Towards a Set-Based IVP Solver for ODEs on the GPU

Ekaterina Auer
University of Applied Sciences Wismar, ekaterina.auer@hs-wismar.de
Lorenz Gillner
University of Applied Sciences Wismar, lorenz.gillner@hs-wismar.de
Julien Alexandre dit Sandretto
ENSTA, Institut Polytechnique de Paris, julien.alexandre-dit-sandretto@ensta.fr

Abstract. Reachability analysis is a key component of (formal) verification techniques for cyber-physical systems subject to uncertainty, as it enables the computation of sets of states that the system can reach over time. However, such methods are often computationally expensive, making algorithmic and implementation-level optimizations essential. In this paper, we propose a verified interval-based ODE solver for reachability analysis, based on the classical fourth-order Runge–Kutta scheme and designed to execute entirely within a single GPU kernel. This enables efficient subdivision strategies to mitigate overestimation while supporting the parallel propagation of a large number of subboxes. Using the Van der Pol oscillator as a benchmark, we demonstrate the applicability of the proposed approach and compare its performance with non-verified GPU-based uncertainty propagation techniques derived from Müller-type theorems, as well as with the CPU-based reference tool DynIbex.

Keywords: result verification, ODE, Runge-Kutta schemes, GPU

1. Introduction

Verification and uncertainty quantification play a central role in ensuring the reliability of computational models. However, these processes can be very time-consuming, especially for computationally demanding problems. One natural way to accelerate computations is through high-performance computing or parallelization.

GPUs are highly specialized programmable units which are widely used in parallel scientific computing today. They offer an affordable alternative to supercomputers—provided that the problem fits the stream processing model, where the same kernel is applied to different data to produce independent results. Since GPU architectures differ significantly from CPUs, existing CPU-oriented software generally cannot be ported to GPUs without substantial redesign.

In (Auer et al., 2025b), we investigated GPU-based acceleration for parameter identification using least squares and global optimization:

$$F(\mathbf{p}) = \sum_{k=T_b}^{T_e} \sum_{j=1}^{n_m} \left(y_j(t_k, \mathbf{p}) - y_{j,m}(t_k)\right)^2 \overset{\mathbf{p}\in[\mathbf{p}]}{\longrightarrow} \min \ ,$$

where $t \in [T_b, T_e]$, $y_j(t_k, \mathbf{p})$ denotes, at time t_k, the j-th component of the solution to the initial value problem (IVP) for the ordinary differential equations (ODE) of the form

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}(t)) \tag{1}$$

with $\mathbf{y}(t_0) \in [\mathbf{y}_0] \in \mathbb{IR}^{n_y}$, and $y_{j,m}(t_k)$ the corresponding measured value among n_m measurements[1]. The aim is to enclose parameter values $\mathbf{p}$ that achieve the global minimum of $F(\mathbf{p})$ over the search space $[\mathbf{p}]$. Once $[\mathbf{p}]$ is subdivided into subdomains $[\mathbf{p_1}], \ldots, [\mathbf{p}_\kappa]$, the problem exhibits massive data parallelism suitable for GPU evaluation.

When such properties as cooperativity (Hirsch and Smith, 2005) cannot be exploited or a mathematical formula for the exact solution does not exist, computing enclosures of $\mathbf{y}(t, \mathbf{p})$ requires a set-based IVP solver on the GPU. In (Auer et al., 2017), we introduced a method for computing a rough (a priori) enclosure on the GPU, a necessary first step toward a full set-based solver as explained in Section 2.

Our goal in this paper is therefore to develop a verified ODE solver that operates entirely within a single GPU kernel. Such a solver is particularly well suited for parameter identification in dynamical models via brute-force global optimization. It also offers an effective way to reduce overestimation: bounded uncertainties in the initial conditions $[\mathbf{y}_0]$ or in other parameters can be subdivided into smaller interval boxes, whose flows under Eq. (1) are computed in parallel on the GPU. Taking the hull of these flows then yields a tighter enclosure of the solution, as illustrated in Section 4.

The paper is structured as follows. Section 2 reviews the theoretical foundations of verified solution methods for IVPs and summarizes Müller-type theorems for bounding the corresponding ODE flow. Section 2.2 is of particular importance, as it introduces the proposed GPU-based verified method. It is built upon the classical fourth-order Runge–Kutta scheme, with local truncation errors rigorously estimated using the Butcher series expansion. In Section 3, we briefly review the tools required for guaranteed computations, in particular for interval arithmetic and for its use with automatic differentiation. These components are essential prerequisites for a verified GPU implementation. In Section 4, we present a comparative evaluation of the proposed solver using the Van der Pol oscillator as a benchmark. We compare the performance of its different GPU implementations against non-verified GPU-based uncertainty propagation techniques based on Müller-type theorems and against the established verified CPU-based tool DynIbex (Alexandre dit Sandretto and Chapoutot, 2016). Finally, Section 5 summarizes the main results and outlines directions for future research.

2. Verifying the Flow of an Initial Value Problem

ODEs of the form in Eq. (1) are among the most classical continuous differential equations. They possess a unique solution $\mathbf{y}(t)$ provided certain conditions are satisfied, for example, if $\mathbf{f}$ is Lipschitz continuous. In general, closed-form solutions are not available, so that numerical integration schemes must be used to approximate the state of the system over time starting from the initial condition

[1] In the following, we assume that $t_0 = 0$

$\mathbf{y}(0) = \mathbf{y}_0$. The integral formulation

$$\mathbf{y}(t) = \mathbf{y}_0 + \int_0^t \mathbf{f}(\mathbf{y}(s))\, ds, \tag{2}$$

does not in fact represent a solution but a true reformulation because the unknown function $\mathbf{y}(\cdot)$ appears both on the left-hand side and inside the integral on the right-hand side.

A widely used class of numerical integrators for ODEs is given by Runge–Kutta schemes, which can be selected according to the characteristics of the problem and the desired level of accuracy. However, when uncertainty must be handled rigorously or result guarantees are required, standard numerical approximations are insufficient.

To address this, a number of verified IVP solvers have been developed for CPUs over the past decades, including VNODE (Nedialkov et al., 1999) and DynIbex (Alexandre dit Sandretto and Chapoutot, 2016). Although these solvers rely on different underlying techniques, they share a common approach to verification, which we outline in Subsection 2.1. We then present the verified fourth-order Runge–Kutta scheme implemented on the GPU (Subsection 2.2), and finally discuss non-verified approaches capable of handling uncertainty (Subsection 2.3).

2.1. General Verification Scheme

The main approach to verifying a numerical integration method, as described in (Nedialkov et al., 1999), is to structure each step of the integration scheme into two phases.

