Acta Cybernetica **26** (2024) 913–932.

# GPU-Accelerated, Interval-Based Parameter Identification Methods Illustrated Using the Two-Compartment Problem

Lorenz Gillner[ab] and Ekaterina Auer[ac]

**Abstract**

Interval methods are helpful in the context of scientific computing for reliable treatment of problems with bounded uncertainty. Most traditional interval algorithms, however, were designed for sequential execution while internally depending on processor-specific instructions for directed rounding. Nowadays, many-core processors and dedicated hardware for massively parallel data processing have become the de facto standard for high-performance computers. Interval libraries have yet to adapt to this heterogeneous computing paradigm.

In this article, we investigate the parallelization of interval methods with an emphasis on modern graphics processors. Using a parameter identification scenario in combination with newly developed or enhanced GPU-based interval software, we evaluate different methods for reducing the size of large interval search domains. For the first time, algorithmic differentiation can be used with intervals on the GPU. Different versions of interval optimization algorithms are compared wrt. their functionality, run times, and energy consumption.

**Keywords:** parameter identification, interval software, parallelization, GPGPU

## 1 Introduction

Since many decades, interval analysis [27] has been used in the context of scientific computing for obtaining verified solutions to many problems, for example, in computer graphics or in engineering. One of its advantages is the possibility to quantify or propagate bounded uncertainty through systems in simulations in a deterministic way. Many of the major programming languages have built-in interval capabilities; over 20 libraries for interval arithmetic alone are available today. With the emergence of multi-core processors, parallelization of interval methods using well

[a] University of Applied Sciences Wismar, Germany
[b] E-mail: lorenz.gillner@hs-wismar.de ORCID: 0009-0007-8244-5810
[c] E-mail: ekaterina.auer@hs-wismar.de, ORCID: 0000-0003-4059-3982

DOI: 10.14232/actacyb.306774

established libraries such as C–XSC, Boost, or Profil/Bias has been tested in multi-threaded and distributed systems [11, 23, 28, 35]. However, high-performance computing industry is moving towards developing specialized hardware to accelerate repetitive tasks, for example, graphics processing units (GPUs), data processing units (DPUs), or field-programmable gate arrays (FPGAs). Interval software still has to be adapted to such specific co-processors.

Especially general-purpose graphics processing units (GPGPUs) have inspired much interest among the scientific community. One part of the reason is their low cost and high availability compared to conventional supercomputers or large-scale CPU clusters. The application of GPUs for computations with the focus on uncertainty has been investigated in [2, 3, 9, 10, 29, 34], just to name a few. Nonetheless, popular interval libraries cannot typically be used on the GPU directly, because the co-processors' architecture differs significantly from that of conventional CPUs. For example, the switching of rounding modes commonly applied in CPU-based interval libraries, which has a negative influence on computing performance, is not necessary on the GPU, since most mathematical operations are available as specifically rounded versions. Notable examples of custom interval libraries for the GPU are given in [5, 7, 21]. In this paper, we are interested in the usefulness of GPUs in the area of interval computations. We extend traditional (sequential) ideas to the massively parallel computing paradigm of GPUs and apply them in a classic parameter estimation scenario.

The paper is structured as follows. In Section 2, the basics of the employed interval techniques are described and software implementing them on the GPU is highlighted where applicable. In Section 3, parameter identification methods we implement using the GPU are detailed. In Section 4, we employ the described methods within our testing procedure relying on the well-known example of a two-compartment problem and highlight the comparison results. A perspective on the paper's findings and an outlook on our further research are in Section 5.

## 2 Background on Interval Analysis

In this section, we provide a short overview of the concepts from the area of interval analysis that are applied in the rest of this article. Where necessary, we also highlight the GPU-based software implementing them.

### 2.1 Interval Analysis and Algorithmic Differentiation

Interval analysis (IA), formally introduced by R. Moore in 1966 [26], is a powerful mathematical tool for *verified* computations, that is, computations with an automatically provided guarantee of correctness. In the context of scientific computing, IA offers a way to compensate for numerical uncertainty caused by finite floating point (FP) number representation as specified by the IEEE 754 standard. Instead of working with a crisp FP number approximating a given real number, IA methods rely on an interval with FP bounds containing it, propagating this uncertainty

through a computation. Almost as a by-product, this approach allows us to propagate uncertainty representable by intervals through systems in a deterministic and verified way. A real interval $\mathbf{x}$ is a closed interval defined as

$$\mathbf{x} = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\},$$

whereas a machine interval has to be additionally rounded towards $\pm\infty$ to the next possible representation $[\lfloor \underline{x} \rfloor, \lceil \overline{x} \rceil]$. For two intervals $\mathbf{x}$ and $\mathbf{y}$, arithmetic operations $\circ \in \{+, -, \cdot, /\}$ can be defined as

$$\mathbf{x} \circ \mathbf{y} = \{x \circ y \mid \forall x \in \mathbf{x}, y \in \mathbf{y}\},$$

while elementary functions $\zeta(.)$ (e.g. $\sin x$) can be extended for the use with intervals as

$$\zeta_{[]}(\mathbf{x}) = [\min\{\zeta(a) \mid \forall a \in \mathbf{x}\}, \max\{\zeta(b) \mid \forall b \in \mathbf{x}\}].$$

These simple rules can be used to define (naive) interval arithmetic, that is, a way to evaluate composite functions over intervals directly, without having to solve any optimization problems as in the equation above. There are more involved interval algorithms, for example, those for computing verified enclosures of solutions to (non-linear) systems of (differential) equations.

Although replacing FP values by intervals guarantees an enclosure of the true result, the results can exhibit overestimation, that is, intervals being wider than necessary. This behaviour originates from the so-called *dependency problem*, when multiple occurrences of the same interval variable are interpreted as separate new variables, and the *wrapping effect*, meaning that an interval enclosure of any (multi-dimensional) shape not identical to an axis-aligned box will always contain some amount of superfluous data [27].

The ability to perform automatic differentiation (AD) is essential for any form of (non-naive) interval analysis [14]. For example, the centered form of an interval extension to the real-valued function $g$, in the univariate case defined as

$$g_{[\tau]}(\mathbf{x}) = g(\tau) + g'(\mathbf{x}) \cdot (\mathbf{x} - \tau), \quad \tau \in \mathbf{x}, \tag{1}$$

requires the evaluation of the first derivative of $g$. The same is true for interval root-finding algorithms, such as the interval Newton method [5]. The widely used divided differences method

$$g'(x) \approx \frac{g(x+h) - g(h)}{h} \tag{2}$$

yields only an approximation of the true derivative, with the error increasing as the step size decreases, which makes computations using it less reliable [4]. Although symbolic differentiation (SD) produces expressions for exact derivatives, it might exhibit a considerable computational overhead (since it is a top-down approach) or even increase overestimation when used with intervals [16]. By contrast, automatic differentiation allows us to compute the value of a function's exact derivative at a specific point or its enclosure in a bottom-up approach without errors introduced by floating point approximation and with less of the overhead. In general, AD falls into three steps:

1. Decomposition of $g$ into elementary expressions (e.g., $\ln x$)
2. Differentiation of elementary expressions $\left(\text{e.g., } \frac{\partial}{\partial x} \ln x = \frac{1}{x}\right)$
3. Application of the chain rule, i.e.
   $(y(x(t)))' = \frac{\partial y}{\partial t} = \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial t} = y'(x(t)) \cdot x'(t)$, e.g., $\frac{\partial}{\partial t} \ln x(t) = \frac{1}{x(t)} x'(t)$.

