Advanced interval tools for computing solutions of continuous games

B. J. Kubica
Warsaw University of Life Sciences, 02-776 Warsaw, Poland
Corresponding e-mail: bartlomiej_kubica@sggw.pl

Computing Nash equilibria in continuous games is a difficult problem, but interval methods have already been applied to its solution quite successfully. The purpose of this paper is to briefly survey previous efforts and achievements of the author related to the topic, and to consider some advanced tools for accelerating the interval branch-and-bound-type methods. In particular, we discuss computing eigenvalues of interval matrices, use of algorithmic (automatic) differentiation, memory management techniques as well as advanced parallelization in both shared-memory and distributed-memory environments.

Keywords: Nash equilibria, continuous games, interval computations, eigenvalues, automatic differentiation, memory management, threads, MPI.

Introduction

Game theory tries to predict decisions and/or advise the decision makers on how to behave in a situation when several players (sometimes called "agents") have to choose their behavior that will also influence the others. In the game theory, the behaviour of a separate player can be described by its "strategy", and we suppose that the i-th player chooses the strategy $x^i \in X_i$. Usually, it is assumed that each player tends to minimize their cost function (or maximize their utility) represented by $q_i(x^1, \ldots, x^n)$.

So, each of the decision makers solves the following problem:

$$\begin{aligned} &\text{find } \min_{x^i} q_i(x^1, \ldots, x^n)\,, \\ &\text{subject to} \\ &x^i \in X_i\,. \end{aligned} \tag{1}$$

What solution are they going to choose?

One of the oldest, most famous, and still widely-used concepts is the Nash equilibrium [1]. It can be defined as a situation (an assignment of strategies to all players), when each player's strategy is optimal against those of the others. Formally, the tuple $x^* = (x^{1*}, \ldots, x^{n*})$ is a Nash equilibrium, iff

$$(\forall i \in \{1, \ldots, n\})\,(\forall x^i \in X_i)\; q_i(x^{1*}, \ldots, x^{i-1*}, x^i, x^{i+1*}, \ldots, x^{n*}) \geq q_i(x^{1*}, \ldots, x^{n*})\,. \tag{2}$$

Also, we shall use a shorter notation: $(\forall i \in \{1,\ldots,n\})\,(\forall x^i)\ q_i(x^{\setminus i*}, x^i) \geq q_i(x^{\setminus i*}, x^{i*})$.

There are several "refinements" to the notion, in particular, the strong Nash equilibrium (SNE, for short); see [2]. These are points, for which not only none of the players can improve their performance by changing strategy, but also no *coalition* of players can improve the performance of all of its members by mutually deviating from the SNE. Formally:

$$(\forall I \subseteq \{1,\ldots,n\}) \left(\forall x^I \in \bigotimes_{i\in I} X_i\right) (\exists i \in I) \quad q_i(x^{\setminus I*}, x^I) \geq q_i(x^{\setminus I*}, x^{I*})\,, \tag{3}$$

where "$\otimes$" means direct (Cartesian) product of the sets. Also, the notion of a k-SNE (or k-equilibrium) is sometimes encountered. Its definition is similar to the ordinary SNE, but the coalition I in (3) can consist of k members at most. Obviously, a $(k+l)$-SNE is also a k-SNE (if $l > 0$) and, in particular, a SNE is also a k-SNE for any $k = 1, 2, \ldots, n$.

In this paper, we consider continuous single-stage games; i. e., the case, when the player's strategy is a tuple of numbers (vector) they choose from the given set, i. e. $x^i = \left(x^i_1, \ldots, x^i_{k_i}\right) \in X_i \subseteq \mathbb{R}^{k_i}$. Let us denote

K_i — the set of components of the i-th player decision variable x^i,
k_i — its size,
K_I — the union of all K_i for $i \in I$,
$x = (x^1, \ldots, x^n) = (x^1_1, \ldots, x^1_{k_1}, x^2_1, \ldots, x^2_{k_2}, \ldots, x^n_1, \ldots, x^n_{k_n})$.

Also, we refer to Nash points (equilibria) that are not strong as "plain" Nash equilibria, to distinguish them from SNE.

Computing Nash equilibria — plain or strong ones — of such games is a hard problem in general. We are going to present an approach based on interval analysis, extending our earlier algorithm for plain Nash points; see [3–5]. Throughout the paper, the notation from [6] is adopted.

1. Interval methods for seeking points that satisfy a certain condition

Decision problems (2) and (3) are of the following form:

$$\text{Find } \textit{all}\ x \in X \text{ such that } P(x) \text{ is fulfilled}, \tag{4}$$

where $P(x)$ is a formula with a free variable x and $X \subseteq \mathbb{R}^n$.

In his previous papers and presentations(in particular, [7]), the author stated that interval methods are well-suited for solving this kind of problems. Basics of the interval calculus and interval algorithms can be found in several textbooks (e. g., [8–12]) and are out of the scope of this paper.

A generic algorithm for solving arbitrary problems of type (4) is going to be presented elsewhere. Such an algorithm can be named the *generalized branch-and-bound method* or the *branch-and-bound-type* method (B&BT method).

Algorithm 1 below presents the version of B&BT method, specialized for seeking game equilibria. The "push" operator inserts a member (the second argument) into a specified list (the first argument), while "pop" takes a member from the top of the list pointed out as the only argument.

