Comparison of Higher-Order Approximations to Solve Dynamical Systems Using Interval Constraint Solving

Angel F. Garcia Contreras(✉) and Martine Ceberio

The University of Texas at El Paso, El Paso, TX 79968, USA
afgarciacontreras@miners.utep.edu

Abstract. Phenomena that change over time are abundant in nature. Dynamical systems, composed of differential equations, are used to model them. In some cases, analytical solutions exist that provide an exact description of the system's behavior. Otherwise we use numerical approximations: we discretize the original problem over time, where each state of the system at any discrete time moment depends on previous/subsequent states. This process may yield large systems of equations. Efficient tools exist to solve dynamical systems, but might not be well suited for certain types of problems. For example, Runge-Kutta-based solution techniques do not easily handle parameters' uncertainty, although inherent to real world measurements. If the problem has multiple solutions, such methods usually provide only one. When they cannot find a solution, it is not know whether none exists or it failed to find one. Interval methods, on the other hand, provide guaranteed numerical computations. If a solution exists, it will be found. Interval methods for dynamical systems fall into two main categories: step-based methods (fast but too conservative with overestimation for large systems) and constraint-solving techniques (better at controlling overestimation but usually much slower). In prior research, we developed a promising method that speeds up constraint-based techniques. In this article, we test that method with higher order approximations known as multi-step methods. We compared these approximations based on their accuracy when attempting to include the real solution. We share insightful experimental results.

1 Introduction

Phenomena that change over time are abundant. Their behavior can be modeled using dynamical systems that represent chronological change via differential equations. For some real life problems, we have analytical solutions that describe the behavior exactly. For many other problems, there is no such exact representation, so we use numerical approximations: we take the original problem, which is continuous over time, and we *discretize* it such that we get a series of discrete moments in time, and in which the system state at each discrete moment depends on previous and/or subsequent states. This relationship is described

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
J. C. Figueroa-García et al. (Eds.): WEA 2022, CCIS 1685, pp. 3–18, 2022.
https://doi.org/10.1007/978-3-031-20611-5_1

through a state equation; depending on the desired granularity/precision for the approximation, discretization may generate a very large set of equations.

There are many tools that solve dynamical systems, but these might not be well-suited for certain types of problems. For example, a Runge-Kutta-based technique cannot easily handle a problem with uncertain parameter values obtained from real world measurements. Solutions are heavily reliant on an initial set of parameters. If a problem has multiple solutions, such methods can find *a* solution, but cannot identify *how many* solutions exist, or if the found solution is the best based on given criteria. If the method does not find a solution, it is unknown whether the problem has no solution, or method simply failed to find one.

We want to provide guaranteed numerical computations, which identify all solutions if they exist, and the certainty that our computations return no solution it means none exist. We use *interval-based computations* [13,14], as they provide the desired guarantees. When solving dynamical systems, there are two main categories of interval-based techniques: *step-based* methods that generate an explicit system of equations one discretized state at a atime, and *constraint-solving techniques* that solve the entire system of implicitly discretized equations. Step-based method are fast and have less computational overhead, so they have been widely studied in the past; however, on complex systems (either because they are simulating longer times or the differential system is very non-linear), the solution they find can lead to overestimated ranges. Constraint-solving techniques can better control the overestimation through the entire system at once, but this reduction comes at a computational time cost and may take considerably longer to produce a result.

In previous work [7,8], we introduced a heuristic approach that dramatically improves the computation time of constraint-solving interval techniques for dynamical systems, by solving smaller overlapping sub-problems. In this paper, we take a look at the algorithm's accuracy under a given discretization, then we analyze the performance and accuracy of discretizations with various degrees of complexity. Our results who that careful selection of the discretized form is essential in improving accuracy while also maintaining a reasonable computation time, as discretized forms with higher-complexity do not necessarily provide better solutions.

2 Background

2.1 Dynamical Systems

Dynamical systems model how a phenomenon changes over time. In particular, we are interested in continuous dynamical systems.

Definition 1. A continuous *dynamical system* is a pair (D, f) with $D \subseteq \mathbb{R}^n$ called a *domain* and $f : D \times T \to \mathbb{R}^n$ a function from pairs $(x, t) \in D \times T$ to $\mathbb{R}^n$.

Definition 2. By a *trajectory* of a dynamical system, we mean a function $x : [t_0, \infty) \to D$ for which $\frac{dx}{dt} = f(x, t)$.