This allows us to differentiate not only mathematical functions, but also entire code segments containing them. While there is a wide selection of AD libraries available, some of them even compatible with the GPU [13], interval arithmetic is typically not supported by these libraries. Although many popular interval libraries can perform AD of interval-valued functions on the CPU, to the authors' knowledge, the Julia language is the only open-source tool offering a straightforward way of using AD with interval-valued functions on the GPU [31], see Section 2.3 for details.

## 2.2 Cooperativity

For the so-called *cooperative* dynamical systems, it is possible to perform set-based computations taking into account bounded uncertainty in parameters without necessarily applying methods with result verification. Consider an ODE system in the explicit, autonomous form

$$y' = f(y, p),\ y : \mathbb{R} \mapsto \mathbb{R}^{n_y},\ p \in \mathbb{R}^{n_p},\ f : \mathbb{R}^{n_y + n_p} \mapsto \mathbb{R}^{n_y} \tag{3}$$

depending on a time-invariant parameter $p$ (that is, $p' = 0$) and parameter-invariant initial conditions $y(t_0)$ of the solution function $y$ (that is, $\dfrac{\partial y_i(t_0, p)}{\partial p_j} = 0$), possibly influenced by bounded uncertainty with $y(t_0) \in \mathbf{y_0}$ for a $t_0 \in \mathbb{R}$ and $p \in \mathbf{p}$, where the bold face denotes characteristics described by intervals as introduced in the previous subsection. The property of cooperativity holds for the system in Eq. (3) if

$$\frac{\partial f_i}{\partial y_j} \geq 0 \text{ for all } i \neq j,\ i, j = 1 \dots n_y\,. \tag{4}$$

Müller's theorem [25] combined with the property of cooperativity results in the Smith's theorem [32] that shows the possibility to quantify the uncertainty in the system (3) by working with two systems with crisp parameters that are independent of each other (the bracketing systems)

$$y'_{\mathrm{lb}} = \underline{f}(y_{\mathrm{lb}}, p_{\mathrm{b}}) \quad \text{and} \quad y'_{\mathrm{ub}} = \overline{f}(y_{\mathrm{ub}}, p_{\mathrm{b}}) \tag{5}$$

instead of one system with uncertain parameters. Here, $p_{\mathrm{b}}$ means that the crisp values of the lower or upper bounds of the interval $\mathbf{p} = [p_{\mathrm{lb}}\ ,\ p_{\mathrm{ub}}]$ are used (not the values in between them). The true result $y(t)$ lies in the convex hull $[\underline{y}_{\mathrm{lb}}(t), \overline{y}_{\mathrm{ub}}(t)]$ of the enclosures $\mathbf{y}_{\mathrm{lb}}(t)$, $\mathbf{y}_{\mathrm{ub}}(t)$ of the true solutions $y_{\mathrm{lb}}(t)$, $y_{\mathrm{ub}}(t)$ to (5). Considering such enclosures in each point of time $t_k$ of the chosen time grid gives us the verified flow of the uncertain system (3) over this grid. Good approximations $y_{\mathrm{lb}}^{(k)}$, $y_{\mathrm{ub}}^{(k)}$

to the verified solutions $y_{\mathrm{lb}}(t_k)$, $y_{\mathrm{ub}}(t_k)$ are obtained by solving the systems in (5) using traditional floating point arithmetic (e.g., solvers `odeint` available in the C++ library BOOST). This procedure captures the bulk of output uncertainty and can be useful on the GPU (cf. Section 3.2.1) since appropriate GPU implementations of verified ODE solvers are not available at the time of writing.

## 2.3 Software for IA and IA-Based AD on the GPU

Implementation of interval methods in any programming language requires interval arithmetic as the core component; performing full interval analysis for a practical problem often needs further functionalities and methods. The C–XSC toolbox [20] or INTLAB [30], for example, offer additional modules for AD, root-finding and optimization, but due to their CPU-specific instructions for rounding control, they are not suited for use on the GPU. One very promising tool for GPU computations involving interval-based AD is the Julia language, because its meta-programming approach allows for generating hardware-specific instructions from generically formulated source code. In [31], it was demonstrated that it was indeed possible to compute derivatives of interval functions for both the CPU and GPU, albeit without tight-as-possible rounding and full support for elementary functions. However, the focus of this paper is on native C++ implementations, because the comfort of Julia's generic programming comes at the price of lengthy compile times.

The Compute Unified Device Architecture (CUDA) is the programming framework for NVIDIA-manufactured GPUs. A basic library for interval arithmetic on CUDA-compatible GPUs was presented in [7]. This library has been extended by a selection of elementary set-based functions in [8]. While arithmetic operations such as addition are available in different directed rounding modes in CUDA C++, the same is not true for trigonometric and other elementary functions. Their results, therefore, need to be rounded to the next and previous FP number if the simplest kind of enclosure is to be obtained. Hence, enclosures produced by the GPU are in some cases wider than they would have been on the CPU. We use our own, enhanced version of this interval library for all interval computations in this paper.

To use interval methods requiring derivatives, programmers are required to provide all necessary derivatives directly in the source code at the moment. To enable AD for these libraries, we ported the forward mode of the well-established C++ library FADBAD [6] to the CUDA C++ dialect. It is entirely template-based, which allows it to work both with different FP data types and interval ones, as long as a corresponding arithmetic is defined completely. Compatibility with CUDA-capable devices can be implemented in a fairly straightforward way, at least in the case of forward mode AD. The data structures and functions used by FADBAD are entirely compatible with CUDA-intrinsic functions, so they merely have to be labeled as suitable for CPU (`__host__`) and GPU (`__device__`) use. This modification makes FADBAD the first AD library to fully support IA on both the CPU and the GPU. A topic for our future work is to test the limitations of this implementation on the GPU and to improve it accordingly, if possible.

# 3 Parameter Identification

In this section, we describe set-based optimization methods — some of them based on IA and fully verified — for the general goal of parameter identification on the GPU. Our approach is not to parallelize a given optimization method; rather, we rely on the massive data parallelism of the GPU to employ brute-force techniques that would be too time-consuming on the CPU.