Algorithm 1 The branch-and-bound-type method for seeking SNE

Require: $\boldsymbol{x}^0, \mathsf{q}(\cdot), \varepsilon$
$L_{ver} = L_{pos} = L_{check} = L_{small} = \varnothing$;
$\boldsymbol{x} = \boldsymbol{x}^{(0)}$
loop
  $\boldsymbol{x}^{old} = \boldsymbol{x}$;
  process the box $\boldsymbol{x}$, trying to verify if it does or does not contain a point satisfying the necessary conditions of being a solution;
  if ($\boldsymbol{x}$ was discarded, but not all q_i's are monotonous on it) **then**
    push (L_{check}, $\boldsymbol{x}^{old}$);
    discard $\boldsymbol{x}$
  else if (the tests resulted in two subboxes of $\boldsymbol{x}$: $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$) **then**
    $\boldsymbol{x} = \boldsymbol{x}^{(1)}$;
    push (L, $\boldsymbol{x}^{(2)}$);
    cycle loop
  else if ($\mathrm{wid}\,(\boldsymbol{x}) < \varepsilon$) **then**
    push (L_{small}, $\boldsymbol{x}$)
  end if
  if ($\boldsymbol{x}$ was discarded **or** $\boldsymbol{x}$ was stored) **then**
    $\boldsymbol{x}$ = pop (L);
    if (L was empty) **then**
      break
    end if
  else
    bisect ($\boldsymbol{x}$), obtaining $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$
    $\boldsymbol{x} = \boldsymbol{x}^{(1)}$;
    push (L, $\boldsymbol{x}^{(2)}$)
  end if
end loop
{Second phase — verification}
for all ($\boldsymbol{x} \in L_{small}$) **do**
  check if another solution from L_{small} does not invalidate $\boldsymbol{x}$;
  verify if no box from L_{check} contains a point that would invalidate $\boldsymbol{x}$;
  put $\boldsymbol{x}$ to L_{ver}, L_{pos} or discard it, according to the results
end for
return L_{ver}, L_{pos}

Papers [3] and [5] discuss necessary conditions of Nash and strong Nash equilibria (which are analogous to Karush–Kuhn–Tucker and Fritz John conditions; see, e. g., [10]). Also, they describe the following tools to check these conditions, in the first phase of Algorithm 1:

- a variant of the monotonicity test; see [3] and [5];
- a variant of the "concavity test" — Algorithm 2; see, e. g., [10];
- an interval Hansen-Sengupta operator (a variant of the interval Newton operator that uses interval Gauss-Seidel iterations with the inverse midpoint preconditioner; see, e. g., [8, 10]).

In the second phase, we verify potential solutions, using boxes from L_{check}. The condition, we need to test for plain Nash equilibria is as follows: there is *no* point x in any box in L_{check}, for which

$$\Big((\exists i = 1, \ldots, n) \quad \big(q_i(x) < q_i(x^*)\big)\Big) \text{ and } \Big((\forall j \neq i)\,(x^j = x^{j*})\Big)\,. \tag{5}$$

Hence, for SNE, there can exist *no* point x in any box in L_{check}, for which

$$\Big((\forall i \in \{1, \ldots, n\})\ \big(q_i(x) < q_i(x^*)\big) \text{ or } (x^i = x^{i*})\Big) \text{ and } \Big((\exists i = 1, \ldots, n)\,(x^i \neq x^{i*})\Big)\,. \tag{6}$$

This condition can easily be checked for all other points in the list of potential solutions L_{small}. Boxes in L_{check} are larger, and in general we need to bisect them, performing a "nested" B&BT procedure to verify if they contain a point validating a specific solution or not. For details, the reader is referred to [5], again.

In the remainder, the author would like to discuss some other tools and techniques to accelerate Algorithm 1.

2. Tools and techniques

This section considers the new tools and techniques the author applies to the B&BT method seeking game solutions.

2.1. Estimating eigenvalues of interval matrices

One of the important tests used to discard boxes not containing a game solution is the so-called concavity test. The name "non-convexity test", while still misleading, might be more appropriate to denote a test checking whether an objective function cannot be convex anywhere on a specific box.

The test has already been used for global optimization (see, e. g., [10]), but — according to the author's experiences — it was not very useful there. When we seek the global optimum of a function, the most basic tool we use is sifting away unpromising subboxes, i. e. checking if the values of the objective function are not too high to contain the global optimum. Such a test (often called the "midpoint test"; see [8, 10]) is simpler and computationally cheaper than the concavity test; it does not even require computing any derivatives. And regions where a function is concave must have higher values of this function than the other ones.

When seeking game solutions, the situation is different. We have several objectives there (each player has their own one), and no simple "midpoint tests". Consequently, the concavity test becomes important. A version of this test suitable for seeking game solutions has been presented in [5]. It can be expressed by the pseudocode in Algorithm 2.

What we check here are the values on the main diagonal of the Hesse matrix. If any of them is negative, the matrix cannot be positive definite and the function cannot be convex in the region; this is a simple and well-known necessary condition (cf., e. g., [10]).

In theory, a stronger test could be performed. We could try to enclose all eigenvalues of the interval enclosure of Hesse matrix and check if all of them are non-negative. Such a test, although computationally more intensive, would allow discarding much more boxes.

The question is: how to enclose eigenvalues of an interval matrix? Quite a few papers have been written on the subject.

