Porting the CADNA library to modern C++

Thomas Helfer, Fabienne Jézequel

2017

Context

Round-off errors propagation is a primary concern in numerical simulations. However, only a few tools exists to identify, analyse, debug and correct portion of the code causing numerical instabilities to appear and propagate.

CADNA, developed by the Computer Science Laboratory of the PARIS 6 university is a library aiming solving the previous issues. With CADNA the numerical quality of any simulation program can be controlled. By detecting all the instabilities which may occur at run time, a numerical debugging of the user code can be performed. Another feature (but not the least) provided by CADNA is that data errors can be taken into account for the estimation of the final accuracy. Based on the latter, this paper will also advocate another potential usage of CADNA: building unit tests assessing that numerical algorithms remains stable as their implementation evolves (whatever the reason).

It's official version can be downloaded at: http://www.lip6.fr/cadna

Authors of the CADNA library can be contacted at this adress: cadna-team@lip6.fr

The aim of this article is to present an implementation of the CADNA library based on modern C++ (C++11), to present our design choices to make the numerical types introduced by the library as closed as possible than plain old floatting point numbers.

A introductory example

To have an overview of the library, consider the following example which evaluates the function $f\left(x,y\right)=9\,x^{4}-y^{4}+2\,y^{2}$ for the $10864, 18817$ and $1/3, 2/3$:

```
#include <iostream>
#include"cadna/numeric_type.hxx"

using real = cadna::numeric_type<double>;
```

```
real rump(const real x,
          const real y){
  const real a = 9*x*x*x*x;
  const real b = y*y*y*y;
  const real c = 2*y*y;
  return a-b+c;
};

int main(void){
  real x = 10864;
  real y = 18817;
  std::cout << "f: " << rump(x,y) << std::endl;
  x = 1/real(3);
  y = 2/real(3);
  std::cout << "f: " << rump(x,y) << std::endl;
  return EXIT_SUCCESS;
}
```

The output of the simple test is the following:

```
-----------------------------------------------------------------
CADNA software --- University P. et M. Curie --- LIP6
-----------------------------------------------------------------
f:  @.0
f:  0.802469135802469E+000
-----------------------------------------------------------------
CADNA software --- University P. et M. Curie --- LIP6
There are 2 numerical instabilities
2 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)
-----------------------------------------------------------------
```

This example shows that the numerical types introduced by the `CADNA` library are meant to be drop-in replacement of plain old floating point types. **Using `CADNA` shall be a simple a replacing every floating point type by their `CADNA` counterparts, recompiling and linking against the `cadna_cxx` shared library.**

The first output states that the result computed is pure numerical noise (`@.0`).

By setting a break point at the `cadna::instability` function, one may analyse the cause of the numerical instability in the debugger.

One may also use the `unwind` library (or one of its native equivalent), to display at runtime. To do this, one have to include the `cadna/logstream.hxx` header and append the following line at the beginning of `main`:

```
int main(void){
  cadna::add_logstream(std::cout);
```

The output is now as follows:

```
** cadna library : cancel instability detected
- _ZNSt17_Function_handlerIFvN5cadna14instability_idEENS0_17InstabilityReportEE9_M_invokeERK
- _ZN5cadna25call_instability_handlersENS_14instability_idE at 0x7f5f2efa5540 (0x30)
- _ZN5cadna11instabilityENS_14instability_idE at 0x7f5f2efa4d9d (0x3d)
- <unknown function name>
- _Z4rumpN5cadna12numeric_typeIdEES1_ at 0x40566c (0x4aa6)
- main at 0x40571b (0x9a)
- __libc_start_main at 0x7f5f2e3e7b45 (0xf5)
- _start at 0x400af9 (0x29)

** cadna library : cancel instability detected
- _ZNSt17_Function_handlerIFvN5cadna14instability_idEENS0_17InstabilityReportEE9_M_invokeERK
- _ZN5cadna25call_instability_handlersENS_14instability_idE at 0x7f5f2efa5540 (0x30)
- _ZN5cadna11instabilityENS_14instability_idE at 0x7f5f2efa4d9d (0x3d)
- <unknown function name>
- _Z4rumpN5cadna12numeric_typeIdEES1_ at 0x40566c (0x4aa6)
- main at 0x40571b (0x9a)
- __libc_start_main at 0x7f5f2e3e7b45 (0xf5)
- _start at 0x400af9 (0x29)
```

The library can be instrumented in numerous ways using the `add_instability_handler` which takes a `std::function` object as argument.