## 3.1 General Possibilities for Parameter Identification

The task of identifying $n$ unknown but bounded system parameters can be viewed as a global optimization problem of the form

$$\min_{\mathbf{x}\in\mathbb{R}^n} c(\mathbf{x}),\ c : \mathbb{R}^n \mapsto \mathbb{R}\,, \tag{6}$$

where $c$ is the objective (cost) function wrt. the system parameters $\mathbf{x}$. According to the widely used least-squares principle, the cost function $\Phi$ can be employed to identify unknown parameters $p \in \mathbb{R}^{n_p}$ of a dynamic model of the form in Eq. (3) with initial conditions at $t_0 := T_b - 1$, given a search space $\mathbf{p} \in \mathbb{IR}^{n_p}$ and measured data over a discrete time grid $t_k \in \{T_b, T_b + 1, \ldots, T_e\}$:

$$\Phi(p) = \sum_{k=T_b}^{T_e} \sum_{j=1}^{n_m} \left(y_j(t_k, p) - y_{m,j}^{(k)}\right)^2 \xrightarrow{p\in\mathbf{p}} \min\,, \tag{7}$$

where $y_j(t_k, p)$ is the $j$th component of the solution to the model in Eq. (3); $y_{m,j}^{(k)}$ is the $j$th component of the measurement made for the solution at the time point $t_k \in [T_b, T_e]$; $n_m$ is the number of the measured solution components; $T_e$, $T_b$ are the end and start times of data recording. There are different possibilities to tackle this problem, with varying degrees of verification associated with them. The available options are:

**F0** Do we use the formula in Eq. (7)?

**F0.a (no)** Experimental/neural networks/other **F0.b (yes)** Least squares

**F1** How exactly is $y(t_k, p)$ obtained?

**F1.a** A closed-form solution

**F1.b** Approximation by an expression (e.g., Euler's method)

**F1.c** Numerical solution from a "black-box" solver

**F2** What is the underlying technique for the implementation?

**F2.a** Fixed or floating point **F2.b** Interval **F2.c** Other verified

**F3** How do we represent the measured data $y_{m,j}^{(k)}$?

**F3.a** As provided by sensors: Hundreds of MB of floating point numbers

**F3.b** With the help of any reliable means for data reduction

As an example, consider the set of options F0.a-F1.b-F2.b-F3.a. The true solution $y(t,p)$ of the IVP in Eq. (3) can be approximated by an explicit method (e.g., Euler's) in its interval version taking into account numerical errors but not the discretization error. For Euler's method, the formula is $\mathbf{y}^{(k)} := \mathbf{y}^{(k-1)} + h \cdot f(\mathbf{y}^{(k-1)}, \mathbf{p})$ for a constant step size $h := t_k - t_{k-1}$. The interval approximation $\mathbf{y}^{(k)}$ at $t_k$ is then substituted for the exact solution $y(t_k, p)$ in the cost function (7) and the discretization error ignored. For the example we consider in this paper as an illustration (cf. Section 4.3), the step size is chosen to be equal to the sampling time for the data, $h = 1$. In general, for this approach to give good results, that is, for $\mathbf{y}^{(k)}$ to be an acceptable approximation of the true solution $y(t_k, p)$, this sampling time (or the step size) should be significantly smaller than the dominant time constant of the process described by the IVP. The approximated cost function can then be rewritten as

$$\Phi_{app}(p) = \sum_{k=T_b}^{T_e} \sum_{j=1}^{n_m} \left( y_j^{(k-1)} - y_{m,j}^{(k)} + h \cdot f_j(y^{(k-1)}, p) \right)^2 , \tag{8}$$

where $y^{(T_b-1)}$ is the initial condition. Although the whole process is not verified, even if interval optimization procedures are applied, the overall verification degree is high if the first (and second) derivatives of $\Phi_{app}$ are computed exactly with the help of AD.

Option F3 deserves a separate discussion. On modern multi-processor systems, high-bandwidth interconnect technologies and large amounts of expandable main memory allow for quick access to terabytes of data, either stored locally or in a distributed manner. By contrast, GPUs typically have a fixed amount of on-board memory that is often limited to a fraction of the capacity available on the CPU. So in case of option F3.a, the question arises how large data sets should be handled on the GPU. Data partitioning and sequential processing of small data blocks at a time involves many expensive memory transfers. One idea is to interpolate the data, for example, by a (piecewise) scalar Bézier curve $b(t)$ depending on time with a predefined degree and enclose them in a verified way by a corresponding interval extension $b_{[]}(t)$. That is, the kernel functions depending on data can be implemented much easier and more efficiently on the GPU using Option F3.b.

Using the CPU as the basis, we explored, for example, the option F0.b-F1.b-F2.b-F3.a for different kinds of solid oxide fuel cell models in [1, 19]. The option F0.a-F1.c-F2.a-F3.a is studied for a distributed heating system on the GPU in [2].

## 3.2 Optimization Algorithms

In this subsection, we describe in detail different GPU-based optimization approaches depending on the choices made according to the general options from Section 3.1. All of them are brute-force approaches; the availability of the cheap

GPU computing power makes it possible to carry them out in acceptable time. Note that the algorithms proposed here do not try to parallelize a sequential optimization algorithm, but rather execute the sequential approach for each considered data item in parallel.

### 3.2.1 Experimental Identification

Let us suppose that measurements $y_{m,j}^{(k)}$ for specified solution components $y_j(t)$ are available at all times $t_k \in \{T_b, \dots, T_e\}$. For these, plausibility bounds $\mathbf{y}_{m,j}$ can be derived on the basis, for example, of physical constraints or of tolerances of measurement devices. This information can be used to reduce the initial search box $\mathbf{p}$ for optimal parameters. A general scheme for a brute-force algorithm is

**Step 1** Bisect $\mathbf{p} = \bigcup_{j=1}^{l} \mathbf{p}_j$ until a predefined bound on the width of $\mathbf{p}_j$ is reached

**Step 2** Compute an enclosure $\mathbf{y}_i^{(k)}$ of $\mathbf{y}_i(t_k, \mathbf{p}_j)$ in parallel

**Step 3** Exclude $\mathbf{p}_j$ if $\exists k, i$: $\mathbf{y}_i^{(k)} \cap \mathbf{y}_{m,i}^{(k)} = \emptyset$ (in parallel)

**Result** List $\mathcal{L} \subseteq \{1, \dots, l\}$ containing indices of suitable boxes

Steps 2 and 3 can be performed on the GPU, for example, in floating-point arithmetic using the Smith theorem [32]. Moreover, the plausibility test in Step 3 can be replaced by any other test mentioned in the next subsection. One possible use for the results is to build the hull of $\mathbf{p}_j$ for $j \in \mathcal{L}$ to obtain a reduced search space. The algorithm can then be reiterated if necessary using the tighter search interval with a lower bisection bound. A possibly better use is to compute the convex hull of enclosures for $\mathbf{y}(t_k, \mathbf{p}_j)$ for $j \in \mathcal{L}$ (again on the GPU). This new enclosure can be good enough even without subsequent least squares.

### 3.2.2 Preconditioning

When the parameter domain has a large plausibility range, different preliminary tests can be applied to exclude boxes from the search [18, 33]. In the brute-force algorithm in the previous subsection, the test in Step 3 can be replaced by various other preconditioning approaches.

