



Role of Hull-Consistency in the HIBA_USNE Multithreaded Solver for Nonlinear Systems

Bartłomiej Jacek Kubica^(✉)

Department of Applied Informatics, Warsaw University of Life Sciences,
ul. Nowoursynowska 159, 02-776 Warsaw, Poland
bartlomiej_kubica@sggw.pl

Abstract. This paper considers incorporating a hull-consistency enforcing procedure in an interval branch-and-prune method. Hull-consistency has been used with interval algorithms in several solvers, but its implementation in a multithreaded environment is non-trivial. We describe arising issues and discuss the ways to deal with them. Numerical results for some benchmark problems are presented and analyzed.

Keywords: Nonlinear equations systems · Interval computations
Hull consistency · Multithreading · Solver

1 Introduction

In a series of papers, including [15, 16, 19, 20] the author considered an interval solver for nonlinear systems – targeted mostly at underdetermined equations systems – and its shared-memory parallelization (see also references in [19] for the author’s other papers). The solver described in these papers is called HIBA_USNE (Heuristical Interval Branch-and-prune Algorithm for Underdetermined and well-determined Systems of Nonlinear Equations) and is currently available from the author’s ResearchGate profile under the GPL license [6].

In none of these papers (and in none of previous versions of HIBA_USNE), hull-consistency has been used.

2 Generic Algorithm

HIBA_USNE uses interval methods. They are based on interval arithmetic operations and basic functions operating on intervals instead of real numbers (so that result of an operation on numbers always belongs to the result of operation on intervals that contain the numerical inputs). We shall not define interval operations here; the interested reader is referred to several papers and textbooks, e.g., [12, 13].

The solver is based on the branch-and-prune (B&P) schema that can be expressed by pseudocode presented in Algorithm 1.

Algorithm 1. Interval branch-and-prune algorithm

Require: L, f, ε

```

1:  $\{L$  – the list of initial boxes, often containing a single box  $\mathbf{x}^{(0)}\}$ 
2:  $\{L_{ver}$  – verified solution boxes,  $L_{pos}$  – possible solution boxes $\}$ 
3:  $L_{ver} = L_{pos} = \emptyset$ 
4:  $\mathbf{x} = \text{pop}(L)$ 
5: loop
6:   process the box  $\mathbf{x}$ , using the rejection/reduction tests
7:   if ( $\mathbf{x}$  does not contain solutions) then
8:     discard  $\mathbf{x}$ 
9:   else if ( $\mathbf{x}$  is verified to contain a segment of the solution manifold) then
10:     push ( $L_{ver}, \mathbf{x}$ )
11:   else if (the tests resulted in two subboxes of  $\mathbf{x}$ :  $\mathbf{x}^{(1)}$  and  $\mathbf{x}^{(2)}$ ) then
12:      $\mathbf{x} = \mathbf{x}^{(1)}$ 
13:     push ( $L, \mathbf{x}^{(2)}$ )
14:     cycle loop
15:   else if ( $\text{wid } \mathbf{x} < \varepsilon$ ) then
16:     push ( $L_{pos}, \mathbf{x}$ ) {The box  $\mathbf{x}$  is too small for bisection}
17:   if ( $\mathbf{x}$  was discarded or  $\mathbf{x}$  was stored) then
18:     if ( $L == \emptyset$ ) then
19:       return  $L_{ver}, L_{pos}$  {All boxes have been considered}
20:      $\mathbf{x} = \text{pop}(L)$ 
21:   else
22:     bisect ( $\mathbf{x}$ ), obtaining  $\mathbf{x}^{(1)}$  and  $\mathbf{x}^{(2)}$ 
23:      $\mathbf{x} = \mathbf{x}^{(1)}$ 
24:     push ( $L, \mathbf{x}^{(2)}$ )

```

The “rejection/reduction tests”, mentioned in the algorithm are described in previous papers (specifically [19]), i.e.:

- switching between the componentwise Newton operator (for larger boxes) and Gauss-Seidel with inverse-midpoint preconditioner, for smaller ones,
- a heuristic to choose whether to use or not the BC3 algorithm [19],
- a heuristic to choose when to use bound-consistency [20],
- sophisticated heuristics to choose the bisected component [16, 19],
- an additional second-order approximation procedure [18],
- an initial exclusion phase of the algorithm (deleting some regions, not containing solutions) – based on Sobol sequences [17, 19].