To obtain the state equations of a dynamical system, we *integrate* its differential equations. The type of problems we are interested in often lack an exact integral, instead we use *numerical methods* that *approximate* the actual solution. These methods can provide good results, but as they are approximations their solutions always have a margin of error that must be included in the computation. This error can be minimized by the choice of numerical methods and by tweaking the method's approximation parameters.

2.2 Traditional Methods

Numerical methods to solve dynamical systems are usually classified in two general categories based on the type of approximation they make for the integral: explicit and implicit methods. In *explicit* methods, the state equation for a specific state involves the values of one or more previous states. This means that given an initial state, every subsequent state value can be obtained by evaluating each discretized equation in order, as each state equation already has the values it needs from previous evaluated equations. *Implicit* methods involve past and future states in their discretization. These equations cannot be solved by simple successive evaluation, but with search algorithms instead. The most common type of algorithm used is root-finding methods, such as Newton-Rhapson. Both types of methods are used to solve dynamical system problems, either separately or synergistically.

2.3 Interval Methods

An interval is defined as: $\boldsymbol{x} = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x};\ \underline{x},\ \overline{x} \in \mathbb{R}\}$.
Intervals represent all values between their infimum $\underline{X}$ and supremum $\overline{X}$. In particular, we can use them to represent uncertain quantities. We manipulate them in computations through the rules of interval arithmetic, naively posed as follows: $\boldsymbol{x} \diamond \boldsymbol{y} = \{x \diamond y, \text{ where } x \in \boldsymbol{x}, y \in \boldsymbol{y}\}$, where $\diamond$ is any arithmetic operator, and combining intervals always results in another interval. However, since some operations, like division, could yield a union of intervals (e.g., division by an interval that contains 0), the combination of intervals involves an extra operation, called the hull, denoted by $\square$, which returns one interval enclosure of a set of real values. We obtain: $\boldsymbol{x} \diamond \boldsymbol{y} = \square \{x \diamond y, \text{ where } x \in \boldsymbol{x}, y \in \boldsymbol{y}\}$.

We can extend this property to any function $f : \mathbb{R}^n \to \mathbb{R}$ with one or more interval parameters:

$$f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) \subseteq \square \{f(x_1, \ldots, x_n), \text{ where } x_1 \in \boldsymbol{x_1}, \ldots, x_n \in \boldsymbol{x_n}\}$$

where $f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})$ represents the range of f over the interval domain $\boldsymbol{x_1} \times \ldots, \times \boldsymbol{x_n}$, and $\square \{f(x_1, \ldots, x_n), \text{ where } x_1 \in \boldsymbol{x_1}, \ldots, x_n \in \boldsymbol{x_n}\}$ represents the narrowest interval enclosing this range. Computing the exact range of f over intervals is very hard, so instead we use surrogate approximations. We call these

surrogates *interval extensions*. An interval extension $\boldsymbol{F}$ of function f must satisfy the following property:

$$f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) \subseteq \boldsymbol{F}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})$$

Interval extensions aim to approximate the range of the original real-valued function. In general, different interval extensions can return a different range for f while still fulfilling the above property. For more information about intervals and interval computations in general, see [13,14].

Step-based Interval Methods for Solving Dynamical Systems. These algorithms use explicit discretization explicit discretization schemes, such as Taylor polynomials or Runge-Kutta, that must be evaluated to provide a guaranteed enclosure that includes the discretization error at every step. The solvers implement interval evaluation schemes that reduce overestimation. For example, VSPODE [11] uses Taylor polynomials for discretization and Taylor models [1,12] for evaluation; DynIBEX [6] uses Runge-Kutta discretization and evaluates its functions using affine arithmetic [9,16].

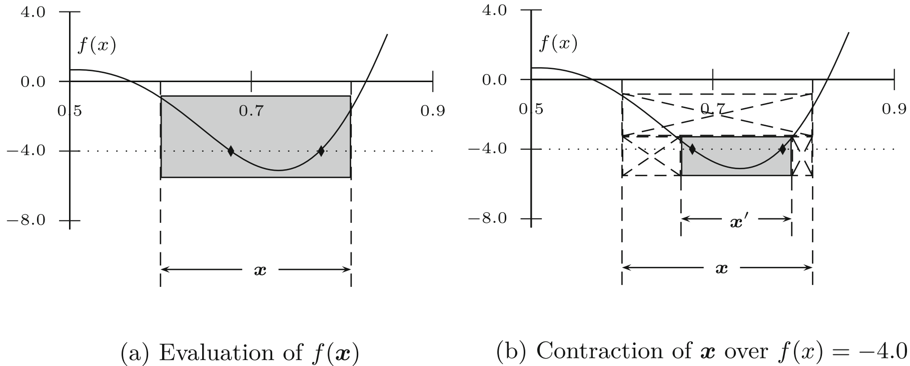
Interval Constraint-Solving Techniques. The methods used to solve a dynamical system using explicit discretization do not work for implicit discretization. We need to solve the entire system. We can do this if we treat the state equations as a system of equality constraints and the dynamical system as an interval Constraint Satisfaction Problem (CSP):

