

A Data-Parallel Algorithm to Reliably Solve Systems of Nonlinear Equations

Frédéric Goualard

Université de Nantes, Nantes Atlantique Universités, CNRS, LINA, UMR 6241
2 rue de la Houssinière, BP 92208, F-44000 NANTES
Frederic.Goualard@univ-nantes.fr

Alexandre Goldsztejn

CNRS, LINA, UMR 6241. 2 rue de la Houssinière, BP 92208, F-44000 NANTES
Alexandre.Goldsztejn@univ-nantes.fr

Abstract

Numerical methods based on interval arithmetic are efficient means to reliably solve nonlinear systems of equations. Algorithm `bc3revise` is an interval method that tightens variables' domains by enforcing a property called box consistency. It has been successfully used on difficult problems whose solving eluded traditional numerical methods. We present a new algorithm to enforce box consistency that is simpler than `bc3revise`, faster, and easily data parallelizable. A parallel implementation with Intel SSE2 SIMD instructions shows that an increase in performance of up to an order of magnitude and more is achievable.

1 Introduction

Interval methods [12] are numerical algorithms that use *interval arithmetic* [11] to avoid rounding error problems intrinsic to floating-point arithmetic [7]. They give enclosures of all solutions of nonlinear systems of equations with the guarantee that no solution is ever lost.

Straight interval extensions of classical numerical algorithms such as the Newton method are not well-suited to problems with many solutions or with large initial domains for the variables. To tackle these shortcomings, elaborate algorithms have been devised in the context of *Interval Constraint Programming* [1]; they are usually employed as the inner stage of a free-steering nonlinear Gauss-Seidel method to exclude parts of a variable's domain that do not contain zeroes of a unidimensional equation. Domain tightening is achieved by enforcing some local consistency property. *Box consistency* [2] is one such consistency notion, which has been proved efficient in handling

hard problems whose solving eluded traditional numerical methods for years [6]. It is usually enforced by Algorithm `bc3revise` [2], which combines a binary search with unidimensional interval Newton steps [11] to isolate leftmost and rightmost zeroes of a unidimensional equation in the domain of a variable.

Thanks to ubiquitous Intel SSE2 SIMD instructions, it is possible to perform many interval operations at roughly the same cost as floating-point operations by computing the two bounds of the result in parallel (*basic interval vectorization*) [5]. We outline in Section 2 a novel way to do even better and to compute an interval function for two different intervals in parallel (a four times speed-up compared to “sequential” interval evaluation).

As all interval methods, Algorithm `bc3revise`—described in Section 3—can benefit from basic interval vectorization without any modification. On the other hand, it cannot take full advantage of the new arithmetic described in Section 2. Hence the introduction of Algorithm `sbc` in Section 4.1: it is a new algorithm that enforces box consistency by “shaving” domains from the left and right bounds inward. Experiments show that a sequential version of `sbc` is already faster than `bc3revise` on a set of test problems. We then describe in Section 4.2 an algorithm that exploits the potential for a high level of data parallelism in `sbc` by using SSE2 instructions to perform interval arithmetic evaluations of functions at four times the speed of a sequential code. Experiments are reported in Section 5 and show increases in performances over `bc3revise` of up to an order of magnitude and more.

2 Interval Arithmetic and its Vectorization

Classical iterative numerical methods suffer from defects such as loss of solutions, absence of convergence, and convergence to unwanted attractors due to the use of floating-point numbers (aka *floats*). At the end of the fifties, Moore [11] popularized the use of intervals to control the errors made while computing with floats. Additionally, interval extensions of iterative numerical methods are always convergent.

In the following, we use the notations sponsored by Kearfott and others [9], where interval quantities are bold-faced.

Interval arithmetic replaces floating-point numbers by closed connected sets of the form $\mathbf{I} = [\underline{I}, \overline{I}] = \{a \in \mathbb{R} \mid \underline{I} \leq a \leq \overline{I}\}$ from the set \mathbb{I} of intervals, where \underline{I} and \overline{I} are floating-point numbers. In addition, each n -ary real function ϕ with domain $\mathcal{D}_\phi \subseteq \mathbb{R}^n$ can be extended to an interval function Φ with domain $\mathcal{D}_\Phi \subseteq \mathbb{I}^n$ in such a way that the containment principle is verified:

$$\forall A \in \mathcal{D}_\phi, \forall \mathbf{I} \in \mathcal{D}_\Phi: A \in \mathbf{I} \implies \phi(A) \in \Phi(\mathbf{I}), \quad (1)$$

as illustrated by the following example.

