

Shaman tool suite

User manual

Version 1.0.0

Nestor Demeure, Cédric Chevalier, Christophe Denis,
Pierre Dossantos-Uzarralde
May 27, 2022

Contents

1	Introduction	3
2	Shaman	4
2.1	Installing Shaman	4
2.1.1	Building Shaman with CMake	4
2.1.2	Adding Shaman to a CMake project	4
2.1.3	Flags	5
2.1.4	Online version	5
2.2	Using Shaman	6
2.2.1	Introducing Shaman in a codebase	6
2.2.2	Unstable tests	8
2.2.3	Nan and infinity	9
2.2.4	User-defined types	9
2.2.5	Integration with other frameworks and libraries	10
3	Shamanizer	11
3.1	Installation	11
3.1.1	Instaling Shamanizer with Spack	11
3.1.2	Manual installation	12
3.2	Usage	12
4	Numerical debugger	13
4.1	Installation	14
4.2	Usage	14
5	Numerical profiler	16
5.1	Installation	16
5.2	Usage	16
6	Benchmarks	17
6.1	Content of the Benchmark	17
6.1.1	Libraries evaluated	17
6.1.2	Applications timed	18
6.2	Building the benchmark	19
6.3	Running the benchmarks	19

List of Code Listings

1	Heron's algorithm.	6
---	----------------------------	---

2	Output of Heron's algorithm.	7
3	Using Shaman's types.	8
4	Output of the Shaman type example.	8
5	User defined instrumented type.	10
6	Using Shamanizer with a compilation database.	12
7	Using Shamanizer on a single file application.	13
8	Finding and using Clang's headers.	13
9	Output of the numerical debugger.	15
10	Using the numerical profiler.	16
11	Output of the numerical profiler.	17
12	Building the docker container.	19
13	Running the benchmark.	20

1 Introduction

The [Shaman tool suite](#)¹ is an ensemble of tools build around [Shaman](#), a C++ library that use operator overloading and encapsulated error to evaluate the numerical accuracy of an application. It is composed of four components:

- Shaman, the core library, described in section 2,
- Shamanizer, an instrumentation tool described in section 3,
- a numerical debugger described in section 4,
- a numerical profiler described in section 5,
- a benchmarking suite described in section 6,

While this document covers the official C++ reference implementation, you can currently find two other implementations in the wild. There is an [official Julia implementation here](#)². There is also an [unofficial Haskell implementation](#)³ within the [HGeometry](#) library.

The theory underlying Shaman and most of these tools is detailed in Nestor Demeure's PhD.⁴ You can reference it with:

```
@phdthesis{demeure_phd,
  TITLE = {{Compromise between precision and performance in
    ↪ high performance computing.}},
  AUTHOR = {Demeure, Nestor},
  URL = {https://tel.archives-ouvertes.fr/tel-03116750},
  SCHOOL = {{{\E}cole Normale sup{\e}rieure Paris-Saclay}},
  YEAR = {2021},
  MONTH = Jan,
  TYPE = {Theses}
}
```

¹https://gitlab.com/numerical_shaman

²https://gitlab.com/numerical_shaman/shaman_julia

³<https://hackage.haskell.org/package/hgeometry-combinatorial-0.12.0.0/docs/Data-Double-Shaman.html>

⁴Available at <https://tel.archives-ouvertes.fr/tel-03116750> and https://www.researchgate.net/publication/348551075_Compromise_between_precision_and_performance_in_high_performance_computing.

2 Shaman

`shaman`⁵ is a C++11 library that use operator overloading and encapsulated error to evaluate the numerical accuracy of an application. It has been designed to target high-performance simulations and, thus, built to be not only accurate but also:

- fast enough to be tested on very large simulations,⁶
- compatible with all the mathematical functions in the `C++ standard library`,
- thread safe and compatible with both `OpenMP` and `MPI`.

2.1 Installing Shaman

You can download the Shaman source code from https://gitlab.com/numerical_shaman/shaman. While one can build and link Shaman to their project manually, we recommend using `CMake` (version 3.9 and above) as explained in the following sections.