Definition 3. An interval *constraint satisfaction problem* is a tuple $P = (X, \boldsymbol{D}, C)$, X is a set of n variables $\{x_1, \ldots, x_n\}$, $\boldsymbol{D}$ is the Cartesian product of the variables' associated interval domains $\boldsymbol{D} = \boldsymbol{x_1} \times \ldots \times \boldsymbol{x_n}$, and C is the set of m constraints $C = \{c_1, \ldots, c_m\}$. [5]

The initial interval domain $\boldsymbol{D}$ represents the entire space in which a real-valued solution to the CSP might be found. With intervals, we want to find an *enclosure* of said solution. This enclosure $\boldsymbol{X^*}$ needs to be narrow: the differences between the infimum and supremum of all interval domains in $\boldsymbol{X^*}$ must be less than a parameter ϵ, representing the *accuracy* of the solution's enclosure. If the entire domain is inconsistent, it will be wholly discarded, which means that the problem has no solution.

An interval constraint solver attempts to find a narrow $\boldsymbol{X^*}$ through consistency techniques. *Consistency* is a property of CSPs, in which the domain does not violate any constraint. For interval CSPs, we want domains that are at least *partially consistent*: if they do not entirely satisfy the constraints, they may contain a solution. Figure 1 shows a visualization of the general concept behind contraction using consistency. Figure 1a shows the evaluation of a function $f(x)$ over an interval $\boldsymbol{x}$, represented by the gray rectangle $y = f(\boldsymbol{x})$. This function is part of a constraint $f(x) = -4$, whose solutions are found in the domain of $\boldsymbol{x}$; however, this interval is too wide, so it must be contracted. In this case, the

range of $f(\boldsymbol{x}) \geq -4.0$ can be discarded, which creates a new interval value for the range of $f(x)$, or y', which can be *propagated* to remove portions of $\boldsymbol{x}$ that are not consistent with y'. This creates the contracted domain x', which is a narrower enclosure of the solutions of $f(x) = -4$, as shown in Fig. 1b.



(a) Evaluation of $f(\boldsymbol{x})$

(b) Contraction of $\boldsymbol{x}$ over $f(x) = -4.0$

Fig. 1. Visual example of interval domain contraction

Contraction via consistency is just part of how interval constraint solver techniques find narrow enclosures of solutions to systems of constraints. For example, the constraint $f(x) = -4$ shown in Fig. 1 has two solutions enclosed inside the domain $\boldsymbol{x}'$, but we need the individual solutions.

Interval constraint solvers use a *branch-and-prune* algorithm. The "prune" part of the algorithm is achieved through contraction via consistency; when "pruning" is not enough to find the most narrow enclosure that satisfies the constraints, the algorithm "branches" by dividing the domain $\boldsymbol{X}$ into two adjacent subdomains by splitting the interval value of one of its variables through a midpoint $m(\boldsymbol{x}) = \frac{\underline{x}+\overline{x}}{2}$. These two new sub-boxes, $\boldsymbol{X_L} = \{\boldsymbol{x_0}, \ldots, [\underline{x_i}, m(\boldsymbol{x_i})], \ldots, \boldsymbol{x_n}\}$ and $\boldsymbol{X_U} = \{\boldsymbol{x_0}, \ldots, [m(\boldsymbol{x_i}), \overline{x_i}], \ldots, \boldsymbol{x_n}\}$, are then processed using the same algorithm. This means that all sub-boxes are put in a queue of sub-boxes, as each sub-boxes and be further "branched" into smaller sub-boxes.

Interval constraint solvers such as RealPaver [10] and IbexSolve [3,4] solve systems of constraints. To solve a dynamical system, we need to generate all required state equations, and provide an initial domain containing all possible state values. While interval solvers can provide good results, when system are too large, they can be slow to find a reasonable solution. For large systems, there have been attempts at making them easier to solve, including generating an alternative reduced-order model [17], and focusing on a subset of constraints at a time [15].