Phase 1: Compute an *a priori* enclosure $[\tilde{\mathbf{y}}_j]$ of the solution such that

- $\mathbf{y}(t; [\mathbf{y}_j])$ is guaranteed to exist for all $t \in [t_j, t_{j+1}]$, that is, over the whole time interval at the current step, and for all $\mathbf{y}_j \in [\mathbf{y}_j]$;
- $\mathbf{y}(t; [\mathbf{y}_j]) \subseteq [\tilde{\mathbf{y}}_j]$ for all $t \in [t_j, t_{j+1}]$.

If adaptive step-size control is employed, the step size $h_j = t_{j+1} - t_j > 0$ is chosen as large as possible while preserving both the existence guarantee and the desired enclosure tightness. Alternatively, for a fixed step size h_j, the algorithm terminates if no suitable interval $[\tilde{\mathbf{y}}_j]$ can be found.

Phase 2: Compute a tighter enclosure of $[\mathbf{y}_{j+1}]$ at time t_{j+1} such that $\mathbf{y}(t_{j+1}, [\mathbf{y}_j]) \subseteq [\mathbf{y}_{j+1}]$.

2.1.1. *A priori enclosure*

Consider the space of continuously differentiable functions $\mathcal{C}^1([t_j, t_{j+1}], \mathbb{R}^n)$ and the Picard-Lindelöf operator

$$\mathbf{p}_\mathbf{f}(\mathbf{y}) = t \mapsto \mathbf{y}_j + \int_{t_j}^t \mathbf{f}(\mathbf{y}(s)) ds \ , \tag{3}$$

where $\mathbf{y}_j$ is the true solution $\mathbf{y}(t_j)$ at time t_j. We can define a simple enclosure function for the right-hand side of the mapping in Eq. (3) such that

$$[\mathbf{p}_\mathbf{f}]([\mathbf{r}]) \overset{\text{def}}{=} [\mathbf{y}_j] + [0, h_j] \cdot \mathbf{f}([\mathbf{r}]) \ . \tag{4}$$

Consequently, if one can find an interval $[\mathbf{r}]$ such that $[\mathbf{p_f}]([\mathbf{r}]) \subseteq [\mathbf{r}]$, then $[\tilde{\mathbf{y}}_j] \subseteq [\mathbf{r}]$ by the Banach fixed-point theorem. That is, we want to find such h_{max}, ρ_{min} and $[\mathbf{r}] := [\mathbf{y}_j - \rho_{min}, \mathbf{y}_j + \rho_{min}]$ that

$$[\mathbf{p_f}]([\mathbf{r}]) = [\mathbf{y}_j] + [0, h_{max}] \cdot \mathbf{f}([\mathbf{r}]) \subseteq [\mathbf{r}] \ . \tag{5}$$

This enclosure is rather coarse, as the true solution is generally not constant over the interval $[t_j, t_j + h_{max}]$. Moreover, when the enclosures $[\tilde{\mathbf{y}}_{j+1}]$, $[\tilde{\mathbf{y}}_{j+2}]$, ... are computed successively from $[\tilde{\mathbf{y}}_j]$, $[\tilde{\mathbf{y}}_{j+1}]$, ... using Eqs. (4)–(5), their widths typically grow monotonically, irrespective of the true behavior of the solution. Higher-order schemes (e.g., those based on Taylor series decomposition) can mitigate this effect to some extent, but they cannot eliminate the resulting overestimation completely.

2.1.2. *Tighter enclosure*
After Phase 1, the a priori enclosure $[\tilde{\mathbf{y}}_j]$ is obtained such that

$$\mathbf{y}(t; [\mathbf{y}_j]) \subset [\tilde{\mathbf{y}}_j] \quad \forall t \in [t_j, t_{j+1}] \ .$$

In particular, we have $\mathbf{y}(t_{j+1}; [\mathbf{y}_j]) \subset [\tilde{\mathbf{y}}_j]$. The goal of Phase 2 is now to compute a tighter enclosure $[\mathbf{y}_{j+1}]$ at time t_{j+1} such that

$$\mathbf{y}(t_{j+1}; [\mathbf{y}_j]) \subset [\mathbf{y}_{j+1}] \subseteq [\tilde{\mathbf{y}}_j] \ .$$

Any formula for improving the a priori enclosure can be seen as consisting of two parts:

- the *approximation part* $\Phi(t, [\mathbf{y}_j]) \approx \mathbf{y}(t; [\mathbf{y}_j])$. It usually follows a formula for a traditional numerical integration method such as Runge-Kutta or a direct Taylor series decomposition.
- the *local truncation error part* $\mathrm{LTE}_\Phi(t, \mathbf{y}, [\mathbf{y}_j])$. It is intended to provide the formula for the distance (enclosure) between the exact solution and the approximate solution produced by the approximation part:

$$\mathrm{LTE}_\Phi(t, \mathbf{y}, [\mathbf{y}_j]) \stackrel{\text{def}}{=} \mathbf{y}(t; [\mathbf{y}_j]) - \Phi(t, [\mathbf{y}_j]) \ . \tag{6}$$

A verified numerical integration method has the following property:

$$\begin{aligned} \exists \xi \in]t_j, t_{j+1}[, \quad \mathbf{y}(t_{j+1}; [\mathbf{y}_j]) \ &= \ \Phi(t_{j+1}, [\mathbf{y}_j]) + \mathrm{LTE}_\Phi(\xi, \mathbf{y}, [\mathbf{y}_j]) \\ &\subset \ \Phi(t_{j+1}, [\mathbf{y}_j]) + \mathrm{LTE}_\Phi([t_j, t_{j+1}], [\tilde{\mathbf{y}}_j], [\mathbf{y}_j]) \ \subset \ [\tilde{\mathbf{y}}_j] \ . \end{aligned}$$

Consequently, the tight enclosure is obtained by evaluating the approximation part at time t_{j+1} and combining it with bounds on the local truncation error derived from the a priori enclosure $[\tilde{\mathbf{y}}_j]$.

2.2. A Verified Runge-Kutta Scheme for a Tighter Enclosure

Let us consider the well-known and widely used fourth-order method from the Runge–Kutta family of explicit integration schemes. It provides an accurate and computationally efficient approximation of the solution of an IVP to an ODE by evaluating the vector field at several intermediate stages within each time step. The method achieves fourth-order accuracy, meaning that the local truncation error scales as $\mathcal{O}(h^5)$ and the global error as $\mathcal{O}(h^4)$ for a step size h. The scheme is defined by

the Butcher tableau given below on the left, which specifies the intermediate stage coefficients and the weights used to combine them into the final update. Eq. (7) on the right shows how the stage values are computed for a current state $\mathbf{y}_n \approx \mathbf{y}(t_n)$ and step size h. The next approximation of the solution is then obtained by the weighted combination in Eq. (8).

