S. Graillat, F. Jézéquel, E. Queiros Martins, M. Spyropoulos, Computing multiple roots of polynomials in stochastic arithmetic with Newton method and approximate GCD, 2021.

`http://hal.archives-ouvertes.fr/hal-03274453`

# Newton based methods

To compute a root with multiplicity $m$ of a polynomial $P$ from an initial approximation $x_0$:

## Newton method

$$x_{n+1} = x_n - \frac{P(x_n)}{P'(x_n)}$$

- single root ($m = 1$): quadratic convergence ☺
- multiple root ($m > 1$): linear convergence ☹

## modified Newton method

$$x_{n+1} = x_n - m\frac{P(x_n)}{P'(x_n)}$$

- quadratic convergence for $m > 1$ ☺
- $m$ is required ☹

# Computation of the (multiple) roots of a polynomial $P$

## Algorithm based on GCD and Newton method

1. $G(x) = gcd(P(x), P'(x))$
2. $P(x)/G(x)$ has single roots → Newton method

Computing multiple polynomial roots and polynomial GCD: ill-posed problems in floating-point arithmetic ⚠

## Effects of a perturbation $\varepsilon$

Let $P_\varepsilon(x) = (x-1)^2 - \varepsilon$

- $\varepsilon = 0$: 1 double root, $\varepsilon > 0$: 2 single roots $1 \pm \sqrt{\varepsilon}$

Let $Q(x) = (x-1)$

- $\varepsilon = 0$: $\gcd(P_\varepsilon(x), Q(x)) = x-1$, $\varepsilon \neq 0$: $\gcd(P_\varepsilon(x), Q(x)) = 1$

# In floating-point arithmetic...

## approximate GCD, quasi-GCD

[Schönhage'85], [Noda& al'91], [Emiris & al'97], [Beckermann & al'98], [Chin & al'98], [Karmarkar'98], [Pan'01], [Zeng'11], [Nagasaka'21]

- related to the computation of the exact GCD of close polynomials
- costful ☹

We propose stochastic versions of

- polynomial GCD
- polynomial Euclidean division
- Newton method

that lead to accurate computation of (possibly multiple) polynomial roots.

# Newton method in classic floating-point arithmetic

Newton iterations: $x_{n+1} = x_n - P(x_n)/P'(x_n)$

**Algorithm 1:** Newton method in classic floating-point arithmetic

$x = x_0$
**do**
  $y = x$
  $x = y - P(y)/P'(y)$
**while** $|x - y| > \varepsilon$ ← *suitable $\varepsilon$?*

# Newton method in stochastic arithmetic

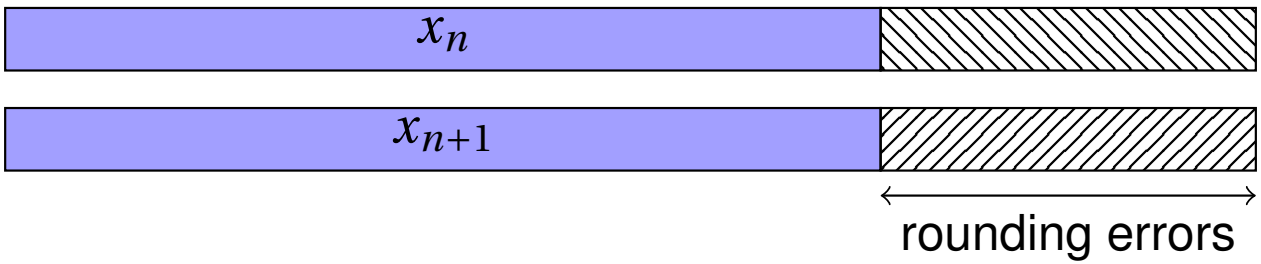
Newton iterations: $x_{n+1} = x_n - P(x_n)/P'(x_n)$

**Algorithm 2:** stochastic Newton method: st-Newton

$x = x_0$
**do**
  $y = x$
  $x = y - P(y)/P'(y)$
**while** $x \stackrel{s}{\neq} y$ ← *optimal stopping criterion*

When $x_n \stackrel{s}{=} x_{n+1}$

- the digits in $x_n$ and $x_{n+1}$ not affected by rounding errors are the same
- $x_n - x_{n+1}$ is numerical noise, so we avoid useless iterations.