Example 1 The natural interval extensions of addition and multiplication are defined by:

$$\begin{aligned} \mathbf{I}_1 + \mathbf{I}_2 &= [\downarrow \underline{I}_1 + \underline{I}_2 \downarrow, \uparrow \overline{I}_1 + \overline{I}_2 \uparrow] \\ \mathbf{I}_1 \times \mathbf{I}_2 &= [\min(\downarrow \underline{I}_1 \underline{I}_2 \downarrow, \downarrow \underline{I}_1 \overline{I}_2 \downarrow, \downarrow \overline{I}_1 \underline{I}_2 \downarrow, \downarrow \overline{I}_1 \overline{I}_2 \downarrow), \\ &\quad \max(\uparrow \underline{I}_1 \underline{I}_2 \uparrow, \uparrow \underline{I}_1 \overline{I}_2 \uparrow, \uparrow \overline{I}_1 \underline{I}_2 \uparrow, \uparrow \overline{I}_1 \overline{I}_2 \uparrow)] \end{aligned}$$

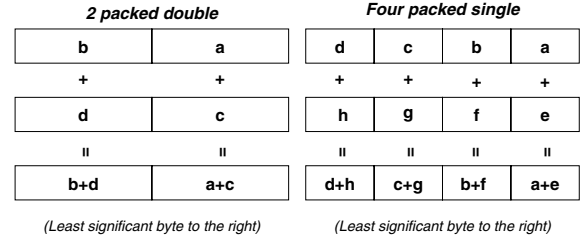
where $\downarrow r \downarrow$ (resp., $\uparrow r \uparrow$) is the greatest floating-point number smaller or equal (resp., the smallest floating-point number greater or equal) to r .

Then, given the real function $f(x, y) = x \times x + y$, we may define its natural interval extension by $\mathbf{f}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \times \mathbf{x} + \mathbf{y}$, and we have that, e.g., $\mathbf{f}([-2, 3], [-1, 5]) = [-7, 14]$, where $[-7, 14]$ is an interval that contains the real domain of f over $([-2, 3], [-1, 5])$ (containment principle (1)).

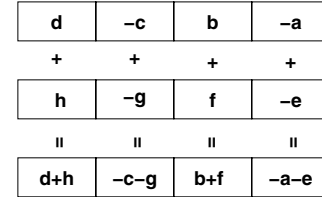
Implementations of interval arithmetic use outward rounding to enlarge the domains computed so as not to violate the containment principle (1), should some bounds be unrepresentable with floating-point numbers.

Interval addition, subtraction, multiplication, division, and integral exponentiation may be computed at roughly the same speed as their floating-point counterpart thanks to SIMD instructions, and in particular, to Intel SSE2 instructions.

Intel SSE2 instructions manipulate 128 bits registers that may be interpreted in various ways. Most notably, the registers may pack 2 double precision or 4 single precision



(a) SIMD floating-point arithmetic



(Least significant byte to the right)

(b) Two interval additions with one SSE2 instruction

Figure 1. Floating-point arithmetic and interval arithmetic in SSE2 registers

floating-point numbers. An SSE2 operator may then compute 2 or 4 floating-point operations in parallel (see Figure 1(a)).

The direction of rounding for SSE2 instructions is selected independently of that of the Floating-Point Unit (FPU). An SSE2 instruction uses the same rounding for all operations performed in parallel. Nevertheless, thanks to simple floating-point properties, it is still possible to write algorithms that compute in parallel the two outward-rounded bounds of the result of interval operations. For example, we may use the property:

$$\downarrow a + b \downarrow = - \uparrow -a - b \uparrow$$

where a and b are floating-point numbers.

By storing the negation of the left bound of an interval, and by setting once and for all the rounding direction for SSE2 instructions to $+\infty$, the two interval additions $[a, b] + [e, f]$ and $[c, d] + [g, h]$ can be performed by the sole SIMD instruction depicted in Figure 1(b). All the other operators may be defined accordingly. Goualard's paper [5] illustrates these principles for the case of basic interval vectorization (two double precision bounds computed in parallel). The algorithms to compute four bounds in parallel are new and are reported in an unpublished paper currently under review.

Armed with an interval library whose operators compute two interval operations in parallel, we may evaluate the interval extension of a function \mathbf{f} for two different intervals for roughly the same cost as one floating-point evaluation of

f . In the following, we note $\llbracket f(I_1), f(I_2) \rrbracket$ such a parallel evaluation of f for two different interval arguments.

3 Box Consistency and the bc3revise Algorithm

Interval Constraint Programming [1] is a successful approach to reliably isolate all solutions of systems of equations. It makes cooperate *contracting operators* to prune the domains of the variables (intervals with floating-point bounds from the set \mathbb{I}) with smart *propagation algorithms* [10]—akin to free-steering nonlinear Gauss-Seidel—to ensure consistency among all the constraints.