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \qquad \begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_n), \\ \mathbf{k}_2 &= \mathbf{f}\big(\mathbf{y}_n + \tfrac{h}{2}\mathbf{k}_1\big), \\ \mathbf{k}_3 &= \mathbf{f}\big(\mathbf{y}_n + \tfrac{h}{2}\mathbf{k}_2\big), \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{y}_n + h\mathbf{k}_3). \end{aligned} \tag{7}$$

$$\begin{aligned} \mathbf{y}_{n+1} &= \Phi_4(t_{n+1}, \mathbf{y}_n) \\ &= \mathbf{y}_n + \frac{h}{6}\Big(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\Big) \end{aligned} \tag{8}$$

The LTE of the this Runge-Kutta scheme can be obtained by comparing the Butcher-series representation of the exact solution with that of the numerical method, as developed in (Butcher, 2021) via the associated rooted-tree formalism. We use the following multilinear derivative notation:

- $f'(\mathbf{y}) \in \mathbb{R}^{n_y \times n_y}$ denotes the Jacobian of $\mathbf{f}$ at $\mathbf{y}$.
- $f''(\mathbf{y})[\mathbf{u}, \mathbf{v}]$ denotes the bilinear application of the Hessian tensor.
- $f^{(k)}(\mathbf{y})[\mathbf{v}_1, \ldots, \mathbf{v}_k]$ denotes the k-th order Fréchet derivative.

Here, $f'(\mathbf{y})\,\mathbf{v} := J_f(\mathbf{y}) \cdot \mathbf{v}$, $f''(\mathbf{y})[\mathbf{u}, \mathbf{v}] := \sum_{i,j} \dfrac{\partial^2 \mathbf{f}}{\partial y_i \partial y_j}(\mathbf{y})\, u_i v_j$. Expressions such as $f' \cdot [f' \cdot [f' \cdot \mathbf{f}]]$ are interpreted as repeated Jacobian-vector products: $J_f(\mathbf{y})(J_f(\mathbf{y})(J_f(\mathbf{y})\,\mathbf{f}(\mathbf{y})))$. Similarly, $f''[\mathbf{f}, \mathbf{f}]$ denotes a Hessian contraction in two identical directions: $f''(\mathbf{y})[\mathbf{f}(\mathbf{y}), \mathbf{f}(\mathbf{y})]$.

The LTE can be written as a linear combination of rooted-tree (Butcher) contributions. For the specific method from this section, this linear combination becomes (with $h = t_{n+1} - t_n$)

$$\begin{aligned} &\mathrm{LTE}_4([t_n, t_{n+1}], [\tilde{\mathbf{y}}_n], [\mathbf{y}_n]) := \\ &h^5 \Big(-\tfrac{1}{120} f' \cdot [f' \cdot [f' \cdot [f' \cdot \mathbf{f}]]] + \tfrac{1}{480} f' \cdot \Big[f' \cdot \Big[f'' \cdot [\mathbf{f}, \mathbf{f}] \Big] \Big] \\ &-\tfrac{1}{240} f' \cdot \Big[f'' \cdot [f' \cdot \mathbf{f},\, \mathbf{f}] \Big] + \tfrac{1}{120} f'' \cdot \Big[f' \cdot [f' \cdot [f' \cdot \mathbf{f}]],\, \mathbf{f} \Big] \\ &-\tfrac{1}{720} f' \cdot [f^{(3)} \cdot [\mathbf{f}, \mathbf{f}, \mathbf{f}]] - \tfrac{1}{480} f'' \cdot [f'' \cdot [\mathbf{f}, \mathbf{f}],\, \mathbf{f}] + \tfrac{1}{160} f'' \cdot \Big[f' \cdot \mathbf{f},\, f' \cdot \mathbf{f} \Big] \\ &+\tfrac{1}{480} f^{(3)} \cdot \Big[f' \cdot \mathbf{f},\, \mathbf{f},\, \mathbf{f} \Big] + \tfrac{1}{2880} f^{(4)} \cdot \Big[\mathbf{f},\, \mathbf{f},\, \mathbf{f},\, \mathbf{f} \Big] \Big) \quad . \end{aligned} \tag{9}$$

Although the formula is written as a nested expression, it should not be evaluated in a naive manner. Instead, it corresponds to a directed acyclic graph (DAG) of tensor contractions (Mullier et al., 2018), where the nodes represent intermediate quantities such as $\mathbf{f}$, $f' \cdot \mathbf{f}$, $f'' \cdot [\mathbf{f}, \mathbf{f}]$, ...; the edges represent dependency relations between contractions. Thus, an efficient algorithm is necessary to compute subexpressions only once and to reuse them (*common subexpression elimination*, CSE).

Direct evaluation of nested expressions in Eq. (9) leads to repeated computation of identical Jacobian and Hessian contractions. Moreover, we cannot represent tensors (especially higher-order derivatives) in their "natural" explicit form as multidimensional arrays on the GPU since this would be expensive in both storage and access cost. Even if we consider flat vectors only, storing all

intermediate results simultaneously may reduce GPU occupancy. Another challenge is the computation of Fréchet derivatives up to fourth order, ideally via automatic differentiation (AD). While AD techniques based on repeated application of the chain rule have been studied extensively (Griewank and Walther, 2000), tracing-based methods commonly used in verified IVP solvers are difficult to implement efficiently on GPUs due to memory and control-flow constraints (cf. Section 3.2). In particular, forward-mode tracing (e.g., FADBAD++ (Bendsten and Stauning, 1996)) can compute Fréchet derivatives implicitly, but may introduce overhead compared to optimized contraction sequences. Instead of runtime evaluation, possibly with AD, it is therefore preferable to transform the expression at compile time into an optimized sequence of tensor contractions. In this work, we present a basic GPU implementation of this approach for the LTE, while extending it to AD remains future work. Automatic code generation for the LTE has been demonstrated on CPUs (Alexandre dit Sandretto, 2025), but without GPU-specific optimizations, and extending such methods to GPUs remains challenging due to differences in execution and memory models.

The final (naive) verified Runge-Kutta method (denoted by RK4) relies on the following formula:

$$[\mathbf{y}_{n+1}] := \Phi_4(t_{n+1}, [\mathbf{y}_n]) + \mathrm{LTE}_4([t_n, t_{n+1}], [\tilde{\mathbf{y}}_n], [\mathbf{y}_n]) \ . \tag{10}$$

When using conventional interval arithmetic (as opposed to, e.g., affine arithmetic as implemented in DynIbex), the method becomes more susceptible to the wrapping effect. Various strategies have been proposed in the CPU context to mitigate this issue, including approaches based on the mean value theorem (MVT) and coordinate transformations such as QR decomposition (Nedialkov et al., 1999). To reduce the wrapping effect while maintaining efficiency on GPUs, one must balance its mitigation against the computational cost of additional work (AD and coordinate transformations). In the our implementation, we incorporate a basic MVT-based step for Φ_4:

$$[\mathbf{y}_{n+1}] := \Phi_4(t_{n+1}, \hat{\mathbf{y}}_n) + J_{\Phi_4}([\mathbf{y}_n])([\mathbf{y}_n] - \hat{\mathbf{y}}_n) + \mathrm{LTE}_4([t_n, t_{n+1}], [\tilde{\mathbf{y}}_n], [\mathbf{y}_n]) \ , \tag{11}$$

with $\hat{\mathbf{y}}_n$ chosen as the midpoint of $[\mathbf{y}_n]$ and J_{Φ_4} being the Jacobean of Φ_4, which we compute using our GPU version of FADBAD++ (cf. Section 3.2). We denote this method by MRK4 in the following.