# Number of exact digits on the computed result

## Theorem [Graillat & al'16]

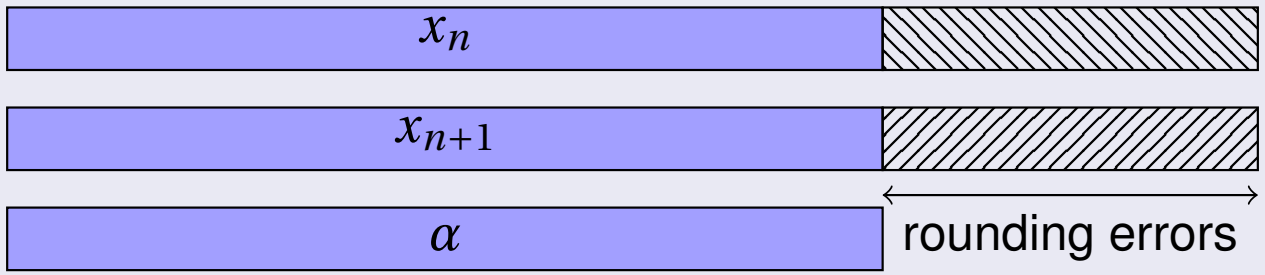
If $x_n$ and $x_{n+1}$ be two successive approximations computed using Newton method of a polynomial single root $\alpha$, $C_{x_n,x_{n+1}} \sim_\infty C_{x_n,\alpha}$.

$C_{a,b}$: number of significant digits common to $a$ and $b$.

$\Rightarrow$ In the convergence zone, the digits common to $x_n$ and $x_{n+1}$ are also in common with the exact root $\alpha$.

## In stochastic arithmetic

When $x_n \stackrel{s}{=} x_{n+1}$, the digits in $x_n$ and $x_{n+1}$ not affected by rounding errors are the same and are in common with the exact root $\alpha$.




$\Rightarrow$ In the computed result, the digits estimated correct by DSA are those of $\alpha$.

**Algorithm 3:** Stochastic polynomial Euclidean division: st-Euclidean-div

**Data:** polynomial $A$ of degree $n$, polynomial $B$ of degree $m$, with $0 \leq m \leq n$

**Result:** polynomials $Q$ and $R$ s.t. $A = B * Q + R$

$R = A$

**for** $i = n - m$ *to 0* **do**

  **if** $lc(R) \not\stackrel{s}{=} 0$ ← *we discard numerical noise* **then**

    // tests if $degree(R) = m + i$, $lc(R)$: leading coefficient of $R$

    $q_i = lc(R)/lc(B)$

    $R = R - q_i x^i B$

  **end**

  **else**

    $q_i = 0$

  **end**

**end**

**return** $Q = \sum_{i=0}^{n-m} q_i x^i$ *and* $R$

**Algorithm 4:** stochastic polynomial GCD: st-gcd

**Data:** polynomials $A$ and $B$
**Result:** stochastic GCD of $A$ and $B$
$R_0 = A$
$R_1 = B$
$i = 1$
**while** $R_i \stackrel{s}{\neq} 0$ **do**
  $R_{i+1}$ =remainder (st-Euclidean-div($R_{i-1}$, $R_i$))  // $R_{i+1} = R_{i-1}$ Mod $R_i$
  $i = i + 1$
**end**
**return** $R_{i-1}$

## Computation of $gcd(P, P')$ with $P(x) = (3x-1)^n$

$R_2$ should be null, and the returned result should be $R_1 = P'$ of degree $n-1$.

- In classic floating-point arithmetic, $R_2$ may have non-zero coefficients with different orders of magnitude (stopping criterion ?)
  $\Rightarrow$ unexpected iterations, incorrect results
- In stochastic arithmetic, $R_2$ coefficients are @.0
  $\Rightarrow$ returned polynomial with correct degree.

**Algorithm 5:** Computation of polynomial roots based on st-gcd and st-Newton

**Data:** a polynomial $P$ and an array $X_0$ of initial approximations of its roots
**Result:** an array $X$ of approximations of the roots
$G$ =st-gcd$(P, P')$
$Q$ =quotient(st-Euclidean-div($P$, $G$))
$d$ =degree($Q$)
**if** $d \leq 4$ **then**
  computation of $X$ using adequate formulas
  `// Cardan's method if d is 3, Ferrari's if d is 4 [Kurosh'88]`
**end**
**else**
  **for** *i=1 to* $d$ **do**
    $X[i]$=st-Newton($Q$, $X_0[i]$)
  **end**
**end**

- $Q$ computed once whatever the number of roots.
- If $d \geq 5$, Newton method applied to a (low-degree) polynomial having only single roots.