The amount of pruning obtained from one equation is controlled by the level of consistency enforced. Box consistency [2] is defined as follows:

Definition 1 (Box consistency) *An equation of the form $f(x_1, \dots, x_n) = 0$ is box consistent with respect to a variable x_i and a box $B = I_1 \times \dots \times I_n$ if and only if:*

$$\begin{cases} 0 \in f(I_1, \dots, I_{i-1}, [\underline{I}_i, \underline{I}_i^+], I_{i+1}, \dots, I_n) \\ \text{and} \\ 0 \in f(I_1, \dots, I_{i-1}, [\bar{I}_i^-, \bar{I}_i], I_{i+1}, \dots, I_n), \end{cases} \quad (2)$$

where $I = [\underline{I}, \bar{I}]$ is an interval with floating-point bounds, a^+ (resp., a^-) is the smallest floating-point number greater than (resp., the largest floating-point number smaller than) the floating-point number a , and f is the natural interval extension of f .

Given a real function $f: \mathbb{R}^n \rightarrow \mathbb{R}$, and a box of domains $B = I_1 \times \dots \times I_n \in \mathbb{I}^n$, we define $g_i^B: \mathbb{I} \rightarrow \mathbb{I}$ as the i th unary interval projection with respect to B of its interval extension f :

$$g_i^B(x) = f(I_1, \dots, I_{i-1}, x, I_{i+1}, \dots, I_n), \quad i \in \{1, \dots, n\}.$$

In the following, we will mostly manipulate g_i^B instead of f . The original real function f , the box B of domains considered and/or the variable x_i on which the projection is performed will often be left implicit and omitted from the notation of g .

In order not to lose any potential solution, an algorithm that enforces box consistency of an equation with respect to a variable x_i and a box of domains must return the largest domain $I'_i \subseteq I_i$ that verifies Eq. (2).

Algorithm `bc3revise` [2], presented by Algorithms 1 and 2, considers the unary projection of an n -ary equation on a variable x_i (where all variables but x_i have been replaced by their current domain) and a domain I_i . It enforces box consistency by searching the leftmost and rightmost *canonical*

Algorithm 1 Computing a box consistent interval with respect to g the usual way

```
[bc3revise] in:  $g: \mathbb{I} \rightarrow \mathbb{I}$ ; in:  $I \in \mathbb{I}$ 
# Returns the largest interval included in  $I$  that is
# box consistent with respect to  $g$ 
begin
  1  $I_l \leftarrow \text{left\_narrow}(g, I)$ 
  2 if  $I_l \neq \emptyset$ :
  3    $I_r \leftarrow \text{right\_narrow}(g, [I_l, \bar{I}])$ 
  4   # Returns the smallest interval w.r.t.
  5   # set inclusion that contains  $I_l \cup I_r$ 
  6   return  $\square(I_l \cup I_r)$ 
  7 else:
  8   return  $\emptyset$ 
```

end

Algorithm 2 Computing a box consistent left bound with Newton steps and a binary search

```
[left_narrow] in:  $g: \mathbb{I} \rightarrow \mathbb{I}$ ; in:  $I \in \mathbb{I}$ 
# Returns an interval included in  $I$ 
# with the smallest left bound  $l$  such that  $0 \in g([l, l^+])$ 
begin
  1 if  $0 \notin g(I)$ : # No solution in  $I$ 
  2   return  $\emptyset$ 
  3 else:
  4   if  $\underline{I}^+ \geq \bar{I}$ : # canonical( $I$ ):
  5     return  $I$ 
  6   else:
  7      $I \leftarrow \text{Newton}(g, g', I)$  # Interval Newton
  8     if  $0 \in g([\underline{I}, \underline{I}^+])$ : # Box consistency on left?
  9       return  $I$ 
 10    else:
 11      #  $I_1 \leftarrow [\underline{I}, m(I)]; I_2 \leftarrow [m(I), \bar{I}]$ 
 12       $(I_1, I_2) \leftarrow \text{split}(I)$ 
 13       $I \leftarrow \text{left\_narrow}(g, I_1)$ 
 14      if  $I = \emptyset$ :
 15        return  $\text{left\_narrow}(g, I_2)$ 
 16      else:
 17        return  $I$ 
```

end

domains* for which g evaluates to an interval containing 0. The search is performed by a dichotomic search aided with Newton steps to accelerate the process. Algorithm 2 describes the search of a quasi-zero to update the left side of I_i . The procedure `right_narrow` to update the right bound works along the same lines and is, therefore, omitted.