Note that any of the described methods can be naturally combined with a subdivision strategy on the GPU. A classical branch-and-bound (brute-force) approach proceeds as follows:

1. **Subdivide:** Decompose the parameter box $[\mathbf{p}]$ into subboxes $[\mathbf{p}] = \bigcup_{j=1}^{\ell} [\mathbf{p}_j]$, where $[\mathbf{p}]$ represents an interval vector of model parameters (CPU or GPU execution).
2. **Evaluate:** For each subbox $[\mathbf{p}_j]$, integrate the system in Eq. (1) from t_0 to t_{end} using a suitable method (e.g., RK4 or MRK4), running all ℓ subproblems in parallel on the GPU.
3. **Combine:** Compute an enclosure of the reachable set by taking the interval hull of the results, $\mathrm{hull}\left(\bigcup_{j=1}^{\ell} [\mathbf{y}_{end,j}]\right)$, which can be carried out on either the CPU or GPU.

2.3. Cooperative Systems and Monotone Inclusions

If the objective is to propagate uncertainty in the initial conditions or other parameters through a dynamical system in a reliable manner, a non-verified solver may be combined with suitable monotonicity properties of the system. The property of *cooperativity* (or *competitivity*) holds for the system in Eq. (1) if

$$\frac{\partial f_i}{\partial y_j} \geq 0 \ \left(\text{or } \frac{\partial f_i}{\partial y_j} \leq 0 \right) \text{ for all } i \neq j,\ i,j = 1 \dots n_y \ . \tag{12}$$

Müller's theorem (Meslem and Ramdani, 2011), combined with the first property in Eq. (12) and positivity of y_j, yields Smith's theorem (Hirsch and Smith, 2005). It shows that the uncertainty in a cooperative system can be quantified by solving two systems with crisp parameters instead of one system with uncertain parameters:

$$\mathbf{y}'_{\text{lb}} = \underline{\mathbf{f}}(\mathbf{y}_{\text{lb}}) \quad \text{and} \quad \mathbf{y}'_{\text{ub}} = \overline{\mathbf{f}}(\mathbf{y}_{\text{ub}}) \tag{13}$$

(two bracketing systems). For cooperative (or competitive) systems, these are decoupled and can be solved in parallel. If the system is not monotone, both $\underline{\mathbf{f}}$ and $\overline{\mathbf{f}}$ may depend on $\mathbf{y}_{\text{lb}}$ and $\mathbf{y}_{\text{ub}}$, leading to a coupled system of dimension $2n_y$:

$$\mathbf{y}'_{\text{lb}} = \underline{\mathbf{f}}(\mathbf{y}_{\text{lb}}, \mathbf{y}_{\text{ub}}) \quad \text{and} \quad \mathbf{y}'_{\text{ub}} = \overline{\mathbf{f}}(\mathbf{y}_{\text{lb}}, \mathbf{y}_{\text{ub}}) \tag{14}$$

The bounding functions $\underline{\mathbf{f}}$ and $\overline{\mathbf{f}}$ can be obtained via interval arithmetic, though often conservatively. A tighter alternative is to compute global extrema of the right-hand side of (1), which is generally difficult due to nonlinear optimization.

The true solution $\mathbf{y}(t)$ lies in the interval $[\underline{\mathbf{y}}_{\text{lb}}(t), \overline{\mathbf{y}}_{\text{ub}}(t)]$. Evaluating such enclosures at each grid point t_k yields a verified flow of (1). A practical approximation is given by $\mathbf{y}_{\text{lb}}^{(k)}$, $\mathbf{y}_{\text{ub}}^{(k)}$ obtained from (13) or (14) using standard floating point arithmetic with sufficiently small step sizes. This captures most of the output uncertainty but is not fully verified; we use monotone inclusions as a baseline for comparison with the verified solver from Section 2.2.

3. Methods with Result Verification on the GPU

As shown in Section 2.2, set-based arithmetics and exact derivatives are required to verify Runge–Kutta schemes. In this section, we describe existing software for performing interval computations on NVIDIA GPUs using CUDA (NVIDIA, 2026). We then review available approaches to AD on GPUs, with particular attention to those compatible with interval arithmetic.

3.1. Interval Analysis on the GPU

Interval arithmetic is well supported on CPUs, where mature libraries such as Boost.Interval (Boost C++ Libraries, 2026) and INTLAB-style systems (Rump, 2026) provide good implementations. In

contrast, GPU support remains limited and is largely driven by research rather than production-ready libraries. Modern NVIDIA GPUs offer IEEE-754 compliant floating point arithmetic and support directed rounding via hardware intrinsics, which are essential for interval methods.

Early CUDA-based interval arithmetic was developed in academic work by Collange et al. (Collange et al., 2011), demonstrating the feasibility of Boost-style approaches on GPUs. However, these efforts have not resulted in currently maintained library. In practice, a common approach is to rely on established CPU interval libraries while adapting them for GPU use through compatible interfaces. For example, Boost.Interval can be employed with policy specializations that enable using the same library on the CPU and the GPU, see our implementation at `https://github.com/lorenzgillner/boost-interval`.

We reuse a simple GPU-based algorithm for a priori enclosure computation from (Auer et al., 2017) in a slightly modified form. While the original method employs adaptive step sizes (cf. Section 2.1.1), we restrict ourselves to constant step sizes to avoid branching and reduce thread divergence. Nevertheless, adaptive strategies may still improve performance by allowing larger steps in regions with mild dynamics. Moreover, we developed a GPU-based global optimization solver for parameter identification problems described in the Introduction, with such applications as for cooperative solid oxide fuel cell systems (Auer et al., 2025b). This problem class has received significant attention in both historically and recently; see, for instance, (Sanders, 2022).

3.2. Automatic Differentiation on the GPU

Recent developments in machine learning have strongly promoted both GPU computing and AD. As a consequence, there are more and more AD tools for the GPU. For successful use on the GPU, AD methods must be fast and fit the synchronous kernel execution. Symbolic computations, especially at runtime, are often not suitable, as they may introduce overhead or irregular execution patterns. Ideally, an AD algorithm should be able to provide the n-th derivative or all derivatives up to order n without requiring n to be fixed in advance. Two major approaches to AD are:

Source code transformation (SCT): Derivatives are generated by manipulating the program code itself. The original source code is analyzed and transformed into new code that explicitly computes the derivatives.