Step-Based or Constraint-based? The solvers that use step-based interval methods (VSPODE, DynIBEX) work well, up to a point. As with non-interval

approaches, there is an approximation error; however, due to the need for guaranteed computations, these methods compute an *enclosure* of each state plus its range of approximation errors. These bounds on the error can introduce a small amount of overestimation into the solution, which compounds after computing multiple states, each with their own bounded approximation error. Even when dynamically computing the step size between states (which helps reduce the overestimation), this additional range accumulates at every iteration. If the interval value of a state becomes too large, these solvers cannot compute a new step size and stop the simulation even before reaching a desired final state.

Our motivation behind using constraint-based interval methods is that they can explore the entire system. Solving a full system using interval constraint-solving techniques can explore multiple realizations of the system with a desired width for the enclosure. To potentially increase the contraction, each state is evaluated multiple times over one or multiple domains. Even with a static value of h for all states, implicit approximations used increase the approximation's accuracy. The drawback of these methods is tied to this particular strength: contractors need to evaluate each equation multiple times, and branch-and-prune-based solvers incorporate multiple contractors. Additionally, the "branching" process creates problems exponentially based on the number of variables; each sub-problem needs to be processed, including potential further sub-divisions, so the number of subproblems being generated becomes exponential. Thus, these techniques provide a strong computational guaranteed, with a trade-off of a considerably higher computation time.

Prior Work. In previous work [7,8], we showed a *Sliding Windows* scheme to improve the computation time to find a solution using interval constraint methods. The main idea of this type of algorithm is to take advantage of the structure in a dynamical system (specifically, an *initial value problem*) to create and solve subproblems made of subsets of *contiguous state variables* and their respective equations. We take the state variables $X_{\text{sub}} = \{x(j), \ldots x(j+w)\}$ with domains $\{\boldsymbol{x}(\boldsymbol{j}), \ldots, \boldsymbol{x}(\boldsymbol{j}+\boldsymbol{w})\}$, along the following set of state equations as a system of constraints C_{sub}:

$$g_i\left(x(j), \ldots x(j+w), t_i\right) = f\left(x(i), t_i, h\right), \ \forall i \in \{j, \ldots j+w\}$$

where function g_i is a discretization of $\frac{dx}{dt}$ at t_i. We call this subproblem $P_{\text{sub}} = (X_{\text{sub}}, C_{\text{sub}})$ a *window* of size w. Our technique aims to speed up the computation process of interval constraint solvers by sequentially creating and solving a series of subproblems with size w.

We can solve the first subproblem normally, as it is exactly an initial value problem. However, subsequent problems cannot be treated the same: if we took the last values of the previous problems as initial values, we would lose the trajectory created by the previous states.

The second element involves maintaining this trajectory by *transfering* multiple state values between subproblems. Solving the k-th subproblem $P_k = (X_k, C_k)$ yields a reduced domain X_k^* representing w states, from t_j to t_{j+N_k}. For

the next subproblem, $P_{k+1} = (X_{k+1}, C_{k+1})$, we take the last o interval values of X_k^* and use them as the initial domain for the *first* o values of X_{k+1}:

$$\{\boldsymbol{x_{k+1}}\,(\mathbf{1})\,, \ldots, \boldsymbol{x_{k+1}}\,(\boldsymbol{o})\} = \{\boldsymbol{x_k}\,(\boldsymbol{w} - \boldsymbol{o})\,, \ldots, \boldsymbol{x_k}\,(\boldsymbol{w})\}$$

We then solve subproblem P_{k+1} using interval constraint-solving techniques, yielding a new reduced domain used to repeat the process again. We call o the *overlap* between subproblem *windows* of size w. Figure 2 shows a graphical representation of how o states are transferred from one subproblem to the next.

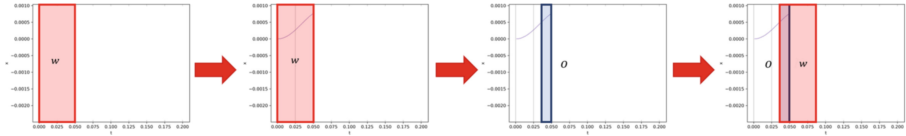


Fig. 2. Graphical plot of overlap transfer

With interval constraint solvers, a series of subproblems with a smaller number of variables is faster to solve than one with more variables: the number of subproblems generated from domain division is reduced, which speeds up the overall process.