Algorithm 2 The "concavity" test

Require: $\boldsymbol{x}, \boldsymbol{x}^{(0)}, \boldsymbol{x}^{old}, \mathsf{q}(\cdot)$
1: $n_{conc} = 0$
2: **if** (**not** $\boldsymbol{x} \subset \mathrm{int}\boldsymbol{x}^{(0)}$) **then**
3: **return**
4: **end if**
5: **for** $(i = 1, \ldots, n)$ **do**
6: {check the Hesse matrix of $\mathsf{q}_i(\boldsymbol{x})$ with respect to x^i}
7: **if** ($\frac{\partial^2 \mathsf{q}_i(\boldsymbol{x})}{\partial (x^i_k)^2} < 0$ for some $k = 1, \ldots, k_i$) **then**
8: increment n_{conc}
9: **end if**
10: **end for**
11: **if** $(n_{conc} > 0)$ **then**
12: **if** $(n_{conc} < n)$ **then**
13: push (L_{check}, $\boldsymbol{x}^{old}$)
14: **end if**
15: discard $\boldsymbol{x}$
16: **end if**

Probably the oldest and the most celebrated result is the theorem of Rohn [13]: if we represent an interval matrix $[\underline{A}, \overline{A}]$ in the midpoint-radius manner: $[\mathrm{mid}\,\boldsymbol{A} - \mathrm{rad}\,\boldsymbol{A}, \mathrm{mid}\,\boldsymbol{A} + \mathrm{rad}\,\boldsymbol{A}]$, the eigenvalues of all $A \in \boldsymbol{A}$ can be bounded in the following way:

$$\lambda(A) \in \left[\, \lambda(\mathrm{mid}\,\boldsymbol{A}) - \rho(\mathrm{rad}\,\boldsymbol{A}), \lambda(\mathrm{mid}\,\boldsymbol{A}) + \rho(\mathrm{rad}\,\boldsymbol{A}) \,\right], \tag{7}$$

where $\rho(\cdot)$ is the spectral radius of a matrix.

So, bounding eigenvalues of an interval matrix can be reduced to computing eigenvalues of two floating-point matrices: $\mathrm{mid}\,\boldsymbol{A}$ and $\mathrm{rad}\,\boldsymbol{A}$, which can be done efficiently, using the LAPACK procedure DSYEV. The problem is that these bounds are not guaranteed. The same regards to other proposed algorithms, e. g., [14, 15].

Actually, to the best knowledge of the author, reliable algorithms for bounding eigenvalues are rare; see [16,17]. In control theory, zeros of the characteristic polynomial are used to check the stability of a dynamic system, instead of eigenvalues of the system matrix (cf. [9]).

Consequently, as for now, we can use eigenvalues if we are interested in usual two-sided approximations, but for actual validated computations, we have to stick to the weaker test described by Algorithm 2.

2.2. Automatic (algorithmic) differentiation

Almost all interval B&BT methods use derivatives of some functions, and the same relates to methods for seeking game solutions. This is being done not only in monotonicity or concavity tests, but also to seek points satisfying first-order necessary conditions of being game solutions.

As stated above, such conditions are analogous to the Karush–Kuhn–Tucker condition and the Fritz John condition. In the unconstrained case (or for solutions from the interior of the feasible set), they reduce to the conditions that some partial derivatives are equal to zero. For plain Nash equilibria, it can be expressed precisely as follows:

$$\frac{\partial q_i(x)}{\partial x^i} = 0, \qquad i = 1, \ldots, n, \tag{8}$$

where we use the following notation for the gradient:

$$\frac{\partial q}{\partial x^j} = \left(\frac{\partial q}{\partial x^j_1}, \ldots, \frac{\partial q}{\partial x^j_{k_j}} \right).$$

See [3] for details.

In [5], we formulated similar conditions for strong Nash equilibria; they are slightly more complicated as they form an overdetermined system of equations, but similar, in general.

In any case, efficient computation of gradients and Hesse matrices is pretty important. How can it be done? Numerical procedures based on finite differences are of little use as the error is large and hard to bound, in their case. Symbolic differentiation is rarely applied, because of its difficulties (see, e. g., [10]). Hence, algorithmic differentiation (AD; also, often called "automatic differentiation") is often chosen, as it has several advantages over the alternatives (cf., e. g., [9, 10]).

Yet, it is not simple to find an appropriate library for algorithmic differentiation. The code, from C-XSC library [18] has several drawbacks:

- there are distinct classes (`GradType`, `HessType`, `DerivType`), implemented in distinct files (`grad_ari.cpp`, `hess_ari.cpp`, `ddf_ari.cpp`) for computing the first or second derivatives and for univariate or multivariate functions;
- there are global variables (`GradOrder`, `HessOrder`, `DerivOrder`) to distinguish the order of computed derivative — these variables have to be checked at runtime, several times during the computation; they also affect multithreaded implementations;
- extended interval division (for divisors containing zero) is not supported in the automatic differentiation library;
- computing derivatives of higher order would require to implement a separate (but analogous) class;
- although, the developers of C-XSC have provided several useful classes for sparse matrices and vectors, their AD code makes no use of it.

Recently, the author has developed a novel algorithmic differentiation library, based on C++ templates (see, e. g., [19]). It is called ADHC, which stands for Algorithmic Differentiation and Hull Consistency [21].

Virtues of template meta-programming allow us to obtain several useful features of the ADHC library. The same source code can be used to generate distinct procedures for computing function values, gradients, Hesse matrices and — potentially — higher derivatives. We can use the same source code to differentiate uni- and multivariate functions and to use sparse or dense representations of vectors and matrices of partial derivatives. And C-XSC library provides us pretty nice implementations of sparse vectors and matrices (cf. [20]).

Proper types are generated from the so-called typelist (see, e. g., [19]) and derivatives' values are stored in a tuple. This word, in the C++ context, means a sequence of values of (possibly) various types, but indexed by a number of field and not its name, as it is for structures or classes. The current implementation (alpha.0.1) of the ADHC library uses Loki library [22] for typelists and tuples, but migration to C++11 (variadic templates and `std::tuple`'s) is planned in future versions. The library is available from the author's ResearchGate profile [21].

As we shall see in Section 3, the improvement gained by ADHC library is very significant.

