Numerical Algorithms (2023) 94:1673–1704
https://doi.org/10.1007/s11075-023-01551-8

ORIGINAL PAPER

# Algorithmic differentiation and hull-consistency enforcing using C++ template meta-programming

**Bartłomiej Jacek Kubica**[1]

Received: 22 July 2021 / Accepted: 30 March 2023 / Published online: 8 June 2023


**Abstract**
Algorithmic differentiation is a tool used in several branches of computational science, both in conjunction with the interval calculus, and apart of it. This paper presents the ADHC library [14] developed by the author, and identified as package "na60" by *Numerical Algorithms* journal. This library makes intensive use of the C++ template meta-programming, and it has several unique features. ADHC seems particularly useful for interval-related applications. The library has been used by some solvers, also developed by the author, including HIBA_USNE. The paper describes the library, presenting its features, focusing on the new ones, added in version 2.0; in particular, we describe bounding subdifferentials of non-smooth functions and computing derivatives over various datatypes. Efficiency comparison with respect to other packages is also presented. Then some examples of ADHC applications are given, involving the use of HIBA_USNE and standard benchmark problems for solving nonlinear systems. A particular emphasis is put on examples related to modern machine learning, but not limited to them. Planned extensions and possible directions for future development of ADHC are also outlined and discussed.

**Keywords** Algorithmic differentiation · Hull-consistency · C++ template meta-programming · Interval calculus

## 1 Introduction

While solving many problems, like constraint satisfaction, nonlinear equations systems, or optimization, our programs need to analyze complicated mathematical formulae. This is the case, when we want to compute the derivative of a function

✉ Bartłomiej Jacek Kubica
bartlomiej_kubica@sggw.edu.pl

[1] Institute of Information Technology, Warsaw University of Life Sciences – SGGW, ul. Nowoursynowska 159, 02-776 Warsaw, Poland

(for instance in Newton operators), or enforce some kind of consistency, like, e.g., the so-called hull-consistency (HC) [19].

To name a few examples:

- When solving nonlinear equations systems, we need the Jacobi matrix (or its analog [38]) for the Newton operator.
- When solving an unconstrained optimization problem, we need gradients and Hesse matrices — also for the Newton method.
- When solving a constrained optimization problem, we often need to solve some necessary conditions system, like the Kuhn-Tucker or Fritz John system; this requires gradients and Hesse matrices not only of the objective function, but also of the constraints.
- A similar situation can be obtained for seeking Pareto-optimal points of a multi-criteria problem (see, e.g., [50]).

In all above cases (and many others), we need to process mathematical formulae, to obtain the desired derivatives (or enforce consistency).

A particular area, when we encounter sophisticated expressions to differentiate (or process in another manner), is modern machine learning (ML), using many kinds of artificial neural networks, in particular — deep neural networks, described in [28] (or, in a more popular manner, in [27]).

There are a few approaches to perform computing of the derivatives, but a very promising one is the so-called *algorithmic differentiation* (AD), also known as *automatic differentiation*. This technique is based on the following observation: each function evaluated by a computer is described by a computer program that consists of several elementary operations (arithmetic operations, transcendental functions, etc.). As it is known, how to compute derivatives of such elementary operations, we can enhance this program, so that it computed derivative(s) together with the original function.

There are many packages and tools to perform it. One of the oldest is ADOL-C [6], written in C++. There are also several other packages for C++, as well (e.g., [5, 7]); also Python has its TensorFlow [13], very popular nowadays. A good survey of several approaches is [34].

In this paper, we are going to focus on the ADHC library [14], written by the author, and identified as "na60" by *Numerical Algorithms* journal. It is a free library, available under the GPLv2 license. As we shall see, it has some unique features:

- it allows creation of procedures to enforce hull-consistency, as well as differentiation,
- the same (template) class is used for all kinds of computations,
- it allows both sparse and dense gradient and Hesse matrix representations,
- it allows not only computing gradients of smooth functions, but also bounding subdifferentials of non-smooth ones,
- since ADHC version 2.0, computations can be done not only using the `cxsc::interval` datatype, but also other types: pointwise and interval-valued ones, double- and extended precision ones, real-valued and complex ones, traditional floating-point and multiple-precision ones, etc. (cf. Section 3.2).

While other libraries (e.g., [1, 3, 5–7]) often have some of the above features, this set seems to be unique (but often with some gaps, as we shall see). In particular, using interval-valued types will be especially beneficial for bounding the subdifferentials of non-smooth functions. Details will be given in Section 3.1.2)

ADHC, has been used in a few programs of the author, in particular in the HIBA_USNE publicly available solver [15], since its version Beta2.0. For details of this solver, the reader can consult [44, 47–49], and the references therein.

The paper is organized as follows. After the introduction in Section 1, Section 2 recapitulates the basic ideas of the interval calculus, algorithmic differentiation, hull-consistency, and template meta-programming. Section 3 is the most important: it describes the ADHC library, its features and potential. In Section 4, the provided library is compared to some alternatives: the old AD code from the original C-XSC [3], PROFIL/BIAS [1], and IBEX [8]. Next, in Section 5, we get a few examples of ADHC's applications: some (but not all) of them related to machine learning and neural networks. Section 6 describes the possibilities of further research, and Section 7 presents the summary and conclusions of the paper. In the Appendix, we describe the Survive-CXSC library.

## 2 Recapitulation of basic ideas

### 2.1 Interval methods

Although the interval calculus is not directly linked to AD, many interval algorithms and solvers make use of AD techniques. A good introduction can be found in many classical textbooks, including, i.a., [33, 35, 38, 55, 56, 58] (or a most recent one [50]). Hence, examples of interval software using AD are: the MATLAB package INTLAB [10], and a few C++ libraries, like PROFIL/BIAS [1] or C-XSC [3], which the author has been using (see the Appendix).

What is the interval calculus? It is a branch of numerical analysis and mathematics that operates on intervals rather than numbers.

Arithmetic (and other) operations on intervals are designed, so that the following condition was fulfilled:

$$\odot \in \{+, -, \cdot, /\},\ \ a \in \mathbf{a},\ \ b \in \mathbf{b} \text{ implies } a \odot b \in \mathbf{a} \odot \mathbf{b}. \tag{1}$$

In other words, the result of an operation on numbers will be contained in the result of an analogous operation on intervals, containing these numbers.

This results in the following formulae for arithmetic operations (cf., e.g., the aforementioned textbooks):

$$\begin{aligned}
&[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] = [\underline{a} + \underline{b}, \overline{a} + \overline{b}],\\
&[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] = [\underline{a} - \overline{b}, \overline{a} - \underline{b}],\\
&[\underline{a}, \overline{a}] \ \cdot\ [\underline{b}, \overline{b}] = [\min(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), \max(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})],\\
&[\underline{a}, \overline{a}] \ /\ [\underline{b}, \overline{b}] = [\underline{a}, \overline{a}] \cdot \left[1 / \overline{b}, 1 / \underline{b}\right], \qquad 0 \notin [\underline{b}, \overline{b}].
\end{aligned} \tag{2}$$

It is worth noting that the above formulae are not the only possible ones. Alternative (and even more general) formulations are possible as well. Details can be found, i.a., in Chapter 2 of [50]. Also, please note that the division by an interval containing zero is also possible — in the extended Kahan-Novoa-Ratz arithmetic [38]:

$$\mathbf{a}/\mathbf{b} = \begin{cases} \mathbf{a} \cdot [1/\overline{b}, 1/\underline{b}] & \text{for } 0 \notin \mathbf{b} \\ [-\infty, +\infty] & \text{for } 0 \in \mathbf{a} \text{ and } 0 \in \mathbf{b} \\ [\overline{a}/\underline{b}, +\infty] & \text{for } \overline{a} < 0 \text{ and } \underline{b} < \overline{b} = 0 \\ [-\infty, \overline{a}/\overline{b}] \cup [\overline{a}/\underline{b}, +\infty] & \text{for } \overline{a} < 0 \text{ and } \underline{b} < 0 < \overline{b} \\ [-\infty, \overline{a}/\overline{b}] & \text{for } \overline{a} < 0 \text{ and } 0 = \underline{b} < \overline{b} \\ [-\infty, \underline{a}/\underline{b}] & \text{for } 0 < \underline{a} \text{ and } \underline{b} < \overline{b} = 0 \\ [-\infty, \underline{a}/\underline{b}] \cup [\underline{a}/\overline{b}, +\infty] & \text{for } 0 < \underline{a} \text{ and } \underline{b} < 0 < \overline{b} \\ [\underline{a}/\overline{b}, +\infty] & \text{for } \underline{a} < 0 \text{ and } 0 = \underline{b} < \overline{b} \\ \emptyset & \text{for } 0 \notin \mathbf{a} \text{ and } 0 = \mathbf{b} \end{cases} . \tag{3}$$

This formula will turn out very useful to us, as we shall see.

Similarly to the arithmetic operations, we can define the power of an interval:

$$[\underline{a}, \overline{a}]^n = \begin{cases} [\underline{a}^n, \overline{a}^n] & \text{for odd n} \\ [\min\{\underline{a}^n, \overline{a}^n\}, \max\{\underline{a}^n, \overline{a}^n\}] & \text{for even n and } 0 \notin [\underline{a}, \overline{a}] \\ [0, \max\{\underline{a}^n, \overline{a}^n\}] & \text{for even n and } 0 \in [\underline{a}, \overline{a}] \end{cases} , \tag{4}$$

and other functions (cf., e.g., Section 2.3 of [50]).

*Dependency problem* The arithmetic defined by formulae (2) has properties very different than the arithmetic of real numbers.

Let us consider the simplest example: what is the value of $\mathbf{x} - \mathbf{x}$? According to (2), we obtain $[\underline{x} - \overline{x}, \overline{x} - \underline{x}]$, which is not necessarily zero; it would be zero only for the degenerate case of $\underline{x} = \overline{x}$.

