# Efficient Hardware Primitives for Interval Contractors in Robotics and Integration to a Custom RISC-V ISA Extension

Pierre Filiol †
Théotime Bollengier †
Luc Jaulin †
Jean-Christophe Le Lann †

2025 23rd IEEE Interregional NEWCAS Conference (NEWCAS) | 979-8-3315-3256-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/NEWCAS64648.2025.11107029

***Abstract*—Interval analysis is commonly used in mobile robotics to solve nonlinear problems. Typical implementations mostly rely on software libraries which perform poorly on low-power embedded devices. This article shows how interval primitives can be ported on FPGA and integrated to the RISC-V architecture using a custom ISA extension called *xinterval*. The novelty of this approach is the emphasis on elementary interval contractors which are higher-level abstractions than plain interval arithmetic especially well suited for robotics applications.**

***Index Terms*—Interval computing, contractors, robotics, RISC-V, FPGA**

## I. Introduction

Many engineering problems require to solving set of equations and inequalities or find the optimal solution to a particular cost function. This task is far from trivial if the considered equations/functions are nonlinear, which is sadly the norm in most of real-world applications. Interval analysis [13] is a numerical method which allows to build a guaranteed approximation of the set of all the solutions even for nonlinear problems.

A lot of recurring tasks encountered in robotics such as localization or robust control can be described as constraint-satisfaction problems and solved with interval analysis [8], [15]. Among the main benefits of this approach over classical ones are speed, guaranteed computations and fault-tolerance which are always big concerns in autonomous systems. Typical programs are generally implemented using general-purpose interval software libraries which can lead to performance issues in the most time-critical applications. Ideally, the user should be able to adjust the trade-off between speed and precision of all interval computations without sacrificing guarantees. Obviously this acceptable threshold will differ from an application to another.

The goal of this article is to accelerate interval arithmetic for mobile robotics using dedicated hardware primitives integrated as a RISC-V ISA custom extension. The resulting architecture uses the most recurrent elementary interval contractors as building blocks for more complex logic. Our studies and first simulations confirm the validity of this approach, with execution times reduced by a factor between 2 and 10 depending of the considered problem.

This article is organized as follows. In Section II, we will formalize the notion of elementary interval contractors and show how they can be used in real-world problems. In Section III, we present the state of the art in software implementations. In Section IV, we present the integration issues that arise when integrating intervals to RISC-V. Then, Section V details our hardware architecture. Section VI presents the first performance results, proving the validity of our approach. Finally we conclude and explore future works in Section VII.

† : Labsticc, Ensta-Bretagne

## II. From Interval Arithmetic to Contractors

Robotics mainly uses the *set-based* interval model as specified in the IEEE-1788 standard [9]. The set of mathematical interval $\mathbb{IR}$ comprises all the subsets $\mathbf{x}$ of the real line $\mathbb{R}$ which are closed and connected in the topological sense. This includes the empty set $\emptyset$ and all non-empty intervals $[x]$ such as :

$$[x] = [\underline{x}, \bar{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \bar{x}\}, \tag{1}$$

where $\underline{x}$ (resp. $\bar{x}$) is the interval lower (resp. upper) bound. The standard also gives a list of recommended arithmetic operators and their expected behaviours. For example, interval addition is defined as follow :

$$[x] + [y] = \begin{cases} \mathbb{IR} \times \mathbb{IR} \to \mathbb{IR} \\ [\underline{x} + \underline{y}, \bar{x} + \bar{y}] \end{cases} \tag{2}$$

Above this layer of arithmetic operators, we can construct interval contractors [2], [10] to tackle complex constraints-solving problems at an higher level of abstraction.

A *contractor* $C$ for the set $\mathbb{X} \subset \mathbb{R}^n$ is an operator $\mathbb{IR}^n \to \mathbb{IR}^n$ which satisfies:

$$C([\mathbf{x}]) \subset [\mathbf{x}] \ (contractance) \tag{3}$$

$$[\mathbf{x}] \subset [\mathbf{y}] \implies C([\mathbf{x}]) \subset C([\mathbf{y}]) \ (monotonicity) \tag{4}$$

$$C([\mathbf{x}]) \cap \mathbb{X} = [\mathbf{x}] \cap \mathbb{X} \ (consistency) \tag{5}$$



Authorized licensed use limited to: ENSTA. Downloaded on December 10,2025 at 23:15:06 UTC from IEEE Xplore. Restrictions apply.

where $\mathbb{IR}^n$ is the set of axis-aligned boxes of $\mathbb{R}^n$.