# Comparison with modified Newton method

Computation of multiple roots based on modified Newton iterations:

$$x_{n+1} = x_n - m\frac{P(x_n)}{P'(x_n)}$$

$\Rightarrow$ $m$ is required

## Proposition [Yakoubsohn'03]

Let $(x_n)$ be the sequence of approximations computed using Newton method of the root $\alpha$ of multiplicity $m$ of a polynomial. Then

$$\lim_{i\to\infty} \frac{x_{i+2} - x_{i+1}}{x_{i+1} - x_i} = 1 - \frac{1}{m}.$$

$\Rightarrow$ $m$ can be estimated from 3 successive iterates of Newton method

## Algorithm in stochastic arithmetic [Graillat & al'16]

- Newton method $\Rightarrow$ m
- modified Newton applied iteratively
  working precision doubled at each iteration

# Numerical experiments

Carried out using SAM

## Working precision (refers also to the initial precision in [Graillat & al'16])

$$\text{Precision} = \text{Requested\_accuracy} * \text{Rate} \quad \text{with} \quad \text{Rate} > 1$$

## Examples of tested polynomials (more in [Graillat & al'21])

$$Q_n(x) = (3x-2)^{n_1}(7x-3)^{n_2}(13x-4)^{n_3}(19x-2)^{n_4}(23x-1)^{n_5}$$

Roots denoted as $\beta_1 = 2/3$, $\beta_2 = 3/7$, $\beta_3 = 4/13$, $\beta_4 = 2/19$, and $\beta_5 = 1/23$

Degrees of polynomials $Q_n$:

| $n = \sum_{i=1}^{5} n_i$ | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ |
|---|---|---|---|---|---|
| 55 | 13 | 12 | 11 | 10 | 9 |
| 105 | 18 | 19 | 21 | 22 | 25 |
| 5000 | 1000 | | | | |

# Computation based on st-gcd and st-Newton

| Poly. | #Digits | | Rate | Performance | |
|---|---|---|---|---|---|
| | Requested | Exact | | Time (s) | Ratio |
| $Q_{55}$ | 100 | 109-111 | 1.3 | 3.63E-3 | 5.8e+01 |
| | 500 | 530-532 | 1.1 | 8.51E-3 | 1.1e+03 |
| | 1000 | 1079-1081 | 1.1 | 1.60E-2 | 3.5e+03 |
| $Q_{105}$ | 100 | 107-109 | 1.3 | 5.04e-03 | 1.8e+02 |
| | 500 | 577-579 | 1.2 | 1.36e-02 | 5.6e+03 |
| | 1000 | 1276-1278 | 1.3 | 2.26e-02 | 2.3e+04 |
| $Q_{5000}$ | 100 | 104-106 | 1.5 | 1.77e-01 | N.A. |
| | 500 | 503-506 | 1.1 | 3.35e-01 | N.A. |
| | 1000 | 1054-1056 | 1.1 | 6.40e-01 | N.A. |
| | 5000 | 5454-5456 | 1.1 | 5.00e+00 | N.A. |

- Exact: #decimal digits in common with the exact roots
  $=$ #digits estimated by DSA, depends on the root
- Rate: minimum rate s.t. accuracy requirement satisfied
  (determined starting from 1.1 and increasing by steps of 0.1)
- Time to compute all the roots
- Performance ratio: w.r.t. modified Newton method

# Computation using modified Newton method

In [Graillat & al'16]: polynomial with at most 4 roots of degree 52.

| Poly. | Root | #Digits Requested | #Digits Exact | Rate | Time (s) |
|---|---|---|---|---|---|
| $Q_{105}$ | $\beta_1$ | 100 | 100 | 2.7 | 1.01e-01 |
| | $\beta_2$ | | 100 | 2.9 | 1.20e-01 |
| | $\beta_3$ | | 100 | 4.1 | 2.48e-01 |
| | $\beta_4$ | | 102 | 3.1 | 1.64e-01 |
| | $\beta_5$ | | 102 | 4.1 | 2.63e-01 |
| | $\beta_1$ | 500 | 506 | 3.8 | 8.33e+00 |
| | $\beta_2$ | | 507 | 3.9 | 9.04e+00 |
| | $\beta_3$ | | 508 | 5.1 | 1.75e+01 |
| | $\beta_4$ | | 506 | 5.0 | 1.70e+01 |
| | $\beta_5$ | | 501 | 5.8 | 2.48e+01 |