2.3. Memory management

In B&BT algorithms, several boxes have to be processed. The most typical implementation stores them on the heap, which means that, in C++, we have to use `new/delete` operators to maintain them. This is suboptimal for two reasons:

- the default C++ allocator is not tuned for allocating relatively small objects; it is a simple wrapper over the C `malloc/free` operations, yet with some additional overhead; cf. [19];
- memory management in a multithreaded environment turns out to be particularly inefficient: the heap is the resource of a process, not a thread, and allocating memory requires synchronization (the overhead might be minor for the today OS's, but it is inevitable).

How can we improve the performance of memory management? It is possible to use a specialized memory allocator, e. g., the Small Object Allocator from the Loki library [22]. This choice is still far from optimal, as the allocator of Loki library, although MT-safe, is not optimized for use with threads. Nevertheless, it results in a significant improvement, as we shall see in Section 3.

Yet another possibility is to use the move semantics, introduced in the C++11 standard (see any modern book on C++ or, e. g., [23]). This allows to avoid using dynamic variables, but can hardly be applied if we decide to store boxes on linked lists.

2.4. Parallelization

Parallelization is crucial for efficient implementation of interval B&BT algorithms. We can consider a shared-memory parallelization (usually, in a multithreaded environment) or a distributed-memory one (usually, using MPI [24]).

2.4.1. Multithreaded implementation

Such an implementation has been already discussed in the author's previous papers: [3, 4] and, in particular, [5]. For a multithreaded implementation, it is straightforward to share quantities between threads, so realization of both phases of Algorithm 1 is relatively simple.

Objects that may be shared are the lists: L_{small}, L_{ver}, L_{pos} and L_{check}. In the first phase, when the lists get assembled, either access to them has to be synchronized or each thread might have its own sub-list and merge it with the others at the end of the phase. Synchronization may be done either using locks or atomic operations.

In the second phase, when candidate solutions from L_{small} get verified, the list L_{check} does not change, so it can be shared with no synchronization; hence, the solution lists act as in the previous phase.

All the shared lists of boxes — L_{small}, L_{ver}, L_{pos} and L_{check} are stored as linked lists, working as queues and synchronized using two mutex locks. Details of this approach are explained in [3].

We also tested a different representation: storing boxes in `std::vector` containers. As described in the previous subsection, in this version, no `new/delete` operations are used, but the C++11 move-semantics, instead.

A *single* lock (specifically `std::mutex`) is used for each `std::vector` to protect it. This approach is sufficient for four threads, but would not scale for higher numbers, obviously.

Some libraries, in particular, Intel TBB [25], have proper containers (e. g., `tbb::vector`), but it is out of the scope of our paper.

2.4.2. Parallelization using MPI

Distributed-memory parallelization is much more difficult to implement, as we can have no quantities shared between nodes. In phase 1, each node builds its own sub-lists of L_{small} and L_{check}.

In our implementation, this phase is parallelized in a bit rough manner: we split the search domain in the beginning and each node gets its own sub-box of $\boldsymbol{x}^0$; there is no box migration or load balancing. As it would be reprehensible in a production environment, for our experiments, such a simplification seems appropriate for two reasons:

- the test problem used in the experiments — the game of misanthropic individuals, described in Section 3 — is very symmetric and halving the initial box results in very equal distribution of work, in this case;
- load balancing has already been studied by several researchers (e. g., [26]); in this paper, the author wanted to focus on parallelization of the second phase of Algorithm 1 — the verification phase.

As for this phase, we need to verify all boxes from the list L_{small} using other boxes from L_{small} and boxes from L_{check}.

The solution candidate sub-lists might stay on their "native" nodes for verification in the second phase, but all boxes from L_{check} are needed to verify each of them. How to deal with this? There are at least two possibilities.

All boxes to all nodes. The first approach is to create complete lists L_{check} and L_{small} (concatenating all of their sub-lists) and spread them to all of the nodes. MPI has pretty convenient functions `MPI_Allgather()` and `MPI_Allgatherv()` for this purpose.

Now, each node is going to verify boxes from its sub-list of L_{small}, using all boxes of complete lists L_{check} and L_{small} (as in lines 19 and 20 of Algorithm 1).

This approach is nice, when there are relatively few boxes in the lists L_{check} and L_{small}, but for a larger number of them, it wastes the memory of the whole system. Different nodes, could store different boxes in their memory, allowing this way to verify solutions, using a great deal of boxes; possibly exceeding the memory amount of a single node. We have similar approaches, e. g., for multiplying large matrices, like for instance the Cannon algorithm [27] or several out-of-core algorithms.

Exchanging the sub-lists. This approach requires a ring topology of our network, but MPI aids us by maintaining virtual topologies. In the ring topology, each node has its predecessor and successor. After using all boxes in its sub-list, it sends the sub-list to its successor and waits on the sub-list from its predecessor. When the list returns to its "native" node, it means that the whole cycle has been done and the verification is finished. Actually, this happens after the number of steps equal to the number of nodes.

Also, please note that MPI has a convenient `MPI_Sendrecv()` function, which might be pretty handy here. If our interval arithmetic library does not support the direct use of this function (and this seems to be the case for CXSC-MPI [28]; cf. [20, 29]), it can be replaced by a sequence of calls to non-blocking MPI operations. Details are given below.

The question is: which list should wander over the nodes, as described? We have two possibilities here:

- to migrate the list(s) of boxes possibly dominating over solutions or
- to migrate the list(s) of potential solutions.

Which is more beneficial? This might be problem-dependent, but a look at Tables 1 and 2 (and also on Table 1 in [5]) suggests (lines "boxes after 1st phase" and "possibly dominating"), moving the solutions would be much more efficient, as there are far fewer of them (at last, for problems we have considered). So, what is transferred between subsequent nodes, are sub-lists of L_{small}.

There is an additional benefit: if we verify a box *not* to contain any solutions, we do not have to check it on further nodes; such a node is just not transferred further.