Other possible variants (see, e.g., [15]) are not going to be considered.

3 Hull-Consistency

Hull-consistency (also known under the name of 2B-consistency) has been used in several interval programs over the years; see, e.g., [7, 8]. It can be defined as follows.

Definition 1. A box $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n)^T$ is hull-consistent with respect to a constraint $c(x_1, \dots, x_n)$, iff:

$$\forall i \mathbf{x}_i = \square\{s \in \mathbf{x}_i \mid \exists x_1 \in \mathbf{x}_1, \dots \exists x_{i-1} \in \mathbf{x}_{i-1}, \exists x_{i+1} \in \mathbf{x}_{i+1} \dots \exists x_n \in \mathbf{x}_n \\ c(x_1, \dots, x_{i-1}, s, x_{i+1}, \dots, x_n)\} .$$

Following [14], the symbol “ \square ” denotes the interval hull.

Other words, \mathbf{x} is hull-consistent iff for each i we can find two points x^a and x^b , satisfying the property c , for which $x_i^a = \underline{x}_i$ and $x_i^b = \bar{x}_i$.

Now, let us describe, how to check if a box is hull-consistent and how to enforce hull-consistency on a box.

3.1 Algorithms for Enforcing Hull-Consistency

For simple constraints, checking and/or enforcing hull-consistency is relatively simple.

As a simple example, let us consider an equation $x_1 + x_2 - 3 = 0$. By obvious symbolic transformations, we obtain formulae for both variables that can be used to obtain their consistent domains:

$$\begin{aligned} \mathbf{x}_1 &= 3 - \mathbf{x}_2 \text{ and} \\ \mathbf{x}_2 &= 3 - \mathbf{x}_1. \end{aligned}$$

Using the above consistency operators, we can simply check consistency for any box or compute its sub-box containing all consistent values. For instance, a box $[-4, 2] \times [-2, 4]$ is not hull-consistent, but it can be reduced to the hull consistent one, by applying:

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{x}_1 \cap (3 - \mathbf{x}_2) = [-4, 2] \cap [-1, 5] = [-1, 2], \\ \mathbf{x}_2 &= \mathbf{x}_2 \cap (3 - \mathbf{x}_1) = [-2, 4] \cap [1, 7] = [1, 4]. \end{aligned}$$

This box is hull-consistent indeed, as points $(-1, 4)$ and $(1, 2)$ are solutions of the initial constraint $x_1 + x_2 - 3 = 0$.

However, for a more sophisticated constraint, obtaining a consistent box is not as straightforward. Let us consider the constraint:

$$x_1^3 + x_1^2 - \exp(x_2) = 0. \tag{1}$$

Again, by relatively simple symbolic transformations we can extract x_2 from Eq. (1), but not x_1 . The solution is to decompose such an equation into primitive ones, by adding additional variables and apply hull-consistency to such a decomposed system. For the constraint (1), we could obtain:

$$\begin{aligned} t_1 - x_1^3 &= 0, \\ t_2 - x_1^2 &= 0, \\ t_3 - t_1 - t_2 &= 0, \\ t_4 - \exp(x_2) &= 0, \\ t_3 - t_4 &= 0. \end{aligned}$$

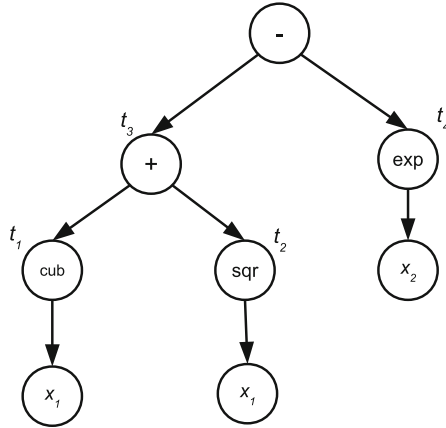


Fig. 1. Expression tree of constraint (1)

The algorithm HC4 [7] (cf. also [11]) performs such a decomposition, creating a tree of the initial constraint, where a variable corresponds to each node: By traversing the tree *forward* and *backward*, we enforce hull-consistency on subsequent variables (Fig. 1).