A contractor is said to be minimal if for all boxes $[x]$, we have $C([\mathbf{x}]) = [[\mathbf{x}] \cap \mathbb{X}]$ where $[\mathbf{A}]$ is the *interval hull* ie the smallest box enclosing $\mathbf{A}$.

From that point, it is possible to build a pair of contractors (denoted respectively as *forward* and *backward*) associated to every common interval operator. In this article forward and backward contractors on operator/function $*$ are respectively defined as $\overrightarrow{C_*}$ and $\overleftarrow{C_*}$. Some operators have trivial contractors such as addition where the constraint $x1 + x2 = x3$ leads to :

$$\overrightarrow{C_+} : \{[x_3] = [x_3] \cap ([x_1] + [x_2]) \tag{6}$$

$$\overleftarrow{C_+} : \begin{cases} [x_1] = [x_1] \cap ([x_3] - [x_2]) \\ [x_2] = [x_2] \cap ([x_3] - [x_1]) \end{cases} \tag{7}$$

Non-monotonic or partially defined operators such as square, square-root, sine or cosine have more complex minimal contractors. Figure 1 shows a graphic representation of the square function contractor where red boxes get contracted into green boxes.

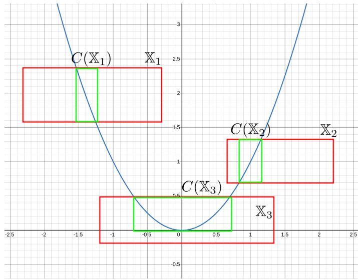


Fig. 1. Contractor for the square function

Tackling real-world robotics problems is achieved by combining elementary minimal contractors with algorithms such as HC4-revise which procedurally derives forward-backward contractors based on a set of input constraints. This process is often depicted in literature for a large scope of applications [1], [11]. An example of complete workflow for a simple localization problem involving a robot attempting to localize itself inside a field of landmarks with known positions is detailed in one of our previous article [6].

## III. State-of-the-Art and Previous Works

Robotics algorithm involving intervals rely on a limited set of software libraries with various levels of abstraction over interval arithmetic. The ecosystem of available tools and their dependencies is depicted on Figure 2.

Libraries can be classified into two broad categories in term of floating-point layer. Those on the left-side directly depends on INRIA's *crlibm* whose primary goal is to provide correctly rounded floating-point operators. This is achieved by performing the operations at a much higher precision than the input value (eg. float32) and then truncating the result to secure the last ulp (unit in the last place). Naturally there is a computing overhead associated to this process compared to an implementation like libc. Libraries on the right-side depends on INRIA's MPFR library which puts an emphasis on custom-precision floating-point operators with correct rounding. This is done by emulating custom-float numbers using large integers from the GMP library to reach perfect precisions. With this method, no use is made of the floating-point pipeline in the underlying CPU thus massively reducing execution speed.

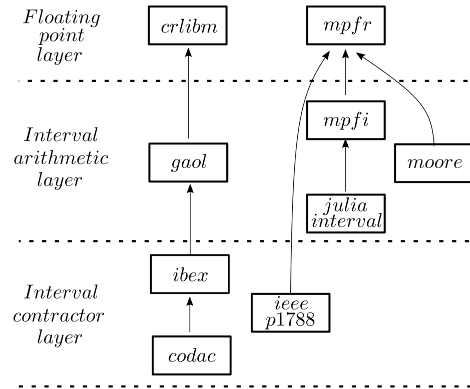


Fig. 2. The ecosystem of interval software libraries in robotics

This article advocates an approach which prioritizes the speed of interval computations at the price of an increased pessimism (larger intervals) but without sacrificing guarantees (the true value belongs to the interval result). Typical use-cases of mobile robotics involve feeding a stream of continuous sensor measurements in a complex interval contractor to compute a parameter estimation. It is generally more rewarding to perform another round of contractor evaluation with fresh measurements rather than spending extra time validating the last ulp of an interval bound.