Also the distributivity of multiplication with respect to addition is not directly fulfilled for intervals. Instead, we have the so-called *subdistributivity* principle:

$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) \subseteq \mathbf{ab} + \mathbf{ac}.$$

In general, formulae that are equivalent for real numbers do not have to be (and usually are not) equivalent in the space of intervals; whenever the same quantity is encountered in an expression more than once, the result is likely to be overestimated. This property is often called the *dependency problem*, and it will be also significant for ADHC, as we shall see (in Section 3.3).

### 2.2 Algorithmic differentiation

Algorithmic differentiation is a useful alternative to using finite differences or symbolic methods for computing derivatives of the function's implementation. As already stated, its essence is to enhance the procedure computing a function, so that it computes its derivative(s), as well.

In his classical book [31], Griewank states that AD has been "rediscovered and implemented many times, yet its application still has not reached the full potential".

AD is based on the *chain rule*:

$$\frac{\partial}{\partial x} f\big(g(x)\big) = \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial x}\,, \tag{5}$$

where $w = g(x)$.

There are several approaches to AD, and many libraries are available. A nice survey can be found in [34]; also Chapter 3 of [50], Chapter 9 of [35], or Appendix D of [27] list several methods.

The most notable distinction is whether accumulation in Formula (5) is performed forwards or backwards. Both versions — the forward and backward mode of AD — have their applications, but the reverse mode (usually based on using so-called Wengert tapes — see, e.g., [50]) has a lower performance bound for functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$, where $m$ is significantly smaller than $n$ [31]. However, it is also much more difficult to implement, and hence less frequently used.

As we shall see, the ADHC library [14], on which we focus in this paper, while using the forward mode, will provide its very efficient implementation thanks, in particular, to using sparse data types. Details will be explained in Example 3.

Specifically, ADHC uses the forward mode of AD, based on operator overloading. There are objects (of type `adhc_ari`), representing expressions.

Using (5), expressions can be decomposed to "atoms'' that can be differentiated, and we can "assemble" the derivative from these "building blocks".

For instance, for basic arithmetic operations, we have the following formulae:

$$\begin{aligned}
\langle \mathbf{u}, \mathbf{u}' \rangle + \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u} + \mathbf{v},\ \mathbf{u}' + \mathbf{v}' \rangle,\\
\langle \mathbf{u}, \mathbf{u}' \rangle - \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u} - \mathbf{v},\ \mathbf{u}' - \mathbf{v}' \rangle,\\
\langle \mathbf{u}, \mathbf{u}' \rangle \cdot \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u} \cdot \mathbf{v},\ \mathbf{u} \cdot \mathbf{v}' + \mathbf{u}' \cdot \mathbf{v} \rangle,\\
\langle \mathbf{u}, \mathbf{u}' \rangle \,/\, \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u}/\mathbf{v},\ (\mathbf{u}' \cdot \mathbf{v} - \mathbf{u} \cdot \mathbf{v}')/\mathbf{v}^2 \rangle.
\end{aligned}$$

Other operations, e.g., power or transcendental functions, can be extended in an analogous manner, e.g.:

$$\langle \mathbf{u}, \mathbf{u}' \rangle^n = \big\langle \mathbf{u}^n, n\mathbf{u}^{n-1}\mathbf{u}' \big\rangle,$$

or the exponens function:

$$\exp\langle \mathbf{u}, \mathbf{u}' \rangle = \big\langle \exp(\mathbf{u}), \mathbf{u}' \cdot \exp(\mathbf{u}) \big\rangle.$$

Let us consider a simple example, to make the things more explicit.

**Example 1**

$$
\begin{aligned}
&f(x) = x^3 - 7x^2 + 2,\\
&\mathbf{x} = [-1, 2],\\
&\langle \mathbf{x}, \mathbf{x}' \rangle = \langle [-1, 2], [1, 1] \rangle,\\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \mathsf{f}\Big( \langle \mathbf{x}, \mathbf{x}' \rangle \Big),\\
&\mathsf{f}\Big( \langle \mathbf{x}, \mathbf{x}' \rangle \Big) = \langle \mathbf{x}, \mathbf{x}' \rangle^3 - 7 \cdot \langle \mathbf{x}, \mathbf{x}' \rangle^2 + \langle 2, 0 \rangle,\\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \langle [-1, 2], [1, 1] \rangle^3 - 7 \cdot \langle [-1, 2], [1, 1] \rangle^2 + \langle 2, 0 \rangle,\\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \langle [-1, 8], 3 \cdot [0, 4] \cdot [1, 1] - 7 \cdot \langle [0, 4], 2 \cdot [-1, 2] \cdot [1, 1] \rangle + \langle 2, 0 \rangle,\\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \langle [-1, 8], [0, 12] \rangle + \langle [-28, 0], [-28, 14] \rangle + \langle [2, 2], [0, 0] \rangle,\\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \langle [-27, 10], [-28, 26] \rangle.
\end{aligned}
$$

We can consider also computing higher derivatives — then the record has more fields, e.g., $\langle \mathbf{u}, \mathbf{u}', \mathbf{u}'', \mathbf{u}''' \rangle$; the formulae are analogous (they may become quite complicated for higher derivatives, though).

The differentiated function may be multivariate — then higher derivatives are represented by some containers: vectors, matrices, etc., instead of singleton values.

An earlier version of ADHC has been briefly described in [52], but the library has evolved since, and significant extensions have been added.

### 2.3 Hull-consistency

Enforcing so-called *partial consistency* (or *local consistency*) is a concept derived from constraint logic programming. To solve a system of several constraints (e.g., equations, inequalities, quantified relations), we try to filter the search domain by discarding points (or subdomains) that cannot satisfy a relaxed version of the original problem. For instance, such a partial consistency may just require that, taken individually ("locally"), the constraints are consistent [23].

For discrete finite domains, a common form of such partial consistency is the so-called *arc-consistency* (AC). In case of finite sets, the domain of each variable can be represented as the set of all possible values. AC demands that values inconsistent with any of the relations between variables ("arcs") are removed from these sets.

Let us consider a simple example: we have two integer-valued variables: $i$ and $j$, both with initial domains $\{1, 2, \ldots, 10\}$. There is a constraint $c(i, j)$ on these variables: $i - 2j - 1 = 0$.

It is easy to verify that, for instance, the value $i = 1$ is impossible: there is no corresponding value of $j$ to satisfy $c$; similarly the value $j = 10$ is inconsistent, as well.

Precisely, only the following domain would result from applying AC to this constraint: $i \in \{3, 5, 7, 9\}$, $j \in \{1, 2, 3, 4\}$.

For continuous domains, the admissible sets cannot be enumerated; we can only store intervals, their unions, or other (more cumbersome to represent) sets of real

numbers. Hence, AC is usually not directly applicable to continuous domains; a further relaxation is required. Benhamou et al. [19] propose a few such notions: *interval-consistency*, *box-consistency*, and *hull-consistency*.

Hull-consistency (also known under the name of *2B-consistency*) has been used in several interval programs over the years (cf. [18, 19, 23, 29, 30, 49]). It can be defined as follows.

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

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

Following [41], the symbol "$\square$" denotes the interval hull.

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

As a simple example, let us consider an equation $x_1 + x_2 - 4 = 0$. By obvious symbolic transformations, we obtain formulae for both variables that can be used to obtain their consistent domains:

$$\mathbf{x}_1 = 4 - \mathbf{x}_2 \ \textit{and} \\ \mathbf{x}_2 = 4 - \mathbf{x}_1.$$

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:

$$\mathbf{x}_1 = \ \mathbf{x}_1 \cap (4 - \mathbf{x}_2) = [-4, 2] \cap [0, 6] = [0, 2], \\ \mathbf{x}_2 = \ \mathbf{x}_2 \cap (4 - \mathbf{x}_1) = [-2, 4] \cap [2, 8] = [2, 4].$$

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

However, for a more sophisticated constraint, obtaining a consistent box is not as straightforward. Let us consider the constraint (Fig. 1):

$$x_1^2 + \exp(x_1) - x_2^3 = 0. \tag{6}$$

Again, by relatively simple symbolic transformations we can extract $x_2$ from equation (6), but not $x_1$. The solution is to decompose such an equation into primitive ones,

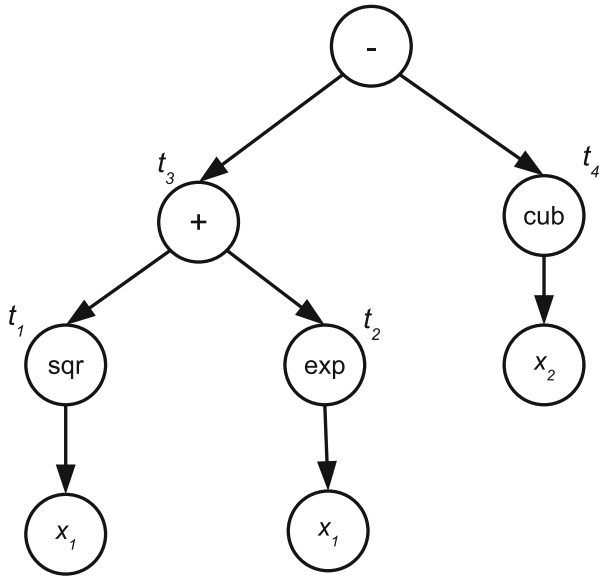


**Fig. 1** Expression tree of constraint (6)

by adding additional variables and apply HC to such a decomposed system. For the constraint (6), we could obtain:

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

The algorithm HC4 [18] (cf. also [30]) 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.

Details of the approach used in ADHC will be described in Section 3 (cf. also [49]).