2.1.1 Building Shaman with CMake

To build Shaman in the `/path/to/dir` folder using `CMake`, run the following commands:⁷

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/dir .
make install
```

2.1.2 Adding Shaman to a CMake project

To add Shaman to a project that is built with `CMake`, open the corresponding `CMakeLists.txt` file and:

- add `find_package(shaman)` to the top of your `CMakeLists.txt` file,

⁵https://gitlab.com/numerical_shaman/shaman

⁶As significant numerical error is more likely to appears when large number of operations are in play.

⁷Assuming you are on a UNIX system. See the following link to run `CMake` on systems that are not compatible with `Make`: <https://cmake.org/runningcmake/>

- add `PUBLIC shaman::shaman` to the end of the `target_link_libraries` line.

If Shaman's installation folder is not known to CMake, you can set the `shaman_DIR` variable with the path to the CMake files that were autogenerated when building Shaman. To do so, add the the followig line to your `CMakeLists.txt`: `set(shaman_DIR /path/to/dir/lib/cmake/shaman)`

Warning: Do not forget to enable Fused-Multiply-Add instructions at compilation (on most compilers, it is done by passing the `-mfma` flag). Shaman *will* keep functioning correctly without it but, some operations (`*`, `/`, `sqrt`) will be computed significantly slower.

2.1.3 Flags

There are three flags that can be added when building *and* linking Shaman:

- `SHAMAN_ENABLE_TAGGED_ERROR` which enables the use tagged error to locate the sources of error with finer granularity.
- `SHAMAN_ENABLE_UNSTABLE_BRANCH` which enables the count and detection of unstable branches. The `Shaman::displayUnstableBranches` function can then be used to print the number of unstable tests performed by the application (and additional localization information if tagged error is enabled).
- `SHAMAN_FLUSH_NANINF` when used this ensures that the numerical error is a finite number, flushing other values to zero (see section 2.2.3).
- `SHAMAN_DEBUGGER` to pinpoint unstable operations with a numerical debugger or profiler (see section 5). Note that this flag *has been deprecated* in favor of tagged error and only works in the *paper branch* (see section 4.1).
- `SHAMAN_DISABLE` to disable shaman and use traditional types instead. This is useful when one wants to keep the main branch of an application instrumented with Shaman.

2.1.4 Online version

There is also an *online version of Shaman*⁸ served by *Replit*. While it relies on an *older header-only branch* and does not implement tagged error, it can

⁸<https://repl.it/@nestordemeure/ShamanDemo?lite=true>

be used to measure the numerical error of a simple computation and quickly familiarize oneself with Shaman before using the up to date version of the package in a larger project.

2.2 Using Shaman

2.2.1 Introducing Shaman in a codebase

Once Shaman is compiled and linked to your project (see section 2.1), you just need to:

- include `shaman.h` at the top of your files,
- replace your floating point datatypes (`float` , `double` , `long double`) with their Shaman equivalents (`Sfloat` , `Sdouble` , `Slong_double`),
- replace mathematical functions (from the `std` namespace) with their homonyms from the `Sstd` namespace.

Here is an example of Heron's square root algorithm implemented in C++ and instrumented with Shaman:

```
1  #include <shaman.h>
2
3  Sdouble x = 2;
4  Sdouble r = x/2;
5
6  while(1e-15 < Sstd::abs(r*r - x))
7  {
8      r = (r + x/r) / 2;
9      printf("sqrt=%.15e\n", r.number);
10     printf("error=%.15e\n\n", r.error);
11 }
```

Listing 1: Heron's algorithm.

The code outputs the following:

```
sqrt=1.5000000000000000e+00  
error=-0.0000000000000000e+00  
  
sqrt=1.4166666666666667e+00  
error=1.480297366166875e-16  
  
sqrt=1.414215686274510e+00  
error=1.393221050510000e-16  
  
sqrt=1.414213562374690e+00  
error=4.079910214529664e-17  
  
sqrt=1.414213562373095e+00  
error=1.253716726897950e-16
```

Listing 2: Output of Heron's algorithm.

Once a computation has been instrumented, one can access the numerical error of any intermediate result directly. The following code excerpt illustrate other quantities that can be obtained with an instrumented number as well as the possibility to display only significant digits of a result by simply printing a Shaman type with the C++ streaming operator (our recommended way to use Shaman):