3 Problem Statement and Experiments

Our first experiments with the Sliding Windows proved we could find solutions to dynamical systems in reasonable time using interval-based constraint solving techniques. These experiments were carried out with a simple approximation to the dynamical system called *backwards Euler*, which involves just two variables per state equation, and without bounding the approximation error. Existing step-based solvers use *high order approximations* that involve multiple prior variables while including techniques that bound the approximation error.

Without any changes, the backwards Euler discretization used in previous experiments with the Sliding Windows algorithm is not a perfectly accurate enclosure, as it does not bound the real solution. This motivates our next research question: how can we improve the enclosures of the approximations in the Sliding Windows algorithm?

To look for an answer, we took a look at the existing, step-based solvers. They use explicit discretization methods that result in polynomials with a more accurate approximation. VSPODE uses high-order Taylor polynomials, generating the coefficients using automatic differentiation of the original ODE. DynIBEX uses an interval-based version of Runge-Kutta, which uses intermediate points between states in its computation of a specific state. Both of these methods bound the approximation error by computing upper and lower bounds of the

local truncation error on each step and add it to the discretized equations to guarantee an enclosure of the solution.

Without a computation of the bounds (for now), we want to find more accurate implicit approximations that work with the Sliding Windows algorithm. We focused on experiments in two main areas: *higher-order approximations* and *artificially-inflated enclosures*.

3.1 Higher-Order Approximations

In general, an approximate state equation that involves more states increases the accuracy of the approximation. Existing solvers apply this concept with explicit discretizations by creating *intermediate sub-states* that are computed on evaluation but not stored outside of finding the state for that equation. This kind of approach would be inefficient for Sliding Windows: if two main states share one or more points, they would have to be computed multiple times, unless we added them to our full set of states.

If adding new sub-states is not an option, the best alternative we have is to have approximations that involve *multiple existing states*; these type of approximations are known as *linear multistep methods*, and each state equation can involve multiple states from before or after the current state. For these experiments, we selected the *Adams-Moulton* method.

The Adams-Moulton Method. The Adams-Moulton method is the implicit version of the explicit Adams-Bashfort method [2]; in non-interval solvers, these two methods work together in a *predictor-corrector method*: the solver finds an "initial guess" to the complete behavior of the system using Adams-Bashfort as a slightly inaccurate predictor, then uses this behavior as an "initial point" to solve an Adams-Moulton approximation to "correct" the results and make them more accurate.

As we are working with interval constraint-solvers, and we want to test their accuracy on their own, we can use interval evaluation and constraint-solving on the Adams-Moulton method only. The general formula for this approximation is:

$$x_{n+s} = x_{n+s-1} + h \sum_{m=0}^{s} b_s f\left(x_{n+m}, t_{n+m}\right)$$

where t_i is a discrete time, x_i is a state variable at t_i, $f(x_i, t_i)$ is an ordinary differential equation s.t. $\dot{x} = f(x, t)$, and b_s is a unique coefficient. These coefficients are independent from x_i and t_i, and its value depends on the *order of the approximation* s, which represented the number of states involved in each state equation.

We believe that such a multi-step discretization method is a good fit for solving dynamical systems using interval constraint-solving techniques. Each state equation involves multiple state values, maintained naturally by the interval box

that represents the entire set of states from the full problem – or, in the case of the Sliding Windows algorithm, an individual window.

A higher order approximation using Adams-Moulton means $s > 0$ ($s = 0$ is backwards Euler), which means each state equation will involve more terms and coefficients. This can impact the interval evaluation and contraction processes. Given the conditions of the Sliding Windows algorithm and these tests, we will also no include bounding the error in any way. This means we know in advance that our approximations will not bound the real solution, but we can use this lack of computational rigor to identify in advance potential issues with higher-order enclosures, such as excessive overestimation or inaccuracy.

We measure said inaccuracy by the *error* on each state enclosure, which we define as the distance between an interval state enclosure and the real solution outside its bounds. For each state enclosure $\boldsymbol{x_i} = \left[\underline{x_i}, \overline{x_i}\right]$ and real solution x_i^*, the *error* in $\boldsymbol{x_i}$ is:

$$\begin{aligned} \tau(\boldsymbol{x_i}) &= 0 && \text{if } x_i^* \in \boldsymbol{x_i} \\ \tau(\boldsymbol{x_i}) &= \underline{x_i} - x_i^* && \text{if } x_i^* < \boldsymbol{x_i} \\ \tau(\boldsymbol{x_i}) &= x_i^* - \overline{x_i} && \text{if } x_i^* > \boldsymbol{x_i} \end{aligned}$$

The first objective of the experiments is to use these metrics to explore the accuracy of various orders of Adams-Moulton discretizations. In particular, we focus on the Adams-Moulton discretizations of order $s = \{0, \ldots, 8\}$. While we are not including bounds on the approximation error, we can still attempt to replicate the impact of including these bounds into the process. We do this by manipulating the state enclosures using artificial intervals into the models given by the Adams-Moulton approximations.