3.2 ADHC Implementation

The ADHC library [5] (Algorithmic Differentiation and Hull Consistency enforcing), developed by the author, contains procedures for constructing the expression tree and for the HC4 algorithm.

Thanks to the virtues of C++ template metaprogramming, the same source code can be used to generate binary procedures computing function values, gradients and Hesse matrices, and to generate the procedure creating the expression tree, in the form of a dynamic data structure.

4 Hull-Consistency Vs Multithreading

Since the very beginning (cf. [15]) the HIBA_USNE solver has been implemented as parallel. The early version has been parallelized using OpenMP, but then the author switched to Intel TBB (Threading Building Blocks [3]). Parallelization of the HIBA_USNE solver, i.e., of Algorithm 1, is done on several levels. Firstly, operations on different boxes form different tasks that can be executed by different threads.

Also, some of the procedures applied on a single box are parallel. Such a concurrent implementations has been particularly useful for the procedure enforcing bound-consistency [20], but enforcing box-consistency (see, e.g., [8]) can be parallelized, also – and such version is applied at least for the initial box.

Parallel implementation of the HC4 algorithm is also possible, but it does not seem worthwhile. The cost of enforcing hull-consistency is far smaller than box-consistency (which, in particular, requires computing derivatives – at least for BC3 and BC4 algorithms; cf. [7]).

Hence, the HC4 implementation we use in the current version of the solver (Beta 2.5; cf. Sect. 5). Still, it is not easy to implement the HC4 algorithm in a MT-safe (multithreaded-safe) manner. The procedure requires the expression tree representation. There are, in general, three possibilities:

- there is a shared expression tree and access to it is synchronized,
- there is a shared expression tree, but domains of variables associated to each node are thread-specific,
- each thread has its own copy of the expression tree, to compute the domains of variables for various boxes.

The first approach seems absolutely unacceptable for a solver that is supposed to be scalable with the number of threads. The second one seems interesting, but is somewhat cumbersome to implement. Also, it might result in suboptimal cache usage as domains of each variable will have to be placed outside the node of the expression tree. The third approach is currently implemented in HIBA_USNE. It uses some memory, as each of the threads has a separate copy of the data structure (and this might become an issue for higher number of threads, e.g., on the MIC architecture, where 240 threads can work in parallel), but, in our experiments, it seems to be acceptable.

5 Computational Experiments

Numerical experiments have been performed on a machine with two Intel Xeon E5-2695 v2 processors (2.4 GHz). Each of them has 12 cores and on each core two hyper-threads (HT) can run. So, $2 \times 12 \times 2 = 48$ HT can be executed in parallel. The machine runs under control of a 64-bit GNU/Linux operating system, with the kernel 3.10.0-123.e17.x86_64 and glibc 2.17. They have non-uniform turbo frequencies from range 2.9–3.2 GHz.

As there have been other users performing their computations also, we limited ourselves to using 24 threads only.

The Intel C++ compiler ICC 15.0.2 has been used.

The solver has been written in C++, using the C++11 standard. The C-XSC library (version 2.5.4) [2] was used for interval computations. The parallelization was done with the packaged version of TBB 4.3 [3].

The following test problems have been considered: two underdetermined ones: 5R planar and Puma7, and six well-determined: Brent10, BT50 (Broyden-tridiagonal), BB30 (Broyden-banded), BB24-mod, Transistor, EF200 (Extended-Freudenstein). Their formulation (and used accuracies) has been described in [19, 20] and references therein. Function BB24-mod is the Broyden-banded function BB24 minus 1; such a minor modification results in a much harder problem. It is worth noting that it was the function BroyN-mod that was

used in previous papers ([15, 19, 20], etc.) under the name of the Broyden-banded function.

