Implementing an interval computation library for OCaml on x86/AMD64 architectures

J-M. Alliot   J-B. Gotteland   C. Vanaret   N. Durand   D. Gianazza

OCaml Users and Developers Workshop 2012
International Conference in Functional Programming
September 14, 2012
Introduction
- Objectives
- Interval arithmetic
- Why a new interval library?

A functional implementation for OCaml
- Unexpected problems
- Modules
- Implementation principles

Performance comparison
1 Introduction
   - Objectives
   - Interval arithmetic
   - Why a new interval library?

2 A functional implementation for OCaml
   - Unexpected problems
   - Modules
   - Implementation principles

3 Performance comparison
Objectives

Bound global solutions of **difficult continuous optimization** problems

\[
\min_{x \in D} f(x) \\
s.t. \quad g_i(x) \leq 0, \quad i \in \{1, \ldots, p\} \\
h_j(x) = 0, \quad j \in \{1, \ldots, q\}
\]

**Hybridization** of optimization methods [ADGG12]

- **Evolutionary Algorithms** (EA)
- **Interval Branch & Bound** algorithm (BB)
  computation of lower/upper bound of \( f \) over a subspace of \( D \)
Interval arithmetic

Numerical analysis method to bound round-off errors [Moo66]

Problem ($\mathcal{P}$)

- $f(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
- $f(77617, 33096) = -0.827396$ (6 digits)
- OCaml 3.12 compiler: $-1.180592.10^{21}$
Interval arithmetic

- Extends to intervals \{+, −, ∗, /\}, \exp, \log, etc.
- An interval extension \( F \) of \( f \) yields a rigorous enclosure of \( f(X) \)
- Computation of lower/upper bound with outward rounding

\[
[a, b] \oplus [c, d] = [a + \text{low } c, b + \text{high } d]
\]
- \( a + \text{low } c \) must be a lower bound of \( a + c \)
- \( b + \text{high } d \) must be an upper bound of \( b + d \)

Problem (\( P \))
Interval arithmetic yields: \([-5.902958 \times 10^{21}, 5.902958 \times 10^{21}]\)
Interval arithmetic

\( F \) interval extension

\[ f(x, y) = x^2 + \exp(y), x \in [-3, 2], y \in [0, 1]: \]
\[ F([-3, 2], [0, 1]) = [-3, 2]^2 + \exp([0, 1]) = [0, 9] + [1, e] = [1, e + 9] \]

\[ f(x) = \sin(x), x \in [1, 2]: F([1, 2]) = [\sin(1), 1] \]
Interval Branch and Bound Algorithms
Why a new interval library?

Existing interval libraries
- C++ libraries: PROFIL/BIAS, Interval in Boost C++, Gaol
- Sun interval library: Fortran 95 or C++ [Mic01]
- MPFI: C or C++ [RR02]
- ...

Motivations for a new library
- Preference for OCaml functional programming
- Need for speed and efficiency
1 Introduction
- Objectives
- Interval arithmetic
- Why a new interval library?

2 A functional implementation for OCaml
- Unexpected problems
- Modules
- Implementation principles

3 Performance comparison
Unexpected problems

C (and OCaml) math functions

- do not return the same results on 32- and 64-bit architectures
- may yield wrong results in low and high rounding modes

Necessity to write **low-level functions** in assembly language
(restricted to x86/Amd64 architectures)
A functional implementation for OCaml

Modules

- **Chcw**: elementary float functions
  - in all rounding modes (nearest, low, high)
  - written in C and assembly language

- **Fpu**: OCaml binding to Chcw

- **Interval**: OCaml interval arithmetic

- **Fpu_rename**: redefinition of OCaml float functions
  - Except float operators: (+.) (−.) (∗.) (/.)

- **Fpu_rename_all**: redefinition of all OCaml float functions
  - Including float operators
Module Fpu

val ffloat: int -> float
val ffloat_high: int -> float
val ffloat_low: int -> float

(** Float functions. The float function is exact on 32-bit machines but not on 64-bit machines with ints larger than 53 bits *)

val fadd: float -> float -> float
val fadd_low: float -> float -> float
val fadd_high: float -> float -> float

(** Floating-point addition in nearest, low and high modes *)

val fsub: float -> float -> float
val fsub_low: float -> float -> float
val fsub_high: float -> float -> float

(** Floating-point substraction in nearest, low and high modes *)
Module Fpu_rename_all

val (+.): float -> float -> float
val (-.): float -> float -> float
val (/ .): float -> float -> float
val ( * .): float -> float -> float

val mod_float: float -> float -> float
val sqrt: float -> float
val log: float -> float
val exp: float -> float

val ( ** ): float -> float -> float

- Opening the Fpu_rename_all module ensures that these functions return the same results on all architectures
**Interface of module Interval**

```ocaml
type interval = {
  low: float; (** lower bound **)
  high: float (** upper bound **)
}

val pi_l: interval

val float_i: int -> interval

val (+$.): interval -> float -> interval
val (+$): interval -> interval -> interval

val sqrt_l: interval -> interval
val pow_l_i: interval -> int -> interval
```
Implementation principles

Rules

- No “empty” interval (raise an exception instead)
- Interval bounds may be infinite
- Interval bounds are not supposed to be `nan`
  Lower bound is not supposed to be `infinity`
  Upper bound is not supposed to be `neg_infinity`
- `nan`, `infinity` and `neg_infinity` operands are not handled

Examples

- `1/[0, 0]` fails
- `1/[0, 1]` returns `[1, +∞]`
- `1/[−1, 1]` returns `[−∞, +∞]`
Module Interval

(...)

let inv_l {low = a; high = b} =
  let sa = compare a 0. and sb = compare b 0. in
  if sa = 0 then
    if sb = 0 then failwith "inv_l"
    else {low = fdiv_low 1. b; high = infinity}
  else if 0 < sa || sb < 0 then {low = fdiv_low 1. b; high = fdiv_high 1. a}
  else if sb = 0 then {low = neg_infinity; high = fdiv_high 1. a}
  else {low = neg_infinity; high = infinity}

(...)

Charlie Vanaret (ENAC-IRIT) Implementing an interval library for OCaml OUD 2012 17 / 23
Introduction
- Objectives
- Interval arithmetic
- Why a new interval library?

A functional implementation for OCaml
- Unexpected problems
- Modules
- Implementation principles

Performance comparison
Performance comparison

Comparison with an OCaml binding to MPFI
(runtime for $10^6$ operations)
Conclusion

Native implementation

- low-level redefinition of elementary functions
- reliable, fast despite the functional paradigm
- available under GNU Lesser General Public License

Successfully used in our hybrid optimization algorithm

- computation of optima of Michalewicz function (improvement for deterministic methods)
- applications to aeronautical problems (air traffic conflict resolution)
References


Questions

For implementation details, please contact

Jean-Marc Alliot
jean-marc.alliot@irit.fr

Jean-Baptiste Gotteland
gottelan@cena.fr
Implementing an interval computation library for OCaml on x86/AMD64 architectures

J-M. Alliot  J-B. Gotteland  C. Vanaret  N. Durand  D. Gianazza

OCaml Users and Developers Workshop 2012
International Conference in Functional Programming

September 14, 2012