Implementing an interval computation library for OCaml on x86/amd64 architectures

Jean-Marc Alliot[1] and **Jean-Baptiste Gotteland**[1,2] and **Charlie Vanaret**[1,2] and **Nicolas Durand**[1,2] and **David Gianazza**[1,2]

Abstract. In this paper we present two implementation of interval arithmetics for Ocaml on x86/amd64 architectures. The first one is simply a binding to the classical MPFI/MPFR library. It provides access to multi-precision floating point arithmetic and multi-precision floating point interval arithmetic. The second implementation has been natively written in assembly language for low-level functions and in ocaml for higher-level functions ans is as fast as classical C or C++ implementations of interval arithmetic.

1 Fundamentals of interval arithmetic

Interval arithmetic has been used in computer science and numerical computations for years [5]. Its main goal was to create computing environments where the exact value of a computed result lies with certainty within an interval, which might be paramount for some critical applications. Floating point units (FPU) only work with a fixed size for the mantissa of the operands, and numerical errors are unavoidable. For example, $1/3$ rounded to two decimals is neither 0.33 nor 0.34 but the exact value lies within the interval $[0.33, 0.34]$. Most FPU are compliant with the IEEE-754 standard regarding numerical computations. The most widely used format is the IEEE-754 double precision. This format gives around 16 decimal digits of precision. In this context, numerical errors might seem insignificant or irrelevant. However, the accumulation of many numerical errors with ill-conditioned functions can lead to disastrous results. Let us consider the function: $f(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$ The correct result with six digits of $f(77617, 33096)$ is -0.827396. However, when computed with an IEEE-754 compliant x87 FPU in double precision, the result is $-1.180592.10^{21}$ with OCaml 3.12 compiler. Interval arithmetic yields a lower bound ($-5.902958.10^{21}$) and an upper bound ($5.902958.10^{21}$), which immediately points out the ill-conditioned nature of the function at this point.

Over the last 30 years, interval arithmetic has expanded to new grounds; since Rokne and Ratschek book [2], interval arithmetic has been widely used in global optimization. Branch and bound algorithms associated with interval arithmetic are now used to find global optima of deceptive functions (such as Griewangk and Michalewicz functions) with up to 20 variables.

The IEEE-754 standard requires that every FPU is able to make any elementary computation in four rounding modes: toward $+\infty$ (upper rounding), toward 0, toward $-\infty$ (lower rounding) and toward the nearest representable number (nearest rounding). It is thus quite easy to implement the most basic functions of interval arithmetic with proper roundings of both endpoints. The addition of two intervals is for example defined by: $[a, b] + [c, d] = [a +_{low} c, b +_{up} d]$. Multiplication and division are slightly more complex tasks that require to take into account the position of the intervals relatively to 0. For example, if $a < 0 < b$, we have $1/[a, b] = [-\infty, 1/a] \cup [1/b, +\infty]$. As the second member of the equation has to be reduced to a single interval, we simply have: $1/[a, b] = [-\infty, +\infty]$ There is thus a loss of information, which is unavoidable. The usual real-valued functions (cos, sin, arcsin, log, etc.) can also be extended to interval arithmetic. The extension is trivial for monotonic functions (such as $\exp(x)$), as the image of interval $[a, b]$ by exp is simply $[\exp_{low}(a), \exp_{up}(b)]$. Computing interval extensions of periodic functions such as $\sin(x)$ or $\cos(x)$ is a much more complex task. When elementary functions are defined, more complex functions can be computed by composition.

[1] Institut de Recherche en Informatique de Toulouse, name.surname@irit.fr
[2] Laboratoire “Mathématiques Appliquées et Informatique” de l’ENAC

2 Existing implementations

There are numerous interval arithmetic implementations with various bindings to different languages such as Profil/BIAS (a C++-class library developed in 1993 at the Hamburg University of Technology), a template class for interval arithmetic in the Boost C++ libraries, Gaol (a C++ interval arithmetic library that implements operators for interval constraint programming), MPFI [6] (a multi-precision interval arithmetic library for C or C++), and the SUN interval arithmetic implementation for Fortran 95 or C++ [4]. There is a proposal to have interval arithmetic integrated to the standard C++ language [1] and an IEEE interval standard is currently under development.

3 Bindings to the MPFR/MPFI libraries

We chose to develop bindings to the MPFR/MPFI libraries since these libraries provide a unique feature: the possibility to work with an arbitrary precision. This feature is extremely valuable when precision (and not time) is paramount. The MPFR library is a multi-precision arithmetic library which implements a very large number of functions on arbitrary-precision floating-point numbers. We only implemented the bindings to MPFR functions which are needed when using the MPFI bindings (more complete MPFR bindings are available in the APRON library). The MPFI library uses the MPFR library to provide arbitrary-precision floating-point interval arithmetic. OCaml bindings exist for almost all MPFI functions with almost no syntactic sugar. Thus, an interval is an opaque structure that must be allocated with the *init* function prior to use. After allocation, it can be used and modified in place. The semantic of the functions is thus not functional whatsoever. Marshalling of these objects is not possible yet.