Here we give used accuracies:

- 5R planar: $\varepsilon = 0.02$,
- Puma7: $\varepsilon = 0.05$,
- Brent10: $\varepsilon = 10^{-7}$,
- BT50: $\varepsilon = 10^{-6}$,
- BB30, BB24-mod: $\varepsilon = 10^{-6}$,
- Transistor: $\varepsilon = 10^{-8}$,
- EF200: $\varepsilon = 10^{-6}$.

The following algorithm versions have been considered:

- “Beta 2.0” – HIBA_USNE Beta 2.0, using box and bound-consistency, but no hull-consistency,
- “HC only” – hull-consistency used instead of box-consistency and 3B consistency, instead of bound-consistency,
- “Beta 2.5” – HIBA_USNE Beta 2.5, combining box and hull-consistency, in a manner similar to BC4 [7]: algorithm HC4 is used always and BC3 is applied after it, but only if there is more than one occurrence of the variable in the formula for the constraint.

Also, please note, execution times of parallel programs are to some extent random. We try to present median results, but please note all of them may vary in a few-seconds interval.

The following notation is used in the tables:

- fun.evals, grad.evals, Hesse evals – numbers of functions evaluations, functions’ gradients and Hesse matrices evaluations (in the interval automatic differentiation arithmetic),
- bisecs – the number of boxes bisections,
- preconds – the number of preconditioning matrix computations (i.e., performed Gauss-Seidel steps),
- Sobol excl. – the number of boxes to be excluded generated by the initial exclusion phase,
- Sobol resul. – the number of boxes resulting from the exclusion phase (cf. [17, 19]),
- bc3 – the number of calls of bc3revise; see [19],
- hc – the number of calls of hc.enforce,
- 3B/bnd.cons. – the number of calls to the procedure enforcing a higher-order consistency, i.e., – depending on the algorithm variant – bound-consistency, 3B consistency or a mixed one (when BC4 is used),
- pos.boxes, verif.boxes – number of elements in the computed lists of boxes containing possible and verified solutions,
- Leb.pos., Leb.verif. – total Lebesgue measures of both sets,
- time – computation time in seconds.

Table 1. Computational results for the “Beta 2.0” solver version

Problem	5R planar	Puma7	Brent10	BT50	BB30	BB24-mod	Transistor	EF200
fun. evals	215,370,202	24,716,399	9,288,556	546,517,232	48,025,431	2,560,784,337	177,883,219	13,531,560
grad.evals	53,540,850	25,687,861	6,479,385	139,774,706	3,929,878	288,405,687	19,786,538	1,300,186
Hesse evals	307,648	357	264,252	30	30	91,564	127,749	200
bisections	5,445,519	1,718,076	50,299	617057	21,644	3,903,618	33,246	1,300
preconds	10,056,243	3,206,635	78,424	221427	26,019	6,225,697	67,554	2
bc3.rev	86,889,730	1,039,374	2,218,450	241,468,708	23,193,747	1,190,987,170	84,359,918	1,454,871
hc	—	—	—	—	—	—	—	—
3B/bnd.cons	110,745	222	95	0	48	8,190	10,408	0
pos.boxes	1,878,238	681,004	401	2	0	0	0	0
verif.boxes	3,458	202,694	820	1	1	1	1	1
Leb.poss	0.000333	3e-47	3e-83	1e-323	0.0	0.0	0.0	0.0
Leb.verif	1e-6	3e-11	1e-82	5e-324	3e-12	3e-233	2e-102	5r-324
time	54	14	11	209	11	662	32	39