The theoretical foundations of the `CADNA` library: probabilistic arithmetics based on the `CESTAC` approach

Basically, `CADNA` replaces the standard floating point number by so-called stocastic types. Internally, those type contain N floating point numbers which all represents an approximation of the result. Each floatting point operations is repeated N with a random bias. If the N numbers representing the final result are closed from each others, this result is very accurate. If those number differ significantly, this result is unreliable. The comparison of those numbers also provide an estimation of the number of significant digits in the final result. Moreover, by instrumenting every floating point operation, one can identify the ones responsible for significant accuracy loss. In practice, N is taken egal to 3.

The driving reason for this new implementation

Removing code generation based on `m4` macros

First versions of `CADNA` were written in `Fortran` and targeted this language, the `de facto` standard in numerical simulations at that time.

To provide a drop-in replacement of native types, one had to implement:

- All the possible numerical operations between two `CADNA` floating point objects, and all the operations between one `CADNA` floating point object and all the native ones (including integers).
- All the overloads of the standard math library functions (`cos`, `exp`, etc.).

As `Fortran` lacks supports for templates, the `CADNA` developers recoursed to `m4` macros to generate those operations. For maintenance and portability, replacing `m4` macros by template functions and operators is a clear gain for a modern `C++` implementation of this library.

This strategy has been reused the create the "`C`" version of `CADNA` described below.

Better support of `C++`

The "`C`" version was indeed based on `C++` to provide its numeric types, leveraging the compatibility of `C` and `C++` to allow most `C` code to be modified.

However, this `C` version was not meant for software heavily relying on the `C++` standard:

- It provided overload for the mathematical function declared the `math.h` header, but no overload their counterparts defined in the `::std` namespace.
- It did not provide appropriate overload of several usefull traits classes defined is the standard, such as `std::numeric_limits`, `std::is_floating_point`, `std::is_scalar`, etc.

The `C++` standard extensively protects the `::std` namespace in C++11 17.6.4.2.1, but specifically allows the overloading of traits classes in paragraphs 1 and 2:

> The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std` unless otherwise specified. A program may add a template specialization for any standard library template to namespace std only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.
>
> [...] A program may explicitly instantiate a template defined in the standard library only if the declaration depends on the name of a

user-defined type and the instantiation meets the standard library requirements for the original template.

Overloading functions in the `std` namespace is not legal and we rely on Argument Dependent Lookup to declare our implementations in a dedicated namespace.

However, in some cases, it is mandatory to also overload the standard mathematical functions, as some users may fully qualify the mathematical functions in their libraries. Indeed fully-qualifying mathematical functions can be considered a good practice as it prevents dumb mistakes, such as unattended casts. Consider the case of the `abs` functions which is defined in both `cstdlib` and `cmath` headers. If one forgets to include the `cmath` header and do not qualify the `abs` function, the `abs` function argument will be casted to integers, leading to a subtle bug.

Better portability

We choose to rely on the `cmake` build system for maximal portability.

Our implementation works with the following compilers:

- GNU compiler
- Intel compiler
- Clang compiler
- Microsoft Visual Studio compiler

Support extended precision numbers

Aside the `fortran` and `C` versions of `CADNA`, a library called `SAM` (Stochastic Arithmetic in Multiprecision) has been created to support arbitrary multiprecision numbers, as introduced by the `MPFR` libray.

The `Boost Multiprecision` library provides integer, rational and floating-point types in C++ that have more range and precision than C++'s ordinary built-in types.

Combining the `Boost Multiprecision`library and our implementation is straight-forward by providing an appropriate specialization of the `cadna::default_accuracy` traits class.

This approach has one intrinsic limit: contrary to `SAM`, the accurary of the extended floating-point types is fixed a compile-time.

Support for `python` and `fortran` bindings

Our implementation support `python` and `fortran` bindings.

Implementations details