We count the steps and consider verified the boxes that have not been rejected after being verified on all N nodes, i. e., sent $N-1$ times.

Also, there is a possibility that some of the transferred sub-lists become empty. In this case, during further steps, the interested nodes receive a message, with the empty list.

There is another feature that makes this verification a bit awkward in a distributed system. After verifying a sub-list of L_{small}, using a sub-list of L_{check}, un-rejected boxes from L_{small} get categorized into two lists: L_{ver} and L_{pos}. Please note that this instance of L_{ver} does not contain boxes that are actually "verified", but that still might become verified solutions; it depends on the result of verification procedure, using other sub-lists of L_{check}. Hence, the instance of L_{pos} contains boxes that have no likelihood to become "verified"; if they do not get rejected, they will stay "possible" solutions.

This means, we need to transfer *two* lists of solutions between adjacent nodes (and distinguish them with different tags). And merge the lists properly. This is much more cumbersome than the case of the multithreaded implementation, when a single procedure has been resulting in final classification of solutions.

Also, it is worth to describe the implementation of the transfer operation of both lists, between adjacent nodes. It would seem the most natural to perform two `MPI_Isend()` operation for L_{ver} and L_{pos}, two `MPI_Irecv()`'s and then a `MPI_Waitall()`, but — unfortunately — CXSC-MPI does not implement `MPI_Irecv()`, either. Consequently, after sending both lists in a non-blocking manner, we wait on data from the predecessor using plain `MPI_Recv()`. Finally, we call `MPI_Waitall()` to make sure that both send operations have been completed.

After all steps of the verification procedure are complete, the root node collects all solutions, using an `MPI_Gatherv()` operation and unpacks them in a loop (each sub-vector has its own length, stored in the packed memory area), inserting into a common vector.

Finally, the global statistics are computed, by adding together statistics from all nodes, using `MPI_Reduce()`. All gathered statistics are stored as long integer numbers, so they can be reduced using a single collective operation, after putting them into a contiguous memory area.

Remarks on CXSC-MPI. To transfer C-XSC [18] interval-related types over MPI, the library CXSC-MPI [28] has been applied. It is worth noting that using this library turned out much more cumbersome than expected. In particular, the following operations on interval data have not been implemented yet in CXSC-MPI:

- non-blocking operations other than sending, in particular `MPI_Irecv()`,
- `MPI_Sendrecv()` or `MPI_Sendrecv_replace()`,

- collective operations more sophisticated than `MPI_Bcast()`, in particular `MPI_Scatter()`, `MPI_Gather()` or `MPI_Gatherv()`.

These limitations can be worked around in a few ways. For instance, an interval could be replaced with a pair of floating-point numbers, but this requires wasting time on transforming data structures.

The author has managed to implement on his own some suitable functions, which are:

- overloaded `MPI_Pack()` and `MPI_Unpack()` functions for the representation of a box (containing two `cxsc::ivectors`: $\boldsymbol{x}$ and $\boldsymbol{y}$),
- the function to perform `MPI_Gatherv()` operation on vectors of boxes: the vectors are `MPI_Pack()`ed, sizes of all sub-lists `MPI_Gather()`ed and finally an `MPI_Gatherv()` is executed on contiguous memory tiles, containing packed data.

It is undebatable that the CXSC-MPI library needs updates promptly, as long as manual implementation of these operations required significant efforts.

3. Numerical experiments

Numerical experiments have been performed on a computer with four cores (allowing hyper-threading), namely, an Intel Core i7-3632QM with 2.2GHz clock. The machine runs under control of a 64-bit Manjaro 0.8.8 GNU/Linux operating system with the GCC 4.8.2, glibc 2.18 and the Linux kernel 3.10.22-1-MANJARO.

The solver has been written in C++ and compiled using the GCC compiler. The C-XSC library (version 2.5.4) [18] has been used for interval computations. The shared-memory parallelization has been done using the threads of the C++11 standard, while distributed-memory — using MPI [24] and CXSC-MPI [28]. OpenBLAS 0.2.15 [30] has been linked for BLAS operations.

Experiments with MPI have been performed on the same machine; unfortunately, the author had no access to a computer cluster, that would be appropriate for tests of the distributed application. On a really distributed system, the MPI-based version of the solver would probably have a higher overhead, related to communication. Yet, the used shared-memory simulation allows drawing some conclusions, as we shall see. OpenMPI [31] has been used, version 1.6.5.

Considered example. We present results for "The game of misanthropic individuals" introduced in [5].

Consider n players choosing their positions on a "compact board", a two-dimensional domain for which we choose the rectangle $D = [-3, 3] \times [-2, 2]$. Their objective is to be as far from the others as possible. Specifically, we assume that each of the players (let us give him the number $i = 1, \ldots, n$) maximizes, by choosing position $(x_i, y_i) \in D$, the following function:

$$q_i(x_i, y_i) = \sum_{j=1, j\neq i}^{n} \big((x_i - x_j)^2 + (y_i - y_j)^2\big). \tag{9}$$

Solutions of the game. Depending on n, the game can have different numbers of Nash equilibria, all or none of them being strong.

For two players, we have 4 Nash equilibrium points, each of them are strong. Their structures are obvious: one of the individuals is located in one of the four corners and the

other one — diagonally opposite to him. It is clear that all of them are SNE — cooperation of both players cannot increase their distance in any way. This case is a "degenerate" case of a game, as both players maximize the same function — the (square of) the distance between them.

For three players we have 36 Nash equilibria: 24 with all three individuals located in different corners ($4 \times 3 \times 2$) and 12 with one of the three individuals in a corner and both others diagonally opposite to him (one of the 3 individuals $\times$ 4 corners). In all cases, one of the individuals has a better position than the two others. And actually, none of these solutions is strong — the two players with worse values can always collude to change their positions and improve their payoffs at the expense of the third player.