4 A native implementation

The MPFR/MPFI library is widely used, but is quite slow due to its versatility. We thus decided to implement a faster library limited to double-precision floating-point interval arithmetic. There were two possibilities: developing bindings to an existing library, or implementing the library by ourselves. From an overview of the existing implementations, we concluded that developing bindings to these libraries (usually in C++) would be tedious and would require the use of an extra layer that would slow us down. We chose to directly implement the library in assembly language for the lower-level functions, and in OCaml for the higher-level functions. The latter implements the "logical" part of the function, while the former implements the elementary function computations for both rounding modes. The target architecture was the Intel processor family (x86/amd64), on the three main operating systems (Windows, Linux and Mac OS X). We decided to use a functional semantics for all functions: implemented operators such as $+, $-, $*, $/ extend standard operators to interval arithmetic. This allows us to write *let a = b $+ c* with a, b and c being interval objects. Implementations choices regarding inclusion functions and throwing exceptions are based on the following properties: Let f be a function and F its extension to interval arithmetic, then

- $\forall x \in [a, b]$, if $f(x)$ is defined then $f(x) \in F([a, b])$
- if $\{f(x) \mid x \in [a, b]\}$ is empty then $F([a, b])$ raises an exception

All the elementary operations with both rounding modes are available via the **Fpu** module of our library, while the interval function extensions are available in the **Interval** module.

We had to deal with some unexpected problems. The x87 is supposed to return the nearest value, the upper and lower bounds for each elementary operation, but this is not always the case: some functions such as cos, sin or tan are not properly implemented everywhere. For example, we computed $\cos(a)$, with a = atan2_low 1. 0. and with the following cosine implementations:

1. the MPFI library (with 128-bit precision),
2. the x87 in round-toward $-\infty$ mode,
3. the x87 in nearest mode (default value for the C and OCaml libraries on 32-bit Linux),
4. the x87 in round-toward $+\infty$ mode,
5. the SSE2 implementation (default value for the C and OCaml libraries on 64-bit Linux):

We got the following results: $\cos_{x87low}(a) < \cos_{x87}(a) = \cos_{x87high}(a) < \cos_{MPFI}(a) < \cos_{SSE2}(a)$, so the upper bound (4) computed by the x87 is clearly incorrect, as it is lower than the correct value computed by the MPFI library. The value computed by the SSE2 (5) is much more precise than the one computed by the x87. However, it is unfortunately impossible to get upper and lower bound values with the SSE2 implementation, and we have no other choice but to use the x87 for computing these (sometimes incorrect) bounds. The problem here is that the value computed by the standard C-lib (or OCaml) $\cos(x)$ function does not always lie within the lower bound/upper bound interval returned by the x87 functions. This can be very prejudicial when executing branch and bound algorithms where the mid-value is expected to lie within the lower/upper interval. We solved this issue by rewriting quite efficiently the trigonometric functions in assembly language. With our new implementation, the lower and upper bounds are properly set and they are always lower (resp. higher) than the values computed by the standard $\cos(x)$ functions on 32 and 64-bit architectures.

Values returned by the standard (C-lib or OCaml) $\cos(a)$, $\sin(a)$ or $\tan(a)$ functions remain different on 32 and 64-bit architectures. In order to obtain the same behavior on both architectures, the *fcos*, *fsin* or *ftan* functions from module **Fpu** can be used. They always return the same values on all architectures, and they can transparently replace standard functions by using the **Fpu_rename** or **Fpu_rename_all** modules.

5 Performance issues

Figure 1 from [3] presents some comparison results of classical interval libraries, including MPFI. The scale is logarithmic.

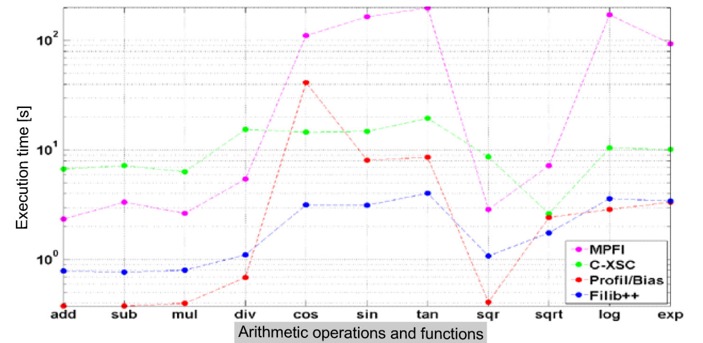

Figure 1. Comparison of standard interval libraries

Table 1 is a comparison of MPFI with our native implementation on 10^6 operations (times are in seconds). The third column is the logarithm of the ratio of the execution times of both implementations ($\log_{10}(M/N)$).

Op	MPFI	Nat	log(M/N)	Op	MPFI	Nat	log(M/N)
+	0.124	0.076	0.21	log	2.696	0.096	1.44
-	0.172	0.068	0.40	exp	4.568	0.224	1.31
*	0.148	0.088	0.23	cos	3.544	0.136	1.42
/	0.240	0.088	0.43	sin	3.868	0.136	1.45

Table 1. Comparison of MPFI bindings and native implementation

From the above figure and table, we can compare our native implementation with some of the fastest implementations of interval arithmetic. We can observe that on most functions, our library is on par with Filib. Moreover, the native implementation suffers from its functional semantics, as it creates at each of the 10^6 operation a new interval object, while the classical libraries (including MPFI) create interval objects once and for all before the computation and modify them in place. So the observed performance is indeed excellent.

References

[1] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion, 'A proposal to add interval arithmetic to the c++ standard library', Technical Report N1843=05-0103, INRIA, (2005).

[2] New computer methods for global optimization, *H. Ratschek and J. Rokne*, Ellis-Horwood, 1988.

[3] R. Dabrowski and B. Kubica, 'Comparison of interval c/c++ libraries in global optimization', Technical report, Warsaw university, (2009).

[4] SUN Microsystems, *C++ Interval Arithmetic programming manual*, SUN, Palo Alto, California, 2001.

[5] R.E. Moore, *Interval Analysis*, Prentice Hall, NJ, 1966.

[6] N. Revol and F. Rouillier, 'Motivations for an arbitrary precision interval arithmetic and the MPFI library', *Reliable computing*, **11**(4), (2005).