- Exact: #decimal digits in common with the exact roots
  $\neq$ #digits estimated by DSA
- higher working precision, depends on the root
- higher execution time
- fails for some polynomials (necessary working precision too high)

# To sum up

- The proposed algorithm efficiently and accurately computes multiple polynomial roots
- Thanks to DSA
  - numerical noise discarded
  - optimal stopping criteria
  - digits estimated correct by DSA in common with the exact roots

# Outline



F. Jézéquel, S. Graillat, D. Mukunoki, T. Imamura, R. Iakymchuk, Can we avoid rounding-error estimation in HPC codes and still get trustworthy results?, NSV'20, LNCS, 12549, p. 163–177, 2020.

`http://hal.archives-ouvertes.fr/hal-02925976`

# How to reduce the cost of numerical validation?

Numerical validation is crucial... but it may be costful ☹

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

# How to reduce the cost of numerical validation?

Numerical validation is crucial... but it may be 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.

# Definitions

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

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

  $\Delta x$ is the backward error associated with $\widehat{y}$.

  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 $\mathbf{u}$)

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

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

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

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

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

# 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 $\mathbf{u}$)

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

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

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

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

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

# Combining DSA and standard floating-point arithmetic

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

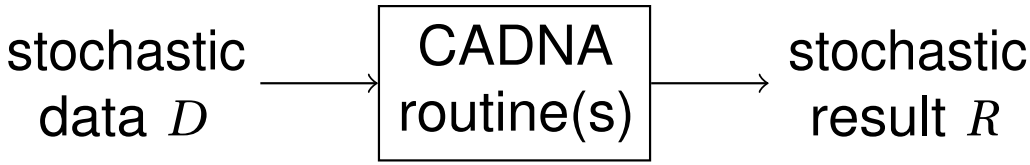
Their input data are affected by errors (rounding errors and/or measurement errors).

# 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).

Computation with a call to CADNA routines:




- $D$ and $R$ consist in stochastic arrays (each element is a triplet).

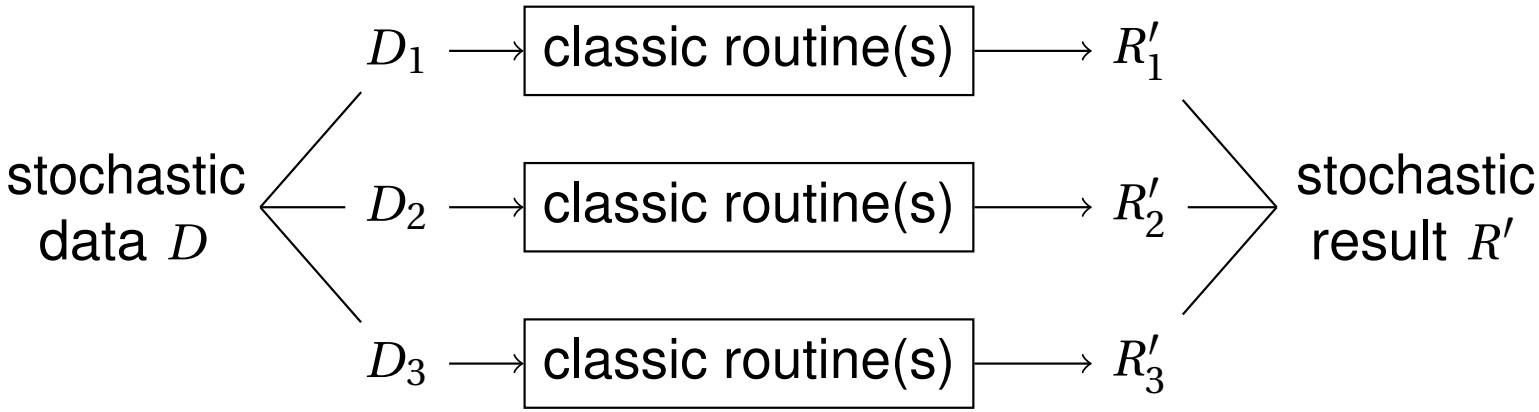
$\Rightarrow$ we compare the number of correct digits (estimated by CADNA) in $R$ and $R'$

# 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).

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.