Algorithm `bc3revise` first tries to move the left bound of I_i to the right, and then proceeds to move its right bound to the left. The Newton procedure computes a fixpoint of the Interval Newton algorithm [11], where a Newton step at iteration $j + 1$ is:

$$I^{(j+1)} \leftarrow I^{(j)} \cap \left(m(I^{(j)}) - \frac{g(m(I^{(j)}))}{g'(I^{(j)})} \right)$$

with $m(I)$ the midpoint of the interval I^\dagger . As in Ratz’s work [13], the Newton step uses an extended version of the interval division to return a union of two semi-open-ended intervals whenever $g'(I^{(j)})$ contains 0. The subtraction and the intersection operators are modified accordingly. The intersection operator is applied to an interval ($I^{(j)}$) and a union of two intervals (result of the subtraction), and returns an interval. Figure 2 presents graphically the steps performed to enforce box consistency. The encircled numbers label the steps.

Algorithms `bc3revise`, `left_narrow` and `right_narrow` do not offer opportunities to exploit full data parallelism as they do not require close evaluations of the same function over different domains. The same holds for the Newton procedure: in the general case, g and g' are different functions, and therefore cannot be evaluated in parallel with SIMD instructions.

4 Box Consistency by Shaving

To obtain a higher level of data parallelism, we propose a new algorithm to enforce box consistency on the projection g_i^B of an equation $f = 0$ on a variable x_i : starting from the original domain I_i for x_i , consider separately its left half and its right half; for the left part, linearize g at I_i as for a Newton step, solve the resulting linear equation and intersect the resulting domain with the left half of I_i ; do the same for the right half by linearizing g at \bar{I}_i . A new smaller domain that preserves solutions is then obtained; Iterate until reaching a fixpoint.

4.1 A Sequential Algorithm: `sbc`

Figure 3 illustrates graphically the process just described, and Algorithm 3 presents the actual algorithm. The

*A non-empty interval $[a, b]$ is canonical if $a^+ \geq b$.

†Note that we are free to choose any point in I , not the midpoint only. We take advantage of this in the algorithms presented in the next section.

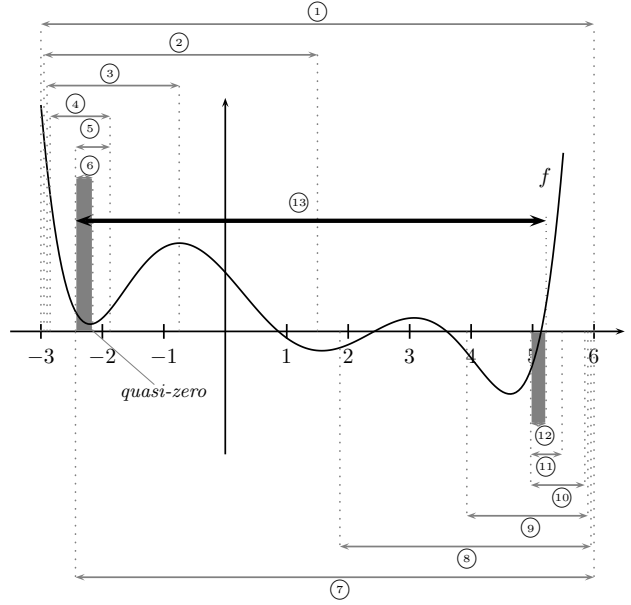


Figure 2. Enforcing box consistency with `bc3revise`. (The algorithm returns $I_{13} = \square(I_6 \cup I_{12})$)

following proposition states its properties: *termination*, *correctness*, and *completeness*.

Proposition 1 Given an n -ary equation $c: f(x_1, \dots, x_n) = 0$, a box $I_1 \times \dots \times I_n$ of domains, and a projection $c: g_i(x) = 0$ of c , we have:

Termination. The call to `sbc`(g_i, I_i) always terminates;

Correctness. The equation c is box consistent with respect to x_i and `sbc`(g_i, I_i);

Completeness. No solution is lost during the tightening process:

$$\forall (r_1, \dots, r_n) \in I_1 \times \dots \times I_n: f(r_1, \dots, r_n) = 0 \implies r_i \in \text{sbc}(g_i, I_i).$$

Proof. In the following, the interval I corresponds to the domain I_i of x_i .

(Termination). Consider the loop on Lines 2–26: Irrespective of the outcome of the Newton step on Line 11, I_l is either narrowed down to the empty set (Line 8), or its left bound is displaced to the right (Line 6), or it is declared consistent (Line 13). The same holds for I_r . As the interval I is set at the end of each iterate as the “hull” of the union of I_l and I_r , the previous argument means that either I is declared left and right consistent—in that case, we