The midpoint test is one possibility for such preconditioning. A box $\mathbf{p}_j$ is excluded from the search space if $\underline{c_{[]}(\mathbf{p}_j)} > \overline{\mathbf{z}^*}$, where $\mathbf{z}^*$ is an enclosure of the global minimum and $c_{[]}$ an interval extension of the cost function $c$ [18]. For example, we know that the least squares cost function $\Phi(p)$ from Eq. (7) has zero as its global, ideal minimum. This global minimum, however, might not be known beforehand in general, or cannot be reached. On the GPU, where all tests are virtually executed at the same time, this value must be chosen carefully.

A more elaborate option is the monotonicity test. A box $\mathbf{p}_i$ is excluded from the search if it fails to satisfy the condition $0 \in \nabla c_{[]}(\mathbf{p}_i)$ ($\nabla$ meaning the exact gradient). In this paper, we apply the monotonicity test in parallel on the GPU, see the example in Section 4. We plan to consider the convexity test in our future work.

### 3.2.3 Set Inversion Approach

Nonlinear parameter identification can be characterized as a set inversion problem. SIVIA — set inversion via interval methods — is an algorithm frequently employed in optimization and parameter estimation tasks involving quantities with bounded uncertainty [15]. There are different variations of the algorithm; Algorithm 1 describes its simplest form. Given an interval function $c_{[]}$, an image $\mathbf{z}$ and an initial search domain $\mathbf{p}_0$, the algorithm tests whether $c_{[]}(\mathbf{p}_i)$ is a real subset of $\mathbf{z}$, $\mathbf{p}_i$ being sub-boxes of $\mathbf{p}_0$ obtained by bisection, until the width of a box is at most $\epsilon$ in its width. Boxes that certainly are preimages of $\mathbf{z}$ are part of the solution set $\mathcal{S}$ (also called *paving*), boxes containing no solution are members of $\mathcal{N}$ and boxes with the width smaller than $\epsilon$ belong to the boundary set $\mathcal{E}$.

**Algorithm 1** Generic SIVIA algorithm.

**Require:** $c_{[]}$, $\mathbf{z}$, $\mathbf{p}_0$, $\epsilon$
1: $\mathcal{S} \leftarrow \emptyset$, $\mathcal{N} \leftarrow \emptyset$, $\mathcal{E} \leftarrow \emptyset$, $\mathcal{L} \leftarrow \{\mathbf{p}_0\}$
2: **while** $\mathcal{L} \neq \emptyset$ **do**
3: $\quad \mathbf{p} \leftarrow \text{pop}(\mathcal{L})$
4: $\quad$ **if** $c_{[]}(\mathbf{p}) \subset \mathbf{z}$ **then**
5: $\quad\quad \text{push}(\mathcal{S}, \mathbf{p})$
6: $\quad$ **else if** $c_{[]}(\mathbf{p}) \cap \mathbf{z} = \emptyset$ **then**
7: $\quad\quad \text{push}(\mathcal{N}, \mathbf{p})$
8: $\quad$ **else if** $diam(\mathbf{p}) < \epsilon$ **then**
9: $\quad\quad \text{push}(\mathcal{E}, \mathbf{p})$
10: $\quad$ **else**
11: $\quad\quad \mathbf{p}_L, \mathbf{p}_R \leftarrow \text{bisect}(\mathbf{p})$
12: $\quad\quad \text{push}(\mathcal{L}, \mathbf{p}_L)$
13: $\quad\quad \text{push}(\mathcal{L}, \mathbf{p}_R)$
14: $\quad$ **end if**
15: **end while**
16: **return** $\mathcal{S}, \mathcal{N}, \mathcal{E}$

The approach from Algorithm 1 can be parallelized in two basic ways:

PSIVIA: a priori subdivision of $\mathbf{p}_0$ into small boxes with subsequent inclusion test in parallel, or

NSIVIA: coarse subdivision of $\mathbf{p}_0$, then parallel execution of SIVIA on sub-boxes.

In the first version, PSIVIA, the bisection step is eliminated by the pre-computation of a grid, making the remaining part of the algorithm *perfectly parallelizable*. Since all boxes are similar in size, the inclusion of a box into the set $\mathcal{E}$ is no longer determined by the box width. Instead, the remaining boxes, being neither a subset of $\mathbf{z}$ nor having an empty intersection with $\mathbf{z}$, satisfy the condition $\mathbf{p}_i \cap \mathbf{z} \neq \emptyset$, qualifying as boundary boxes. This type of lightweight algorithm benefits the most

**Algorithm 2** Parallelized SIVIA.

**Require:** $c_{[]}$, $\mathbf{z}$, $\mathbf{p}_0$
$\mathcal{S} \leftarrow \emptyset$, $\mathcal{N} \leftarrow \emptyset$, $\mathcal{E} \leftarrow \emptyset$
$\mathcal{L} \leftarrow \{\text{subdivide}(\mathbf{p}_0)\}$
**for all** $\mathbf{p}_i \in \mathcal{L}$ **do**
  **if** $c_{[]}(\mathbf{p}_i) \subset \mathbf{z}$ **then**
    push($\mathcal{S}$, $\mathbf{p}_i$)
  **else if** $c_{[]}(\mathbf{p}_i) \cap \mathbf{z} = \emptyset$ **then**
    push($\mathcal{N}$, $\mathbf{p}_i$)
  **else**
    push($\mathcal{E}$, $\mathbf{p}_i$)
  **end if**
**end for**
**return** $\mathcal{S}, \mathcal{N}, \mathcal{E}$

from parallel processing, especially on the GPU. Algorithm 2 shows the reformulated, parallelizable SIVIA procedure. In parameter identification scenarios, we are interested in enclosures containing the system parameters. Depending on the subdivision strategy, the optimal enclosure box might be situated directly on the boundary of two boxes in the subdivision. Hence, forming a convex hull of all boxes contained in $\mathcal{S}$ might seem sensible. However, the solution set $\mathcal{S}$ might consist of non-adjacent boxes, or be a set of adjacent boxes enveloping a set that does not contain any solutions. In the worst case, the hull of such a paving will be marginally smaller than, or equal to, the initial search domain $\mathbf{p}_0$. While this box might be guaranteed to contain optimal parameters, it might also get too large to lead to any meaningful conclusions about the problem under study. From this perspective, building a convex hull over solutions $y(t_k, \mathbf{p}_i)$ to Eq. (3) over each $\mathbf{p}_i \in \mathcal{S}$ (or, possibly, $\mathcal{S} \cup \mathcal{E}$) at each $t_k$ might be a more useful option. Another possibility is to treat boxes from $\mathcal{S}$ as entities in an $n$-dimensional feature space so that we can apply density-based clustering to group suitable boxes into smaller sets as described in Algorithm 3. The result of that algorithm is a set of boxes $\mathcal{C}$, where each box contains at least one box belonging to $\mathcal{S}$, ultimately containing the entire $\mathcal{S}$.

# 4 Test Application: Two-Compartment Model

In this section, we test the performance of the methods introduced in Section 3 by applying them to the example of the two compartment model similarly to [17].