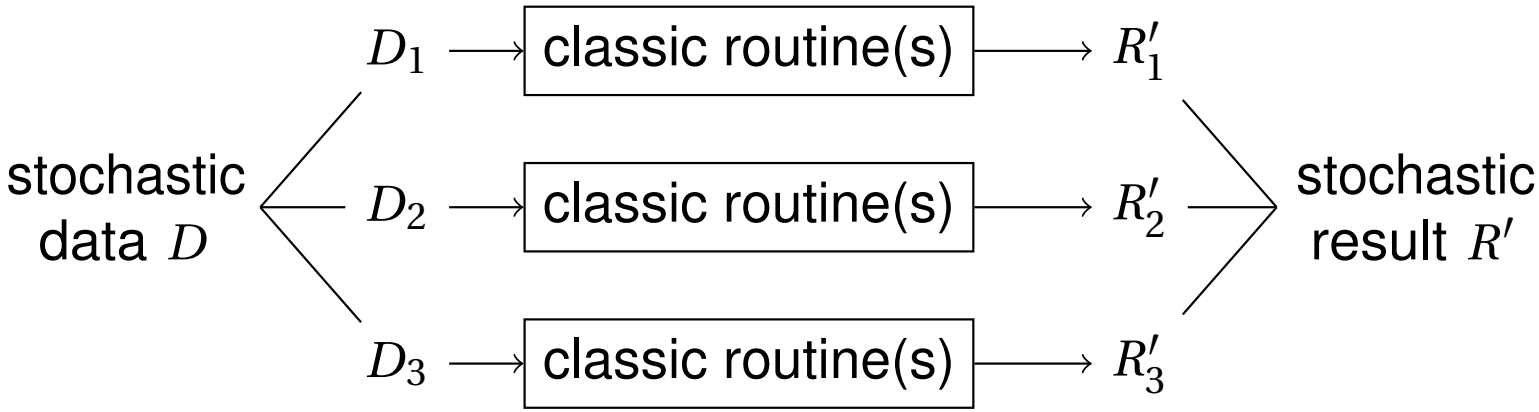
$\Rightarrow$ we compare the number of correct digits (estimated by CADNA) in $R$ and $R'$

# 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).

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 $R'$

# Accuracy comparison

## Experimental setup

Each random input value is perturbed with a relative error $\delta$.

For $i = 1, \ldots, n^2$ (matrix mult.) or for $i = 1, ..., n$ (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

in double precision

| $\delta$ | accuracy of $R$ | | accuracy difference between $R$ & $R'$ | |
|---|---|---|---|---|
| | mean | min-max | mean | max |
| Multiplication of matrices of size 500 | | | | |
| 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 |
| Multiplication of a matrix of size 1000 with a vector | | | | |
| 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 |

- As the order of magnitude of $\delta$ $\nearrow$ the mean accuracy $\searrow$ by 1 digit.
- Low difference between the accuracy of $R$ & $R'$