### 2.4 Template meta-programming

Template meta-programming (see, e.g., [16, 26, 59]) is a powerful C++ technique (also present in a few other programming languages — interesting, but rarely encountered, like D, Curl, or XL). It is worth noting that in Java and C# we have a concept similar to C++ templates — the so-called generic classes — but its capabilities are way more limited than their C++ counterparts.

What is the template meta-programming, actually? Its essence is processing information at compile-time, instead of runtime. What kind of information can be processed in this manner? Actually, almost arbitrary information, as template meta-programming is proven to be Turing-complete. Thanks to moving some computations from runtime to compile-time, the actual program becomes more efficient.

The C++ templates are, to some extent, "classes" of classes (or functions), parameterized either by a type or by an enumerable value. The information is processed at compile-time.

The most common application of C++ meta-programming is to process the type-related data. In the simplest case, templates behave identically for various datatypes; more sophisticated templates adapt their behavior to the value of template parameters. The classical book [17] describes several examples and techniques.

## 3 The ADHC library

Now, let us describe the library itself.

### 3.1 What is the ADHC library?

Since 2016, the author has been providing a novel algorithmic differentiation library, based on C++ templates. The package is named ADHC, which stands for Algorithmic Differentiation and Hull-Consistency enforcing [14]. Just lately (in July 2021), the version 2.0 of this library has been released, featuring a few important innovations.

#### 3.1.1 Basic features of ADHC

Virtues of template meta-programming allowed obtaining several useful features of the ADHC library. This includes efficiency and versatility. The same source code can be used to generate distinct procedures for computing function values, gradients, Hesse matrices and — potentially — higher derivatives. There is no runtime penalty for their generation (obviously, there will be one for computing the gradient or Hesse matrix).

Also, 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 efficient and relatively easy to use implementations of sparse vectors and matrices (cf. [43]) that can directly be used in ADHC.

Proper types are generated, using the so-called typelist (see, e.g., [17]). They are specializations of the following template:

```
template<int level,
         sparsity_t sparse_mode,
         int num_vars,
         typename T = cxsc::interval>
struct adhc_ari {
    // ...
};
```

The four template parameters are:

- `level` — information on what should be computed; number of computed derivatives (for nonnegative values) or construction of the syntactic tree (for value -1),
- `sparse_mode` — should sparse or dense matrix/vector representation be used,
- `num_vars` — the number of variables,
- `T` — the basic type of represented "numbers": `cxsc::real`, `cxsc::interval`, `cxsc::l_interval`, etc.

Please note, `num_vars` is the template parameter of class, so expressions using potentially different numbers of variables are of inherently different types. Hence, naïvely performing an operation on such incompatible objects will be detected at compile-time, already. For instance, the following code:

```
adhc::adhc_ari<2, sparse, 2, cxsc::interval> x;
adhc::adhc_ari<2, sparse, 3, cxsc::interval> y;
//...
z = x + y;
```

will not compile. In contrast to that, the old AD code from C-XSC library had to use a dedicated function `TestSize()` while performing virtually any arithmetic operation, which resulted in a certain overhead at runtime.

The type of the second template parameter, `sparse_mode`, is an instance of enumerable class type `sparsity_t` and can have the following values:

```
enum class sparsity_t {dense, sparse, highly_sparse,
another_sparse};
```

Their meaning is as follows:

- `dense` — both, the gradient and Hesse matrix are represented using dense datatypes,
- `sparse` — the gradient is represented as a dense vector, and the Hesse matrix as a sparse matrix,
- `highly_sparse` — both, the gradient and Hesse matrix are represented using sparse datatypes,
- `another_sparse` — the gradient is represented as a sparse vector, and the Hesse matrix as a dense matrix,

As it has already been stated, the C-XSC library contains usefvul classes for sparse matricevs and vectors (cf. [43]): `cxsc::srvector`, `cxsc::scvector`, `cxsc::sivector`, `cxsc::scivector`, `cxsc::srmatrix`, `cxsc::scmatrix`, `cxsc::simatrix`, `cxsc::scimatrix`. More about them will be described in Section 3.2.3.

It is worth noting that in virtually all experiments, the best sparsity mode turned out to be the `highly_sparse` one, i.e., both the gradient and the Hesse matrix are represented by sparse types. The author had even considered removing this template parameter, but decided against it, to keep the backward compatibility of the package. Also, the other sparsity modes can (at least in theory) turn out to outperform

highly_sparse, for some applications. But the author's recommendation at the moment is to always use the "highly sparse" mode.

Let us illustrate using ADHC, by presenting the code to compute the function from Example 3:

```
const int lev = 2; // we compute the function value,
                   // the gradient and the Hesse matrix
const int n = 1;

adhc_ari<lev, sparse_mode, n, T>
f(const adhc_ari<lev, sparse_mode, n, T> &x) {
    adhc_ari<lev, sparse_mode, n, T> result;
    result = power(x, 3) - 7.0*sqr(x) + 2.0;
    return result;
}
```

### 3.1.2 Using non-smooth functions

ADHC is the package for *differentiation*, so it might seem obvious that it should handle smooth functions. As it might sound peculiar, many non-smooth functions can be handled as well, by algorithmic differentiation.

Non-smooth functions do not have gradients in strict sense, but they do have so-called *subgradients*. The set of all possible subgradients at point $x$ is called the *subdifferential* of the function at $x$. We shall not give precise definitions of these notions here, referring the interested reader to [22].

What is important to us is that subdifferentials can be bounded using the interval calculus (cf. [57]). Needless to say that non-interval packages, like ADOL-C [6] or Adept [5], even if they have (like these two libraries) a possibility to differentiate non-smooth functions, they cannot bound the subdifferential in the points of non-differentiability. On the other hand, popular interval packages for AD (like the codes in the old C-XSC [3] or PROFIL/BIAS [1]) do not take non-smooth functions into account (an exception is the IBEX library [8], but it does not allow computation of the Hesse matrix, only of the gradient).

ADHC attempts to fill this gap, which seems particularly useful, as the interval algorithms require no (or very minor) changes to process non-smooth functions. Succinctly, they are agnostic to whether the bounded quantity is the "proper" gradient or a subdifferential.

In the current version of ADHC, only one non-smooth function is defined: the max function. It seems the most prominent, and most important of all the non-smooth functions; it is also used in some neural networks, as we shall see in Section 5. Functions like min or abs can be implemented using the max function; specific implementations are likely to be added in future versions of ADHC.

**Example 2** Consider the function $f(x) = \max(0, x - 1)$.

What is its subdifferential? Clearly, it is:

$$\frac{Df}{Dx} = \begin{cases} 0, & \text{for } x < 1, \\ 1, & \text{for } x > 1, \\ [0,1], & \text{for } x = 1. \end{cases}$$

Now, let us present formulae for the inclusion functions of max() and its subdifferential. Obviously, the inclusion of maximum of two functions cannot be smaller than any of these functions, so we get:

$$\max\left(\mathsf{f}_1(\mathbf{x}), \mathsf{f}_2(\mathbf{x})\right) = [\max(\underline{f}_1(\mathbf{x}), \underline{f}_2(\mathbf{x})), \max(\overline{f}_1(\mathbf{x}), \overline{f}_2(\mathbf{x}))]. \tag{7}$$

And the "derivative" (i.e., subdifferential) is:

$$\frac{D\max\left(\mathsf{f}_1(\mathbf{x}), \mathsf{f}_2(\mathbf{x})\right)}{D\mathbf{x}} = \begin{cases} \frac{D\mathsf{f}_1(\mathbf{x})}{D\mathbf{x}}, & \text{for } \underline{f}_1(\mathbf{x}) > \overline{f}_2(\mathbf{x}), \\ \frac{D\mathsf{f}_2(\mathbf{x})}{D\mathbf{x}}, & \text{for } \overline{f}_1(\mathbf{x}) < \underline{f}_2(\mathbf{x}), \\ \square\left(\frac{D\mathsf{f}_1(\mathbf{x})}{D\mathbf{x}} \cup \frac{D\mathsf{f}_2(\mathbf{x})}{D\mathbf{x}}\right), & \text{otherwise.} \end{cases} \tag{8}$$

What about the "second derivative"? Surprisingly, this notion makes a perfect sense! As the author has already proposed in [45], we can consider the functions not in the space of "proper" functions, but "generalized functions", also known as *distributions*. The most well-known example of such a generalized function is the Dirac's delta function:

$$\delta(x) = \begin{cases} +\infty, & \text{for } x = 0, \\ 0, & \text{otherwise} \end{cases}$$

To be precise, also the condition $\int_{-\infty}^{+\infty} \delta(x)dx = 1$ should be fulfilled, but in our considerations, it will not be used.

The interval extension of the "function" $\delta(x)$ is easy to obtain:

$$\delta(\mathbf{x}) = \begin{cases} [0, +\infty], & \text{for } 0 \in \mathbf{x}, \\ [0, 0], & \text{otherwise} \end{cases} \tag{9}$$

And now, the "second derivative" of function $f$ from Example 2 is: $\frac{D^2 f}{Dx^2} = \delta(x-1)$, with the obvious interval inclusion function. These kinds of derivatives are also called *weak derivatives*.

Details are beyond the scope of this paper; a separate one is planned on this — interesting and important — topic. Fortunately, for solving equations systems, the first derivative is crucial (as it is used in the Newton operator [35, 38, 50]), and the delta function does not appear there.

*Comment* It is worth noting that a similar approach has been proposed by Kearfott [39, 40]. However, in his papers no relationship to weak derivatives calculus was realized. Consequently, the obtained formulae were less elegant, and their justification was weaker.

It will be very interesting to compare using *weak derivatives* to using so-called *slopes* (see, e.g., [33, 37, 38]), in terms of the efficiency. In the aforementioned paper

of Kearfott [37], such a comparison has even been suggested, but no numerical results seem to be available.