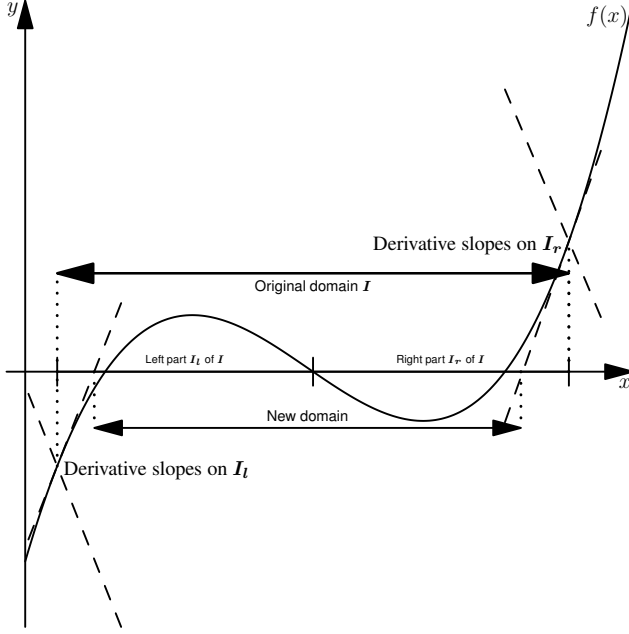


Figure 3. Domain reduction obtained with one iteration of the loop in sbc

leave the loop on Line 26—or it is shaved on at least one of its bounds. Since we operate on intervals whose bounds are floating-point numbers, of which there are finitely many, we cannot narrow an interval indefinitely. Consequently, we will eventually reach the point to which I is narrowed down to the empty set, or to some interval for which `left_consistent` and `right_consistent` are true, at which point we will leave the loop.

(Correctness). We leave the loop 2–26 if I is empty or if the canonical intervals at left and right bounds contain solutions of g_i (Lines 5 and 15). In the latter case, we have the two conditions of Eq (2) for box consistency; the former case occurs if both I_l and I_r do not contain a solution of c (Lines 7 and 17) or if the Newton steps on Lines 11 and 21 narrow them down to \emptyset . By correctness of the interval Newton method, this case only occurs if, once again, I_l and I_r do not contain a solution of c .

(Completeness). By completeness of the Newton operator, as tightening only occurs either through Newton steps, or by discarding intervals that have been proved on Lines 5, 7, 15, or 17 not to contain solutions. \square

For each iteration of the loop 2–26 in Algorithm `sbc`, we have to compute g for intervals $[I_l, I_l^+]$, $[\overline{I_r}^-, \overline{I_r}]$, I_l , I_r , $[I_l, I_l]$, and $[\overline{I_r}, \overline{I_r}]$. We also have to compute g' for intervals I_l and I_r . All these evaluations are candidates for parallelization with SIMD instructions, as presented in the

Algorithm 3 Enforcing box consistency by shaving

[sbc] in: $g: \mathbb{I} \rightarrow \mathbb{I}$; in: $I \in \mathbb{I}$

Returns the largest interval included in I that is
box consistent w.r.t. g

begin

1 (`left_consistent`, `right_consistent`) \leftarrow (`false`, `false`)

2 do:

3 (I_l , I_r) \leftarrow `split`(I)

4 if \neg `left_consistent`: # Updating the left bound

5 if $0 \notin g([I_l, I_l^+])$: # I not box consistent on left?

6 $I_l \leftarrow [I_l^+, \overline{I_l}]$ # Consider the remainder of I_l

7 if $0 \notin g(I_l)$: # No solution in remainder of I_l ?

8 $I_l \leftarrow \emptyset$

9 else:

10 # One Newton step

11 $I_l \leftarrow I_l \cap (I_l - g([I_l, I_l])/g'(I_l))$

12 else:

13 `left_consistent` \leftarrow `true`

14 if \neg `right_consistent`: # Updating the right bound

15 if $0 \notin g([\overline{I_r}^-, \overline{I_r}])$: # Not right box consistent?

16 $I_r \leftarrow [\underline{I_r}, \overline{I_r}^-]$ # Consider remainder of I_r

17 if $0 \notin g(I_r)$: # No solution in remainder of I_r ?

18 $I_r \leftarrow \emptyset$

19 else:

20 # One Newton step

21 $I_r \leftarrow I_r \cap (\overline{I_r} - g([\overline{I_r}, \overline{I_r}])/g'(I_r))$

22 else:

23 `right_consistent` \leftarrow `true`

24 # Returns the “hull” $[\min(\underline{I_l}, \underline{I_r}), \max(\overline{I_l}, \overline{I_r})]$

25 $I \leftarrow \square(I_l \cup I_r)$

26 while $((I \neq \emptyset) \wedge$
$(\neg$ `left_consistent` \vee \neg `right_consistent`))

27 return I

end

next section.