## 4.1 Problem Description

The two-compartment model described in [14] has been a standard example in parameter identification since many years. It describes the evolution of two interconnected compartments after an impulse. This dynamic system can be described

**Algorithm 3** Fast interval clustering.

**Require:** $\mathcal{B},\ \theta$
1: $\mathcal{C} \leftarrow \emptyset$
2: **for all** $\mathbf{b}_i \in \mathcal{B}$ **do**
3: $\quad processed \leftarrow false$
4: $\quad$ **for all** $\mathbf{q}_j \in \mathcal{C}$ **do**
5: $\quad\quad$ **if** $\max_k |\mathbf{b}_{i,k} - \mathbf{q}_{j,k}| \leq \theta$ **then**
6: $\quad\quad\quad \mathbf{q}_i \leftarrow \mathbf{b}_i \cup \mathbf{q}_j$
7: $\quad\quad\quad processed \leftarrow true$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: $\quad$ **if** $\neg processed$ **then**
11: $\quad\quad$ push($\mathcal{C}$, $\mathbf{b}_i$)
12: $\quad$ **end if**
13: **end for**
14: **return** $\mathcal{C}$

by a two-dimensional first-order ODE of the form

$$\begin{aligned} \dot{y}_1 &= -(p_3 + p_1) \cdot y_1 + p_2 \cdot y_2 \\ \dot{y}_2 &= p_3 \cdot y_1 - p_2 \cdot y_2 \end{aligned} \tag{9}$$

with the initial condition $\mathbf{y}(0) = (1, 0)^T$. We are interested in the values of $p_1, p_2, p_3$ given (measurement) data. The problem is formulated such that only the second component $y_2$ is measured (i.e., the data is given only for $y_2$). These linear ODEs can be solved analytically:

$$y_2(t) = \alpha \cdot \left(e^{-\lambda_1 \cdot t} - e^{-\lambda_2 \cdot t}\right) \tag{10}$$

with its macro-parameters defined as

$$\begin{aligned} D &= \sqrt{(p_1 - p_2 + p_3)^2 + 4p_2 \cdot p_3}\,, \quad \alpha = \frac{p_3}{D}\,, \\ \lambda_1 &= \frac{1}{2}(p_1 + p_2 + p_3 - D)\,, \quad \lambda_2 = \frac{1}{2}(p_1 + p_2 + p_3 + D)\,. \end{aligned} \tag{11}$$

Following [17], this exact solution can be reformulated to reduce overestimation as

$$y_2(t) = \left(p_3 \cdot e^{\frac{-p_3 \cdot t}{2}}\right)\left(e^{\frac{-p_2 \cdot t}{2}}\right)\left(e^{\frac{-p_1 \cdot t}{2}}\right)\left(e^{\frac{D \cdot t}{2}} - e^{-\frac{D \cdot t}{2}}\right)/D\,. \tag{12}$$

We implemented three GPU-based approaches to reduce the search space for the parameter estimation and applied them to the two-compartment model: the experimental approach from Section 3.2.1 involving the black-box solver `odeint`, taking advantage of the system's cooperativity (I), the monotonicity test from Section 3.2.2 (II) and PSIVIA from Section 3.2.3 with the image $\Phi_{[]}(\mathbf{p}) \subset [0, 0.005]$

(III). All three methods have been tested with raw data points for $y_2$ in combination with the respective plausibility bounds copied to the GPU's memory and a continuous enclosure of the measurements by interval Bézier curves. Methods II and III were additionally tested with the exact solution from Eq. (12) and an approximation of $y(t)$ by using Euler's method as described in the formula from Eq. (8). The scenarios we considered in this paper are therefore defined as follows with respect to the parameter identification options mentioned in Section 3.1: Ia and Ib correspond to F0.a-F1.c-F2.a-F3.a/b; IIa/IIIa and IIb/IIIb to F0.b-F1.a-F2.b-F3.a/b; IIc/IIIc and IId/IIId to F0.b-F1.b-F2.b-F3.a/b. Note that IIa/IIIa and IIb/IIIb use only verified operations and algorithms and in this way can be considered verified on the GPU.

## 4.2 Reference System and Testing Setup

With an emphasis on reproducibility of the experiments, measurement data were synthesized by the following procedure. First, system parameters were defined as $\mathbf{p} = (0.232718, 1.925403, 0.145076)^T$. Inserting $\mathbf{p}$ into the exact solution from Eq. (10), we then calculated the values of $y_2(t)$ from $T_b = 1$ to $T_e = 16$ with a step size $h = 1$. Afterwards, small pseudo-random FP numbers were added to each value to achieve a simulated measurement noise. For random number generation, the well-known Classic Mersenne Twister [24] with a seed value of 1788 was used. The resulting uniform distribution was then transformed into a normal distribution with a mean value of $\mu = 0$ and a standard deviation $\sigma = \frac{1}{2^9}$. We used the C++ standard library function `std::normal_distribution`, which is based on the Box-Muller transform. Finally, the results were truncated after the sixth decimal place. Table 1 shows the data points acquired in that way.

Table 1: Artificial measurements for $y_2$ (with added plausibility bounds of $\pm$ 0.007).

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $y_{m,2}$ | 0.051801 | 0.046753 | 0.040787 | 0.032543 | 0.024291 | 0.021511 | 0.020577 | 0.012213 |

| $t$ | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| $y_{m,2}$ | 0.008919 | 0.007634 | 0.004929 | 0.008158 | 0.006349 | 0.002386 | 0.005543 | 0.004355 |

All computations were performed on

- a CPU system with an Intel Xeon Gold 5215 64 bit CPU with 192 GB of main memory and
- a GPU system with an NVIDIA Quadro RTX 6000 GPU with 24 GB of memory,

running the operating system Ubuntu 22.04 with CUDA 12 and the NVIDIA driver 525.105 installed. Experiments on the GPU were written in the CUDA C++. For interval computations on the CPU, we used the `interval` library from Boost. On the GPU, we used a modified version of the extended `cuda_interval_lib` from [8],

originally published in [7]. Device-side automatic differentiation of interval data types was made possible by a custom port of the forward-mode differentiation from FADBAD [6] to CUDA C++ (both described in more detail in Section 2.3). Function execution times were measured by wrapping the relevant parts of the program between two calls to `std::chrono::high_resolution_clock::now()` and calculating their difference at nanosecond resolution. The execution time of entire programs was captured by the standard UNIX command `time`. Since the testing environment was accessed exclusively via a remote connection, measurements of power consumption were limited to software-based tools. According to [22], the reported sampling time of power usage information on NVIDIA GPUs is 20 ms. Taking into account that a kernel might complete its operation in less time for the considered relatively small example, these measurements are only rough approximations.

## 4.3 Results

We conducted a series of (standardized) tests to compare the approaches presented in the previous sections. Given the measurements in Table 1, we chose the (hypothetical) plausibility bounds $\mathbf{y}_{m,2} = [y_{m,2} \pm 0.007]$, the search space $\mathbf{p} = [0.01, 1] \times [1, 2] \times [0.05, 2]$, and a maximum box width $w = 0.05$, which resulted in a subdivision containing $2^{16}$ possible boxes. The search space was chosen in such a way as to exclude possible symmetric parameter values; theoretically, only one optimum is contained in $\mathbf{p}$ if we consider the data generation procedure described in Section 4.2. The averaged results for 100 test trials for each method are shown in Table 2.