It should be noted that, in the present version of ADHC, the max() function can be used over interval-valued domains only. Using it over, e.g., the `cxsc::real` datatype would require several changes: we would need to choose a single subgradient, instead of bounding the whole subdifferential. This is another extension that may be added to ADHC in future versions; however the focus of the author is on interval-related applications.

### 3.1.3 Constructing the expression tree

What would happen, if `lev = -1` was used in the above program? Then, instead of computing the function's or its derivatives' values, the code would create a dynamically linked tree data structure, representing the expression.

Such expression trees are represented using the type `adhc_node`. This kind of objects stores the interval of possible values of the expression, and a union with the data, specific to the kind of expression: a constant, a variable, a dyadic operation ($+$, $-$, $\times$, $\div$, max, etc.), or an unary function. The specialization of the template class `adhc_ari`, for the first template parameter equal to -1, inherits from the class `adhc_node`. Specializations for nonnegative values of this parameter, obviously do not.

How to enforce HC on the tree nodes? As already indicated in Section 2.3, we have to traverse the tree forwards, and then backwards. Firstly, we compute the intervals of possible values, for each of the nodes. In the second step, the domains are narrowed.

For instance, when we have a sum of two expressions: $t_3 = t_1 + t_2$, we can narrow the domains of variables $t_1$ and $t_2$, using the obvious expressions:

$$\begin{aligned} \mathbf{t}_1 &\longleftarrow \mathbf{t}_1 \cap (\mathbf{t}_3 - \mathbf{t}_2), \\ \mathbf{t}_2 &\longleftarrow \mathbf{t}_2 \cap (\mathbf{t}_3 - \mathbf{t}_1). \end{aligned}$$

Enforcing HC on arguments of transcendental functions can be more tedious; it is trivial only for monotonic functions. But let us consider the square function: $t_2 = t_1^2$. What are the feasible points of $\mathbf{t}_1$? Please note, they can belong to one of the two intervals:

$$\mathbf{t}_1 \cap [\sqrt{\underline{t}_2}, \sqrt{\overline{t}_2}] \; or \; \mathbf{t}_1 \cap [-\sqrt{\overline{t}_2}, -\sqrt{\underline{t}_2}]. \tag{10}$$

Using the formula:

$$\mathbf{t}_1 \leftarrow \mathbf{t}_1 \cap \square\Big([-\sqrt{\overline{t}_2}, -\sqrt{\underline{t}_2}] \cup [\sqrt{\underline{t}_2}, \sqrt{\overline{t}_2}]\Big),$$

would not be efficient, as it may severely overestimate some domains (if only one of the intervals (10) coincides with the domain). A better alternative (used in the current version of ADHC) is:

$$\mathbf{t}_1 \leftarrow \square\Big(\big(\mathbf{t}_1 \cap [-\sqrt{\overline{t}_2}, -\sqrt{\underline{t}_2}]\big) \cup \big(\mathbf{t}_1 \cap [\sqrt{\underline{t}_2}, \sqrt{\overline{t}_2}]\big)\Big), \tag{11}$$

but even this formula can overestimate the result. Assume, for instance, we have the domain $\mathbf{t}_1 = [-4, 4]$ and $\mathbf{t}_2 = [9, 16]$.

The box $\mathbf{t}_1 \times \mathbf{t}_2$ is hull-consistent, and cannot be reduced any further, as both endpoints of $\mathbf{t}_1$: $-4$ and $+4$ are feasible. Nevertheless, not all values from the domain $\mathbf{t}_1 = [-4, 4]$ are feasible; actually, these are only values from $[-4, -3] \cup [3, 4]$. But to obtain such narrowing, we would need to use interval-consistency [19], instead of hull-consistency.

Using the interval-consistency is another possible addition for future versions of ADHC. It may be useful not only for the square functions, but also for many other non-monotonic functions, including trigonometric ones. The current version of ADHC implements a similar formula, to the given above, for functions sin and cos.

### 3.2 Computations over various types

The most significant innovation in ADHC 2.0 is the fourth template parameter $T$, not present in earlier versions of the library. This parameter allows us to generate functions operating on various argument types: interval-valued or pointwise, real-valued or complex, and having various precision.

#### 3.2.1 Pointwise types vs interval types

It is a well-known fact that interval arithmetic operations are significantly more expensive than pointwise ones. Addition and subtraction require only twice more arithmetic operations than pointwise ones, but multiplication, division, and other operations are even more expensive — cf. the formulae in (2). What is more, all of these operations require changing the rounding mode, which can also be costly.

Needless to say that whenever we do not need either to bound the function on an area, or to obtain a verified result, replacing interval operations with pointwise floating-point ones can be very worthwhile.

An important application, when we may need it, is hybrid algorithms, combining interval and non-interval approaches. For instance, in [21], a global optimization algorithm is presented, where we first obtain an approximation of the global minimum, using non-verified methods (and we can use fast, highly efficient today solvers for this purpose), and then, using constraint propagation, suboptimal areas are removed in the branch-and-bound-type process.

An analogous approach can be applied to seeking Pareto sets of a multicriteria problem (cf. Chapter 6 of [50]), as many non-interval algorithms for covering Pareto sets (SPEA, SPEA2, NSGA, NSGA-II, etc.; cf., e.g., [24, 61] and the references therein) have been developed.

#### 3.2.2 Higher precision types

The C++ language standards define three floating-point types:

- single-precision numbers (`float`), i.e., 32-bits numbers,
- double-precision numbers (`double`): 64-bits numbers — most commonly used,

– extended-precision numbers (`long double`): they are usually 80-bits numbers, but some compilers might have another size of these variables (the C++ standards do not give the precise size of the `long double` type).

Lately, some devices (mostly GPUs) started implementing *half-precision numbers*, as well. These 16-bits floating-point numbers allow even faster computations, and it turned out that they are quite sufficient in many important applications [32]. Nowadays, they are only implemented in CUDA for Nvidia GPUs, but some libraries allow emulating them on CPU, as well.

And what types do we have in C-XSC? There are no single-precision types, yet we get very interesting multiple-precision types, based on various kinds of the so-called staggered-precision arithmetic [20, 42, 43, 55].

The high-precision types we use are: `cxsc::l_real`, `cxsc::l_complex`, `cxsc::l_interval`, and `cxsc::l_cinterval`. All of them have corresponding vector and matrix types (although, unfortunately, only dense ones).

The new types `cxsc::lx_real`, `cxsc::lx_complex`, `cxsc::lx_in`, `terval`, and `cxsc::lx_cinterval`, representing the *extended staggered types*, with the "extremely wide exponent range" [20] do not have corresponding matrix types. Such types are one of the planned additions to subsequent versions of the survive-CXSC library [12]. That would allow using them in ADHC, as well.

### 3.2.3 Using sparse data types

Since version 2.4.0, the C-XSC library has been providing us with the eight aforementioned sparse data types — four for real numbers: `cxsc::srvector`, `cxsc::srmatrix`, `cxsc::sivector`, `cxsc::simatrix` (cf. [43, 60]), and four analogous for complex ones. They represent sparse pointwise vector, pointwise matrix, interval vector, and interval matrix, respectively.

Sparse vectors are represented as a pair of `std::vector` objects: one storing values and the other one — indices of non-zero elements. Sparse matrices are stored in a *compressed column storage* format (CCS). Details can be found, e.g., in [60].

Unfortunately, there are no sparse representations for the multiple-precision vectors and matrices. Dense representations (`cxsc::l_ivector`, `cxsc::l_imatrix`, etc.) have to be used, instead. Hopefully, during the future development of survive-CXSC, such types will get implemented.

But why is using sparse datatypes so worthwhile? At the first glance, it might seem that their use will be beneficial for sparse problems only, but it is not so. Let us explain it on a specific example.

**Example 3** Suppose, we intend to compute the gradient of:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^2. \tag{12}$$

Obviously, neither the function nor its gradient are sparse.

But consider, how the derivatives are going to be computed. By augmenting the (overloaded) operations + and `sqr()` with the expression's gradient evaluation, we obtain:

$$\nabla f(x_1, \ldots, x_n) = \nabla\left(x_1^2 + x_2^2 + \cdots + x_n^2\right) =$$
$$= 2 \cdot x_1 \cdot \nabla x_1 + 2 \cdot x_2 \cdot \nabla x_2 + \cdots + 2 \cdot x_n \cdot \nabla x_n,$$

which results in:

$$\nabla f(x_1, \ldots, x_n) = 2 \cdot x_1 \cdot \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + 2 \cdot x_2 \cdot \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \cdots + 2 \cdot x_n \cdot \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Consequently, it is necessary to multiply $n$ vectors by scalars and then perform $n-1$ vector additions. For a dense representation, it results in $n^2$ interval multiplications and $n \cdot (n-1)$ interval additions.

But if vectors are represented in a sparse manner, the complexity is quite different! For each of the vectors, only a single component has a non-zero value, which decreases the number of interval multiplications from $n^2$ to $n$. As for the additions, each vector has its non-zero component at a different index. So: virtually no additions are needed, only some non-floating-point (and hence cheap) index checking operations.

This is a more than very significant reduction of the computational effort. It is worth noting that this problem may be considered better suited for the reverse-mode AD. Using sparse data types allowed us to reduce the difference. And let us remind once more that interval operations require several floating-point ones, including switching of the rounding mode (cf., e.g., [38]).

To sum up: contrary to his initial assumptions, the author observed that it is almost always worthwhile to use sparse representations — especially for gradients.