The proposed solution is to handle elementary interval contractors directly at hardware level by the mean of a FPGA target. This strategy offers a finer control over the speed/precision trade-off of floating-point computations which is directly reflected in the overall interval performances. A benefit regarding energy consumption is also to be expected since FPGA implementations are typically more power-efficient than their software counterparts.

There are few studies in the literature on interval arithmetic support in hardware. One noticeable work is [16] by Schulte describes an ASIC implementation for a multi-precision interval arithmetic ALU (addition, multiplication, division). This fairly old study is oriented toward high-precisions and gives a most likely outdated baseline for circuit areas and latencies. A similar work initiated by Kulisch [12] presents theoretical circuits and algorithms for the basic interval arithmetic operators. Some thoughts are made regarding possible integration on RISC processors.

The novelty of our approach is to give a performance baseline of a FPGA implementation centered around interval contractors for use in mobile robotics. Another specificity is



Authorized licensed use limited to: ENSTA. Downloaded on December 10,2025 at 23:15:06 UTC from IEEE Xplore. Restrictions apply.

the integration to RISC-V architecture which is a continuation of our previous work [6].

## IV. Interfacing intervals with RISC-V

The proposed hardware primitives for interval contractors are implemented in the context of a custom RISC-V [17] extension named *xinterval* that we specified and developed in [6] . It is compatible with either 32 or 64-bit processors as long as the standard extension D (double precision floating-point) is supported.

Intervals are represented using a 64-bit data-structure as depicted in Figure 3 and hosted in floating-point registers. Bounds are represented using a custom 31-bits floating-point shorter of 1 exponent bit compared to the classical float32, thus decreasing the range of representable numbers. This concession is not too penalizing in typical robotic use-cases and gives room for two extra bits to mark an interval as empty (useful in arithmetic) or to indicate the iota flag [7].

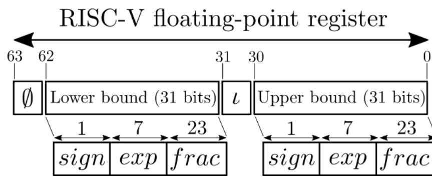


Fig. 3. Interval representation in xinterval

All instructions from *xinterval* share the opcode $0xB$ (dedicated to custom extensions) and use the R-format specified by the RISC-V standard (figure 4) .

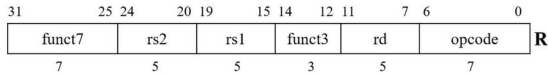


Fig. 4. RISC-V R instruction encoding

The instructions presented in Table I are used as a bridge between floating-point and intervals. The role of *itvmake* and *itvmake.i* is to use two fp32 bounds in $rs_1/rs_2$ and to return the associated representation from Figure 3 (with iota flag set in the second case). Internally, fp32 bounds are converted to fp31 and their respective order is checked to produce a valid interval (unordered bounds return ∅). Instructions *itv.e*, *itv.i*, *itv.lb*, *itv.ub* are respectively used to retrieve the empty flag, iota flag, lower bound and upper bound of the input interval in $rs_1$. For *itv.lb* and *itv.ub*, a conversion from fp31 to fp32 is performed and the result is written back in floating-point registers.