Table 2. Computational results for the “HC only” algorithm version

Problem	5R planar	Puma7	Brent10	BT50	BB30	BB24-mod	Transistor	EF200
fun. evals	34,874,569	20,989,100	898,053	279,863,240	65,789,226	n/a	2,615,238	11,031,042
grad.evals	35,558,735	23,491,133	1,696,949	310,595,033	88,392,235	n/a	3,637,807	1,080,256
Hesse evals	682,103	820	295,851	82,063	222	n/a	789,087	200
bisections	5,777,688	1,603,978	56,435	3,694,559	1,472,865	n/a	138,951	1,801
preconds	10,486,933	2,997,809	87,508	3,570,485	2,190,128	n/a	286,216	3
bc3.rev	15	21	1,092	1,056	4,656	n/a	52	2,400
hc	282,907	1,406	4,141	8,756,612	2,577,003	n/a	298,050	4,388
3B/bnd.cons	313,406	452	1,268	1	863,681	n/a	181,261	0
pos.boxes	1,872,259	627,067	421	0	0	n/a	0	0
verif.boxes	4,098	203,577	816	2	1	n/a	1	1
Leb.poss	0.000323	1e-47	8e-84	0.0	0.0	n/a	0.0	0.0
Leb.verif	2e-6	1e-12	5e-72	1e-323	5e-324	n/a	1e-112	5e-324
time	41	13	3	192	698	> 3,600	16	9

Table 3. Computational results for the “Beta 2.5” solver version

Problem	5R planar	Puma7	Brent10	BT50	BB30	BB24-mod	Transistor	EF200
fun. evals	173,706,494	28,125,076	9,348,171	118,071	39,817,448	2,343,834,369	80,230,271	12,156,051
grad.evals	47,579,449	31,068,946	6,414,680	118,555	3,075,583	140,805,784	10,551,334	1,161,508
Hesse evals	333,876	637	260,631	60	270	94,705	318,036	200
bisections	5,471,725	2,124,805	50,193	1	16,574	1,265,552	53,239	1,298
preconds	10,087,309	3,947,726	78,336	6	19,441	1,771,075	100,787	2
bc3.rev	67,838,381	235,058	2,288,678	29,147	19,327,021	1,142,463,509	39,421,250	762,070
hc	94,045	1,282	713	325	31,932	2,471,674	110,407	3,314
3B/bnd.cons	117,320	393	99	1	65	6,465	48,197	0
pos.boxes	1,868,601	846,350	419	2	0	0	0	0
verif.boxes	3,415	247,477	820	0	1	1	1	1
Leb.poss	0.000334	2e-47	2e-82	1e-323	0.0	0.0	0.0	0.0
Leb.verif	1e-6	1e-12	7e-65	0.0	5e-7	4e-9	3e-118	5e-324
time	53	17	11	< 1	9	390	21	29

For comparison, let us consider some results, obtained using another solver, *Realpaver* [1] – a mature interval solver that can be considered the current state-of-the-art:

- 5R-planar – 17 min (for `Bisection precision = 2.0`, much less accurate than the presented solver) and did not cover the whole solution set (“Property: non reliable process (some solutions may be lost)”).
- Brent10 – 55 sec to find all solutions (1065); parameter `-number 2000` must be set to loose no solution.
- Transistor – 30 sec to find the solution for the default setting.

6 Analysis of the Results

Replacing `box-` with hull-consistency resulted in a minor speedup, for 5R-planar and Puma7 problems and a major one for Brent10, Transistor and Extended-Freudenstein200 (see Tables 1 and 2. Hence for problems BT50, BB30 and BB24-mod, we obtained a significant slowdown.

Combining both consistencies (Table 3) resulted in reasonable runtimes for all problems. The time for problems BT50 and BB24-mod have been particularly good – better than for any of the previous algorithm versions. Unfortunately, the speedup for Brent10 and EF200 problems, that had been observed for the “HC only” version, has not been preserved. The author has not managed to design a better heuristic.

As for *Realpaver* – our solver performed better on all problems; in earlier versions (e.g., [20]), it had been outperformed for problems, where hull-consistency was very efficient, like the Transistor problem.