For the sake of completeness, let us present the C++ function, based on ADHC, computing the function from Example 3:

```
template<int lev, sparsity_t sparse_mode, int n,
 class T>
adhc_ari<lev, sparse_mode, n, T>
f_example(const adhc_vector<lev, sparse_mode, n, T> &x)
{
    adhc_ari<lev, sparse_mode, n, T> result;
    result = sqr(x[1]);
    for (int i = 2; i <= n; ++i) result += sqr(x[i]);
    return result;
}
```

Ranging in the vector indices from 1 to $n$, instead of more typical in C++ from 0 to $n-1$, is inherited from the C-XSC library. The use of the above function, might look, e.g., like the following:

```
//...
const int N = 8;
cxsc::ivector x(N);
//...
adhc_vector<1, SPARSITY, N, T> x_(x);
adhc_ari<1, SPARSITY, N, T> y_ = f_example(x_);
std::cout << y_ << "\n";
//...
```

More code snippets are given in the documentation of the ADHC [14] (see file `doc/manual.tex`).

### 3.3 Intersection of two expressions

It has already been stated that interval expressions tend to generate overestimated values due to the *dependency problem*. If we are able to provide a formula, where all variables and parameters occur only once, we can bound the results precisely (up to the numerical precision). Yet, in many situations, there is no such possibility,

**Example 4** Consider the function:

$$f(x, y) = x + x \cdot y + y \,. \tag{13}$$

Both variables: $x$ and $y$ occur twice in it. We can provide interval extensions, like: $\mathsf{f}_1(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot (1 + \mathbf{y}) + \mathbf{y}$, or $\mathsf{f}_2(\mathbf{x}, \mathbf{y}) = \mathbf{x} + (\mathbf{x} + 1) \cdot \mathbf{y}$, but in both cases one of the variables occurs twice in the expression. There is no simple way to overcome the dependency problem, in this case.

Yet, while it cannot be fully overcome, it can still be mitigated. We can compute the *intersection* of inclusion functions $\mathsf{f}_1$ and $\mathsf{f}_2$.

For instance, for $\mathbf{x} = [-1, 2]$, $\mathbf{y} = [-5, -1]$, we obtain:

$$\begin{aligned} \mathsf{f}_1(\mathbf{x}, \mathbf{y}) &= [-1, 2] \cdot (1 + [-5, -1]) + [-5, -1] = [-13, 3], \\ \mathsf{f}_2(\mathbf{x}, \mathbf{y}) &= [-1, 2] + ([-1, 2] + 1) \cdot [-5, -1] = [-16, 2], \\ \mathsf{f}_1(\mathbf{x}, \mathbf{y}) \cap \mathsf{f}_2(\mathbf{x}, \mathbf{y}) &= [-13, 3] \cap [-16, 2] = [-13, 2], \end{aligned}$$

which is narrower than both "partial" results!

The ADHC library, provides the function `intersection()` that allows to provide more than one expression for the same quantity. During the computational process, the computed interval for the expression value (and, of course, for all its available derivatives) will be intersected. Also, the procedure enforcing HC, narrows the terms in all "branches" of the intersection.

To the best knowledge of the author, no other libraries provide such a feature.

## 4 Comparison of ADHC to its main competitors

As indicated in Section 1, it is difficult to compare ADHC to other libraries, as its features are pretty unique. Nevertheless, let us compare the efficiency of differentiation, with respect to major interval libraries.[1]

### 4.1 Competitors

*C-XSC* This was the "starting point" of the author's consideration: the code provided by the C-XSC library, in the file `hess_ari.cpp` [3]. The forward AD mode is implemented there, using operator overloading. The decision whether to compute the function value only, the gradient or the Hesse matrix is done at runtime, using a static variable `HessOrder`.

In its original version, the code is not thread-safe, but it can be fixed relatively easily (cf. Section 6 of [54]).

*PROFIL/BIAS* It is another interval library, or more precisely, a pair of libraries: PROFIL (Programmer's Runtime Optimized Fast Interval Library), and its underlying BIAS (Basic Interval Arithmetic Subroutines). The page [1] provides the tarball to be downloaded, as well, as the documentation in Gzipped PostScript. PROFIL/BIAS has been pretty popular, at least at some time (it has not been updated since 2009). The book [35] describes some code of this package. AD is one of its features.

The author was able to find little information about AD in the PROFIL library, but the code suggests, the forward mode is used there. It seems, always both the gradient and the Hesse matrix get computed, even if we do not need them both. The aforementioned documentation at [1] states that the variable `INTERVAL_AUTODIFF::ComputeHessian` should allow to change it, but the author was not able to obtain such a behavior. Also examples, attached to the library, do not use such switching: the Hesse matrix gets computed always.

*IBEX* The name stands for Interval-Based Explorer [8]. This is the most modern of the considered pieces of software. IBEX is feature-rich: it contains not only basic interval functions, but also, i.a., several contractor operators (using various consistency enforcing methods, and linear programming), and two stand-alone solvers for constraint systems and global optimization.

IBEX does not have its own implementation of the interval type or its arithmetic. Instead, it can use one of the three interval libraries as its base: GAOL [4] (which is the default one), the aforementioned BIAS [1], or Filib++ [2]. All of them are considered in the presented comparison.

According to its online documentation [8], IBEX can use both AD and symbolic differentiation. Very little is written about algorithms used in both cases, yet it seems that the reverse mode is used for the AD.

It is the only of presented AD codes (except my own ADHC), which allows to bound the "gradient" of some non-smooth terms, like the max function. Yet, it allows computing first derivatives only: gradients, Jacobi matrices, or Hansen slope matrices.

[1] The codes used in this section are available at https://gitlab.com/bkubica/ad-comparison.

The author has found no way to obtain the Hesse matrix — neither using AD, nor symbolic differentiation.

### 4.2 Benchmark problems

The author has investigated three problems. For each of them we compute either the function value, the gradient, or the Hesse matrix. The computation is repeated one million times for random intervals belonging to the domain.

The first benchmark is a simple smooth function of two variables:

$$f(x_1, x_2) = x_1^2 + \sin(x_2), \quad x_1, x_2 \in [-10, 10]. \tag{14}$$

The second one is also smooth, but it can have arbitrary many variables. We considered $n = 10$:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^2 + \prod_{i=1}^{n} \cos(x_i), \quad x_i \in [-10, 10] \text{ for } i = 1, \ldots, 10. \tag{15}$$

The third function is non-smooth, as it uses the max operation:

$$\begin{aligned} &f(x_1, \ldots, x_n) = \sum_{i=1}^{n} \max(x_i, 0) + \max\Big(\sum_{i=1}^{n} x_i, \prod_{i=1}^{n} x_i\Big), \\ &x_i \in [-10, 10] \text{ for } i = 1, \ldots, 10. \end{aligned} \tag{16}$$

### 4.3 Comparison

The AD codes to be compared are as follows:

- ADHC — the presented ADHC library,
- CXSC — the AD code from the old C-XSC (`hess_ari.cpp`),
- PROFIL — the AD code from PROFIL/BIAS,
- IBEX+GAOL — the gradient evaluation for IBEX using GAOL,
- IBEX+BIAS — the gradient evaluation for IBEX using BIAS,
- IBEX+FILIB — the gradient evaluation for IBEX using Filib++.

As the IBEX library requires to use the WAF build system, based on the obsolete Python 2.7 (and incompatible with the modern Python versions), the last three codes have been built in Docker containers based on the `python:2.7` image.

For each AD code, three levels of evaluation are considered:

- 0 — function evaluations,
- 1 — gradient evaluations,
- 2 — Hesse matrix evaluations (if available).

**Table 1** Computational times (in seconds) of AD codes for function (14)

| level | ADHC | CXSC | PROFIL | IBEX+GAOL | IBEX+BIAS | IBEX+FILIB |
|---|---|---|---|---|---|---|
| 0 | 0.15 | 0.69 | 2 | 0.28 | 0.34 | 0.20 |
| 1 | 0.50 | 0.67 | 2 | 0.56 | 0.85 | 0.66 |
| 2 | 1.62 | 0.71 | 2 | —- | —- | —- |

### 4.4 Comments

The above comparison has clearly shown that the feature set of ADHC is pretty unique. It was the only of compared packages that was able to perform computations of both gradient and Hesse matrix (or their analogs for non-smooth functions) for all three functions.

Its efficiency seems also quite satisfactory. It outperformed the old C-XSC AD code and PROFIL/BIAS in virtually all cases; only for bounding the Hesse matrix of function (14) — a dense $2 \times 2$ matrix — it was marginally slower (see Table 1). For larger systems (cf. Table 2), the benefits of using sparse vector and matrix types turned out to be obvious (Table 3).

The IBEX AD code outperformed ADHC in some cases; particularly when GAOL was used as the interval library. It is worth noting that results of IBEX varied to the high extent, depending on the interval library it was using. In particular, when using BIAS, results did not really outperform ADHC. Hence, it seems that the efficiency of interval operations is crucial, and can have more impact on the efficiency of the software than further improvement of the AD code.

Also, it is worth noting, that IBEX was not able to compute Hesse matrices; this is a serious drawback with respect to ADHC.

It turns out that, thanks to using the sparse formats and its other features, ADHC, based on forward-mode AD, can compete with reverse-mode AD systems, being only marginally slower than them.

## 5 Examples and applications

In the previous section (and earlier, in [52]), ADHC has already been compared to the older AD code from C-XSC (now also to a few other libraries). In this section, let us focus on the new features of ADHC 2.0.

