A Direct Method to Assess Floating-Point Accuracy

Nestor Demeure[1,2], Cédric Chevalier[2], Christophe Denis[3], Pierre Dossantos-Uzarralde[2]

[1] Université Paris-Saclay, ENS Paris-Saclay, CNRS, Centre Borelli, F-94235, Cachan, France [2] CEA, DAM, DIF, F-91297 Arpajon, France

[3] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Motivations

Floating-point numbers are an approximation of real numbers: only a finite set of numbers is represented. As a consequence, when doing a computation with floating-point arithmetic, each operation is approximated leading to small numerical errors at each step:

$$(0.1 + 0.2) + 0.3 = (0.3 \; + error_1) + 0.3$$
$$(0.1 + 0.2) + 0.3 = 0.6 \; + error_1 + error_2$$
$$(0.1 + 0.2) + 0.3 = 0.6000000000000009$$
$$\neq$$
$$0.1 + (0.2 + 0.3) = 0.5999999999999998$$

While their impact tend to be small when the number of operations stays small, it can easily become significant on large applications that do millions of arithmetic operations per second.

Thus, there is need for a method to measure the impact of floating-point arithmetic on computations and, ideally, pinpoint its origins.

Goals

- **Fine-grained error measuring**
  Access the accuracy anywhere in an application,
  able to signal noticeable numerical errors in real time.
- **Localization of error sources**
  Traces the numerical error back to sections of interest in the application.
- **Suitable for high performance computing**
  Limited overhead compared to the state of the art,
  compatible with parallelism.

Measuring numerical error: a new method

Numbers are replaced with $(value, error)$ couples such that $value$ is the output one would have with classical floating-point arithmetic and $error$ verifies:

$$value + error \approx analytical\ value$$

To do so, the numerical error produced locally is computed at each operation (using either an *Error Free Transform* operation when it is available or higher precision arithmetic) while the errors comings from the inputs are propagated.

The simplest example is the addition which is defined as :

$$(x, error_x) + (y, error_y) = (x + y, error_x + error_y + error_+)$$

Using this representation we have access to the result $(value)$ but also its numerical error $(error)$ anywhere in an application and thus can quantify the number of significant digits of any value using the following formula :

$$digits(value, error) = \left\lfloor -\log_{10} \left| \frac{error}{value} \right| \right\rfloor$$

Numerical analysis of an integration scheme

The accuracy of our method is illustrated on the integration of a function using the rectangle rule. As the number of steps increase, the discretization error converges toward zero leaving us with the numerical error due to the sum of the areas.