# Performance comparison

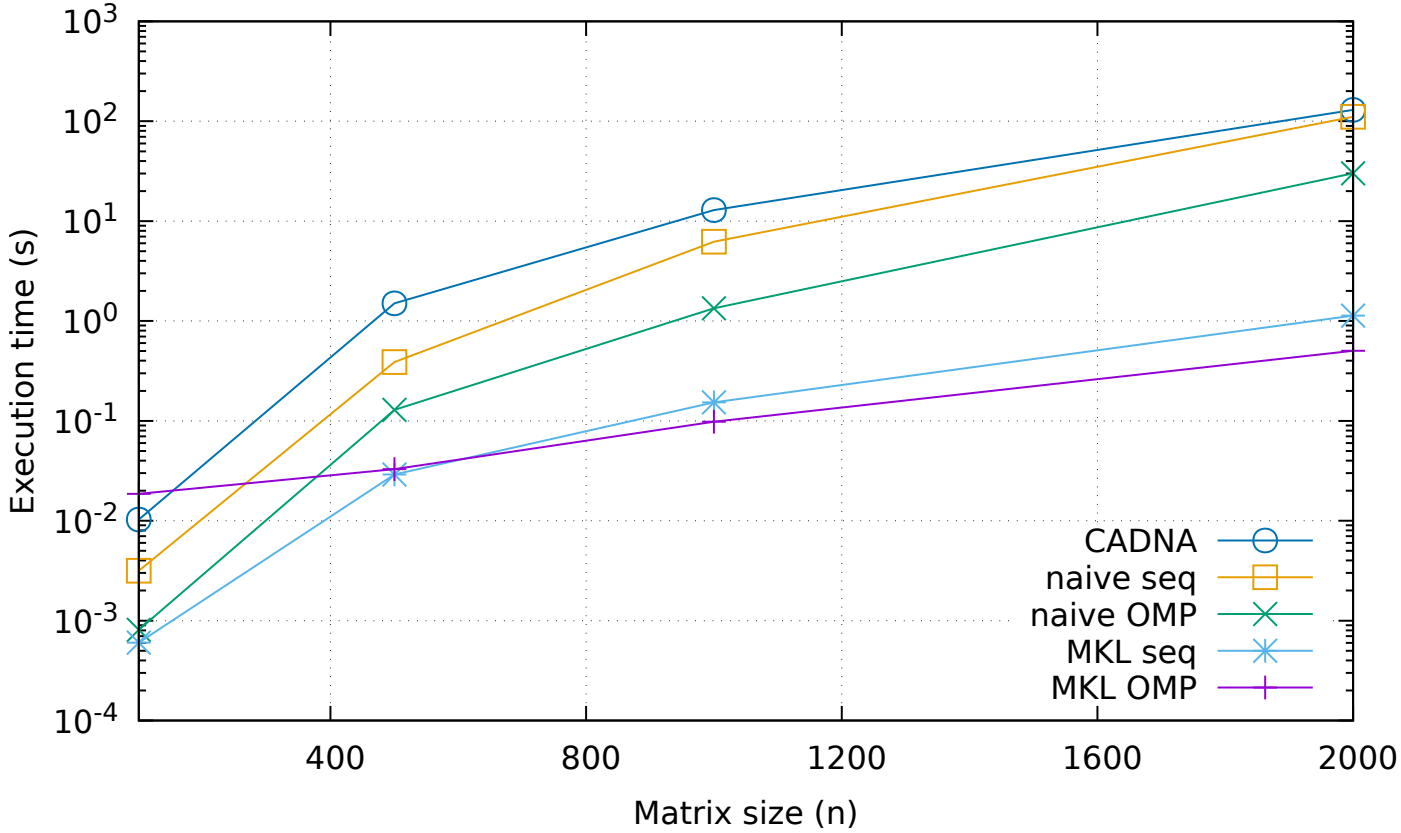
We compare the performance of the CADNA routine with codes using:

- a naive floating-point algorithm
- the Intel MKL implementation.