The enclosure $\mathbf{p}^*$ is the convex hull of all suitable boxes after search space reduction. To achieve a fair comparison between all methods, the enclosure of the SIVIA-like method includes the boundary set $\mathcal{E}$ as well, since the other two methods make no distinction between uncertain boxes and definite solutions. In Column *Boxes*, the number of boxes in the corresponding solution list is shown. Kernel time $t_{kern}$ represents the execution time of each method on its own, while $t_{wall}$ is the total number of seconds elapsed (wall-clock time) for the entire program execution, including post-processing. During each test, peaks in CPU and GPU power usage were recorded. The highest enduring peaks during kernel execution $P_{peak}$ are listed in the last column.

The scenarios I, II and III were described in Section 4.1. For comparison, we also tested the example with a CPU-based C++ implementation of SIVIA (IV) and with the `@infsup/fsolve` function (V) provided by the `interval` package for GNU Octave [12]. Decimal values in the table are rounded outwards/to nearest to four decimal places for improved readability.

Based only on enclosure size, the best results are obtained by the experimental method. Although its execution time is also fairly low, this method has the highest peak in power consumption. This comes as no surprise since the use of the black-box solver `odeint` adds a significant amount of computations. This impact on energy efficiency might be a factor to consider when solving more complex problems. Moreover, although the bulk of uncertainty is captured by this approach, the

Table 2: Test results for the parameter space $\mathbf{p} = [0.01, 1] \times [1, 2] \times [0.05, 2]$.

| Method | | $\mathbf{p}^*$ | Boxes | $t_{kern}$ (s) | $t_{wall}$ (s) | $P_{peak}$ (W) |
|---|---|---|---|---|---|---|
| GPU | Ia | $[0.0718, 0.505] \times [1, 2] \times [0.05, 0.2024]$ | 597 | 0.0579 | 0.3664 | 39.2905 |
| | Ib | $[0.0718, 0.505] \times [1, 2] \times [0.05, 0.2024]$ | 548 | 0.0568 | 0.3664 | 39.1510 |
| | IIa | $[0.01, 1] \times [1, 2] \times [0.05, 2]$ | 47 531 | 22.9567 | 25.3034 | 90.5923 |
| | IIb | $[0.01, 1] \times [1, 2] \times [0.05, 2]$ | 47 509 | 22.9817 | 25.3283 | 90.5746 |
| | IIc | $[0.01, 1] \times [1, 2] \times [0.05, 2]$ | 65 536 | 28.547 | 31.8037 | 90.8255 |
| | IId | $[0.01, 1] \times [1, 2] \times [0.05, 2]$ | 65 536 | 28.3206 | 31.7399 | 90.5315 |
| | IIIa | $[0.01, 1] \times [1, 2] \times [0.05, 0.6899]$ | 11 215 | 0.0056 | 0.4193 | 37.6086 |
| | IIIb | $[0.01, 1] \times [1, 2] \times [0.05, 0.6899]$ | 11 178 | 0.0288 | 0.4358 | 37.6487 |
| | IIIc | $[0.01, 1] \times [1, 2] \times [0.05, 0.8727]$ | 12 910 | 0.0038 | 0.6337 | 36.2425 |
| | IIId | $[0.01, 1] \times [1, 2] \times [0.05, 0.8727]$ | 12 899 | 0.0274 | 0.625 | 36.0214 |
| CPU | IVa | $[0.01, 1] \times [1, 2] \times [0.05, 0.5985]$ | 8 877 | 0.4384 | 0.4548 | 46.7245 |
| | Va | $[0.01, 1] \times [1, 2] \times [0.05, 0.7875]$ | 1 067 | 26.781 | 29.117 | 49.572 |

computations are not verified.

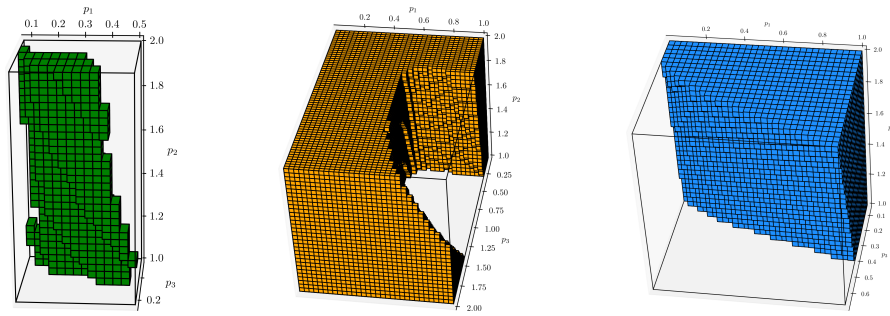
Although the monotonicity test should actually be less computationally expensive, it takes considerably longer than the previous method, while reducing the initial search space only slightly (not even noticeable from the convex hull). Because only a fraction of the search space has been discarded (18 005 out of 65 536), this test alone is not sufficient for search space reduction, at least in this scenario. The high execution time is a result of our GPU version of FADBAD still having essentially the same structure as the version on the CPU, where memory access is not as limited and expensive as on the GPU. Using the Euler-based approximation of the solution to (3) in IIc/d does not help to exclude any boxes.

Out of the three GPU-based methods considered here, PSIVIA shows the best performance in terms of computation time and power usage due its simplicity, even surpassing the traditional CPU-based SIVIA implementation in execution time, despite being a brute-force approach. In Figure 1, visualizations of the reduced parameter space are shown after each of the methods with the exact solution and measurement data provided was applied.

If only the set of guaranteed boxes $\mathcal{S}$ is considered for PSIVIA, the remaining enclosure is

$$[0.2265, 0.4122] \times [1.2187, 2] \times [0.0804, 0.2024],$$

outperforming the experimental approach wrt. the width of the convex hull. When we apply Algorithm 3 with $\theta = 0.25$ as part of the post-processing to the solution set produced by PSIVIA (using the exact solution and no interpolation), the resulting boxes are



(a) $\mathbf{p}^*_{\text{Ia}}$ (b) $\mathbf{p}^*_{\text{IIa}}$ (c) $\mathbf{p}^*_{\text{IIIa}}$

Figure 1: Visualizations of the reduced parameter space after GPU computations.

$$[0.2265, 0.4122] \times [1.7187, 2] \times [0.1109, 0.2024],$$
$$[0.2575, 0.4122] \times [1.4375, 1.7188] \times [0.1109, 0.1718],$$
$$[0.2884, 0.4122] \times [1.2187, 1.4375] \times [0.0804, 0.1415],$$

the first of which is a small enclosure of the original parameters used for synthesizing our measurements (see Section 4.1).