Tracing: Derivatives are computed at runtime using special data structures. The program execution is recorded, and derivative information is propagated according to the chain rule. This approach usually offers somewhat more flexibility with respect to derivative order.

3.2.1. *Source Code Transformation Versus Tracing on the GPU*

The decisive distinction between SCT and tracing in the context of GPU computing concerns the stage at which derivative information is generated and how it is integrated into the execution model. In SCT, derivatives are computed ahead of time, typically during compilation. The source code is transformed to generate additional derivative code, which is then compiled together with the original program. The resulting executable is deployed to the GPU, where multiple worker threads execute it. Since the derivative logic is already embedded, each worker can evaluate derivatives

Table I. Different AD tools based on SCT

Tool	Type	Intermediate rep.	CUDA	Interval support	Reference
Clad	Compiler plugin	Clang AST	yes	no	(Ifrim et al., 2023)
Enzyme	Compiler plugin	LLVM IR	yes	no	(Moses et al., 2022)
Tapenade	Preprocessor	IL Syntax Tree	manual transfer to GPU		(Hascoet and Pascual, 2013)
OpenAD	Preprocessor	Xaif	manual transfer to GPU		(Utke et al., 2008)

immediately at launch without additional runtime construction. Thus, derivative computation is effectively shifted to the compilation phase.

Tracing follows a fundamentally different approach. The source code is compiled without explicitly generating separate derivative functions. During execution on the GPU, derivatives are computed dynamically. Special data structures record operations and propagate derivative information via the chain rule, so each worker both evaluates the function and propagates its derivative at runtime.

Using SCT, derivative routines are embedded directly into the compiled program. Since NVIDIA GPU programming is typically performed in C++ variants (e.g., CUDA) compiled into GPU-specific assembly, transformation mechanisms can be integrated into the compilation pipeline. Compiler toolchains may be extended via preprocessors or plugins that support SCT, enabling derivative generation at compile time. This requires a suitable intermediate representation for symbolic manipulation. However, current tools (cf. Table I) are limited in scope and usually support only basic data types such as integers, floats, and doubles.

3.2.2. *Tracing on the GPU*

When tracing is used on the GPU, derivatives are computed during program execution, introducing computational overhead but offering greater flexibility. In particular, AD can be implemented with custom data types using C++ templates, making tracing-based approaches easily adaptable to intervals. A central challenge for tracing on GPUs is memory usage. Stack memory is limited but fast, whereas heap allocations reside in slower global memory. Since tracing often relies on dynamically allocated structures, it may lead to performance decrease. In this setting, forward mode is generally preferred, as it better fits the limited fast local memory of GPU threads.

These aspects become evident when porting AD libraries from the CPU to the GPU. The C++ library FADBAD++, originally designed for CPUs, offers a choice between stack and heap memory usage in forward mode, while the backwards mode and the Taylor coefficients computation rely solely on heap allocations. Although our GPU implementation of FADBAD++, available at `https://github.com/lorenzgillner/FADBAD-NG`, supports higher-order derivatives in principle, it is not optimized for GPU architectures. As a result, execution is slow and resource-intensive, particularly in heap mode (Auer et al., 2025a).

To address these limitations, we developed a GPU-optimized proof-of-concept implementation called TADA (available at `https://github.com/lorenzgillner/tada`). This approach follows a

generic design tailored to GPU execution. It currently supports first- and second-order derivatives, but not arbitrary orders. An additional advantage is its architectural flexibility: it runs on both CPUs and GPUs and works out of the box with interval data types.

AD on GPUs is currently limited in efficiency and scalability for higher-order derivatives, particularly when interval-valued data types are used. Consequently, in our LTE implementation, we do not employ AD for the involved tensors. The only use of AD in our framework is for the implementation of the MRK4 scheme in Eq. (11), where we utilize our GPU-based version of FADBAD++ in forward mode with a stack-based memory model.

4. Illustration: Van der Pol Equation

The ODE form of the Van der Pol equation (VDP) is given by

$$\begin{bmatrix} \dot{y}_1 = y_2 \\ \dot{y}_2 = -y_1 + \mu y_2 \left(1 - y_1^2\right) \end{bmatrix} \tag{15}$$

It is a well-known example that describes a non-conservative oscillator with non-linear damping, which can become stiff for large values of μ. The system is neither cooperative nor competitive.

In this section, we illustrate possibilities available for set-based solving of IVPs on the GPU using the VDP example with $\mu = 1$. In Subsection 4.1, the options provided by monotone inclusions are highlighted and compared to a CPU-based verified solver DynIbex. In Subsection 4.2, a DynIbex-like implementation of RK4 along with MRK4 for VDP on the GPU is investigated. We conduct all GPU-related tests and simulations in this section using an Intel® Core™ i7-9700 CPU (3.00 GHz) and an NVIDIA GeForce GTX TITAN Black GPU, running on Ubuntu 22.04.5 LTS.

4.1. Monotone Inclusions for the Van der Pol Equation

Müller's theorem allows one to construct order-preserving bounding systems even when the original system is not monotonic (cf. Section 2.3). We can apply Müller's theorem to VDP as given by Eq. (15). Suppose $y_1 \in [\underline{y}_1, \overline{y}_1]$, $y_2 \in [\underline{y}_2, \overline{y}_2]$. The flow of VDP can be bounded by considering the following system:

$$\begin{bmatrix} \dot{\underline{y}}_1 = \underline{y}_2 \\ \dot{\overline{y}}_1 = \overline{y}_2 \\ \dot{\underline{y}}_2 = -\overline{y}_1 + \underline{y}_2 - \overline{P} \\ \dot{\overline{y}}_2 = -\underline{y}_1 + \overline{y}_2 - \underline{P} \end{bmatrix} \tag{16}$$

where $[\underline{P}, \overline{P}] = [\min P, \max P]$ with $P = \{m\underline{y}_2, m\overline{y}_2, M\underline{y}_2, M\overline{y}_2\}$ and

$$M = \max(\underline{y}_1^2, \overline{y}_1^2), \quad m = \begin{cases} 0, & 0 \in [\underline{y}_1, \overline{y}_1] \\ \min(\underline{y}_1^2, \overline{y}_1^2), & \text{otherwise} \end{cases} \tag{17}$$