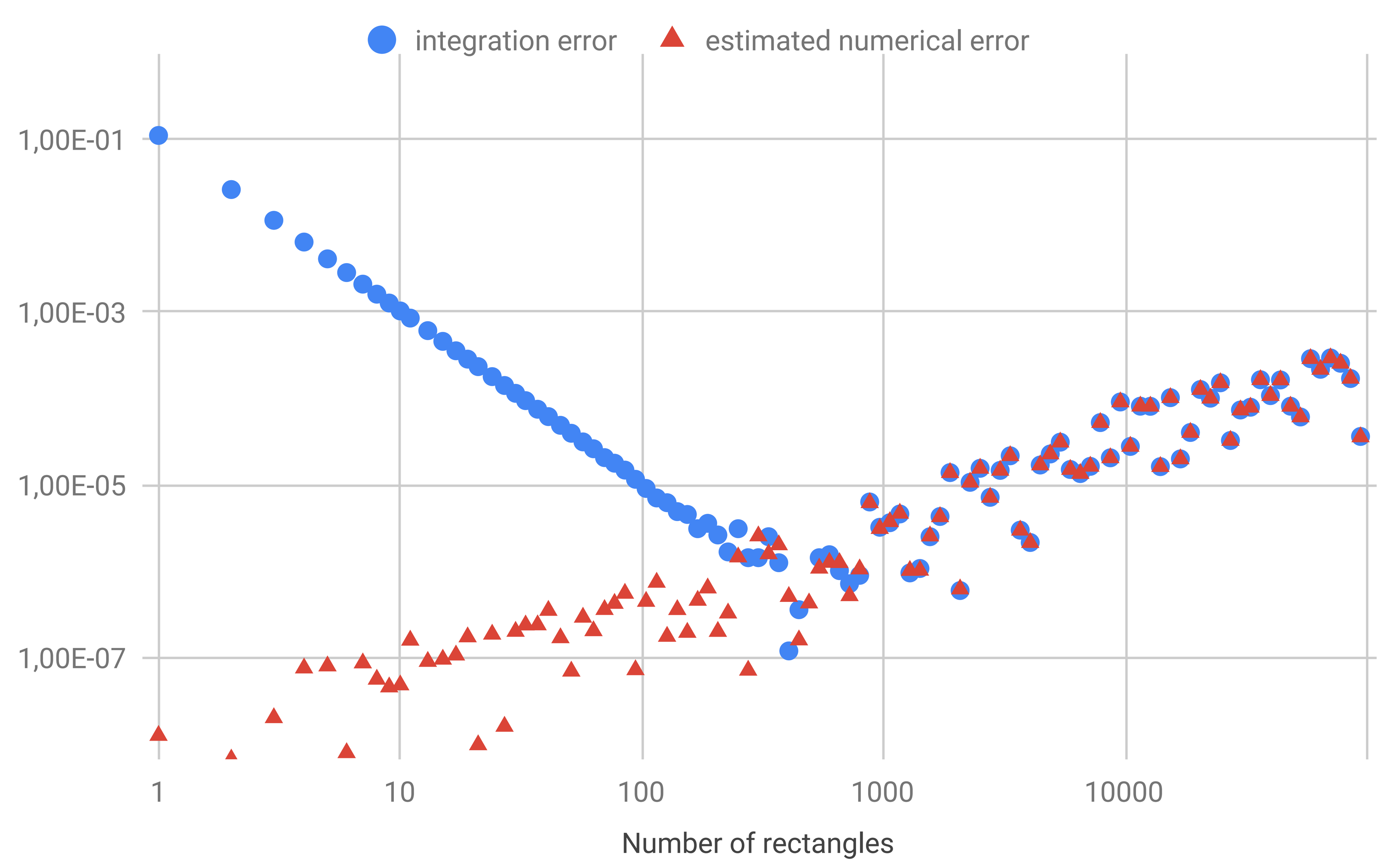

Figure 1: Integrating cos(x) between 0 and $\pi/2$ using the rectangle rule.

The Shaman library

Our method is implemented in the **Shaman** library. It can be used to instrument mixed precision C++ source code and is compatible with the Eigen linear algebra library, OpenMP and MPI. Furthermore, it can be hooked to a debugger such as gdb in order to trigger a breakpoint on noticeable operations such as unstable branches.

Tracing numerical error

The representation can be refined in order to trace the main sources of error. To do so, $error$ is expanded into $[error_{tag}]$ such that :

$$value + \sum_{tag} error_{tag} \approx analytical\ value$$

Each *tag* is a unique identifier associated with a section of interest in the instrumented application. With this refinement, the formula to compute an addition in the section s becomes:

$$(x, [error_{x,tag}]) + (y, [error_{y,tag}]) = (x + y, [error_{x,tag} + error_{y,tag} + \delta_{s,tag} * error_+])$$

The relative impact of a section s to the numerical error of a result $(value, [error_{tag}])$ can now easily be computed as follows :

$$\frac{error_s}{\sum_{tag} error_{tag}}$$

Sources of error in a numerical solver

The sources of numerical error in the conjugate gradient algorithm are traced for matrices of increasing condition number.