```

#include <shaman.h>

// computation
Sdouble largeNum = 2e30;
Sdouble smallNum = 1;
// this should be one
// but ends up being zero due to a cancellation
Sdouble sum = (largeNum + smallNum) - largeNum;

// by default Shaman displays only significant digits
std::cout << "default: " << sum << std::endl;

// number that would have been obtained without Shaman
std::cout << "number: " << sum.number << std::endl;
std::cout << "cast: " << static_cast<double>(sum) << std::endl;

// approximation of the numerical error
std::cout << "error: " << sum.error << std::endl;

// approximation of the number of significant digits
std::cout << "digits: " << sum.digits() << std::endl;

```

Listing 3: Using Shaman's types.

The code outputs the following, where `~numerical-noise~` signifies that the output had no significant digits and could thus not be displayed:

```

default: ~numerical-noise~
number: 0
cast: 0
error: 1
digits: 0

```

Listing 4: Output of the Shaman type example.

2.2.2 Unstable tests

A test is said *unstable* if numerical error could have impacted its output (which can change the branches being taken by a computation and deeply impact the

end result). To detect unstable tests, pass the `SHAMAN_UNSTABLE_BRANCH` flag at compile time (see section 2.1.3). They will then be monitored and counted, calling the `Shaman::displayUnstableBranches()` function inside the code will display the number of unstable tests detected so far⁹.

Whenever an unstable test is detected, the `Shaman::unstability` function is called. You can get the exact localization of an unstable test in real time by putting a breakpoint on the function and running the code with a debugger (such as `GDB`). As an unstable test might be triggered hundreds of times in a loop, making it unpractical to use a debugger to study other instabilities in a computation, you can also run the code with the numerical profiler (see section 5) to produce a summary of all unstable test, their position in the code and how many times they were triggered.

2.2.3 Nan and infinity

Shaman is able to manipulate `nan` and `inf` correctly but, they might play havoc with the numerical error computation. This is not a problem in general as the numerical error associated with a `nan` or `inf` is usually meaningless.

However, some computation (such as a number divided by infinity) manage to recover gracefully without resulting in `nan` in the output. When displaying the result of such a computation, Shaman might display `~nan~`. This means that the number being displayed is finite but its numerical error is not-a-number.

You can solve that problem by using the `SHAMAN_FLUSH_NANINF` flag (see section 2.1.3). This will flush all non-finite numerical error terms to zero during the computations. This is not the default behavior as, sometimes, an infinite numerical error *is* the correct value.

2.2.4 User-defined types

All IEEE-754 compatible types can be instrumented with Shaman. furthermore, the user can pick any type to manipulate the numerical error and do the higher precision computations.

For example, given an emulated 16-bits precision type named `half`, the following would be a type behaving like `half` but whose numerical error is stored and manipulated in `float` precision and that relies on `double` precision when needing higher precision:

⁹If you are using tagged error, via the `SHAMAN_ENABLE_TAGGED_ERROR` flag, it will also display the tags corresponding to the sections of the code in which the tests happened.