**Table 2** Computational times (in seconds) of AD codes for function (15)

| level | ADHC | CXSC | PROFIL | IBEX+GAOL | IBEX+BIAS | IBEX+FILIB |
|---|---|---|---|---|---|---|
| 0 | 1 | 3 | 226 | 2 | 3 | 2 |
| 1 | 11 | 15 | 219 | 5 | 12 | 8 |
| 2 | 67 | 195 | 220 | — | — | — |

**Table 3** Computational times (in seconds) of AD codes for function (16)

| level | ADHC | CXSC | PROFIL | IBEX+GAOL | IBEX+BIAS | IBEX+FILIB |
|---|---|---|---|---|---|---|
| 0 | 1 | — | — | 1 | 2 | 2 |
| 1 | 8 | — | — | 2 | 7 | 7 |
| 2 | 61 | — | — | — | — | — |

All experiments have been performed on the author's laptop computer, with AMD Ryzen 5-4600H CPU (6 cores, 12 hardware threads; 3GHz). The machine ran under control of a 64-bit Manjaro 21.07 GNU/Linux operating system with glibc 2.33 and the Linux kernel 5.10.42-1-MANJARO (with SMP and PREEMPT options).

The software was written in C++ and compiled using the GCC compiler (GCC 11.1.0). The parallelization (8 threads) was done with TBB 2020.3-1 [9]. OpenBLAS 0.3.17 [11] was linked for BLAS operations.

As for the the author's libraries, the following versions have been used:

- ADHC 2.2.2,
- survive-CXSC 2.6.1,
- HIBA_USNE 2.8.9-1.

Let us start the presentation of experiments with the multiple-precision Gauss-Seidel operator.

### 5.1 Multiple-precision Newton operator

New versions of ADHC allow us to use the same expression to generate functions computing the expression (and its derivatives) for various datatypes. Thanks to this, the author was able to incorporate to the HIBA_USNE solver (in the version 2.8) a code generating multiple-precision version of the equations system under consideration.

Also, a multiple-precision version of the Newton operator has been implemented. It is based on the staggered-precision type `cxsc::l_interval`; `cxsc::lx_interval` cannot be used yet, because of the reasons described in Section 3.2.2.

Unlike the version for the double-precision `cxsc::interval` type, the multiple-precision Newton cannot handle components of the Jacobi matrix that contain zeros. The reason is that the extended division (3) is not implemented for multiple-precision types. This is another feature that will hopefully be added in future versions of survive-CXSC.

To show the impact of the new operator, we shall solve a few benchmark equations systems, using HIBA_USNE. We shall choose such problems, where without the multiple-precision Newton, we get relatively many boxes non-verified either to contain or not to contain a solution. Hence, we shall consider five equations systems (two well-determined and three underdetermined), and we shall apply HIBA_USNE for them in two configurations: with or without the aforementioned contractor. All of the test problems have been considered in [47–49].

Let the first of them be the Puma problem — eight equations in eight variables:

$$\begin{aligned}
&x_1^2 + x_2^2 - 1 = 0,\ x_3^2 + x_4^2 - 1 = 0,\\
&x_5^2 + x_6^2 - 1 = 0,\ x_7^2 + x_8^2 - 1 = 0,\\
&0.004731x_1x_3 - 0.3578x_2x_3 - 0.1238x_1 - 0.001637x_2 - 0.9338x_4 + x_7 = 0,\\
&0.2238x_1x_3 + 0.7623x_2x_3 + 0.2638x_1 - 0.07745x_2 - 0.6734x_4 - 0.6022 = 0,\\
&x_6x_8 + 0.3578x_1 + 0.004731x_2 = 0,\\
&-0.7623x_1 + 0.2238x_2 + 0.3461 = 0,\\
&x_1, \ldots, x_8 \in [-1, 1].
\end{aligned} \tag{17}$$

The problem is related to the kinematics of a 3R robot, and it is often used as a benchmark for nonlinear systems solvers. Accuracy $\varepsilon = 10^{-6}$ was set.

The second problem is the Brent problem — it is a well-determined algebraic problem, supposed to be "difficult" (cf. [47]):

$$\begin{aligned}
&3x_1 \cdot (x_2 - 2x_1) + \frac{x_2^2}{4} = 0,\\
&3x_i \cdot (x_{i+1} - 2x_i + x_{i-1}) + \frac{(x_{i+1} - x_{i-1})^2}{4} = 0, \quad i = 2, \ldots, N-1,\\
&3x_N \cdot (20 - 2x_N + x_{N-1}) + \frac{(20 - x_{N-1})^2}{4} = 0,\\
&x_i \in [-10^8, 10^8], \quad i = 1, \ldots, N.
\end{aligned} \tag{18}$$

Presented results have been obtained for $N = 10$; accuracy was set to $\varepsilon = 10^{-7}$.

The third problem, first of the underdetermined ones is the Hippopede problem — two equations in three variables:

$$\begin{aligned}
&x_1^2 + x_2^2 - x_3 = 0,\\
&x_2^2 + x_3^2 - 1.1x_3 = 0.\\
&x_1 \in [-1.5, 1.5],\ x_2 \in [-1, 1],\ x_3 \in [0, 4].
\end{aligned} \tag{19}$$

Accuracy $\varepsilon = 10^{-7}$ was set.

The fourth and fifth problems are (as was the Puma problem) related to robotics, specifically the inverse kinematics of a planar $nR$ manipulator with 5 joints. We shall use two formulations of this problem: the one based on trigonometric functions, used in [47] and [48], and the algebraic formulation, like in Chapter 9 of [50].

Hence, the fourth problem is the system of the following three equations in $N$ variables:

$$\begin{aligned}
&\sum_{i=1}^{N} l_i \cdot \prod_{j=1}^{i} \cos\Big(\sum_{k=1}^{j} x_k\Big) - 1 = 0,\\
&\sum_{i=1}^{N} l_i \cdot \prod_{j=1}^{i} \sin\Big(\sum_{k=1}^{j} x_k\Big) - 1 = 0,\\
&\sum_{i=1}^{N} x_i - \frac{\pi}{2} = 0\,,\\
&x_i \in \big[-\frac{\pi}{2}, \frac{\pi}{2}\big], \quad i = 1, \ldots, N.
\end{aligned} \tag{20}$$

Accuracy $\varepsilon = 0.02$ was used.

And the fifth problem is:

$$\begin{aligned}
&\sum_{i=1}^{n-1} l_i \cdot c_i - 1 = 0,\\
&\sum_{i=1}^{n-1} l_i \cdot s_i + l_n - 1 = 0,\\
&s_i^2 + c_i^2 - 1 = 0, \quad i = 1, \ldots, n-1,\\
&s_i, c_i \in [-1, 1], \quad i = 1, \ldots, n-1.
\end{aligned} \tag{21}$$

In both cases, all $l_i$'s are equal to 1. For $n = 5$ joints, we get $n + 1 = 6$ equations in $2 \cdot n - 2 = 8$ variables.

The accuracy is set to $\varepsilon = \frac{1}{64} = 0.015625$.

Results are given in Tables 4 and 5.

The following notation is used in all of the tables:

- fun.evals, grad.evals, Hesse evals — numbers of functions evaluations, functions' gradients and Hesse matrices evaluations (in the interval algorithmic differentiation arithmetic),
- bisecs — the number of boxes bisections,
- preconds — the number of preconditioning matrix computations (i.e., performed Gauss-Seidel steps),
- bis.Newt, del.Newt — numbers of boxes bisected/deleted by the Newton step,
- high.Newt. — the number of performed multiple-precision Gauss-Seidel steps, i.e., the ones using the staggered arithmetic of the `cxsc:l_interval` datatype,
- ver.high.Newt, del.high.Newt — numbers of boxes verified/deleted by the aforementioned multiple-precision Newton step,
- Sobol excl. — the number of boxes to be excluded generated by the initial exclusion phase,

**Table 4** Computational results for well-determined problems

| problem | no multiple-prec | | multiple-prec. GS | |
|---|---|---|---|---|
| | (17) | (18) | (17) | (18) |
| fun. evals | 10,719 | 9,372,689 | 10,734 | 9,576,346 |
| grad.evals | 10,013 | 5,997,647 | 10,009 | 6,104,334 |
| Hesse evals | 40 | 260,376 | 40 | 260,997 |
| bisections | 45 | 48,650 | 45 | 48,902 |
| preconds | 93 | 76,191 | 119 | 77,161 |
| bis.Newt. | 12 | 11,041 | 12 | 11,127 |
| del.Newt. | 6 | 14,664 | 6 | 14,897 |
| high.Newt. | — | — | 26 | 413 |
| ver.high.Newt. | — | — | 0 | 0 |
| del.high.Newt. | — | — | 14 | 179 |
| Sobol excl. | 64 | 100 | 64 | 100 |
| Sobol resul. | 1,045 | 1,276 | 1,045 | 1,107 |
| pos.boxes | 26 | 413 | 12 | 234 |
| verif.boxes | 4 | 815 | 4 | 815 |
| Leb.poss. | 2e-100 | 1e-81 | 1e-127 | 9e-91 |
| Leb.verif. | 2e-105 | 2e-78 | 3e-103 | 7e-47 |
| time (sec.) | $< 1$ | 6 | $< 1$ | 6 |

- Sobol resul. — the number of boxes resulting from the exclusion phase (cf. [46], [47]),
- 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.

*Comments* It turns out that the multiple-precision version of the Newton operator is costly (which is not surprising), but it allows to discard some of the boxes. In no case, it was able to *verify* any new box: normal floating-point precision seems sufficient for this purpose. For boxes that cannot be verified using the Newton operator, we have yet other verification tools (based, e.g., on the theorem of Karol Borsuk or computing the topological degree; cf., e.g., [25, 36, 53] or the survey in Section 5.4 of [50]).

Is it worthwhile to use the staggered-precision Newton? For the Puma problem, 14 out of 26 of the possible boxes have been discarded. For the Brent problem, the improvement is even more significant (discarding 170 out of 420 boxes), but for the underdetermined problems, the result is less impressive. While for the $nR$ problem in version (20), the additional test, while time-consuming, at least had discarded over 100 thousands of boxes (out of 1.8 millions), for problems (19) and (21), the results were discarding 1.5 thousand boxes out of over 170 thousands, or less than 5 thousands out of 1.7 millions, respectively. Provided the increase of the computation time, this would rarely be worthwhile.