In both cases: sequential and OpenMP 4 cores

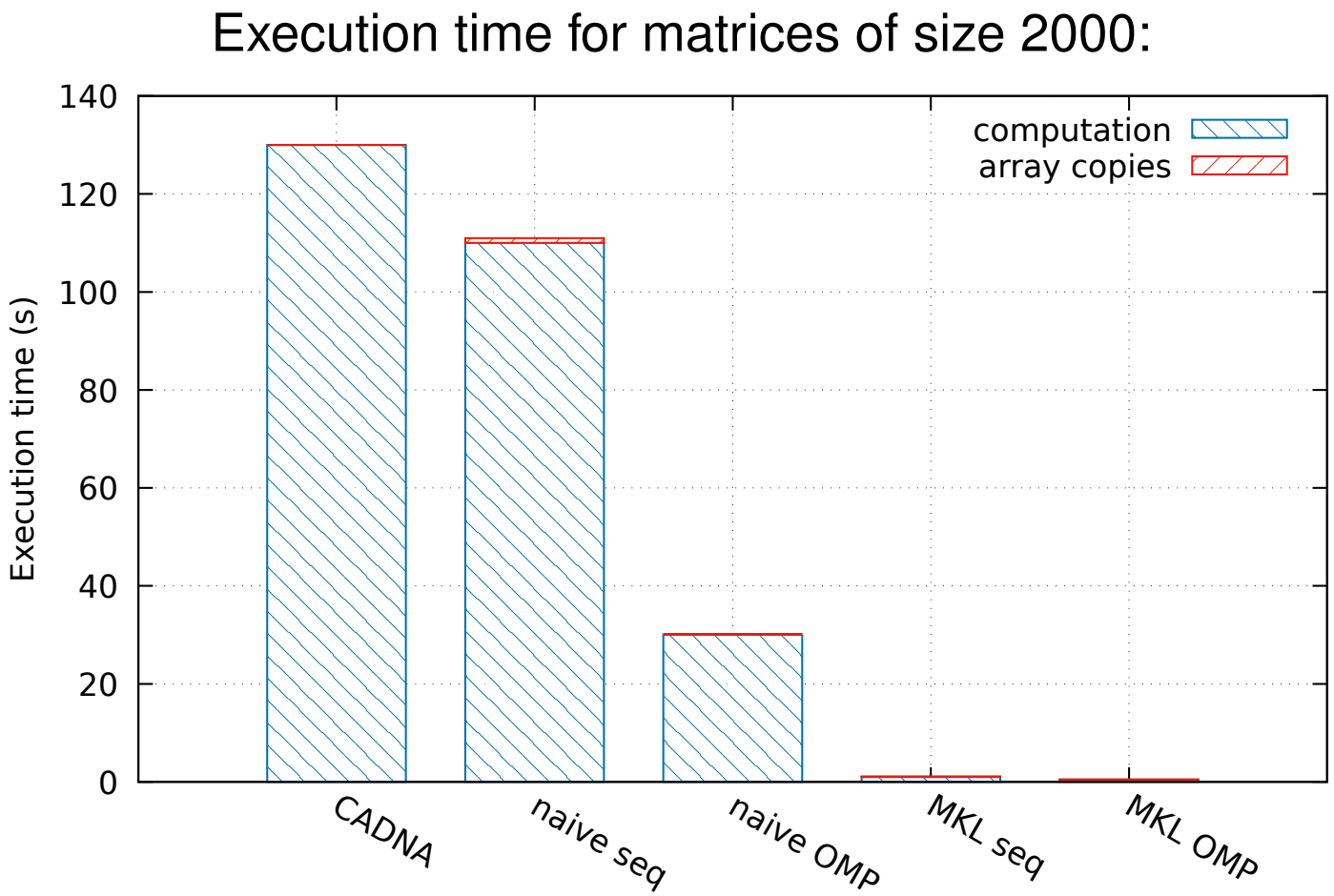
# Performance for matrix multiplication

Execution time including matrix multiplications and array 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 (this gain increases on many-cores).