A tighter construction is possible if we use the two-dimensional VDP form given by Liénard's theorem and find the bounding system for it. The Liénard's form is given below

$$\left[\begin{array}{c} \dot{y}_1 = y_1 - \frac{1}{3}y_1^3 - y_2 \\ \dot{y}_2 = y_1 \end{array}\right] , \tag{18}$$

with the bounding system given by

$$\left[\begin{array}{c} \dot{\underline{y}}_1 = -\overline{y}_2 + \min\left(\underline{y}_1 - \frac{1}{3}\underline{y}_1^3, \overline{y}_1 - \frac{1}{3}\overline{y}_1^3, -\frac{2}{3}\right) \\ \dot{\overline{y}}_1 = -\underline{y}_2 + \max\left(\underline{y}_1 - \frac{1}{3}\underline{y}_1^3, \overline{y}_1 - \frac{1}{3}\overline{y}_1^3, \frac{2}{3}\right) \\ \dot{\underline{y}}_2 = \underline{y}_1 \\ \dot{\overline{y}}_2 = \overline{y}_1 \end{array}\right] \tag{19}$$

because local extrema of $y_1 - \frac{1}{3}y_1^3$ occur at ± 1. Note that the notation above means that the value $-2/3$ is taken into account for the minimum only if $-1 \in [\underline{y}_1, \overline{y}_1]$, and, analogously, the value $2/3$ is considered for the maximum only if $1 \in [\underline{y}_1, \overline{y}_1]$. Furthermore, using the exact extrema of the right-hand side via min and max does not violate the assumptions of Müller's theorem. Indeed, the resulting right-hand side remains locally Lipschitz continuous, which guarantees that the solutions remain well defined and unique.

The simulation results for the VPD system, obtained by monotone inclusions using Thrust and a floating point IVP solver from the Boost library[2] as described in (Auer et al., 2025b), are shown in Figure 1. Only the state y_1 is displayed, as it represents the oscillator flow and is identical in both forms (15) and (18). For comparison, we also show the solution computed on the CPU using DynIbex relying on the verified Runge-Kutta method of order 4 (yellow line) with the VDP form in (18). The initial conditions are $y_1(0) \in [1.9, 2.1]$ and $y_2(0) = 0$ when using (15), and $y_2(0) = -2/3$ when using (18), ensuring that y_1 is the same in both forms. In all cases, the precision of 10^{-11} was used. The blue lines represent solutions obtained for sampled crisp initial conditions within the interval $[1.9, 2.1]$ using the same Boost IVP solver.

We observe that applying subdivision to the first initial condition on the GPU considerably reduces overestimation, particularly when the Liénard form as given by Eq. (19) is used (red line). In this respect, the CPU-based solution provided by DynIbex for the Liénard form in Eq. (18) also yields tight enclosures (yellow line). In contrast, when Müller's theorem is applied directly with interval arithmetic without subdivision, the enclosure grows rapidly, as illustrated by the black line. Even if used with subdivision, interval arithmetic alone does not provide very viable lower and upper bracketing systems (green and orange lines); cf. also Table II.

Note that we report the wall-clock time for DynIbex in Table II only as a reference, since it runs entirely on the CPU, in cases 4 and 5 without parallelization; therefore, a direct comparison of runtimes would not be meaningful. Nevertheless, the results show that the fully verified solver DynIbex performs substantially better for the VDP formulation in Eq. (18) than the approaches based on monotone inclusions with respect to the break-down time, that is, the time t after which the flow enclosure becomes too conservative. Increasing the number of subintervals in the subdivision

[2] the Dormand-Prince method of order 5 with adaptive step size (DP5)

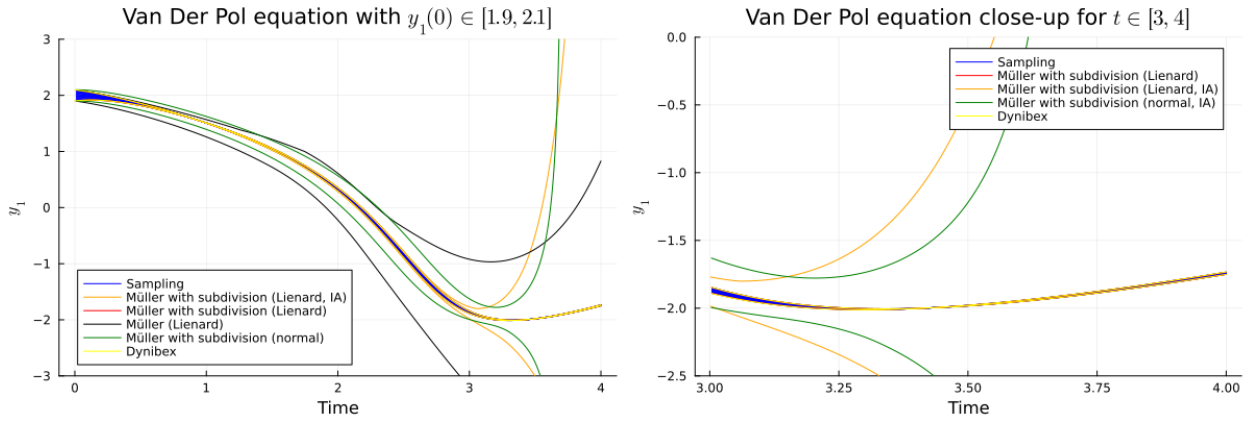

Figure 1. Different types of monotone inclusions for VDP on the GPU in comparison to DynIbex (CPU).

Table II. Subdivision on the GPU (2^{14} boxes) for different types of monotone inclusions for VDP in comparison to DynIbex on the CPU (adaptive step size for all).

		Variant	Kernel (GPU) or real (CPU) time in ms ($t \in [0, 3]$)	Break-down t	Break-down w($[y_1]$)
GPU	1	(19)	85	13.2	≈ 20
	2	(18) + IA for bracketing systems	78	3.79	≈ 13
	3	(15) + IA for bracketing systems	338	3.70	≈ 20
CPU	4	(18) + DynIbex	214,688	> 40	≈ 0.25
	4′	+ parallelization (16 boxes, 8 cores)	64,462	> 40	$\approx 10^{-5}$
	5	(15) + DynIbex	387,869	3.43	≈ 54
	5′	+ parallelization (16 boxes, 8 cores)	114,804	18.22	≈ 56

strategy on the GPU has only a limited effect since the initial uncertainty is relatively small. For example, considering variant 2 from Table II with 2^{20} boxes instead of 2^{14} reduces the width of $[y_1]$ in $t = 3$ from 0.22 to 0.05. However, this improvement comes at a significant computational cost, as the runtime increases to 1312 ms, which is mainly due to the specific block-thread structure of the employed GPU. This suggests that a well-designed verified implementation using techniques for enclosure tightening can outperform monotone inclusions for uncertainty propagation, especially if the system is not cooperative or competitive. This observation motivates our effort to develop such a verified solver for the GPU.

4.2. RK4 and MRK4 Algorithms for VDP

In this subsection, we apply the approach from Section 2.2 to the Liénard form of VDP in Eq. (18). For comparison with variant 5 in Table II, we also report results for the formulation in (15), cf. Table III. Due to the lack of efficient interval-based AD tools, verifying the IVP flow on the GPU currently requires external generation of derivative tensor code, in particular for third- and fourth-

order terms. The first derivative required for MRK4 is obtained via tracing on the GPU using our version of FADBAD++. We further employ the same uniform linear subdivision of the first initial condition as in the previous subsection.