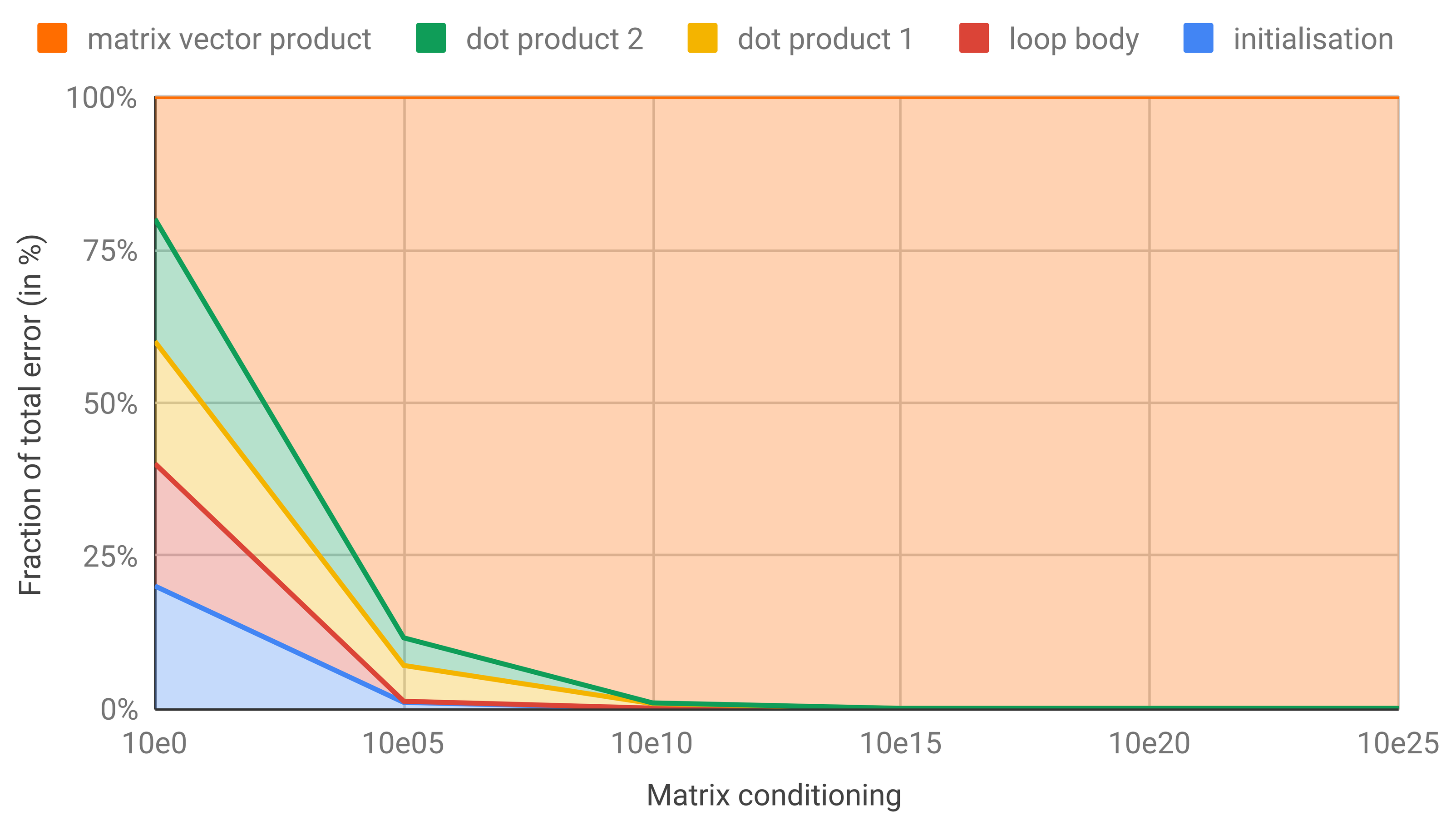

Figure 2: Distribution of the numerical error in the output of the conjugate gradient algorithm.

Replacing the matrix vector product with a numerically stable implementation, as it quickly dominates the numerical error, leads to a three-fold reduction of the residual on badly conditioned matrices.