The instructions presented in Table II are used to perform set operations on intervals. An intersection (resp. Union) between two intervals in $rs_1/rs_2$ can be computed using *itv.set.i* (resp *itv.set.u*). These instructions are used in forward and backward contractors to re-intersect the output with its original value as in Equations 6 and 7. Without them, most contractor instructions would require the R4-format specified by the RISC-V standard and two opcodes would be necessary (and thus bloating the instruction encoding space).

| Opcode | Funct3 | Funct7 | HW circuit | Asm mnemonic |
|---|---|---|---|---|
| 0xB | 0x0 | 0x0 | ItvMake | itvmake |
| 0xB | 0x1 | 0x0 | ItvMake | itvmake.i |
| 0xB | 0x0 | 0x1 | ItvData | itv.e |
| 0xB | 0x1 | 0x1 | ItvData | itv.i |
| 0xB | 0x2 | 0x1 | Mk1ItvData | itv.lb |
| 0xB | 0x3 | 0x1 | Mk1ItvData | itv.ub |

TABLE I
Instructions for FP32 ↔ ITV

| Opcode | Funct3 | Funct7 | HW circuit | Asm mnemonic |
|---|---|---|---|---|
| 0xB | 0x0 | 0x2 | ItvSet | itv.set.u |
| 0xB | 0x1 | 0x2 | ItvSet | itv.set.i |

TABLE II
Instructions for ITV set operators

Finally, Table III contains the forward and backward half-contractors (ie not re-intersected with the original output) for a set of usual arithmetic and transcendental functions.

| Opcode | Funct3 | Funct7 | HW circuit | Asm mnemonic |
|---|---|---|---|---|
| 0xB | 0x0 | 0x3 | ItvAdd | itv.add |
| 0xB | 0x1 | 0x3 | ItvAdd | itv.sub |
| 0xB | 0x0 | 0x4 | ItvMul | itv.mul |
| 0xB | 0x1 | 0x4 | ItvMul | itv.sqr |
| 0xB | 0x0 | 0x5 | ItvDiv | itv.div |
| 0xB | 0x0 | 0x6 | ItvSqrt | itv.sqrt |
| 0xB | 0x1 | 0x6 | ItvSqrt | itv.sqrrevhalf |
| 0xB | 0x0 | 0x7 | ItvExp | itv.exp |
| 0xB | 0x0 | 0x8 | ItvLog | itv.log |
| 0xB | 0x0 | 0x9 | ItvSinCos | itv.cos |
| 0xB | 0x1 | 0x9 | ItvSinCos | itv.sin |
| 0xB | 0x0 | 0xA | ItvAsinAcos | itv.cosrevhalf |
| 0xB | 0x1 | 0xA | ItvAsinAcos | itv.sinrevhalf |

TABLE III
Instructions for ITV contractors

Listing IV shows how everything can be used to compute the backward contractor from 7 using C language.

```
/* interval as double (64 bits ) */
typedef itv_t double ;

/* Inlining of xinterval instruction itv.set.i */
inline itv_t __attribute__((always_inline))
itv_set_i(itv_t itv1, itv_t itv2) {
  itv_t result;
  asm("itv.set.i %0, %1, %2" : "=f"(result) : "f"(
    itv1), "f"(itv2));
  return result;
}

/* Inlining of xinterval instruction itv.sub */
inline itv_t __attribute__((always_inline))
itv_sub(itv_t itv1, itv_t itv2) {
  itv_t result;
  asm("itv.sub %0, %1, %2" : "=f"(result) : "f"(itv1
    ), "f"(itv2));
  return result;
}
```



Authorized licensed use limited to: ENSTA. Downloaded on December 10,2025 at 23:15:06 UTC from IEEE Xplore. Restrictions apply.

```
20 /* Backward ctc for x1 */
21 itv_t addbwctc1(itv_t x1, itv_t x2, itv_t x3) {
22     itv_t sub = itv_sub(x3, x2);
23     itv_t inter = itv_set_i(x1, sub);
24     return inter;
25 }
```

Listing 1. Addition backward contractor using *xinterval*

## V. Hardware Architecture

This section presents the generic workflow used to implement the hardware circuits behind the instructions from table III and gives a detailed example for the multiplication contractor.

The performances of hardware interval operators are heavily impacted by the quality of the underlying floating-point primitives. Most textbook implementations of IEEE-754 algorithms [5] are targeted at ASIC chips and typically perform poorly on FPGA as architectures differ vastly [14, p. 269ff]. The proposed implementation uses the INRIA's FloPoCo project [3], [4] which is a VHDL code generator offering FPGA-optimized pipelined circuits for recurring floating-point primitives. A nice feature is the support for exotic floating-point formats such as the float31 from Figure 3.

