
# ADVANCED METHODS FOR STUDYING NUMERICAL STABILITY

**Nikolai Bakouta**
**Electricité de France (EDF)**
**nikolai.bakouta@edf.fr**

E-MUG Meeting - Brno – 2025, April 7-11th

# Outline

1. Introduction
2. Motivation
3. Advanced methods
4. Perspectives

# Introduction

- Verrou is an open-source tool developed by EDF R&D (project InterFLOP) that helps identify floating-point round-off errors in programs. The latest version can be downloaded from https://github.com/edf-hpc/VERROU/releases/latest.

- VERROU is an external backend tool developed for Valgrind, which is a programming tool used for memory debugging, memory leak detection, and profiling. For more information, see https://en.wikipedia.org/wiki/Valgrind. Running VERROU involves executing Valgrind with VERROU selected as an external tool:

```
$ valgrind --tool=verrou [verrou options] {program} [program options]
```

`{program}` refers to an executable of interest (e.g. `maap.exe`)

- The VERROU documentation is available as a dedicated chapter in the Valgrind manual http://edf-hpc.github.io/verrou/vr-manual.html.

# Motivation

VERROU introduces numerical noise (rounding of arithmetic operations on the order of $\pm 10^{-16}$) into the original executable.

No source code is needed. Recompile the executable with `-g` option.

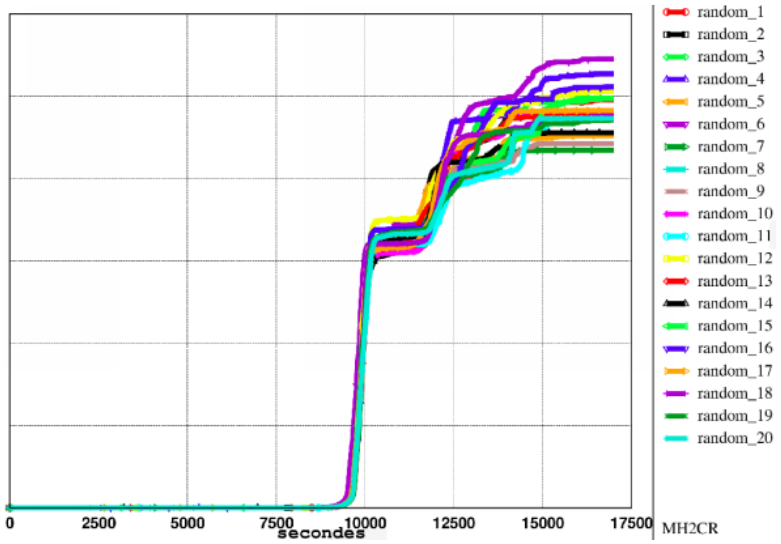
Basic usage for diagnostic: if the numerical noise amplifies, then the code is unstable.

Diagnostic:

- Run the same calculation multiple times with random rounding

```
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
valgrind --tool=verrou --rounding-mode=random ${MAAP} ${INP}
```

- Analyze discrepancies in the parameter of interest



# Motivation

The diagnostic does not improve numerical stability.

Advanced methods focus on:

- Identifying sources of instability
- Reducing instability

# Advanced methods

Controlling the scope of perturbations

**Delaying numerical noise with macros `VERROU_START_INSTRUMENTATION`**

- Requires modification of the user program's source code

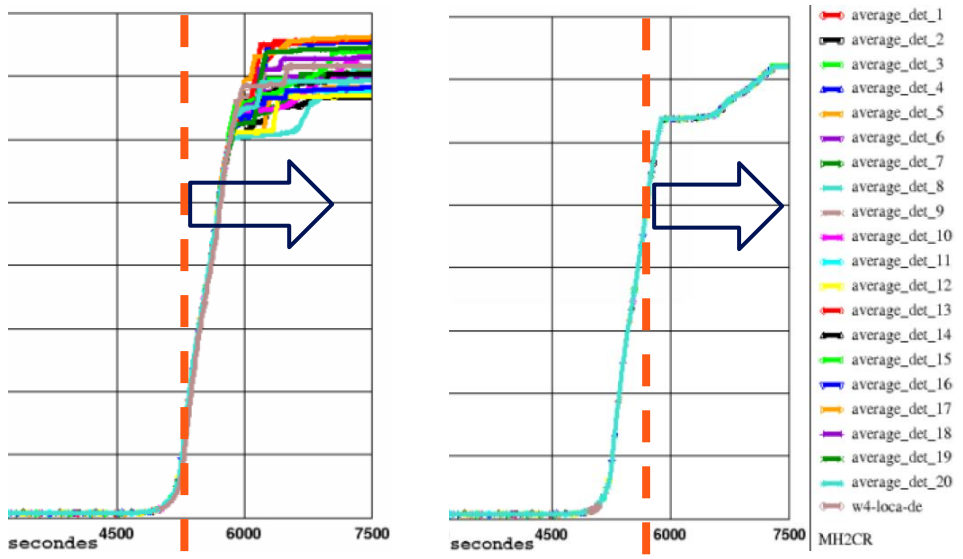
```
#include <valgrind/VERROU.h>

if(tim>start_verrou ) {
  VERROU_START_INSTRUMENTATION;
}
```

- Initially, run the program without numerical noise

```
valgrind --tool=verrou --instr-atstart=no ${MAAP} ${INP}
```

Instabilities grow before `tim`=5800 s =>




`start_verrou=`**5000** `start_verrou=`**5800**

# Advanced methods

Controlling the scope of perturbations

**Exclusion files: allows you to avoid perturbing the files described in the exclusion list.**

For example, to excluded the math library `libm-2.28.so` linked to the executable:

- Add the following text to the file `libm.ex` :

```
#FNNAME          OBJNAME
*                /lib/x86_64-linux-gnu/libm-2.28.so
```

- Run VERROU with the `--exclude` option

```
valgrind --tool=verrou --exclude=libm.ex ${MAAP} ${INP}
```

# Advanced methods

Controlling the scope of perturbations

**Limit perturbations to specific lines of the source code.**

The lines must be declared in the source list file.

For example, to perturb only the `cos` function from the `s_sin.c` file, ligne #12:

- Add the following text to the source list file `src.inc`

```
#FILENAME      LINENUM        SYMNAME
s_sin.c        12             cos
```

- Run VERROU with `--source` option

```
valgrind --tool=verrou --source=src.inc ${MAAP} ${INP}
```

EDF

# Advanced methods

**Delta-debugging: searching for source code line(s) whose perturbation produces the most significant errors.**

Let's consider a simple example:

- Perturbing all subsequent lines results in instability.

```
mwn(i) += mwn(i2) ± 10-16;
uwn(i) += uwn(i2) ± 10-16;
mstn(i) += mstn(i2) ± 10-16;
nstn(i) += nstn(i2) ± 10-16;
```

- Perturbing only the first three lines results in stability.

```
mwn(i) += mwn(i2) ± 10-16;
uwn(i) += uwn(i2) ± 10-16;
mstn(i) += mstn(i2) ± 10-16;
nstn(i) += nstn(i2);
```

Conclusion: instability in the program is caused only by perturbing the fourth line `nstn(i) += nstn(i2);`
The first three lines can be ignored.

EDF

# Advanced methods

More explanation on **Delta-Debugging** from http://edf-hpc.github.io/verrou/vr-manual.html:

- Provide a complete list of all lines to be perturbed. Exclude from the list all symbols that should produce unperturbed results (like math library, etc.). The final list is called the search space.
- By splitting the search space in two parts, and perturbing each half separately, it is possible to determine whether each perturbed half produces inexact results.
- Going on like this and performing a bisection of the search space, the algorithm eventually finds a subset of functions whose only perturbation is enough to produce inexact results. This Trial and Error algorithm is called Delta-Debugging.

# Advanced methods

- Reminder: Delta-Debugging helps identify the source code lines whose perturbation causes the most significant errors.
- Delta-Debugging does not identify where and when an initial microscopic perturbation ($\pm 10^{-16}$) amplifies to a macroscopic discrepancy
- While a deep dive into the unstable calculation with a debugger could help track the amplification of the perturbation, practical use of the debugger (GDB) with Valgrind/VERROU is not feasible.
- Solution: The 'unstable' line of the program can be instrumented to eliminate the need for Valgrind and enable the use of GDB.
- The term 'instrumented' refers to replacing the intrinsic operator with a VERROU function.