For LTE computation, we consider three options. First, we use the hand-optimized formula in Eq. (9) together with manually precomputed tensors evaluated at runtime (OP1). The DAG of the LTE expression is optimized once since it remains unchanged. All tensors are stored in flat arrays on the GPU with local memory allocations, and common subexpressions are evaluated only once. The second option follows (Alexandre dit Sandretto, 2025) (OP2). Symbolic evaluation of Eq. (9), including tensors and their products, yields the following expression (pre-generated on the CPU) for the Liénard form of VDP:

$$
\begin{aligned}
LTE_{4,y_1} &= h^5 \Bigg(-\frac{y_1^2 \left(-\frac{y_1^3}{3} + y_1 - y_2\right)^3}{120} + \frac{y_1(1-y_1^2)\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)}{120} \\
&- \frac{y_1\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)^2}{80} \\
&- \frac{y_1\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\left(\frac{y_1^3}{3} - y_1 + y_2 + (1-y_1^2)\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)\right)}{60} \\
&- \frac{(1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)^2\left(\frac{2y_1^3}{3} - 2y_1 + 2y_2\right)}{720} \\
&+ \frac{\left(2y_1 - 2(1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)^2}{480} \\
&+ \frac{\left(-2y_1(1-y_1^2)^2\left(-\frac{y_1^3}{3} + y_1 - y_2\right)^2 + 2y_1\left(-\frac{y_1^3}{3} + y_1 - y_2\right)^2\right)}{480} \\
&- \frac{1}{120}\Bigg[-\frac{y_1^3}{3} + y_1 - y_2 - (1-y_1^2)\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right) + (1-y_1^2)\Bigg(y_1 \\
&- (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right) + (1-y_1^2)\left(\frac{y_1^3}{3} - y_1 + y_2 + (1-y_1^2)\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)\right)\Bigg)\Bigg]\Bigg) \\
LTE_{4,y_2} &= h^5 \Bigg(-\frac{h^5 y_1(1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)^2}{240} + \frac{h^5 y_1\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)}{120} \\
&- \frac{h^5\left(y_1 - (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right) + (1-y_1^2)\left(\frac{y_1^3}{3} - y_1 + y_2 + (1-y_1^2)\left(-y_1 + (1-y_1^2)\left(-\frac{y_1^3}{3} + y_1 - y_2\right)\right)\right)\right)}{120} \\
&- \frac{h^5\left(-\frac{y_1^3}{3} + y_1 - y_2\right)^2\left(\frac{2y_1^3}{3} - 2y_1 + 2y_2\right)}{720}\Bigg)
\end{aligned}
$$

The code generator from (Alexandre dit Sandretto, 2025) is not specifically optimized for RK4 and can produce LTE formulas for arbitrary Runge-Kutta methods and ODE models. Although

the generated LTE expressions are optimized via subexpression factorization, further manual reformulation with GPU-specific considerations can improve efficiency. In particular, restructuring the expressions to reuse common subexpressions, enhance memory coalescing, and enable fused operations can better exploit the GPU architecture and accelerate execution. The workflow of OP2 is more involved than OP1: code is first generated on the CPU and then manually transferred to the GPU. Its advantage is that no matrix–vector multiplications are required within the GPU kernel.

Option OP1 can be implemented more efficiently if the LTE expression is constructed using a compile-time expression template system[3] (OP3). VDP is represented as a function object, and symbolic operators such as the Jacobian are composed into expression trees at compile time. These trees encode repeated derivative applications without numerical evaluation during construction; computation is deferred to runtime, where the expressions are evaluated for a given state stored in a computational context object. At runtime, the evaluator traverses the tree and computes intermediate results using preallocated buffers, allowing for efficient GPU execution.

The results are presented in Table III and Figure 2. In contrast to monotone inclusions and DynIbex, both RK4 and MRK4 are evaluated with a fixed step size $h = 0.01$, which affects both enclosure quality and computational cost. Since OP1, OP2, and OP3 implement essentially the same formula and thus yield identical solution flows, only OP2 for Eq. (18) and OP1 for Eq. (15) are shown in the figure, while the table reports performance data for all options.

The naive RK4 implementation based on Eq. (10) is more susceptible to the wrapping effect than MRK4 in Eq. (11), but achieves lower runtimes. Both approaches yield weaker enclosure quality for Eq. (18) than the CPU-based DynIbex (cf. Table II), although they are significantly faster. Compared to the monotone inclusion methods in Table II (Variants 1 and 2), MRK4 provides tighter enclosures while remaining almost fully verified. The only non-verified component is the subdivision, which is currently non-overlapping and may contain gaps of up to 2 ULP.

Notably, improved enclosure quality is achieved even for the reduced number of subdivisions, as reflected in the last two columns of Table III as compared to Table II, which decreases kernel runtimes. For a fixed number of subdivisions, however, MRK4 is slower than the monotone inclusion methods, mainly due to the use of AD. Furthermore, the monotone approaches are implemented using Thrust, benefiting from optimized memory handling, whereas our RK4 and MRK4 implementations rely on a yet unoptimized CUDA design. OP1/OP3 produce slightly wider intervals than OP2 for both RK4 and MRK4. Since the fourth-order tensor for VDP is zero and not used, OP1 achieves slightly better computation times than OP2. In the case of OP3, the runtime of RK4 is reduced by approximately a factor of two compared to OP1, since no AD is applied. For MRK4, however, the performance gain is less pronounced, as FADBAD++ performs derivative computations at runtime.

5. Conclusions

As far as we know, this is the first implementation of a verified IVP solver for tight enclosure of ODE flows on a GPU. While its performance does not yet match that of established CPU-based approaches, particularly when additional subdivision is used on the CPU, it already yields

[3] The same is true for our FADBAD++ implementation for the GPU, but this is left for future work

Table III. Solving VPD using subdivision (2^{12} and 2^{14} boxes) on the GPU for RK4 and MRK4 with different implementation options (constant step size 0.01).