The implemented interval contractors can be broadly separated in 3 categories :

- Group I : ALU arithmetic operators. eg : $\overrightarrow{C_*}$ and $\overleftarrow{C_*}$ for $+, -, \times, \div$
- Group II : Functions involving a partially-defined operator in either forward or backward part. eg : $\overrightarrow{C_*}$ and $\overleftarrow{C_*}$ for $x^2, \sqrt{x}, \exp, \log$.
- Group III : Non-monotonic and periodic trigonometric functions. eg : $\overrightarrow{C_*}$ and $\overleftarrow{C_*}$ for $\sin, \cos$.

Contractors from groups I and II closely match their respective arithmetic operator counterparts as depicted in Equations 6 and 7. Some subtleties are involved when input intervals cross several monotonicity domains for a given function ($\mathbb{X}_3$ in figure 1). In contractors involving partially-defined functions like square-root or logarithm, the result must be carefully re-intersected with the domain of definition.

Contractors from group III are more complex. Trigonometric functions are notoriously difficult to evaluate on a FPGA and require the use of fix-point arithmetic in combination with techniques such as CORDIC or polynomial approximations. The results are often available only for a specific range of input (eg $[-\pi, \pi]$ for sine/cosine) due to limited hardware resources. A direct consequence is that additional circuits dedicated to range reduction are necessary to perform the conversion between an argument and its equivalent in the supported input range. The associated interval algorithms are far more complicated due to the periodicity/non-monotonicity of the trigonometric functions and require much more control logic in average than the ALU operators.

To demonstrate the implementation process, the circuit of the interval contractor responsible for the multiplication/square (*ItvMul* from Figure III) is detailed below.

The IEEE-1788 standard defines the result of the interval multiplication $[x] \times [y]$ as

$$[\underline{z}, \bar{z}] \; with \; \begin{cases} \underline{z} = \min(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}) \\ \bar{z} = \max(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}) \end{cases} \quad (8)$$

To compute the result efficiently in hardware, the input intervals can be classified according to Table IV:

| Encoding | Type |
|---|---|
| 000 | $\emptyset$ |
| 001 | $\{0\}$ |
| 010 | $]-\infty, +\infty[$ |
| 011 | $\subset [a, 0] \mid a < 0$ |
| 100 | $\subset [0, a] \mid a > 0$ |
| 101 | $[a, b] \mid a < 0 < b$ |

TABLE IV
Input classification for multiplication

Multiplications involving $\emptyset$, 0 and $]-\infty, +\infty[$ as one of their inputs produce the same result (in this order of priority). Other cases are summarized in Table V.

| Encoding x and y | 011 | 100 | 101 |
|---|---|---|---|
| 011 | $[\bar{x}\bar{y}, \underline{x}\underline{y}]$ | $[\underline{x}\bar{y}, \bar{x}\underline{y}]$ | $[\underline{x}\bar{y}, \underline{x}\underline{y}]$ |
| 100 | $[\underline{x}\bar{y}, \bar{x}\underline{y}]$ | $[\underline{x}\underline{y}, \bar{x}\bar{y}]$ | $[\bar{x}\underline{y}, \bar{x}\bar{y}]$ |
| 101 | $[\underline{x}\bar{y}, \underline{x}\underline{y}]$ | $[\bar{x}\underline{y}, \bar{x}\bar{y}]$ | $[\underline{t}, \bar{t}] = [\min(\underline{x}\bar{y}, \bar{x}\underline{y}), \max(\bar{x}\bar{y}, \underline{x}\underline{y})]$ |

TABLE V
Bound computations for multiplication

Figure 5 and Table VI shows how this algorithm is implemented in hardware. Red lines represent float31 values and blue lines other control signals.