**Table 5** Computational results for underdetermined problems

| problem | no multiple-prec | | | multiple-prec. GS | | |
|---|---|---|---|---|---|---|
| | (19) | (20) | (21) | (19) | (20) | (21) |
| fun. evals | 1,323,947 | 174,162,828 | 29,921,974 | 1,323,947 | 174,162,828 | 29,921,974 |
| grad.evals | 1,520,372 | 47,644,353 | 32,624,082 | 1,520,372 | 47,644,353 | 32,624,082 |
| Hesse evals | 498 | 334,994 | 18,372 | 498 | 334,994 | 18,372 |
| bisections | 369,359 | 5,472,732 | 2,715,977 | 369,359 | 5,472,732 | 2,715,977 |
| preconds | 659,902 | 10,088,496 | 4,967,088 | 830,314 | 11,956,950 | 6,689,002 |
| bis.Newt. | 45 | 7,484 | 38 | 45 | 7,484 | 38 |
| del.Newt. | 79,285 | 2,816,973 | 532,702 | 79,285 | 2,816,973 | 532,702 |
| high.Newt. | — | — | — | 170,412 | 1,868,454 | 1,721,914 |
| ver.high.Newt. | — | — | — | 0 | 0 | 0 |
| del.high.Newt. | — | — | — | 1,568 | 116,304 | 7,771 |
| Sobol excl. | 9 | 25 | 64 | 9 | 25 | 64 |
| Sobol resul. | 68 | 473 | 1,077 | 68 | 473 | 1,077 |
| pos.boxes | 170,412 | 1,868,454 | 1,721,914 | 168,844 | 1,752,150 | 1,714,143 |
| verif.boxes | 20,494 | 3,429 | 2,699 | 20,494 | 3,429 | 2,699 |
| Leb.poss. | 8e-18 | 0.000334 | 9e-12 | 8e-18 | 0.000294 | 9e-12 |
| Leb.verif. | 0.001195 | 1e-6 | 2e-11 | 0.001195 | 1e-6 | 2e-11 |
| time (sec.) | 1 | 41 | 11 | 3 | 1186 | 158 |

Obviously, implementing the Kahan's extended division (3) for staggered-precision types, may improve these results.

### 5.2 Neural networks

Many important applications of numerical algorithms, in particular of AD, are related to machine learning, a very hot and timely research field. In particular, we can use them in investigating neural networks.

In [51], we have presented localizing all stationary points of a Hopfield-like network. In these experiments, a well-known activation function has been used: the sigmoid:

$$\sigma(t) = \frac{1}{1 + \exp(-\beta \cdot t)}. \tag{22}$$

This function is obviously smooth, as are many other activation functions: the hyperbolic tangent, the arctan or ELU (Exponential-Linear Unit) [27].

Nevertheless, nowadays, non-smooth activation functions get more and more popular. We shall check how efficient ADHC will be, for problems using such functions. Two such activation functions — ReLU (Rectified Linear Unit) and "Leaky ReLU" — are going to be considered.

Both ReLU and Leaky ReLU are significantly less expensive to compute, than ELU or arctan. Formulae for these activation functions are:

$$ReLU(t) = \max(t, 0)\ . \tag{23}$$

and:

$$LeakyReLU(t) = \max(t, \alpha \cdot t)\ , \tag{24}$$

where $\alpha$ shows the amount of "leaking". A typical value for $\alpha$ is 0.01.

Both functions can easily be implemented in ADHC, thanks to the existence of the max() function, as described in Subsubsection 3.1.2.

The problem we are solving, as in [51] is to find all stationary points of a recurrent Hopfield-like network (see Fig. 2).

It can be formulated as follows:

$$\begin{aligned} &\text{Find } x_i,\ i = 1, \ldots, n,\ \text{such that :} \\ &x_i - \sigma\Big(\sum_{j=1}^{n} w_{ij} x_j\Big) = 0,\ \text{for } i = 1, \ldots, n, \end{aligned} \tag{25}$$

where $\sigma(\cdot.)$ is the activation function.

We consider the network with $n = 8$ neurons, storing 3 vectors. The first vector to remember is $(1, 1, \ldots, 1)$. The second one consists of $\frac{n}{2}$ values $+1$ and $\frac{n}{2}$ values $-1$. The third one consists of $n - 2$ values $+1$ and 2 values $-1$.

The neurons were trained (i.e., the weights $w_{ij}$ set), using the Hebb's rule [27, 28]. The accuracy $\varepsilon = 10^{-6}$ was used, when solving (25). Results can be found in Table 6.

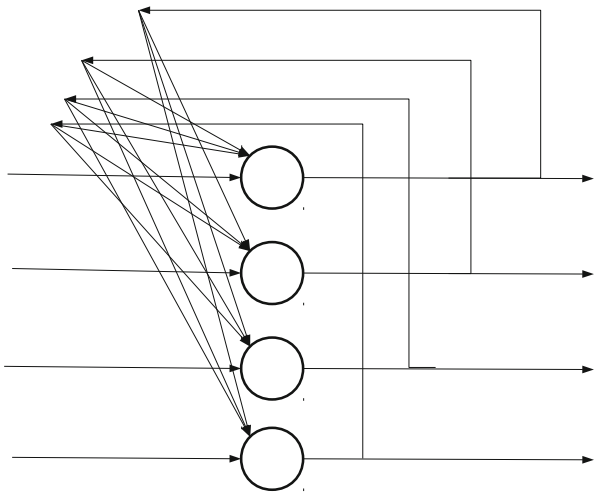

**Fig. 2** A Hopfield-type neural network

**Table 6** Computational results for Problem (25)

| activation function | no multiple-prec (22) | (23) | (24) | multiple-prec. GS (22) | (23) | (24) |
|---|---|---|---|---|---|---|
| fun. evals | 366,584 | 3,516 | 15,499 | 366,790 | 3,516 | 15,498 |
| grad.evals | 472,893 | 12,840 | 14,590 | 473,165 | 12,840 | 14,538 |
| Hesse evals | 3,608 | 16 | 8 | 3,608 | 16 | 8 |
| bisections | 28,738 | 274 | 280 | 28,755 | 274 | 280 |
| preconds | 32,898 | 282 | 288 | 32,906 | 291 | 297 |
| bis.Newt. | 0 | 0 | 0 | 0 | 0 | 0 |
| del.Newt. | 5,751 | 0 | 0 | 5,742 | 0 | 0 |
| high.Newt. | — | — | — | 2 | 9 | 9 |
| ver.high.Newt. | — | — | — | 0 | 0 | 0 |
| del.high.Newt. | — | — | — | 0 | 8 | 8 |
| Sobol excl. | 62 | 63 | 63 | 62 | 63 | 63 |
| Sobol resul. | 1,179 | 1,052 | 1,052 | 1,179 | 1,098 | 1,052 |
| pos.boxes | 2 | 9 | 9 | 2 | 1 | 1 |
| verif.boxes | 1 | 0 | 0 | 1 | 0 | 0 |
| Leb.poss. | 6e-69 | 4e-49 | 3e-50 | 5e-128 | 4e-49 | 3e-50 |
| Leb.verif. | 6e-60 | 0.0 | 0.0 | 6e-60 | 0.0 | 0.0 |
| time (sec.) | 1 | $< 1$ | $< 1$ | 1 | $< 1$ | $< 1$ |

*Comments* The HIBA_USNE solver turned out to work very well with both the ReLU and LeakyReLU functions. It is worth noting that the staggered-precision version of the Newton operator (cf. the next subsection) has dealt perfectly with the "clusters" of small boxes, close to the solution. In both cases, all of them one have been deleted, resulting in a single narrow box.

Obviously, the above experiments were only a pure exemplification of using these functions, and they have no practical meaning. In practice, the main advantage of using activation functions like ReLU or LeakyReLU is that they do not saturate, which is important for deep, multi-layered networks, but not for a single-layer network, as in the presented examples.

As problems presented in this subsection were well-determined, we could conclude that the multiple-precision GS operator turned out to be efficient for well-determined problems, and inefficient for underdetermined ones. But such a statement would definitely be premature. It seems, this operator is powerful at reducing clusters of boxes close to the solution (or to the approximate solution), but it is too computationally intensive for large sets of boxes to verify. Would some randomization be helpful?

In general, proper heuristics need to be developed to decide whether to use the staggered-precision Newton operator, or not; this will also be an interesting topic for the further investigations.

It is also worth noting that changing the length of the staggered-precision record (the variable `stagprec`) did not seem to have any influence on the result. This phenomenon should be investigated in the future as well.

## 6 Further work

During the development of the ADHC library, several interesting and important new issues emerged. Many of them have already been mentioned throughout the paper, and at least some of them are going to be investigated in the near future. Let us summarize them briefly:

- adding more types that can be used in `adhc_ari` template class,
- providing interval and non-interval datatypes for other precision levels — in particular, for the half-precision floating-point types,
- providing sparse vectors and matrices for the staggered-precision arithmetic types — this would allow efficient high-precision computations of derivatives,
- other investigations related to using the staggered-precision Newton operator,
- comparing of the efficiency of using weak derivatives and slopes,
- improving the procedure of hull-consistency enforcing, and possibly adding procedures for other consistencies (interval-consistency).

There is also another topic, related to using ADHC in solvers for optimization or multicriteria analysis. In this case, it would be very worthwhile to enforce HC on conditions related to the gradients of some functions, and not the functions themselves. A good example is solving the F. John's conditions [33, 38, 50] not only using the interval Newton method, but also HC (or $k$B [23]) enforcing procedures.

The current version of ADHC allows to construct the expression tree of a function, but not of its derivatives. This is another feature the author is going to add in future versions, but it has not been decided yet on how it should be implemented.