4.2 An SIMD Algorithm for Box Consistency: vsbc

Algorithm 4 presents a modification of Algorithm `sbc` to make good use of its higher level of data parallelism thanks to the SIMD interval arithmetic that has been presented in Section 2. The evaluations of g and g' are reordered to appear in pairs that can be evaluated in parallel. In addition,

Algorithm 4 A data parallel algorithm for box consistency enforcement

[vsbc] in: $g: \mathbb{I} \rightarrow \mathbb{I}$; in: $I \in \mathbb{I}$

Returns the largest interval included in I that is

box consistent w.r.t. g

begin

1 $(left_consistent, right_consistent) \leftarrow (false, false)$

2 do:

3 $(I_l, I_r) \leftarrow split(I)$

4 $(J_l, J_r) \leftarrow \llbracket g([I_l, I_l^+]), g([\overline{I_r}^-, \overline{I_r}]) \rrbracket$

5 if $0 \notin J_l$: # I not box consistent to the left?

6 $I_l \leftarrow [I_l^+, \overline{I_l}]$ # Considering the remainder of I_l

7 else:

8 $left_consistent \leftarrow true$

9 if $0 \notin J_r$: # I not box consistent to the right?

10 $I_r \leftarrow [\underline{I_r}, \overline{I_r}^-]$ # Considering the remainder of I_r

11 else:

12 $right_consistent \leftarrow true$

13 if $\neg left_consistent \vee \neg right_consistent$:

14 $(K_l, K_r) \leftarrow \llbracket g(I_l), g(I_r) \rrbracket$

15 # First checking an obvious absence of solution in I_l

16 if $0 \notin K_l$:

17 $I_l \leftarrow \emptyset$

18 # First checking an obvious absence of solution in I_r

19 if $0 \notin K_r$:

20 $I_r \leftarrow \emptyset$

21 # Performing 2 Newton steps in parallel

22 # to update both bounds

23 # For better performances, we reuse J_l and J_r

24 # instead of $g([I_l, I_l])$ and $g([\overline{I_r}, \overline{I_r}])$

25 $(I_l, I_r) \leftarrow \llbracket I_l \cap ([I_l, I_l^+] - J_l/g'(I_l)), \right.$
 $\left. I_r \cap ([\overline{I_r}^-, \overline{I_r}] - J_r/g'(I_r)) \rrbracket$

26 # Returns the "hull" $[\min(I_l, I_r), \max(\overline{I_l}, \overline{I_r})]$

27 $I \leftarrow \square(I_l \cup I_r)$

28 while $((I \neq \emptyset) \wedge (\neg left_consistent \vee \neg right_consistent))$

29 return I

end

we reuse the evaluation of $g([I_l, I_l^+])$ and $g([\overline{I_r}^-, \overline{I_r}])$ of Line 4 for the Newton steps of Line 25 instead of $g([I_l, I_l])$ and $g([\overline{I_r}, \overline{I_r}])$ as was done in Algorithm sbc. This choice avoids two evaluations of g at the cost of potentially slightly decreasing the tightening ability of the Newton step. The domain computed is unaffected by this optimization. In particular, box consistency is still obtained eventually.

At each iteration of the loop between Lines 2 and 28, we perform 4 *interval* evaluations of g and 2 *interval* evaluations of g' for roughly the same cost as 2 *floating-point* evaluations of f and 1 *floating-point* evaluation of f' .

5 Experiments

To evaluate the impact of our new algorithms, we have selected 20 instances of 12 classical test problems. Some are polynomial and others are not. The characteristics of these test problems are summarized in Table 1. All problems are structurally well constrained (as many equations as variables). We indicate after each name whether all equations are polynomial (quadratic, if no polynomial has a degree greater than 2) or not. A problem is labelled "non-polynomial" if at least one constraint contains a trigonometric, hyperbolic or otherwise transcendental operator. Column "Size" reports the number of equations/variables. All test problems are presented on the COPRIN web page [8].

Table 1. Test problems characteristics

Name	Code	Size
Bronstein (<i>quadratic</i>)	<i>bro</i>	3
Broyden-banded (<i>quadratic</i>)	<i>bb</i>	100, 500, 1 000
Broyden tridiagonal (<i>quadratic</i>)	<i>bt</i>	10 and 20
Combustion (<i>polynomial</i>)	<i>comb</i>	10
Discrete Boundary Value Function (<i>polynomial</i>)	<i>dbvf</i>	10 and 30
Extended Freudenstein (<i>polynomial</i>)	<i>ef</i>	30 and 50
Mixed Algebraic Trigonometric (<i>non-polynomial</i>)	<i>mat</i>	3
Moré-Cosnard (<i>polynomial</i>)	<i>mc</i>	50 and 100
Robot (<i>quadratic</i>)	<i>rob</i>	8
Trigexp 3 (<i>non-polynomial</i>)	<i>te3</i>	5 000
Troesch (<i>non-polynomial</i>)	<i>tro</i>	50, 100, and 200
Yamamura (<i>polynomial</i>)	<i>yam</i>	6 and 8

All experiments were conducted on an Intel Core2 Duo T5600 1.83GHz. The Whetstone test [3] for this machine reports 1111 MIPS with a loop count equal to 100,000. All algorithms have been implemented in an in-house C++ solver, with *gaol* [4] as the underlying interval arithmetic library. The SIMD interval arithmetic presented above has been implemented from scratch using Intel intrinsic instructions. In its current state, the library only contains vector-

ized versions of the addition, subtraction, multiplication, division, and integral power. All other SIMD functions are emulated with sequential interval arithmetic. As a consequence, only polynomial equation systems are entirely solved in an SIMD environment at present.