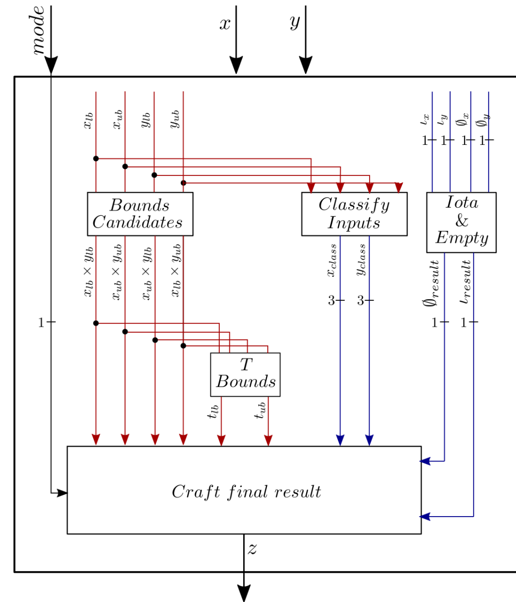


Fig. 5. High-level view of the ItvMul circuit



Authorized licensed use limited to: ENSTA. Downloaded on December 10,2025 at 23:15:06 UTC from IEEE Xplore. Restrictions apply.

| Circuit | Flopoco operators |
|---|---|
| Bound candidates | 4 Fp31 Multipliers |
| Classify inputs | 4 Fp31 Comparators |
| T bounds | 2 Fp31 Comparators |

TABLE VI
FLOPOCO OPERATORS USAGE IN ITVMUL

## VI. RESULTS AND PERFORMANCES

All the aforementioned primitives have been all implemented on the Vivado Artix 7 XC7A35T FPGA and can fit simultaneously. The ability to run on this fairly small chip in term of hardware resources demonstrates the suitability of this approach.

The achieved performances of the FP31 FloPoCo hardware blocks used to build the interval contractors are summarized in Table VII for a clock speed of 100 MHz.

| | LUTs | Reg. | Muxes | DSP | BRAM | Cycles |
|---|---|---|---|---|---|---|
| FpComparator | 90 | 66 | 0 | 0 | 0 | 2 |
| FpAdder | 155 | 199 | 0 | 0 | 0 | 6 |
| FpMultiplier | 136 | 175 | 0 | 2 | 0 | 5 |
| FpDivider | 753 | 568 | 0 | 0 | 0 | 14 |
| FpSqrt | 482 | 373 | 0 | 0 | 0 | 12 |
| FpExp | 324 | 401 | 279 | 1 | 0 | 10 |
| FpLog | 640 | 828 | 24 | 3 | 0 | 12 |
| FpFma | 1081 | 653 | 0 | 2 | 0 | 7 |
| FpSinCos | 706 | 407 | 0 | 2 | 1 | 7 |
| FpAtan2 | 927 | 632 | 30 | 0 | 0 | 7 |

TABLE VII
FP31 FLOPOCO PRIMITIVES ON XC7A35T @ 100MHZ

The associated interval primitives have the performances from Table VIII in the same conditions :

| | LUTs | Reg. | Muxes | DSP | BRAM | Cycles |
|---|---|---|---|---|---|---|
| ItvAdd | 1334 | 880 | 0 | 0 | 0 | 7 |
| ItvMul | 1115 | 1002 | 7 | 8 | 0 | 8 |
| ItvDiv | 3399 | 2893 | 52 | 2 | 0 | 15 |
| ItvSqrt | 896 | 720 | 0 | 0 | 0 | 12 |
| ItvExp | 650 | 805 | 558 | 2 | 0 | 10 |
| ItvLog | 1279 | 1673 | 48 | 6 | 2 | 12 |
| ItvSinCos | 6105 | 5123 | 0 | 20 | 2 | 37 |
| ItvAsinAcos | 5320 | 4125 | 327 | 0 | 0 | 31 |

TABLE VIII
INTERVAL PRIMITIVES ON XC7A35T @ 100MHZ

Measuring performance gains relative to traditional software approaches is tricky as it is easy to perform an unfair comparison. Figure 2 shows that existing implementations does not rely on real floating-point hardware to compute intervals and are thus at a clear disadvantage. As a consequence, we have re-implemented most interval algorithms from MPFI using the real float32 type for bounds and the float functions from the RISC-V *newlib* libc. This library is targeted at embedded applications and is typically less precise that the full-system libc, making it the closer contender for the proposed hardware implementation. The evaluation protocol compares the number of cycles required to evaluate various interval contractors taken from the literature in two scenarios. The first one uses a solution based on the minimal MPFI re-implementation compiled for standard RISC-V (ie without *xinterval*). The other one uses the same algorithm but with only *xinterval* primitives. Internally, clock-cycles counting is performed using a custom RISC-V software emulator and the knowledge of the performances of the interval primitives.