For four players, we have 36 Nash equilibria: 24 solutions with each individual in his own corner ($4 \times 3 \times 2$) and 12 solutions with two pairs of players in opposite corners. Counter-intuitively, formula (9) makes their values identical for both types of solutions. All of these 36 solutions are strong Nash equilibria.

For larger number of players, it is very difficult to analyze all possible solutions and their structures.

Results are presented in Tables 1, 2, 3 and 4. Accuracy $\varepsilon = 10^{-8}$ is set in all cases.

These tables contain the information on:

- the number of cost function evaluations, its gradients and Hesse matrices,
- the number of bisections,
- numbers of boxes deleted by various tools,
- "boxes after 1st ph." — the number of potential solutions, stored in the list L_{small},
- "possibly dominating" — the number of boxes, stored in the list L_{check},
- numbers of verified and possible solutions,
- "time [5] (sec.)" — computation time for the solver version, described in the given paper,
- "time ADHC (sec.)" — computation time for the solver version using ADHC library for algorithmic differentiation,
- "time AHDC + SmallObj (sec.)" — computation time for the version using ADHC and the memory allocator from Loki [22],
- "time ADHC + vector" — computation time for the version using ADHC and boxes stored not in linked lists, but in `std::vector` containers and no `new`/`delete` operations, but the C++11 move-semantics, only (cf. Subsection 2.3),
- "time (sec.)" — computation time for the MPI-based version.

Analysis of the results. Using the author's ADHC library for algorithmic differentiation proved to be very worthwhile. In all but one cases, the computational time is significantly shorter (almost twice in one case!). Obviously, the impact is more visible for problems of higher dimensions.

The exception mentioned computed SNE for the misanthropic individuals game with $n = 6$ players, and the behavior we observed was a bit mysterious. Probably, it was related to the fact that, for this problem, the computation time is dominated by the second phase, where a nested B&BT algorithm is used to check whether no coalition can improve the outcomes of its members. This nested algorithm computes objective function values and gradients, but, apparently, is dominated by comparison operations and recursion.

As for using the Loki's "Small Object" memory allocator, its usage turned out very justified too, which can be seen in Table 1. Why in the second table, where we seek strong

T a b l e 1. Computational results for seeking plain Nash equilibria, using the solver with four threads

Players number	2	3	4	5	6	7
cost fun. evals	5196	47 335	47 602	685 225	1 099 660	14 443 406
gradient evals	0	0	0	0	0	0
Hesse matrix evals	3774	18 141	71 164	300 555	1 136 634	4 677 113
bisections	943	3023	8895	30 055	94719	334 079
deleted monot. test.	0	0	0	168	256	1536
deleted "conc."	928	2960	8640	28 864	90 368	316 160
deleted Newton	0	0	0	0	0	0
boxes after 1st ph.	16	64	256	1024	4096	16 384
possibly dominating	944	3280	10 304	41 972	133 120	516 864
deleted 2nd phase	12	28	220	624	3696	11 484
removed monot.test.	0	0	0	0	0	0
removed "conc."	0	0	0	0	0	0
possible solutions	0	32	0	336	0	4000
verified solutions	4	4	36	64	400	900
time [5] (sec.)	0.474	0.581	1.220	7.296	37	483
time ADHC (sec.)	0.033	0.127	0.700	5.005	24	387
time AHDC + SmallObj (sec.)	0.030	0.123	0.642	5.078	23	337
time ADHC + vector (sec.)	0.025	0.136	0.629	4.781	21	295

T a b l e 2. Computational results for seeking SNE, using the solver with four threads

Players number	2	3	4	5	6	7
cost fun. evals	7616	1164	4 853 056	70 235	5 576 803 158	1 519 735
gradient evals	0	0	728 800	0	1 210 016 856	0
Hesse matrix evals	3774	18 141	71 164	300 555	1 136 634	4 677 113
bisections	943	3023	8895	30 055	94 719	334 079
deleted monot. test.	0	0	220	168	256	1536
deleted strong mon.	0	0	0	0	0	0
deleted "conc."	928	2960	8640	28 864	90 368	316 160
deleted Newton	0	0	0	0	0	0
boxes after 1st ph.	16	64	256	1024	4096	16 384
possibly dominating	944	3280	10 304	41 972	133 120	516 864
deleted 2nd phase	12	64	220	1024	3696	16 384
removed monot.test.	0	0	0	0	0	0
removed "conc."	0	0	0	0	0	0
possible solutions	0	0	36	0	400	0
verified solutions	4	0	0	0	0	0
time [5] (sec.)	0.452	0.555	4.442	5.577	5221	189
time ADHC (sec.)	0.030	0.111	2.867	3.353	6014	99
time AHDC + SmallObj (sec.)	0.025	0.092	2.844	3.389	5876	100
time ADHC + vector (sec.)	0.027	0.118	2.939	3.400	6140	96

Nash equilibria, it did not improve the time? The answer is simple: for this kind of problems, the majority of time is devoted to the second phase, which is implemented in a recursive manner, hence the memory is not allocated or deallocated dynamically.