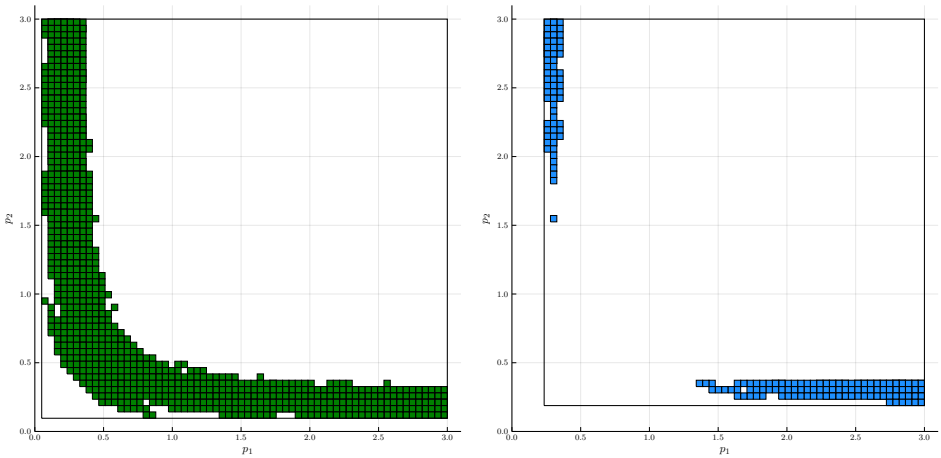
At this scale, preferring interpolated measurement data over raw data points adds some computational overhead, despite global memory access being a performance bottleneck. Furthermore, it can be observed that Euler's method is not suitable for this task. In the case of PSIVIA, the time horizon of 16 steps is long enough for the wrapping effect to significantly impact the resulting enclosure, leading to intervals so large in width that the criterion $\Phi_{app}(\mathbf{p}_i) \subset \mathbf{z}$ cannot be satisfied anymore. As a result, all suitable boxes belong to the boundary set $\mathcal{E}$, while $\mathcal{S}$ remains empty.

Considering a wider search space $\mathbf{p} = [0.05, 3] \times [0.05, 3] \times [0.01, 2]$ with $w$ unchanged, the tests were repeated. Results are shown in Table 3.
Depending on the method used, we observe different increases in execution time. While the experimental approach is more than two times slower than before, both

Table 3: Test results for a larger parameter domain $\mathbf{p} = [0.05, 3] \times [0.05, 3] \times [0.01, 2]$.

| | Method | $\mathbf{p}^*$ | Boxes | $t_{kern}$ (s) | $t_{wall}$ (s) | $P_{peak}$ (W) |
|---|---|---|---|---|---|---|
| GPU | Ia | $[0.05, 3] \times [0.0960, 3] \times [0.0410, 0.2899]$ | 1 420 | 0.1369 | 0.9377 | 60.5923 |
| | IIa | $[0.05, 3] \times [0.05, 3] \times [0.01, 2]$ | 182 603 | 90.0465 | 156.2434 | 95.1462 |
| | IIIa | $[0.05, 3] \times [0.05, 3] \times [0.01, 2]$ | 102 618 | 0.0222 | 1.5053 | 35.5917 |
| CPU | IVa | $[0.05, 3] \times [0.05, 3] \times [0.01, 2]$ | 102 618 | 4.5162 | 4.6874 | 60.91 |

the monotonicity test and PSIVIA need approximately four times of the previous kernel time. However, this larger search space highlights the advantage of using the GPU; now the CPU implementation is significantly slower than both methods I and III. Another factor to take into account is that $\mathbf{p}$ now contains a symmetric solution [14]. In Figure 2, the effectiveness of method III compared to I is highlighted at this scale (under the assumption that only $\mathcal{S}$ and not the boundary $\mathcal{E}$ is considered).




(a) $\mathbf{p}^*_{\mathrm{Ia}}$ (b) $\mathcal{S}_{\mathrm{IIIa}}$

Figure 2: Visual comparison of methods I and III for a larger parameter domain (top view of $p_1$ and $p_2$).

# 5 Conclusions

In this paper, we demonstrated — for the first time — a solid combination of IA with AD capabilities in C++ employed on a graphics processor, using enhanced public domain software. Furthermore, we applied an experimental set-based and two actually verified methods to a well-known example of a two-compartment problem and compared them wrt. computational cost, both in terms of execution time and approximate power consumption, as well as wrt. the quality of the resulting convex hull of the parameter enclosures. In regard to the rising interest in energy-efficient software design and "green computing", the comparison of power consumption is of particular interest. The PSIVIA method seems to be a good compromise between the power consumption and solution quality, even outperforming other considered methods if only the definite solution set $\mathcal{S}$ is taken into account. Using a data reduction method, although easier to implement, could not be shown to lead to better run times in case of the simple test scenario we considered. However, we expect it to be so for more complex examples. Likewise, using the Euler approximation

did not bring any significant reduction of the initial search space in the considered scenario, the cause of which was that the data sampling time of $h = 1$s was too large compared to the dominant time constants of the process. Nonetheless, we expect it to be helpful in more complex scenarios with appropriate sampling times. Finally, we succeeded in obtaining a tight enclosure of the optimum on the GPU.

It remains to be seen how the methods presented in this paper scale up when applied to more complex and close-to-life problems. On the one hand, good performance of our GPU-based set-inversion approach indicates that this type of hardware might allow algorithms like SIVIA to solve problems of higher dimensionality than currently possible. On the other hand, we aim to improve the currently low performance of interval-based AD on the GPU, because our ultimate goal is to implement a GPU-enabled verified solver for differential equations, for which the ability to perform set-based AD is essential.

Our results indicate that AD at run time might not be an ideal concept for the GPU, because it requires every kernel to produce essentially the same derivative, which is expensive computationally. A more elegant approach would be to compute a function's derivatives once and then make them available to all GPU kernels, which we plan to test in our future work. Even better results might be achieved by evaluating derivatives beforehand during compilation. Finally, after an extensive phase of testing the introduced (and extended) GPU-based approaches, we plan to apply the most promising ones in the context of battery systems and fuel cells. To improve our currently purely software-based testing process, we plan to include hardware-based measurement tools for higher accuracy as well as further comparison criteria to assess the performance of our algorithms.

## References

[1] Auer, E., Kiel, S., and Rauh, A. Verified parameter identification for solid oxide fuel cells. In *Proceedings of the 5th International Conference on Reliable Engineering Computing*, pages 41–55, 2012. URL: https://rec2012.fce.vutbr.cz/documents/papers/auer.pdf.

[2] Auer, E., Rauh, A., and Kersten, J. Experiments-based parameter identification on the GPU for cooperative systems. *Journal of Computational and Applied Mathematics*, 371:112657, 2020. DOI: 10.1016/j.cam.2019.112657.

[3] Bagóczki, Z. and Bánhelyi, B. A parallel interval arithmetic-based reliable computing method on a GPU. *Acta Cybernetica*, 23(2):491–501, 2017. DOI: 10.14232/actacyb.23.2.2017.4.

[4] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: A survey. *Journal of Marchine Learning Research*, 18:5595–5637, 2017. DOI: 10.5555/3122009.3242010.

[5] Beck, P.-D. and Nehmeier, M. Parallel interval Newton method on CUDA. In *Proceedinngs of the 11th International Conference on Applied Parallel and*

*Scientific Computing*, pages 454–464. Springer, Berlin, 2013. DOI: 10.1007/978-3-642-36803-5_34.