7 Conclusions

We investigated incorporating of a hull-consistency enforcing procedure to the interval nonlinear systems solver. Contrary to author’s earlier fears (see [19], Sect. 3), we managed to implement this function in a MT-safe and MT-efficient (yet not parallelized itself) manner.

In general, trying to replace `box-` with hull-consistency is often very worthwhile, but there are significant exceptions to this rule; in our experiments hull-consistency turned out to be inefficient on various instances of the Broyden function: BT50, BB30, BB24-mod.

Enforcing hull-consistency is less computationally intensive than box-consistency, but the reduction of the box diameter is usually smaller. An exception to this rule are constraints, where a variable occurs only once; in such cases hull-consistency is definitely superior to box-consistency. This is consistent with results obtained by other researchers, e.g., [10]. Reasonable results have been obtained for the algorithm version, combining hull- and box-consistency enforcing procedures. Unfortunately, these results, while acceptable, are significantly worse than using “HC only”, for some problems. As designing a better heuristic seems difficult, using machine learning might be a proper direction [9].

Acknowledgments. The author is grateful to Roman Wyrzykowski (Częstochowa University of Technology) and the team of the MICLAB project [4], for providing the great machine with Xeon and Xeon Phi processors, on which the computations have been performed.

References

1. Realpaver: Nonlinear constraint solving and rigorous global optimization (2014). <http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/>
2. C++ eXtended Scientific Computing library (2015). <http://www.xsc.de>
3. Intel TBB (2015). <http://www.threadingbuildingblocks.org>
4. MICLAB project (2015). <http://miclab.pl>
5. ADHC, C++ library (2017). https://www.researchgate.net/publication/316610415_ADHC_Algorithmic_Differentiation_and_Hull_Consistency_Alfa-05
6. HIBA_USNE, C++ library (2017). https://www.researchgate.net/publication/316687827_HIBA_USNE_Heuristical_Interval_Branch-and-prune_Algorithm_for_Underdetermined_and_well-determined_Systems_of_Nonlinear_Equations_-_Beta_25
7. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: International Conference on Logic Programming, pp. 230–244. The MIT Press (1999)
8. Benhamou, F., McAllester, D., Hentenryck, P.V.: CLP (intervals) revisited. In: Logic Programming, Proceedings of the 1994 International Symposium, pp. 124–138. The MIT Press (1994)
9. Goualard, F., Jermann, C.: A reinforcement learning approach to interval constraint propagation. *Constraints* **13**(1–2), 206–226 (2008)
10. Granvilliers, L.: On the combination of interval constraint solvers. *Reliable Comput.* **7**(6), 467–483 (2001)
11. Granvilliers, L., Benhamou, F.: Progress in the solving of a circuit design problem. *J. Global Optim.* **20**(2), 155–168 (2001)
12. Hansen, E., Walster, W.: *Global Optimization Using Interval Analysis*. Marcel Dekker, New York (2004)
13. Kearfott, R.B.: *Rigorous Global Search: Continuous Problems*. Kluwer, Dordrecht (1996)
14. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. *Vychislennyye Tiekhnologii (Comput. Technol.)* **15**(1), 7–13 (2010)
15. Kubica, B.J.: Interval methods for solving underdetermined nonlinear equations systems. *Reliable Comput.* **15**, 207–217 (2011)
16. Kubica, B.J.: Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) *PPAM 2011. LNCS*, vol. 7204, pp. 467–476. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31500-8_48
17. Kubica, B.J.: Excluding regions using Sobol sequences in an interval branch-and-prune method for nonlinear systems. *Reliable Comput.* **19**(4), 385–397 (2014)

18. Kubica, B.J.: Using quadratic approximations in an interval method for solving underdetermined and well-determined nonlinear systems. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8385, pp. 623–633. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55195-6_59
19. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. *Numer. Algorithms* **70**(4), 929–963 (2015)
20. Kubica, B.J.: Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. *J. Parallel Distrib. Comput.* **107**, 57–66 (2017)