## 7 Summary and conclusions

The paper has presented the ADHC C++ template library for Algorithmic Differentiation and Hull-Consistency enforcing. Its efficiency has been compared with respect to other AD libraries, supporting interval data types.

New possibilities provided by the aforementioned package have been described; among them bounding subdifferentials of non-smooth functions, and computing the derivatives' values over various datatypes: interval and non-interval ones. Plans and prospects of further research have also been presented. Particular focus has been put on applications related to machine learning — one of the most timely and important areas, where AD can be applied.

**Acknowledgements** The author is grateful to the reviewers for helping to improve the paper, by their valuable suggestions.

**Data availability** The software analyzed during the current study is available in the author's GitLab repository, under the GPLv2 license: [14, 15]; also the benchmarks and tests; cf. footnote 1. No other datasets were generated or analyzed.

## Declarations

**Conflict of interest** The author declares no competing interests.



## Appendix: Survive-CXSC library

All of the author's codes have been using C-XSC libraries [3] for years. The C-XSC libraries have been developed at the University of Wuppertal (Germany), under the leadership of Walter Krämer. Unfortunately, since his passing away in 2014, the further development of these tools have been stalled (the last version of this great software piece is 2.5.4, released on 28th of February 2014).

Over the years, it has become a serious issue, as the C++ language is still evolving, breaking the backward compatibility on occasion. In particular, the old-style exception throwing declarations have become invalidated by the C++17 standard. Also, some of the C-XSC classes turned out not to be thread-safe.

As the migration to another interval library would be cumbersome, and, what is even more important, other interval libraries lack several useful features of C-XSC (sparse matrix and vector classes, or collaboration with BLAS libraries), the author was willing to stick to using C-XSC.

Hence, it was decided to fork the library, starting a new one: "survive-CXSC library". It is currently hosted on GitLab [12]. Its current number is 2.6.1, as the author decided to keep consistent versioning with the original C-XSC.

Three kinds of changes have been done, up to now.

*Exceptions throwing.* No functions declare to throw any exceptions, now. If a function is to inform that it throws *no* exceptions, it is done using the modern `noexcept`, and not the old-style `throw()` declaration.

*Minor bug-fixes.* The `sqr()` function was not correctly called for the `cxsc::complex` class, due to inconsistent `inline` declarations. The typo has been corrected.

*The staggered-precision classes are now thread-safe.* This is one of the most important innovations with respect to C-XSC 2.5.4. It required changing several variable declarations in the `l_imath.cpp` file (70 of them, to be precise) from `static` to

thread-specific (a C++11 `thread_local` keyword is used, now), and a single global variable `stagprec` in file `l_real.cpp`.

Details can be found in the changelog or in the Git log files, of Survive-CXSC.

## References


1. PROFIL/BIAS library. https://www.tuhh.de/ti3/keil/profil/index_e.html (2009)
2. FILIB++ library. http://www2.math.uni-wuppertal.de/wrswt/software/filib.html (2011)
3. C++ eXtended Scientific Computing library. http://www.xsc.de (2014)
4. GAOL – Not Just Another Interval Library. https://github.com/goualard-f/GAOL (2020)
5. Adept (Automatic Differentiation using Expression Templates) C++ library. http://www.met.rdg.ac.uk/clouds/adept (2021)
6. ADOL-C library. https://github.com/coin-or/ADOL-C (2021)
7. CppAD (A C++ Algorithmic Differentiation Package) library. https://coin-or.github.io/CppAD/doc/cppad.htm (2021)
8. IBEX library. http://www.ibex-lib.org (2021)
9. Intel TBB. https://github.com/oneapi-src/oneTBB (2021)
10. INTLAB package for MATLAB and GNU Octave. https://www.tuhh.de/ti3/intlab (2021)
11. OpenBLAS library. https://www.openblas.net (2021)
12. Survive-CXSC, C++ library, a fork of C-XSC. https://gitlab.com/bkubica/survive-cxsc (2021)
13. TensorFlow library. http://www.tensorflow.org (2021)
14. ADHC, C++ library. https://gitlab.com/bkubica/adhc (2022)
15. HIBA_USNE, C++ library. https://gitlab.com/bkubica/hiba_usne (2022)
16. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Portable Documents, Pearson Education (2004)
17. Alexandrescu, A.: Modern C++ design: Generic programming and design patterns applied. Addison-Wesley (2001)
18. 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)
19. 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)
20. Blomquist, F.: Staggered correction computations with enhanced accuracy and extremely wide exponent range. Reliable Computing **15**(1), 26–35 (2011)
21. Caprani, O., Godthaab, B., Madsen, K.: Use of a real-valued local minimum in parallel interval global optimization. Interval Computations **2**, 71–82 (1993)
22. Clarke, F.H.: Optimization and nonsmooth analysis. SIAM (1990)
23. Collavizza, H., Delobel, F., Rueher, M.: Comparing partial consistencies. In: Csendes, T. (ed.) Developments in Reliable Computing, pp. 213–228. Springer, Netherlands (1999)
24. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In: International conference on parallel problem solving from nature, pp. 849–858. Springer (2000)
25. Franek, P., Ratschan, S.: Effective topological degree computation based on interval arithmetic. Mathematics of Computation **84**(293), 1265–1290 (2015)
26. Gennaro, D.D.: Advanced Metaprogramming in Classic C++. Apress (2015)
27. Géron, A.: Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media (2019)
28. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
29. Granvilliers, L.: On the combination of interval constraint solvers. Reliable Computing **7**(6), 467–483 (2001)
30. Granvilliers, L., Benhamou, F.: Progress in the solving of a circuit design problem. Journal of Global Optimization **20**(2), 155–168 (2001)
31. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM (2008)
32. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: International conference on machine learning, pp. 1737–1746. PMLR (2015)

33. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
34. Hoffmann, P.H.: A hitchhiker's guide to automatic differentiation. Numerical Algorithms **72**(3), 775–811 (2016)
35. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. Springer, London (2001)
36. Kearfott, R.B.: An efficient degree-computation method for a generalized method of bisection. Numerische Mathematik **32**(2), 109–127 (1979)
37. Kearfott, R.B.: Interval extensions of non-smooth functions for global optimization and nonlinear systems solvers. Computing **57**(2), 149–162 (1996)
38. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
39. Kearfott, R.B.: Treating non-smooth functions as smooth functions in global optimization and nonlinear systems solvers. Mathematical Research **90**, 160–172 (1996)
40. Kearfott, R.B., Muñoz, H.: Slope interval, generalized gradient, semigradient, and slant derivative. Reliable Computing **10**(3), 163–193 (2004)
41. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Vychislennyie Tiehnologii (Computational Technologies) **15**(1), 7–13 (2010)
42. Krämer, W.: Multiple/arbitrary precision interval computations in C-XSC. Computing **94**(2), 229–241 (2012)
43. Krämer, W., Zimmer, M., Hofschuster, W.: Using C-XSC for high performance verified computing. Lecture Notes in Computer Science **7134**, 168–178 (2012). PARA 2010 Proceedings
44. Kubica, B.J.: Interval methods for solving underdetermined nonlinear equations systems. Reliable Computing **15**, 207–217 (2011). Proceedings of SCAN 2008
45. Kubica, B.J.: The "second derivative" of a non-differentiable function and its use in interval optimization methods. Journal of Telecommunications and Information Technology **4**, 81–85 (2011)
46. Kubica, B.J.: Excluding regions using Sobol sequences in an interval branch-and-prune method for nonlinear systems. Reliable Computing **19**(4), 385–397 (2014). Proceedings of SCAN 2012 (15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics)
47. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. Numerical Algorithms **70**(4), 929–963 (2015). http://dx.doi.org/10.1007/s11075-015-9980-y
48. Kubica, B.J.: Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. Journal of Parallel and Distributed Computing **107**, 57–66 (2017). https://doi.org/10.1016/j.jpdc.2017.03.009
49. Kubica, B.J.: Role of hull-consistency in the HIBA_USNE multithreaded solver for nonlinear systems. Lecture Notes in Computer Science **10778**, 381–390 (2018). Proceedings of PPAM 2017
50. Kubica, B.J.: Interval methods for solving nonlinear constraint satisfaction, optimization and similar problems: From inequalities systems to game solutions, *Studies in Computational Intelligence*, vol. 805. Springer (2019). https://doi.org/10.1007/978-3-030-13795-3
51. Kubica, B.J., Hoser, P., Wiliński, A.: Interval methods for seeking fixed points of recurrent neural networks. In: International Conference on Computational Science, pp. 414–423. Springer (2020)
52. Kubica, B.J., Kurek, J.: Interval arithmetic, hull-consistency enforcing and algorithmic differentiation using a template-based package. In: CPEE 2018 Proceedings. IEEE (2018)
53. Kubica, B.J., Kurek, J.: A parallel method of verifying solutions for systems of two nonlinear equations. Lecture Notes in Computer Science **12044**, 418–430 (2020). Proceedings of PPAM 2019
54. Kubica, B.J., Woźniak, A.: A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment. Lecture Notes in Computer Science **6126/6127** (2010). Accepted for publication. PARA 2008 Proceedings
55. Kulisch, U.: Computer Arithmetic and Validity - Theory. Implementation and Applications. De Gruyter, Berlin, New York (2008)
56. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
57. Ratschek, H., Voller, R.L.: What can interval analysis do for global optimization? Journal of Global Optimization **1**(2), 111–130 (1991)
58. Shary, S.P.: Finite-dimensional Interval Analysis. Institute of Computational Technologies, Sibirian Branch of Russian Academy of Science, Novosibirsk (2013)
59. Vandevoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley (2010)

60. Zimmer, M., Krämer, W., Hofschuster, W.: Sparse matrices and vectors in C-XSC. Reliable Computing **14**, 138–160 (2010)
61. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm. TIK-report 103 (2001)