## VII. CONCLUSION

The implementation of interval calculations on hardware was initiated in part by Schulte and Kulich, but without the use of contractors. In this paper, we have fully implemented these computations on a FPGA board. The validity of this approach has been demonstrated by theory and various implementations. The gains in terms of execution compared with software libraries are a factor of between 2 and 10. The simulation and evaluation protocol will be published in a forthcoming article.

## REFERENCES


[1] F. Benhamou, F. Goualard, L. Granvilliers and J.-F. Puget, “Revising Hull and Box Consistency,” Procs. of the 16th Intl. Conf. on Logic Programming, pp. 230-–244, 1999.

[2] G. Chabert and L. Jaulin, “Contractor programming”, Artificial Intelligence, 173(11), pp. 1079-–1100, 2009.

[3] F. de Dinechin and B. Pasca, “Designing Custom Arithmetic Data Paths with FloPoCo,” IEEE Design & Test of Comp., 28(4), pp. 18–27, 2011.

[4] F. de Dinechin, “Reflections on 10 Years of FloPoCo,” 26th Symposium on Computer Arithmetic (ARITH), Kyoto, Japan, pp. 187–189, 2019.

[5] M. D. Ercegovac and T. Lang, “Digital Arithmetic”, Morgan Kaufmann, 2004.

[6] P. Filiol, T. Bollengier, L. Jaulin and J.-C. Le Lann, “RISC-V Based Hardware Acceleration of Interval Contractor Primitives in the Context of Mobile Robotics”, Acta Cybernetica, 26(4), pp. 889-912, 2024.

[7] P. Filiol, T. Bollengier, L. Jaulin and J.-C. Le Lann, “A New Interval Arithmetic to Generate the Complementary of Contractors,” Acta Cybernetica, 26(4), pp. 817–838, 2024.

[8] R. Guyonneau, S. Lagrange, L. Hardouin and P. Lucidarme, “Guaranteed Interval Analysis Localization for Mobile Robots”, Advanced Robotics, 28(16), pp. 1067-–1077, 2014.

[9] IEEE SA, “IEEE 1788-2015: IEEE Standard for Interval Arithmetic”, https://standards.ieee.org/ieee/1788/4431/, 2015.

[10] L. Jaulin, M. Kieffer, O. Didrit and E. Walter, “Applied Interval Analysis”, Berlin: Springer, 2001.

[11] I. K. Kueviakoe, A. Lambert and P. Tarroux, “Comparison of Interval Constraint Propagation Algorithms for Vehicle Localization”, J. of Soft. Eng. and Appl., 5, pp. 157–162, 2012.

[12] U. W. Kulisch and R. Kirchner, “Hardware Support for Interval Arithmetic,” Reliable Comput. 12, pp. 225-–237, 2006.

[13] R. E. Moore, “Interval Analysis”, Prentice-Hall, 1966.

[14] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol and S. Torres, “Handbook of Floating-Point Arithmetic”, Birkhäuser Cham, 2018.

[15] M. Mustafa, A. Stancu, S. P. Guteirrez, E. A. Codres and L. Jaulin, “Rigid Transformation Using Interval Analysis for Robot Motion Estimation,” 20th International Conference on Control Systems and Computer Science, Bucharest, Romania, pp. 24-31, 2015.

[16] M. J. Schulte and E. E. Swartzlander, “Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[17] A. Waterman, Y. Lee, D. A. Patterson and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: UserLevel ISA, Version 2.1,” Univ. of California at Berkeley, 2016.




Authorized licensed use limited to: ENSTA. Downloaded on December 10,2025 at 23:15:06 UTC from IEEE Xplore. Restrictions apply.