The Adams-Moulton Method. To analyze the impact of the calculation of the approximation error, we created new models based on the systems generated by the Adams-Moulton method that introduce an "artificial bounding" that increases and changes the interval state enclosures. With this, we can explore how much the solution process of higher-order enclosures is changed by these additional bounds.

We explored two different types of enclosure manipulations:

- *Initial value inflation.* We change the initial value of the model by making it a "wider" interval enclosure.
- *State equation inflation.* We "naively" simulate a per-state bound on the error by adding a narrow $[-\Delta, +\Delta]$ interval, to examine the impact of an additional interval term.

3.2 Experiments and Results

We designed experiments along two axes: the *approximation level* and *type of enclosure manipulation.* We generated models for Adams-Moulton involving $s = \{0, \ldots, 8\}$, with $s = 0$ representing the Backwards Euler method examined in

previous experiments. For enclosure manipulation, we settled on the following seven types:

- Case Base: The normal case, with no additional uncertainty added.
- Case All-A: Add to all state equations a constant $\delta = \left[-10^{-16}, 10^{-16}\right]$
- Case All-B: Add to all state equations a constant $\delta = \left[-10^{-8}, 10^{-8}\right]$
- Case All-C: Add to all state equations a constant $\delta = \left[-10^{-4}, 10^{-4}\right]$
- Case Ini-A: Add to all initial states a constant $\delta = \left[-10^{-16}, 10^{-16}\right]$
- Case Ini-B: Add to all initial states a constant $\delta = \left[-10^{-8}, 10^{-8}\right]$
- Case Ini-C: Add to all initial states a constant $\delta = \left[-10^{-4}, 10^{-4}\right]$

We ran each of the models with each of the enclosure manipulation types, for a total of 63 experiments using a three-species food chain model with Holling type II predator response function:

$$\frac{dm_1}{dt} = r_1 m_1 \left(1 - \frac{m_1}{K_1}\right) - a_{12}\left(\frac{m_1 m_2}{m_1 + A_1}\right) \quad (1a)$$

$$\frac{dm_2}{dt} = -d_2 m_2 + a_{21}\left(\frac{m_1 m_2}{m_1 + A_1}\right) - a_{23}\left(\frac{m_2 m_3}{m_2 + A_2}\right) \quad (1b)$$

$$\frac{dm_3}{dt} = -d_3 m_3 + a_{32}\left(\frac{m_2 m_3}{m_2 + A_2}\right) \quad (1c)$$

For the discretization, we used a step size of $h = 0.01$.

Comparison Metrics. Given that we are not bounding the approximation error, the enclosures found with the state equations and the Sliding Windows algorithm will not always enclose the solution at all discrete times. We designed a group of metrics with the intent of using them to compare the Adams-Moulton discretizations for $s = \{0, \ldots, 8\}$:

- *Time states until overestimation.* Number of states contracted before reaching a "window" of states whose interval width is beyond a certain threshold. We consider a state to be overestimating, when the supremum of a state is 10% above its midpoint. For these experiments, we set a goal of $t_f = 40.0$, or $N = 4000$ states, which represents the max value for this metric. We chose this target based on previous experiments [7,8] that showed existing solvers could reach this target number without leading to excessive overestimation.
- *Solution accuracy.* If the approximation was not perfectly accurate, it means that in our solution there were interval states that did not enclose the actual solution to the system. There are various ways to analyze the accuracy of the system:
  - *Coverage.* Defined as the percentage of states that enclose the solution. For example, if in a system of 100 states, the interval solution encloses the actual solution in 90 of those states, we say there is a coverage of 90%. A higher coverage is better in this metric; all solvers that bound the approximation error, such as VSPODE and DynIBEX, have a coverage of 100%.

- *Total accumulated error.* The sum of all errors across all states. States that provide an enclosure have an error of 0, states that do not enclosure the solution have an error equal to the linear distance from the closest bound to the real solution. The smallest this value is, the better is the result.
- *Total average error.* The average of the error across all states, including those that provide an enclosure to the real-based solution. The smallest this value is, the better is the result.
- *Coverage average error.* The average of the error across all states that do not enclose the real-based solution. The smallest this value is, the better is the result.