```
using Shalf = S<half, float, double>;
```

Listing 5: User defined instrumented type.

The `Shalf` type can then be used like `half` but it will keep track of its numerical error.

While Shaman insures that implicit casts are done as they would have been done by their underlying types, some mixed precision operations that are legal with the original types might be rejected by their instrumented equivalent in the absence of an explicit cast (such as `Sfloat(1.5f) + double(1.5)`). To solve the problem, one just need to add an explicit cast (the previous example becoming `Sfloat(1.5f) + Sdouble(1.5)`).

2.2.5 Integration with other frameworks and libraries

The `shaman/helpers`¹⁰ folder contains additional headers that can be included to help when using:

- `std::complex`, to help with casting and insuring that the implementation behaves as the non-instrumented version,
- `OpenMP`, adding support for reduction over Shaman's default types (this requires OpenMP 4.0 or later),
- `MPI`, adding custom `init` and `finalize` functions and support to send Shaman's type and reduce over them,
- `Eigen`, adding traits to support Shaman's default types as well as arrays and matrix definition shortcuts,
- `Trilinos`, adding traits to support Shaman's default type within the Belos, Teuchos and Kokkos libraries.

Those implementation are fairly straightforward and do not require access to shaman's internals. As such, a user should be able to add support manually for any templated library they might need.

¹⁰https://gitlab.com/numerical_shaman/shaman/-/tree/master/src/shaman/helpers

3 Shamanizer

Shamanizer¹¹ is a tool, based on the **Clang compiler** (and in particular **LibTooling**), that can instrument a C++ codebase replacing classical numerical types with Shaman's types and introducing the needed headers.

3.1 Instalation

3.1.1 Instaling Shamanizer with Spack

We recommend that you use **Spack** in order to install Shamanizer without having to worry about its dependencies. To do so you first need to install Spack:

```
git clone https://github.com/spack/spack.git
cd spack
. share/spack/setup-env.sh
```

Then, we recommend that you build a recent C++ compiler and register it as a usable compiler in Spack:

```
spack install gcc@7.3.0 +binutils
spack compiler add $(spack find -p gcc@7.3.0)
```

To add Shamanizer to the Spack repository list, you should clone its repository (https://gitlab.com/numerical_shaman/spack_shaman) then add the path of the cloned folder to the `spack/etc/spack/defaults/repos.yaml` file in your Spack installation:

```
repos:
- path/to/spack_shaman
- $spack/var/spack/repos/builtin
```

Once those configuration steps are done, you should be able to install Shamanizer with Spack:

¹¹https://gitlab.com/numerical_shaman/shamanizer

```
spack install shamanizer
```

3.1.2 Manual installation

To install Shamanizer manually (which we do *not* recommend), you will first need to install the LLVM 3.0 toolchain.

At the moment the official Ubuntu/Debian LLVM packages have broken CMakefiles, you will thus have to get the packages directly from apt.llvm.org. If you cannot use a package manager to help you with the installation, you will need to recompile LLVM and Clang from source (note that you might encounter insufficient RAM problems if you build them with more than one parallel worker).

Once you have installed all the packages from the stable branch, you should be good to go (previously installed LLVM versions could cause problems, in which case the easier solution is to uninstall them) and can build Shamanizer with CMake similarly to how you built Shaman (see section 2.1.1).

3.2 Usage

To instrument a project, you need to first make sure that it compiles with [Clang](#) then build a compilation database for it, [both Clang and CMake are equipped to do so](#)¹². Once you have a compilation database built for your project, you can run the following command to instrument all files in the compilation database:

```
shamanizer -p=my_compilation_database.json .
```

Listing 6: Using Shamanizer with a compilation database.

You can replace the end dot with the name of a file to instrument a single file. Furthermore, for a very simple, single file, application you can omit the compilation database and run only the following command:

¹²<https://sarcasm.github.io/notes/dev/compilation-database.html#how-to-generate-a-json-compilation-database>

```
shamanizer my_file.cpp --
```

Listing 7: Using Shamanizer on a single file application.

Note that Clang’s headers will be needed if Shamanizer is not stored in the `usr/bin` folder. They can be located and passed to Shamanizer using the following commands:

```
HEADERS=$(dirname $(which clang))/../lib/clang/7.0.0/include  
shamanizer -p=my_compilation_database.json . \  
-extra-arg=-isystem=$HEADERS
```

Listing 8: Finding and using Clang’s headers.

The instrumentation *will be canceled* leaving the files untouched if there is any compilation error while Shamanizer runs.

4 Numerical debugger

The Shaman numerical debugger displays a summary of the number of numerical instabilities encountered while running a computation. Furthermore, you can hook a traditional debugger onto it to trigger a break whenever a computation encounters a numerically unstable operation in order to examine the state of the computation in real time.

Note that we now recommend using *tagged error* to trace the sources of numerical error in a computation as it takes the propagation of the numerical error into account rather than pinpoint single operations, leading to less false positives (such as a large cancellation on an intermediate result that will not impact the end result) and false negatives (such as a slow accumulations of errors that would not be detected because they never trigger an actual unstable operation).

4.1 Installation

The numerical debugger comes included in the `paper branch`¹³ of the code.¹⁴

To include it, you just need to compile the corresponding version of the code with the `SHAMAN_DEBUGGER` flag (see section 2.1.3) and have a debugger available (such as GDB).

4.2 Usage

To use the numerical debugger, you need to use Shaman's types in your application and add the `NUMERICAL_DEBUGGER` compilation flag. This will activate the numerical debugger, keep a count of the various numerical instabilities in your computation and display at summary at the end of the run.

For example, here is the output obtained when running our implementation of Heron's algorithm (see section 2.2.1) with the numerical debugger enabled:

¹³https://gitlab.com/numerical_shaman/shaman/-/tree/paper

¹⁴As it has been deprecated in favor of *tagged error*.

```

sqrt=1.5000000000000000e+00
error=-0.0000000000000000e+00

sqrt=1.4166666666666667e+00
error=1.480297366166875e-16

sqrt=1.414215686274510e+00
error=1.393221050510000e-16

sqrt=1.414213562374690e+00
error=4.079910214529664e-17

sqrt=1.414213562373095e+00
error=1.253716726897950e-16

*** SHAMAN ***
There are 5 numerical instabilities
3 CANCELLATION(S)
0 UNSTABLE DIVISION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE POWER FUNCTION(S)
2 UNSTABLE BRANCHING(S)

```

Listing 9: Output of the numerical debugger.

We can go further, as the output does not tell us where and when the cancellations and unstable branches occur, by using a classical debugger to pause the computations on cancellations or unstable branches.

To do so, you need to compile your application with `-g`, use a debugger (such as GDB) and set a breakpoint on the `Shaman::unstability` function. You might also need to reduce the optimization level (as usual when debugging a C++ application).

When running the application with the debugger, it will stop whenever an unstable operation (such as a cancellation or an unstable test) is detected. Doing so with our example, we learn that from the iteration 4 onward the subtraction in the loop condition is a cancellation and that the test and the computation of the absolute value become unstable on the last iteration.

5 Numerical profiler

The **Shaman numerical profiler**¹⁵ builds on the numerical debugger to give you a numerical profile of your application with the number of numerical instabilities detected in each function and the exact line/operations where those instabilities appeared. This is particularly useful on a large application that might have thousands of numerical instabilities, most of them occurring in the same spots and irrelevant to the end result.

While the numerical debugger has been depreciated in favor of tagged error, you can still use the numerical profiler in Shaman's master branch to gather information on unstable tests.

5.1 Installation

The numerical profiler is included in the **shaman/tools/shaman_profiler** folder¹⁶. It consist of a Python script and *requires* gdb-7.11.1 or later (as we rely on their Python API).

The version present in the main branch can be used to track unstable tests in the master branch of Shaman but, you will need to use the **paper branch** if you want to track all unstable operations such as cancellations.

5.2 Usage

To use the Shaman Profiler, you need to add the `NUMERICAL_DEBUGGER`¹⁷ and `-g` flags at compile time (you might also need to reduce the optimization level). You can then run your program with the following line (you can also use the `shaman_prof.sh` shell script as a shortcut):

```
gdb -quiet --command shaman_profiler.py --args ./your_program  
→ args
```

Listing 10: Using the numerical profiler.

¹⁵https://gitlab.com/numerical_shaman/shaman/-/tree/master/tools/shaman_profiler

¹⁶https://gitlab.com/numerical_shaman/shaman/-/tree/master/tools/shaman_profiler

¹⁷You can omit this flag if you are only tracking unstable tests in the master branch but, you will need the `SHAMAN_ENABLE_UNSTABLE_BRANCH` flag as detailed in section 2.2.2.

It will display a message every 100000 numerical instability detected and output its results to the `EXECUTABLE_NAME_shaman_profile.txt` file once the program finishes running. As it relies on breakpoints to collect information you can expect a slowdown proportional to the number of numerical instabilities (which can be consequent on large computations).

Here is the file produced when running our implementation of Heron's algorithm (see section 2.2.1) with the numerical profiler:

```
*** SHAMAN PROFILE ***
5      heron (file main.cpp)
3          operator- (line 4)
1          operator< (line 4)
1          abs (line 4)
```

Listing 11: Output of the numerical profiler.

We find the same results as the ones obtained with the numerical debugger (in section 4.2) with line 6 corresponding to the loop condition of the code.¹⁸ The analysis of this particular example tells us that the stopping criteria of our algorithm is impacted by the numerical error, creating an unstable branching, and that it might benefit from a redesign to avoid instabilities¹⁹.

6 Benchmarks

The `shaman benchmarks`²⁰ are a docker container designed to reproduce our timing benchmark, comparing Shaman with several other library on a variety of applications.

6.1 Content of the Benchmark

6.1.1 Libraries evaluated

We tried to include at least one implementation for each of the most common approaches used to measure the numerical error. We choose these imple-

¹⁸Note that the computation is inside a `heron` function in the `main.cpp` file which have been omitted from our code snippet for brevity.

¹⁹The current criteria uses a naive formula that can, indeed, suffer from convergence problems from small values of x .

²⁰https://gitlab.com/numerical_shaman/shaman_containers

mentations because of their extensive usage in their category and efficiency:

- **Shaman**.
- **MPFR**, with the **MPFR C++** wrapper, which implements arbitrary precision arithmetic (tested with 100 and 200 bit of precision).
- **Boost Interval**, which implements interval arithmetic.
- **Verrou**, which implements a form of stochastic arithmetic.
- **Cadna**, which implements a synchronous variant of stochastic arithmetic.

6.1.2 Applications timed

We instrumented four programs:

- n-body: N-body simulation.
- Spectral norm: computing an eigenvalue using the iterated power method.
- Mandelbrot set: generating the Mandelbrot set at a given resolution; it is the only parallel benchmark of the set.
- Lulesh 1.0: solving explicit hydrodynamics equations on a collection of volumetric elements.

The first three are the tasks that deal with floating-point arithmetic in the **computer benchmark game**,²¹ a well-known benchmarking suite that is used to compare the peak performances of programming languages on different tasks. The last program of our selection is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics code (**Lulesh 1.0**), a proxy program for performance benchmark for exascale computing. All four programs are in double precision (represented by Shaman's `Sdouble` type).

²¹As the computer benchmark game provides several implementations for each task, we instrumented the fastest C++ implementations that did not rely on explicit vectorization or calls to libraries (such as **Eigen**) to do their computations.

6.2 Building the benchmark

You will need **Docker** or a Docker compatible tool (such as **Podman**) to build the containers.

Once you have downloaded the benchmark's folder (from https://gitlab.com/numerical_shaman/shaman_containers), simply run the following command inside the folder (`DOCKER_BUILDKIT=1` is optional but recommended if presents on your system):

```
DOCKER_BUILDKIT=1 make
```

Listing 12: Building the docker container.

If the command fails, you might want to try running it in `sudo` priority, start docker (as described in section 6.3) and restart your computer before building. It will build 5 containers:

- `shaman/shaman`, which contains a **Shaman** installation,
- `shaman/cadna`, which contains a **Cadna** installation,
- `shaman/verrou`, which contains a **Verrou** installation,
- `shaman/standard`, which contains a **ubuntu-20.04** installed with **boost-interval** and **MPFR**,
- `shaman/sandbox`, which is built on top of the previous containers and allows experimentation with all these tools.

6.3 Running the benchmarks

Once the containers are built and Docker is **started**, the benchmarks can be started like this (on a `bash` compatible shell):

```
docker run -v $(pwd)/tests/benchmarks:/home/user/tests -it  
↳ shaman/sandbox  
  
cd ~/tests  
make  
make time
```

Listing 13: Running the benchmark.

It should run all the combinations of applications and libraries one after the other and display the corresponding run times.