Table 2. Experiments

Problem	bc3revise	bc3vd	bc3vf	sbc	sbcvd	vsbc	vsbcvd&sbcvd
<i>bro</i>	2.6	0.9	0.4	0.3	0.3	0.06	0.2
<i>bb 100</i>	10.4	3.3	3.2	2.7	1.2	0.7	0.8
<i>bb 500</i>	123.7	39.1	27.0	24.8	10.8	5.3	5.5
<i>bb 1000</i>	280.2	88.7	56.8	55.1	24.0	11.1	11.4
<i>bt 10</i>	16.9	5.8	3.4	2.1	1.0	0.4	0.5
<i>bt 20</i>	1127.4	382.2	260.3	141.1	66.7	28.6	30.4
<i>comb</i>	1.4	0.6	0.5	0.2	0.08	0.03	0.08
<i>dbvf 10</i>	1.7	0.5	0.6	0.3	0.1	0.08	0.1
<i>dbvf 30</i>	42.7	13.4	20.4	7.7	3.2	4.7	3.8 ∇
<i>ef 30</i>	2.1	0.8	TO	1.1	0.5	TO	0.3
<i>ef 50</i>	5.1	1.8	TO	2.2	0.9	TO	0.6
<i>mat</i>	26.9	13.6	13.2	1.5	1.1	0.4	1.0
<i>mc 50</i>	23.7	6.5	5.7	11.0	4.4	3.0	4.1
<i>mc 100</i>	175.5	49.0	46.2	86.9	36.8	28.0	35.6
<i>rob</i>	4.5	1.5	4.3	0.8	0.4	1.0	0.3
<i>te3 5000</i>	7.5	5.1	17.1	2.9	2.7	3.3	3.8 ∇
<i>tro 50</i>	24.8	15.3	29.1	4.4	3.6	2.6	3.4
<i>tro 100</i>	180.1	112.7	384.2	30.9	25.4	33.7	24.2
<i>tro 200</i>	1341.4	844.4	TO	231.1	188.1	TO	181
<i>yam 6</i>	14.6	4.3	4.2	7.3	2.5	1.2	1.7
<i>yam 8</i>	279.1	84.6	104.0	91.0	35.1	26.1	27.7

Times in seconds on an Intel Core2 Duo T5600 1.83GHz (whetstone 100 000=1111 MIPS)

Best times in bold blue. Time out TO set to 1 800 s.

Numbers followed by the symbol “ ∇ ” correspond to cases for which vsbcvd&sbc performs worse than sbc alone.

Table 2 reports the time spent in seconds to isolate all solutions of the test problems in domains with a width smaller than 10^{-6} , starting from the standard domains given on the COPRIN web page. An entry “TO” indicates a time-out (more than 30 minutes, here). Column *bc3revise* presents the results obtained with Algorithm *bc3revise* implemented with double precision interval arithmetic on the FPU; Column *bc3vd* corresponds to Algorithm *bc3revise* where interval arithmetic is performed in double precision with SSE2 instructions (basic vectorization); Column *bc3vf* corresponds to Algorithm *bc3revise* where interval arithmetic is performed in single precision with SSE2 instructions (we still perform only one interval operation per SSE2 instruction, using only the lower half of SSE2 registers, though); Column *sbc* corresponds to Algorithm *sbc* implemented with double precision interval arithmetic on the FPU; Column *sbcvd* corresponds to Algorithm *sbc* where interval arithmetic is performed in double precision with SSE2 instructions; Column *vsbc* corresponds to Algorithm

vsbc where interval arithmetic is performed in single precision with SSE2 instructions (two interval operations are performed in parallel); lastly, Column *vsbc&sbcvd* corresponds to the cooperation of *vsbc* and *sbcvd*: *vsbc* is used until the domains are all smaller than a size fixed empirically to 0.25; *sbcvd* is used afterwards.

6 Discussion

As can be seen from Column *sbc* of Table 2, enforcing box consistency by shaving is faster than with Algorithm *bc3revise* on all problems of our test set, the ratio *bc3revise/sbc* ranging from 1.9 to 17.9 and beyond. We also believe that *sbc* is simpler to understand and easier to implement correctly than *bc3revise*.

Basic vectorization of interval arithmetic improves speed by up to three times (see *bc3revise* vs. *bc3vd* and *sbc* vs. *sbcvd*) at no cost since algorithms do not have to be modified in order to benefit from it.