While these metrics might seem enough, we want discretizations that produce better enclosures to longer simulations. Interval methods struggle to produce tighter enclosures after a longer simulated time, particularly when the equations are complex (i.e. highly non-linear) or the simulated time is longer/has more states.

There is no guarantee that we will get narrow state enclosures for all $N = 4000$ states, and in fact it is more likely that we will not. Instead of using these metrics as-is, we use the "Time states until overestimation" metric to *weight* the accuracy metrics based on how close the model was to the desired goal of $N = 4000$ states without overestimation:

$$<\text{weighted metric}> = <\text{base metric}> \times \frac{<\text{states until overest.}>}{4000}$$

This creates four new metrics: *weighted coverage*, *weighted accumulated error*, *weighted average error*, and *weighted coverage average error*. Using these metrics we can compare the overall impact of each discretization in improving the quality of the solution (by increasing coverage/decreasing error). Table 1 shows the "Time states until overestimation" metric, while Tables 2, 3, 4 and 5 show the weighted metrics.

Table 1. Comparison of time states until overestimation

Disc	*Base*	*All-A*	*All-B*	*All-C*	*Ini-A*	*Ini-B*	*Ini-C*
s=0	4000	4000	2190	860	4000	4000	2980
s=1	4000	4000	2200	850	4000	4000	2970
s=2	4000	4000	2140	260	4000	4000	2940
s=3	4000	4000	2120	250	4000	4000	2820
s=4	4000	4000	2080	240	4000	4000	2560
s=5	4000	4000	2010	200	4000	4000	2040
s=6	3850	3870	1880	180	3880	2650	1630
s=7	2350	2300	1640	140	2330	1910	230
s=8	1290	1290	470	110	1280	1050	220

Table 2. Comparison of weighted coverage in the experiments.

Disc.	*Base*	*All-A*	*All-B*	*All-C*	*Ini-A*	*Ini-B*	*Ini-C*
s=0	0.0000	0.0000	0.4178	0.1880	0.0000	0.0008	0.0608
s=1	0.0000	0.0038	**0.4733**	0.2060	0.0000	0.0868	**0.7100**
s=2	0.0327	0.2460	**0.5138**	0.0635	0.0345	0.4198	**0.7298**
s=3	0.0330	0.2178	**0.4868**	0.0600	0.0343	0.2650	**0.6895**
s=4	0.2380	0.2558	**0.4778**	0.0578	0.2383	0.2943	**0.6260**
s=5	0.2855	0.2833	0.4605	0.0475	0.2878	0.4610	**0.4963**
s=6	0.3518	0.3568	0.4290	0.0425	0.3620	**0.5480**	0.3958
s=7	0.4590	0.4555	0.3710	0.0325	0.4558	0.4110	0.0478
s=8	0.2530	0.2530	0.0850	0.0250	0.2505	0.2163	0.0475

Table 3. Comparison of weighted total accumulated error in the experiments.

Disc.	*Base*	*All-A*	*All-B*	*All-C*	*Ini-A*	*Ini-B*	*Ini-C*
s=0	26.5617	26.5582	10.6833	22.0304	26.5617	26.5134	14.7647
s=1	0.5927	0.5896	0.2961	0.3136	0.5927	0.5482	0.1982
s=2	**0.0159**	**0.0148**	**0.0184**	0.0898	**0.0159**	**0.0113**	**0.0117**
s=3	0.0910	0.0850	0.0502	0.1432	0.0909	0.0705	**0.0301**
s=4	0.0691	0.0646	0.0449	0.1365	0.0691	0.0481	**0.0287**
s=5	0.0571	0.0575	0.0483	0.1677	0.0568	**0.0265**	0.0369
s=6	0.0401	0.0396	0.0506	0.1844	0.0396	**0.0365**	0.0443
s=7	0.0416	0.0425	0.0584	0.2386	0.0420	0.0511	0.3057
s=8	0.0750	0.0750	0.1983	0.3022	0.0756	0.0917	0.3039

Table 4. Comparison of weighted total average error in the experiments.