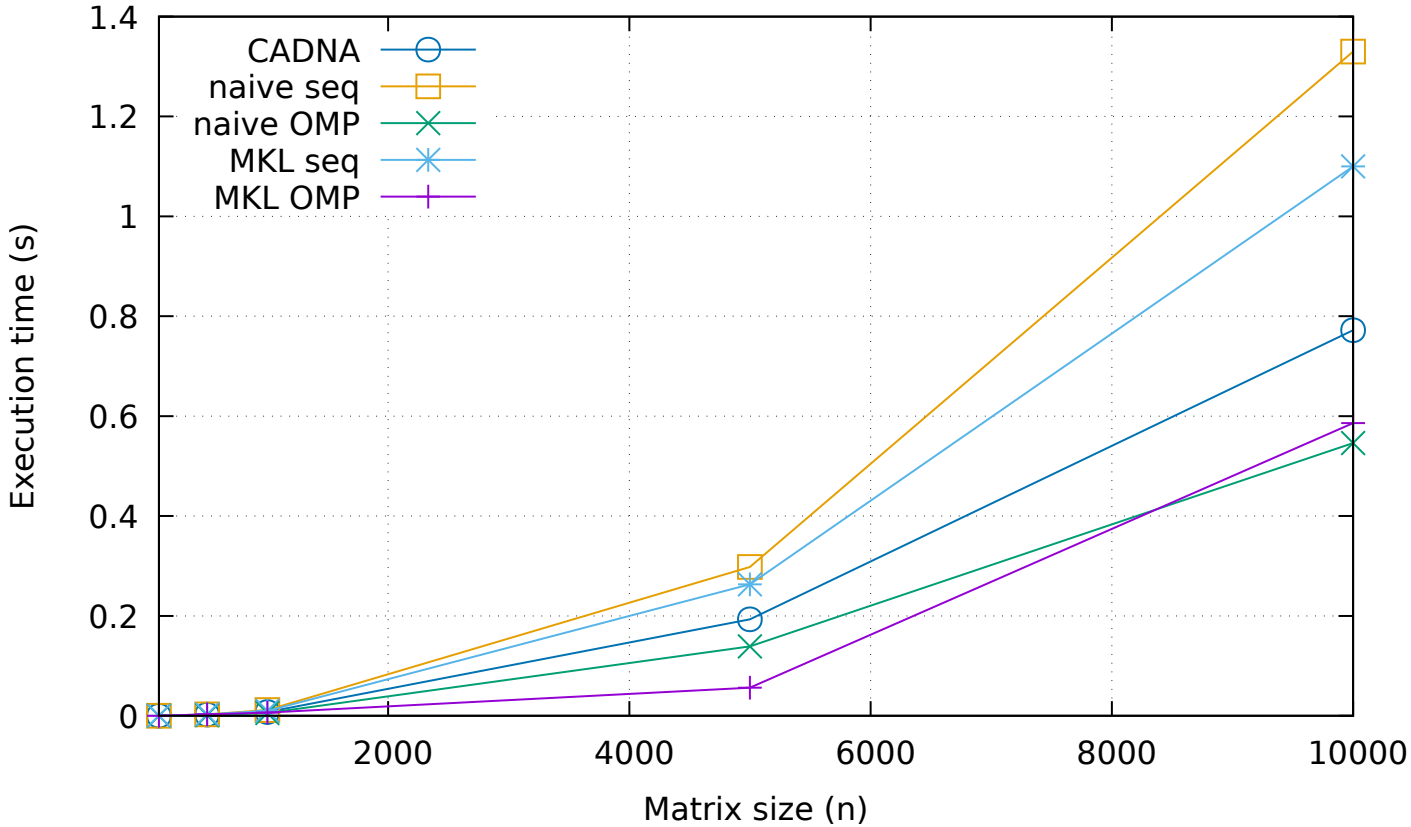
# Performance for matrix multiplication




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

# Performance for matrix-vector multiplication

Execution time including matrix-vector multiplications and array copies:




- The CADNA routine performs better than the other sequential codes.

# Performance for matrix-vector multiplication


- Except with the CADNA routine, the main part of the execution time is spent in array copies.

# To sum up

- In a code controlled using CADNA, if computation-intensive routines are run with data affected by errors,
  - 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.

  CADNA-MPI routines $\Rightarrow$ optimized floating-point MPI routines.

# Conclusion/Perspectives

## Advantages of DSA

- In one execution: accuracy of any result, complete list of numerical instabilities
- Detection of numerical noise $\Rightarrow$ optimal stopping criteria
- Easily applied to real life applications
- Support for wide range of codes (vectorised, GPU, MPI, OpenMP)

## Perspectives

- Extend our approach (3 $\times$ classic routines) to large simulation codes
- Computation of multiple polynomial roots: optimal working precision?
- Floating-point autotuning in arbitrary precision
- Combine mixed precision algorithms and floating-point autotuning

# References

## On Discrete Stochastic Arithmetic (DSA):

J. Vignes, Discrete Stochastic Arithmetic for Validating Results of Numerical Software, Num. Algo., 37, 1–4, p. 377–390, 2004.

## On the computation of multiple roots of polynomials with DSA:

S. Graillat, F. Jézéquel, E. Queiros Martins, M. Spyropoulos, Computing multiple roots of polynomials in stochastic arithmetic with Newton method and approximate GCD, 2021.
http://hal.archives-ouvertes.fr/hal-03274453

S. Graillat, F. Jézéquel, and M. S. Ibrahim, Dynamical Control of Newton's Method for Multiple Roots of Polynomials, Reliable Computing, 21, p. 117–139, 2016.
http://hal.archives-ouvertes.fr/hal-01363961

## On the numerical validation of BLAS routines with perturbed data:

F. Jézéquel, S. Graillat, D. Mukunoki, T. Imamura, R. Iakymchuk, Can we avoid rounding-error estimation in HPC codes and still get trustworthy results?, NSV'20, LNCS, 12549, p. 163–177, 2020.
http://hal.archives-ouvertes.fr/hal-02925976

## Tools related to DSA:

- CADNA: http://cadna.lip6.fr
- SAM: http://www-pequan.lip6.fr/~jezequel/SAM
- PROMISE: http://promise.lip6.fr

Thanks for your attention!