	Variant	Kernel time in ms ($t \in [0,3]$)		Break-down using 2^{12} boxes	
		2^{12}	2^{14}	t	$w([y_1])$
6	(18) + RK4 + OP1	22	71	4.92	≈ 14.6
$6'$	(18) + RK4 + OP2	27	90	4.92	≈ 14.3
$6''$	(18) + RK4 + OP3	12	35		same as OP1
7	(18) + MRK4 + OP1	56	166	16.78	≈ 9.95
$7'$	(18) + MRK4 + OP2	59	184	16.78	≈ 9.88
$7''$	(18) + MRK4 + OP3	45	142		same as OP1
8	(15) + MRK4 + OP1	54	166	10.12	≈ 3.92

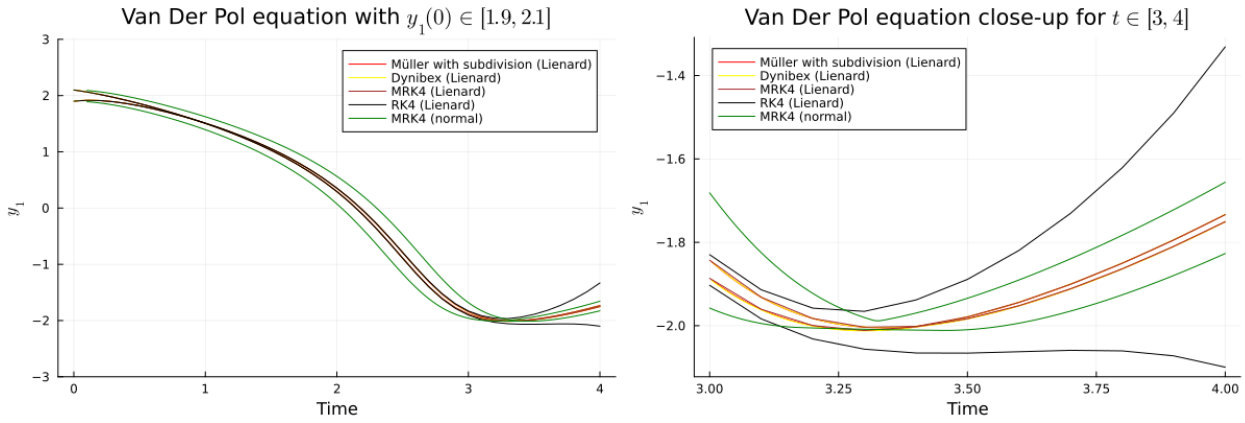

Figure 2. VDP in the Liénard form in Eq. (18) compared with VDP form in Eq. (15) on the GPU as well as DynIbex (CPU) and monotone inclusions (GPU, variant 1 from Table II)

significantly improved enclosure quality for the Liénard form of the Van der Pol oscillator compared to purely a priori enclosure techniques. Moreover, all variants of our MRK4 implementation produce tighter enclosures than monotone inclusion methods while requiring fewer subdivisions, thereby also reducing runtime.

Future work will include more extensive numerical testing, as well as a more efficient workflow for automatic differentiation to compute higher-order derivative tensors (up to order four) within a unified GPU framework. In addition, implementing interval-aware automatic differentiation at compile time appears promising for further reducing computational cost. Finally, improved subdivision strategies remain an important direction for future research.

References

Alexandre dit Sandretto, J.: 2025, 'Symbolic-Numeric Pipeline for Reachability Analysis of Differential Equations with B-Series'. *ACM Communications in Computer Algebra* **59**(3), 57–63. Publisher Copyright: © 2025 Copyright is held by the owner/author(s).

Alexandre dit Sandretto, J. and A. Chapoutot: 2016, 'Validated Explicit and Implicit Runge–Kutta Methods'. *Reliable Computing* **22**(1), 79–103.

Auer, E., A. Ahrens, and L. Gillner: 2025a, 'GPU-Based Interval Optimization in the Context of Optical MIMO Systems'. In: R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski (eds.): *Parallel Processing and Applied Mathematics*. Cham, pp. 360–374, Springer Nature Switzerland.

Auer, E., S. Kiel, and A. Rauh: 2017, 'Towards a Verified ODE Solver for GPU-Based Parameter Identification'. In: *Proceedings of ICOSSAR 2017*. pp. 2039–2049, CRC Press / Taylor & Francis.

Auer, E., A. Rauh, and L. Gillner: 2025b, 'Parameter Identification for Cooperative SOFC Models on the GPU'. In: T. N. Dinh, A. Rauh, S. Z. Yong, and Z. Wang (eds.): *Set-Valued Approaches to Control and Estimation of Uncertain Systems*, Mathematical Engineering. Springer Cham, pp. 211–238.

Bendsten, C. and O. Stauning: 1996, 'FADBAD, a flexible C++ package for automatic differentiation using the forward and backward methods'. Technical Report 1996-x5-94, Technical University of Denmark, Lyngby.

Boost C++ Libraries: 2026, 'Boost Interval Arithmetic Library'. `https://www.boost.org/doc/libs/release/libs/numeric/interval/doc/interval.htm`. Accessed: 2026-04-29.

Butcher, J. C.: 2021, *B-series and Runge–Kutta methods*, pp. 177–209. Cham: Springer International Publishing.

Collange, C., M. Daumas, and D. Defour: 2011, 'Interval Arithmetic in CUDA'. In: W. mei W. Hwu (ed.): *GPU Computing Gems Jade Edition*, No. 978-0-12-385963-1. Morgan Kaufmann, pp. 99–107.

Griewank, A. and A. Walther: 2000, 'Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition'. In: *Frontiers in applied mathematics*.

Hascoet, L. and V. Pascual: 2013, 'The Tapenade automatic differentiation tool: principles, model, and specification'. *ACM Transactions on Mathematical Software (TOMS)* **39**(3), 1–43.

Hirsch, W. M. and H. L. Smith: 2005, *Monotone Dynamical Systems*, Vol. 2, Chapt. 4 of Ordinary Differential Equations, pp. 239–357. Elsevier.

Ifrim, I., V. Vassilev, and D. J. Lange: 2023, 'GPU Accelerated Automatic Differentiation With Clad'. *Journal of Physics: Conference Series* **2438**(1), 012043.

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

Moses, W. S., S. H. K. Narayanan, L. Paehler, V. Churavy, M. Schanen, J. Hückelheim, J. Doerfert, and P. Hovland: 2022, 'Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Press.

Mullier, O., A. Chapoutot, and J. A. dit Sandretto: 2018, 'Validated computation of the local truncation error of Runge–Kutta methods with automatic differentiation'. *Optimization Methods and Software* **33**(4-6), 718–728.

Nedialkov, N. S., K. Jackson, and G. Corliss: 1999, 'Validated solutions of initial value problems for ordinary differential equations'. *Appl. Math. and Comp.* **105**(1), 21 – 68.

NVIDIA: 2026, 'CUDA Toolkit Documentation: CUDA C++ Programming Guide'. `https://docs.nvidia.com/cuda/`. Accessed: 2026-04-29.

Rump, S. M.: 2026, 'INTLAB — Interval Laboratory, Version 11'. `http://www.ti3.tu-harburg.de/rump/intlab/`. Accessed: 2026-04-29.

Sanders, D. P.: 2022, 'Global optimization and interval constraint programming using Julia: Symbolics, code generation, GPU'.

Utke, J., U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch: 2008, 'OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes'. *ACM Transactions on Mathematical Software (TOMS)* **34**(4), 1–36.