If we take advantage of the data parallelism inherent to Algorithm *sbc* to vectorize interval evaluations, leading to Algorithm *vsbc*, we obtain even better results on all problems but *dbvf 30*, *ef 30*, *ef 50*, *rob*, *te3 5000*, *tro 100*, and *tro 200*. All other things being equal, if we emulate SIMD instructions with double precision floating-point operations, we obtain back the times of *sbc*, which means that the very size of single precision floating-point numbers is the culprit here: as we vectorize 2 interval instructions with SSE2 registers, we must switch from double precision floats used in the rest of the program to single precision floats (see Figure 1(b)). The cast leads to less precision in the computation, which in turn has an impact on the ability to reject domains having no solutions. The same problem occurs for *bc3vf*.

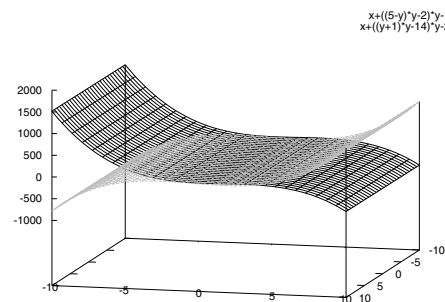


Figure 4. Extended Freudenstein 2

As a consequence, the exploration algorithm has more branching to perform to isolate solutions. This incurs

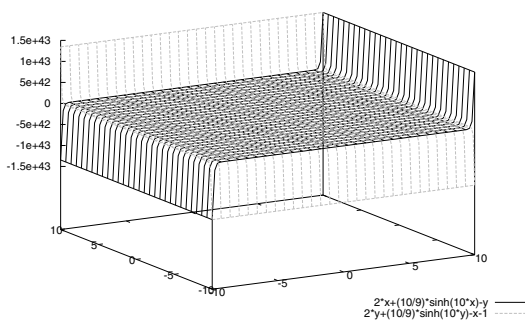


Figure 5. Troesch 2

an increase in the running time that may be drastic for ill-conditioned problems such as *Troesch* or *Extended Freudenstein*: as we may see in Figures 4 and 5 for the case of 2 equations and 2 variables, the curves for these two problems are almost tangent to each other and to the xy -plane on a large surface. Each equation considered separately leads to the computation of many quasi-zeros that cannot be removed easily by the other equation of the problem.

There is currently no easy cure to this problem as micro-processor makers do not seem to be ready to offer SIMD instructions on 4 double precision floats any time soon. It is still possible to quickly isolate regions of interest in “large” domains using *vsbc*, and then switch to *sbcvd* to polish the results and obtain tighter domains if necessary. Column *vsbc&sbcvd* in Table 2 shows that this procedure indeed removes the time-out problems of *vsbc* on ill-conditioned problems, while still preserving better performances compared to *sbcvd* alone. The best cooperation scheme that maximizes performances as much as possible still remains to be found, though.

References

- [1] F. Benhamou. Interval constraints, interval propagation. In P. M. Pardalos and C. A. A. Floudas, editors, *Encyclopedia of Optimization*, volume 3, pages 45–48. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.
- [2] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Procs. Intl. Symp. on Logic Prog.*, pages 124–138. The MIT Press, 1994.
- [3] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *Comput. J.*, 19(1):43–49, 1976.
- [4] F. Goualard. *GAOL 3.1.1: Not Just Another Interval Arithmetic Library*. Laboratoire d’Informatique de Nantes-Atlantique, 4.0 edition, Oct. 2006. Available at <http://sourceforge.net/projects/gaol>.

- [5] F. Goualard. Fast and correct SIMD algorithms for interval arithmetic. In *Proceedings of PARA '08*, May 2008.
- [6] L. Granvilliers and F. Benhamou. Progress in the solving of a circuit design problem. *Journal of Global Optimization*, 20:155–168, 2001.
- [7] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
- [8] INRIA project COPRIN: Contraintes, OPTimisation, Résolution par Intervalles. The COPRIN examples page. Web page at <http://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html>.
- [9] R. B. Kearfott, M. T. Nakao, A. Neumaier, S. M. Rump, S. P. Shary, and P. van Hentenryck. Standardized notation in interval analysis. In *Proc. XIII Baikal International School-seminar “Optimization methods and their applications”*, volume 4 “Interval analysis”, pages 106–113. Irkutsk: Institute of Energy Systems SB RAS, July 2005.
- [10] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 1(8):99–118, 1977.
- [11] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.
- [12] A. Neumaier. *Interval methods for systems of equations*, volume 37 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1990.
- [13] D. Ratz. Inclusion isotone extended interval arithmetic. Technical Report 5/1996, Institut für Angewandte Mathematik, Universität Karlsruhe, 1996.