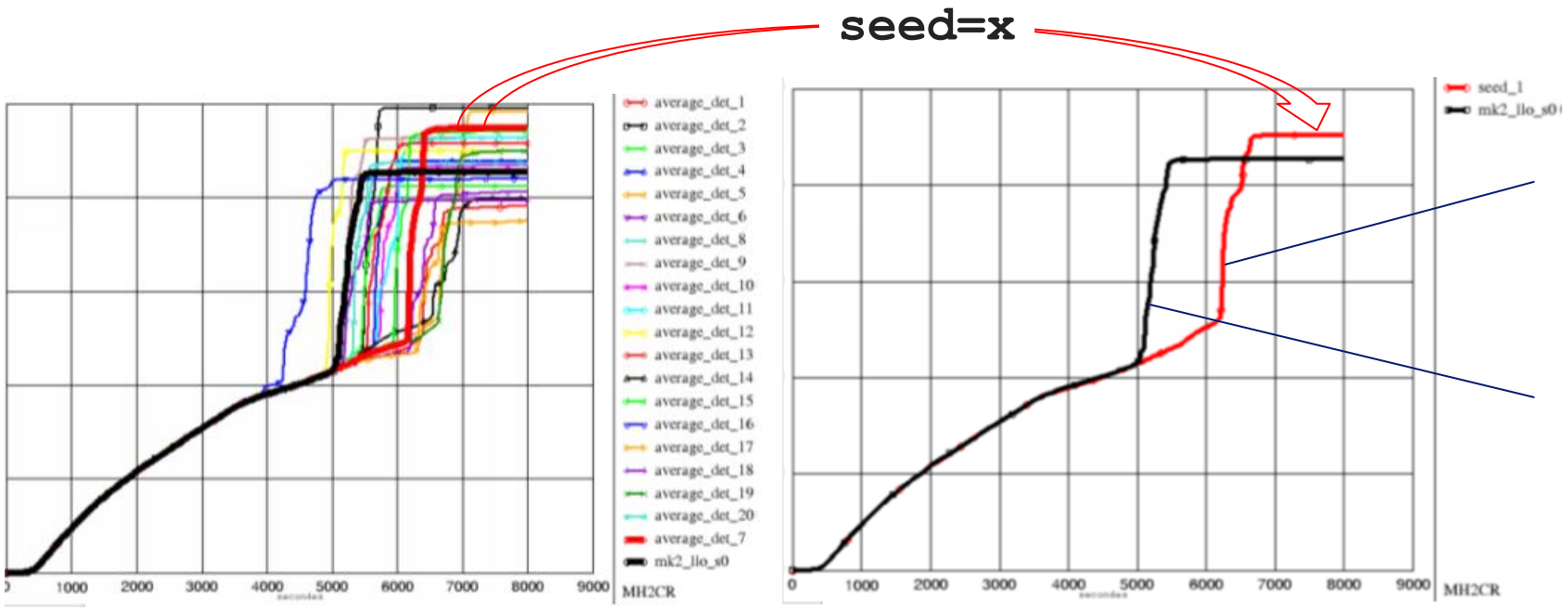
```
double nfrz = mz / cmn.molwzr;
```

```
double nfrz = verrou::div(&mz, &cmn.molwzr);
```

# Advanced methods

- The appropriate seed for the 'random' perturbation law can be identified using Delta-Debugging.
- The seed value is then set in the program using the VERROU interface function.

```
verrou::custom_verrou_init( &seed );
```




Delta-Debugging Valgrind/VERROU

GDB

Unstable calculation with instrumented program `maap_i.exe`

Stable calculation with original program `maap.exe`

# Advanced methods

At this point, we can run two calculations with GDB (without Valgrind/VERROU):