Overhead of the instrumentation

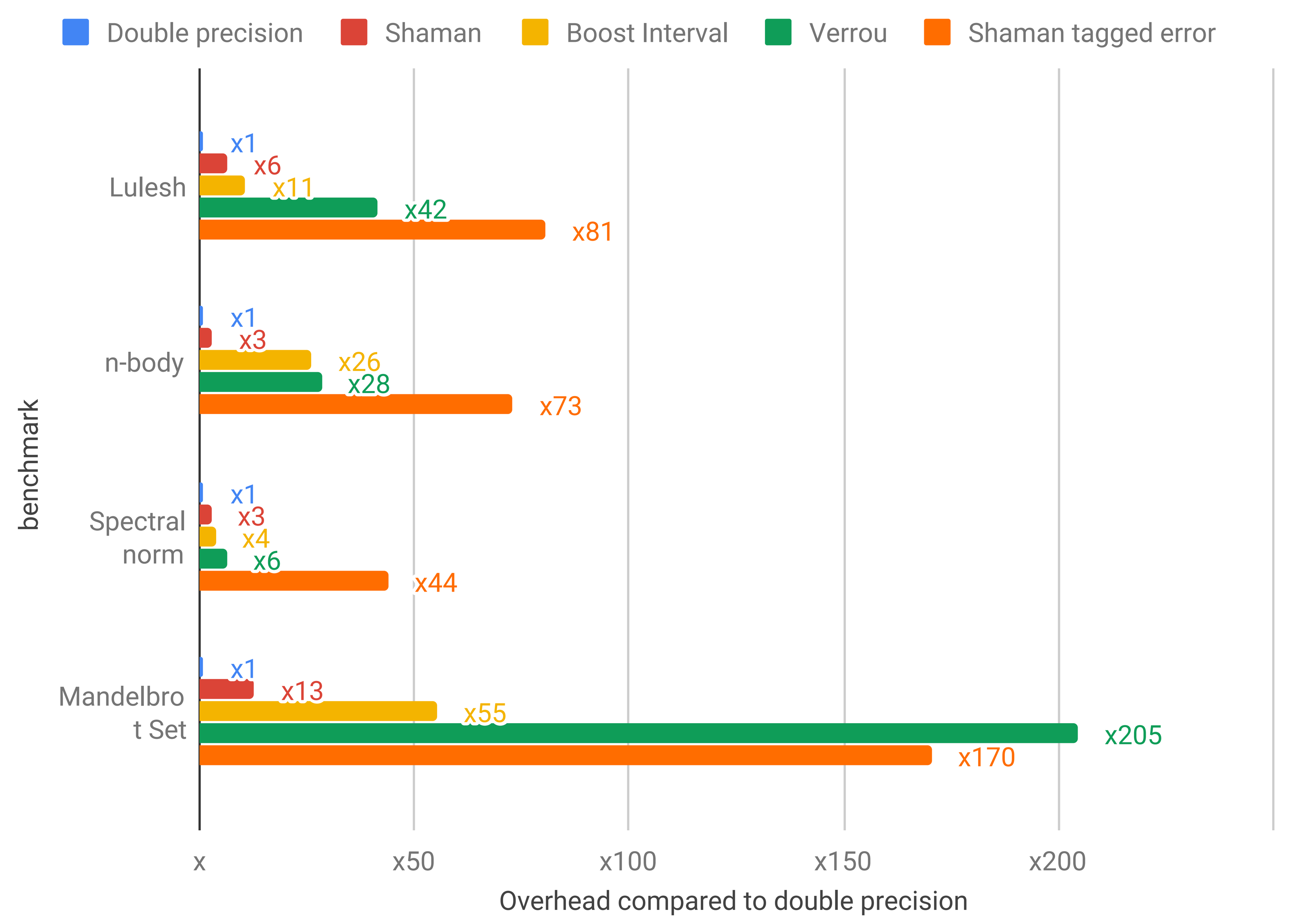

Figure 3: Overhead of the Shaman library compared to the state of the art.

Perspectives

- **Automatic instrumentation**
  Automatic refactoring (Clang),
  Instrumentation at compile time (Clang / LLVM),
  Instrumentation of a binary (Valgrind).
- **Targeted precision improvement**
  Limit the precision of an application where it does not matter to the final result.
- **Error modeling**
  Build an interpretable model of the propagation of error in an application.