[6] Bendtsen, C. and Stauning, O. FADBAD, a flexible C++ package for automatic differentiation. Technical report, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, 1996. URL: https://www2.imm.dtu.dk/~kajm/FADBAD/tr17_96.pdf.

[7] Collange, C., Daumas, M., and Defour, D. Interval arithmetic in CUDA. In Hwu, W. W., editor, *GPU Computing Gems Jade Edition*, chapter 9, pages 99–107. Elsevier, 2012. DOI: 10.1016/b978-0-12-385963-1.00009-5.

[8] Eriksen, M. B. and Rasmussen, S. *GPU accelerated parameter estimation by global optimization using interval analysis*. Master's thesis, Aalborg University, 2013. URL: https://projekter.aau.dk/projekter/files/77291483/report.pdf.

[9] Fan, H., Ferianc, M., and Luk, W. Enabling fast uncertainty estimation: Accelerating Bayesian transformers via algorithmic and hardware optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 325—330, New York, NY, 2022. Association for Computing Machinery. DOI: 10.1145/3489517.3530451.

[10] González-Arribas, D., Sanjurjo-Rivo, M., and Soler, M. Multiobjective optimisation of aircraft trajectories under wind uncertainty using GPU parallelism and genetic algorithms. *Computational Methods in Applied Sciences*, 2018. DOI: 10.1007/978-3-319-89890-2_29.

[11] Grimmer, M. and Krämer, W. An MPI extension for the use of C–XSC in parallel environments, 2005. Preprint 2005/3, Universität Wuppertal. URL: https://www2.math.uni-wuppertal.de/wrswt/preprints/prep_05_3.pdf.

[12] Heimlich, O. Interval arithmetic in GNU Octave. In *SWIM 2016: 9th Summer Workshop on Interval Methods*, 2016. URL: https://swim2016.sciencesconf.org/data/SWIM2016_book_of_abstracts.pdf.

[13] Ifrim, I., Vassilev, V., and Lange, D. J. GPU accelerated automatic differentiation with Clad. In *Journal of Physics: Conference Series*, page 012043. IOP Publishing, 2023. DOI: 10.1088/1742-6596/2438/1/012043.

[14] Jaulin, L., Kieffer, M., Didrit, O., and Walter, E. *Applied Interval Analysis*. Springer, London, 2001. DOI: 10.1007/978-1-4471-0249-6.

[15] Jaulin, L. and Walter, E. Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis. *Mathematics and Computers in Simulation*, 35(2):123–137, 1993. DOI: 10.1016/0378-4754(93)90008-I.

[16] Jerrell, M. E. Automatic differentiation and interval arithmetic for estimation of disequilibrium models. *Computational Economics*, 10:295–316, 1997. DOI: 10.1023/A:1008633613243.

[17] Kieffer, M., Markót, M. C., Schichl, H., and Walter, E. Verified global optimization for estimating the parameters of nonlinear models. In *Modeling, Design, and Simulation of Systems with Uncertainties*. Springer, Berlin, 2011. DOI: 10.1007/978-3-642-15956-5_7.

[18] Kieffer, M. and Walter, E. Interval analysis for guaranteed nonlinear parameter estimation. In *MODA 5 — Advances in Model-Oriented Data Analysis and Experimental Design*, pages 115–125, Heidelberg, 1998. Physica-Verlag. DOI: 10.1007/978-3-642-58988-1_13.

[19] Kiel, S., Auer, E., and Rauh, A. Uses of GPU powered interval optimization for parameter identification in the context of SO fuel cells. In *Proceedings of the 9th IFAC Symposium on Nonlinear Control Systems*, pages 558–563. International Federation of Automatic Control, 2013. DOI: 10.3182/20130904-3-FR-2041.00169.

[20] Klatte, R., Kulisch, U., Wiethoff, A., Lawo, C., and Rauch, M. *C–XSC, A C++ Class Library for Extended Scientific Computing*. Springer, Berlin, 1993. DOI: 10.1007/978-3-642-58058-1.

[21] Kozikowski, G. and Kubica, B. J. Interval arithmetic and automatic differentiation on GPU using OpenCL. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, pages 489–503. Springer, Berlin, 2013. DOI: 10.1007/978-3-642-36803-5_37.

[22] Lang, J. and Rünger, G. High-resolution power profiling of GPU functions using low-resolution measurement. In *Proceedings of the 19th International Conference on Parallel Processing*, pages 801–812. Springer, Berlin, 2013. DOI: 10.1007/978-3-642-40047-6_80.

[23] Marvel, S. W., de Luis Balaguer, M. A., and Williams, C. M. Parameter estimation in biological systems using interval methods with parallel processing. In *Proceedings of the 8th International Workshop on Computational Systems Biology*, pages 129–132, 2011.

[24] Matsumoto, M. and Nishimura, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. DOI: 10.1145/272991.272995.

[25] Meslem, N. and Ramdani, N. Interval observer design based on nonlinear hybridization and practical stability analysis. *International Journal of Adaptive Control and Signal Processing*, 25(3):228–248, 2011. DOI: 10.1002/acs.1208.

[26] Moore, R. E. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[27] Moore, R. E., Kearfott, R. B., and Cloud, M. J. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009. DOI: 10.1137/1.9780898717716.

[28] Pilarek, M. and Wyrzykowski, R. Solving systems of interval linear equations in parallel using multithreaded model and "interval extended zero" method. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics*, pages 206–214. Springer, Berlin, 2012. DOI: 10.1007/978-3-642-31464-3_21.

[29] Rebner, G. and Beer, M. CUDA accelerated fault tree analysis with C–XSC. In *Scalable Uncertainty Management*, pages 539–549. Springer, Berlin, 2012. DOI: 10.1007/978-3-642-33362-0_41.

[30] Rump, S. M. *INTLAB — INTerval LABoratory*. In *Developments in Reliable Computing*, pages 77–104. Springer, Dordrecht, 1999. DOI: 10.1007/978-94-017-1247-7_7.

[31] Sanders, D. P. and Churavy, V. Branch-and-bound interval methods and constraint propagation on the GPU using Julia. In *Proceedings of the 19th International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations*, page 79, Szeged, Hungary, 2021. URL: https://www.inf.u-szeged.hu/scan2020/sites/default/files/scan2020_proceedings.pdf.

[32] Smith, H. L. *Monotone Dynamical Systems: An Introduction to the Theory of Competitive and Cooperative Systems*. Mathematical Surveys and Monographs. American Mathematical Society, Providence, RI, 1995. DOI: 10.1090/surv/041.

[33] Tucker, W. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press, 2011. DOI: 10.1515/9781400838974.

[34] Wojtkiewicz, S. F. and Wojtkiewicz, G. Use of GPU computing for uncertainty quantification in computational mechanics: A case study. *Scientific Programming*, 19(4):199–212, 2011. DOI: 10.3233/SPR-2011-0328.

[35] Zimmer, M. Using C–XSC in a multi-threaded environment, 2011. Preprint 2011/2, Universität Wuppertal. URL: https://www2.math.uni-wuppertal.de/wrswt/preprints/prep_11_2.pdf.