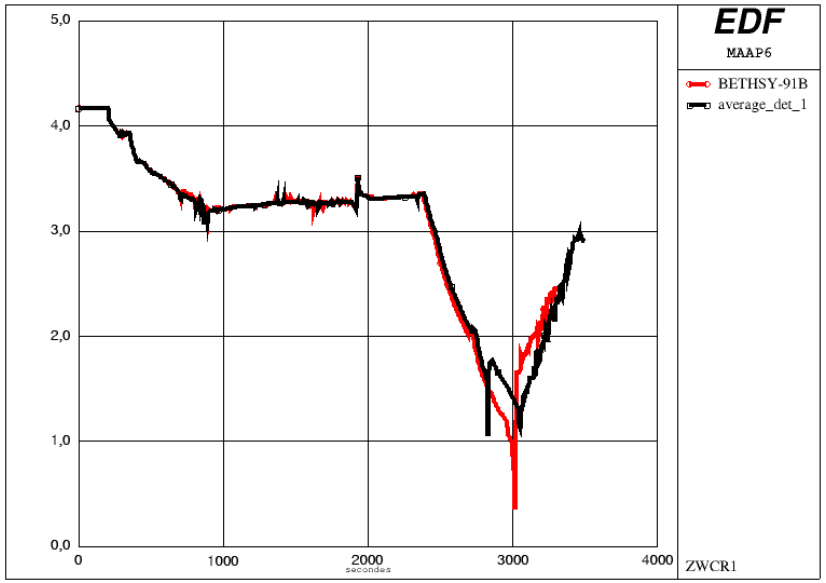
- A stable calculation with the original program
- An unstable calculation with the program modified as previously explained

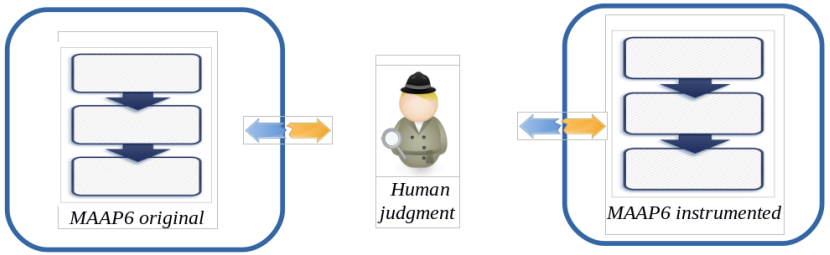
Analyzing the calculations with GDB allowed for updates to the program to improve its stability.

For example, an empirical correlation can be modified to be smoother vs the reference correlation

# Perspectives

- The VERROU interface allows for the creation of an unstable calculation with the instrumented program. This unstable calculation can be compared to the stable calculation with GDB.
- Typically, the stable calculation remains close to the unstable calculation until a visible macroscopic discrepancy appears.
- Generally, the deviating variable depends on other (hidden) variables that deviated earlier. In this case, it is necessary to rerun the calculations to track those variables, and so on.
- This kind of iterative assessment, which relies on human judgment, is very time-consuming.

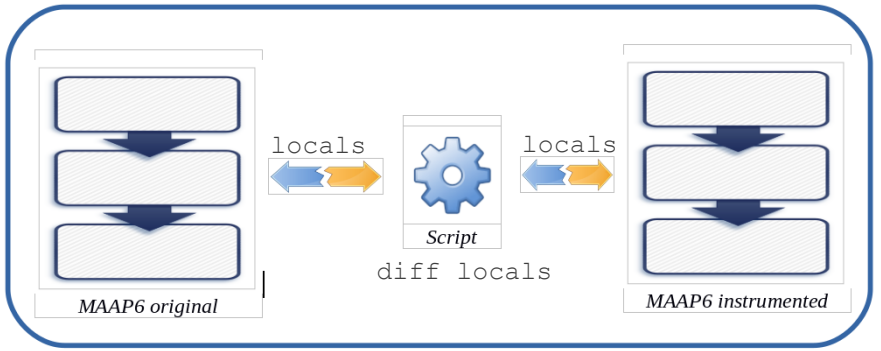


# Perspectives

Automatically running both calculations (original versus instrumented) synchronously could help identify the line where discrepancies become too high.

The GDB makes this possible as long as the two calculations are running inside the same debugging session with so-called *inferiors*, see https://sourceware.org/gdb/onlinedocs/gdb/Inferiors-Connections-and-Programs.html#Inferiors-Connections-and-Programs.




- An `inferior` represents the state of the program. You can switch between inferiors using the command `inferior 1` or `inferior 2`. Then, you can perform debugger operations on the active inferior, such as `info locals`.

- If the discrepancies in locals are close enough, the two calculations move one step forward, and so on. Otherwise, the calculations halt, allowing the user to analyze the situation with GDB.

- All these steps can be implemented using GDB scripting or Python scripting, which is also available within GDB.

# Thank you for your attention