T a b l e 3. Computational results for seeking plain Nash equilibria, using MPI with four processes

Players number	2	3	4	5	6	7
cost fun. evals	8874	51 102	53 153	934 923	1 482 977	42 968 682
gradient evals	0	0	0	0	0	0
Hesse matrix evals	3768	18 132	71 152	300 540	1 136 616	4 677 092
bisections	940	3020	8892	30 052	94716	334076
deleted monot. test.	0	0	0	168	256	1536
deleted "conc."	928	2960	8640	28 864	90 368	316 160
deleted Newton	0	0	0	0	0	0
boxes after 1st ph.	16	64	256	1024	4096	16 384
possibly dominating	944	3280	10 304	41 972	133 120	516 864
deleted 2nd phase	12	28	220	628	3696	11 487
removed monot.test.	0	0	0	0	0	0
removed "conc."	0	0	0	0	0	0
possible solutions	0	32	0	336	1	4272
verified solutions	4	4	36	60	399	625
time (sec.)	1.093	1.182	1.754	6.263	24	273

T a b l e 4. Computational results for seeking SNE, using MPI with four processes

Players number	2	3	4	5	6	7
cost fun. evals	1 478 161	182 458	4 853 032	6 838 563	6 777 913 316	1 511 517
gradient evals	734 349	10 030	728 800	606 248	1 437 068 624	0
Hesse matrix evals	3768	18 141	71 152	300 540	1 136 616	4 677 092
bisections	940	3020	8892	30 051	94 716	334 076
deleted monot. test.	0	0	220	168	256	1536
deleted strong mon.	0	0	0	0	0	0
deleted "conc."	928	2960	8640	28 864	90 368	316 160
deleted Newton	0	0	0	0	0	0
boxes after 1st ph.	16	64	256	1024	4096	16 384
possibly dominating	944	3280	10 304	41 972	133 120	516 864
deleted 2nd phase	12	64	220	1024	3696	16 384
removed monot.test.	0	0	0	0	0	0
removed "conc."	0	0	0	0	0	0
possible solutions	0	0	36	0	400	0
verified solutions	4	0	0	0	0	0
time (sec.)	2.152	1.191	3.591	10.188	8378	96

As for the version using the vector container, with the move-semantics, instead of dynamic allocations on the heap, results for higher dimensions seem to be better than for dynamic allocation — even for the version using the Small Object Allocator. For small problem dimensions, the difference is insignificant.

With regard to the MPI-based version, we can observe several interesting facts.

Initially, the first phase of Algorithm 1 works identically, as for the shared-memory versions. The same number of boxes in L_{small} and L_{check} lists has generated (for the distributed implementation, they are spread between various machines, which is not reflected in Tables 3 and 4). The numbers of bisections differ in a very regular manner: the MPI-based version uses as many bisections as the multithreaded one minus 3. This fact can be easily explai-

ned, by the structure of the author's implementation: the MPI-based version subdivides the initial domain between four machines, which is equivalent to performing exactly three bisections (one for the initial box and one for each of its direct subboxes). Small differences in the number of Hesse matrix evaluations are probably caused by the same reason.

As for numbers of objectives' and their gradients' evaluations, the differences are more significant, because they are used in the second phase. Performance of this verification procedure differs highly between the shared- and distributed-memory versions. In the case of shared-memory, we use all potential solutions from L_{small} and, if the box has not been invalidated, we use boxes from L_{check}. In the case of distributed-memory, we use subsequent sub-lists of L_{small}, followed by corresponding sub-lists of L_{check}.

Hence, computation times between the two implementations tend to differ. In particular, for computing SNE, when the verification phase is very complicated and time-consuming (please compare Tables 2 and 4).

For small dimensions of the problem, the MPI-based version seems to need slightly more time to execute. This is probably the overhead of MPI: starting "heavy" processes, instead of more "lightweight" threads, etc. On a real cluster, this overhead would be likely to yet increase.

For higher dimensions, this overhead becomes less significant. Sometimes, the MPI-based version is even more efficient, e. g., for computing plain Nash equilibrium, for $n = 7$ (cf. Tables 1 and 3). This might be related to the fact, that MPI needs no synchronization in the first phase, while our vector-based multithreaded version has a single lock for each list.

Again, computing SNE for $n = 6$ behaves differently: the process is long in all cases and the difference in computation time between the versions is huge. This behavior must be related to the number of solutions for verification, but still it seems hard to explain.

Finally, we have to note small differences in numbers of verified solutions: for computing plain Nash equilibria in case of $n = 5, 6, 7$. The author has no explanation for this interesting phenomenon; certainly, it requires further careful studies.

4. Future research

The paper opens several topics, important and interesting for further investigation. Many of them are closely connected with the problem of seeking game solutions (e. g., designing specific heuristics for this class of problems or incorporating box or hull consistency checking to the algorithm), but others are more generic, and these are the ones the author would like to focus on.

In particular, the future research can include

- finding (designing? adapting?) a better memory management technique, namely a more scalable allocator, which is crucial for multithreaded implementations, e. g., on Intel Xeon Phi or other modern many-core architectures;
- augmenting the CXSC-MPI library for better support of features, available in MPI for built-in types, but not for interval-related types, e. g., the `MPI_Allgather()` function;
- investigating the possibilities of the modern MPI-3 standard (and the new one-sided communication primitives or non-blocking collective functions — see, e. g., [32], [33]). It looks quite promising.

Conclusions

The paper presented the author's investigations on an interval solver, seeking (strong) Nash equilibrium points of continuous games. It discussed several — lately implemented and considered for implementation — tools and techniques, useful in this solver. The author did not want to focus on features specific to the problem under consideration (e. g., heuristics tuned for seeking game solutions), but issues that are as common as possible to several interval B&BT algorithms.

The problem of seeking game solutions seems particularly hard for distributed-memory parallelization, because of the importance of list L_{check} that cannot be shared, in this case. Nevertheless, the author has presented a working solution, that might be useful for other problem classes, also.

A preliminary comparison of efficiency of the versions has been done. Finally, we have outlined possible directions of further research in the area.

References

[1] **Nash, J.F.** Equilibrium points in n-person games // Proceedings of National Association of Science. 1950. Vol. 36, No. 1. P. 48–49.

[2] **Aumann, R.J.** Acceptable points in general cooperative games // Contributions to the Theory of Games IV. A.W. Tucker and R.D. Luce, eds. Princeton: Princeton Univ. Press, 1959. 287–324 p.

[3] **Kubica, B.J., Woźniak, A.** An interval method for seeking the Nash equilibria of non-cooperative games // PPAM 2009 Proceedings. Lecture Notes in Computer Science, Vol. 6068. Springer, 2010. P. 446–455.

[4] **Kubica, B.J., Woźniak, A.** Applying an interval method for a four agent economy analysis // PPAM 2011 Proceedings. Lecture Notes in Computer Science, Vol. 7204. Springer, 2012. P. 477–483.

[5] **Kubica, B.J., Woźniak, A.** Interval methods for computing strong Nash equilibria of continuous games // Decision Making in Manufacturing and Services. 2015. Vol. 9(1). P. 63–78.

[6] **Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.** Standardized notation in interval analysis // Comput. Technologies. 2010. Vol. 15, No. 1. P. 7–13.

[7] **Kubica, B.J.** A class of problems that can be solved using interval algorithms // Computing. 2012. Vol. 94. P. 271–280.

[8] **Hansen, E., Walster, W.** Global optimization using interval analysis. New York: Marcel Dekker, 2004. 515 p.

[9] **Jaulin, L., Kieffer, M., Didrit, O., Walter. E.** Applied interval analysis. London: Springer, 2001. 378 p.

[10] **Kearfott, R.B.** Rigorous global search: Continuous problems. Dordrecht: Kluwer, 1996. 278 p.

[11] **Moore, R.E., Kearfott, R.B., Cloud, M.J.** Introduction to interval analysis. Philadelphia: SIAM, 2009. 234 p.

[12] **Shary, S.P.** Finite-dimensional interval analysis. XYZ, 2017. Electronic book. (In Russ.)
Шарый С.П. Конечномерный интервальный анализ. XYZ, 2017. Available at: `http://www.nsc.ru/interval/Library/InteBooks/SharyBook.pdf` (accessed 03.04.2017).

[13] **Rohn, J., Deif, A.** On the range of eigenvalues of an interval matrix // Computing. 1992. Vol. 47, No. 3. P. 373–377.

[14] **Hladík, M., Daney, D., Tsigaridas, E.** A filtering method for the interval eigenvalue problem // Applied Mathematics and Computation. 2011. Vol. 217, No. 12. P. 5236–5242.

[15] **Hladík, M.** Bounds on eigenvalues of real and complex interval matrices // Applied Mathematics and Computation. 2013. Vol. 219, No. 10. P. 5584–5591.

[16] **Mayer, G.** Result verification for eigenvectors and eigenvalues // Topics in Validated Computations. Proc. of the IMACS-GAMM Intern. Workshop on Validated Computation, Oldenburg, Germany, 30 August–3 September, 1993 / J. Herzberger, ed. Amsterdam: Elsevier, 1994. P. 209–276.

[17] VERSOFT. Verification software in MATLAB / INTLAB. Available at: `http://uivtx.cs.cas.cz/~rohn/matlab/`.

[18] C-XSC. C++ eXtended Scientific Computing library. Available at: `http://www.xsc.de`, 2015.

[19] **Alexandrescu, A.** Modern C++ design: Generic programming and design patterns applied. Boston: Addison-Wesley, 2001. 352 p.

[20] **Krämer, W., Zimmer, M., Hofschuster, W.** Using C-XSC for high performance verified computing // Applied Parallel and Scientific Computing. 10th Intern. Conf., PARA 2010, Reykjavik, Iceland, June 6–9, 2010. Lecture Notes in Computer Science Vol. 7134. Springer, 2012. P. 168–178.

[21] ADHC template library. Available at: `https://www.researchgate.net/profile/Bartlomiej_Kubica/publications?category=data/`, 2017.

[22] Loki C++ template library. Available at: `http://loki-lib.sourceforge.net`, 2015.

[23] **Williams, A.** C++ Concurrency in Action. New York: Manning Publications, 2012. 528 p.

[24] MPI. Message Passing Interface. Available at: `http://www.mpi-forum.org`, 2015.

[25] Intel Threading Building Blocks. `http://www.threadingbuildingblocks.org`, 2015.

[26] **Gau, C.-Y., Stadtherr, M.A.** Dynamic load balancing for parallel interval-Newton using message passing // Computers & Chemical Engineering. 2002. Vol. 26, No. 6. P. 811–825.

[27] **Cannon, L. E.** A cellular computer to implement the Kalman filter algorithm. Technical report, DTIC Document, 1969. 229 p.

[28] MPI extension for the use of C-XSC in parallel environments. Available at: `http://www2.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html#cxsc_mpi`, 2015.

[29] **Grimmer, M.** An MPI extension for the use of C-XSC in parallel environments. Electronic book. `http://www2.math.uni-wuppertal.de/~xsc/preprints/prep_05_3.pdf`, 2005.

[30] OpenBLAS library. Available at: `http://xianyi.github.com/OpenBLAS/`, 2015.

[31] OpenMPI library. Available at: `https://www.open-mpi.org`, 2015.

[32] **Hoffler, T.** Advanced MPI: New features of MPI-3. Electronic book. `http://htor.inf.ethz.ch/teaching/mpi_tutorials/speedup15/hoefler-advanced-mpi-speedup15.pdf`, 2016.

[33] **Brinsky, M.** An introduction to MPI-3.0 shared memory programming. Electronic book. `https://goparallel.sourceforge.net/wp-content/uploads/2015/06/PUM21-2-An_Introduction_to_MPI-3.pdf`, 2016.

Received 3 April 2017
Received in revised form 11 January 2018