Disc.	*Base*	*All-A*	*All-B*	*All-C*	*Ini-A*	*Ini-B*	*Ini-C*
s=0	6.64e-03	6.64e-03	4.88e-03	2.56e-02	6.64e-03	6.63e-03	4.95e-03
s=1	1.48e-04	1.47e-04	1.35e-04	3.69e-04	1.48e-04	1.37e-04	6.67e-05
s=2	**3.97e-06**	**3.70e-06**	**8.60e-06**	3.45e-04	**3.97e-06**	**2.82e-06**	**3.98e-06**
s=3	2.27e-05	2.12e-05	2.37e-05	5.73e-04	2.27e-05	1.76e-05	1.07e-05
s=4	1.73e-05	1.61e-05	2.16e-05	5.69e-04	1.73e-05	1.20e-05	1.12e-05
s=5	1.43e-05	1.44e-05	2.40e-05	8.39e-04	1.42e-05	**6.61e-06**	1.81e-05
s=6	**1.04e-05**	**1.02e-05**	2.69e-05	1.02e-03	**1.02e-05**	1.38e-05	2.72e-05
s=7	1.77e-05	1.85e-05	3.56e-05	1.70e-03	1.80e-05	2.68e-05	1.33e-03
s=8	5.81e-05	5.81e-05	4.22e-04	2.75e-03	5.90e-05	8.74e-05	1.38e-03

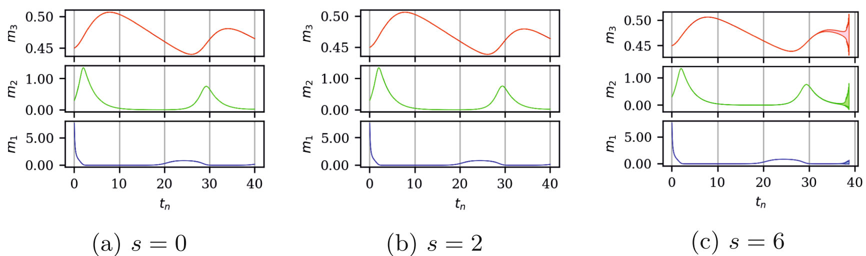
Table 5. Comparison of weighted coverage average error in the experiments.

Disc.	*Base*	*All-A*	*All-B*	*All-C*	*Ini-A*	*Ini-B*	*Ini-C*
s=0	6.64e-03	6.64e-03	2.06e-02	2.04e-01	6.64e-03	6.63e-03	5.39e-03
s=1	1.48e-04	1.48e-04	9.64e-04	1.21e-02	1.48e-04	1.50e-04	1.52e-03
s=2	**4.11e-06**	**4.90e-06**	2.17e-04	1.50e-02	**4.11e-06**	**4.85e-06**	5.57e-04
s=3	2.35e-05	2.72e-05	2.90e-04	1.43e-02	2.35e-05	2.40e-05	4.85e-04
s=4	2.27e-05	2.17e-05	2.66e-04	1.52e-02	2.27e-05	**1.70e-05**	5.12e-04
s=5	2.00e-05	2.01e-05	2.87e-04	1.68e-02	**1.99e-05**	**1.23e-05**	6.71e-04
s=6	**1.64e-05**	**1.62e-05**	3.09e-04	1.84e-02	**1.63e-05**	7.96e-05	9.42e-04
s=7	8.10e-05	8.89e-05	3.74e-04	2.39e-02	8.28e-05	1.92e-04	7.84e-03
s=8	2.70e-04	2.70e-04	1.53e-03	3.02e-02	2.72e-04	4.96e-04	1.01e-02

Analysis of Results

States Until Overestimation. The behavior in Table 1 is as expected. Wider initial states, such as the ones in Cases *Ini-A, Ini-B and Ini-C*, lead to overestimation happening earlier in the simulated set of states. This effect is more noticeable on the *All* Cases, as each state introduces more overestimation. The data we find most interesting is on the discretization complexity: as s increases, these models lead to earlier overestimation, which can be seen in Fig. 3. This suggests that under the conditions of the experiment it is preferable to use a lower complexity model to avoid overestimation. It is important to note that this might not be applicable in all scenarios: it is certainly possible that changes to the default constraint solver in Ibex, or even the implementation of a different constraint solver could lead to improvements with these higher complexity models.

Weighted Coverage. In general, coverage is a measure of how well a specific experiment managed to enclose the solution. In general, adding wider intervals to the models leads to better coverage, as seen in case *Ini-C*. Among the *All*



(a) $s = 0$ (b) $s = 2$ (c) $s = 6$

Fig. 3. Visual comparison of simulated results for Case Base.

cases, it is surprising that *All-B* gets better weighted coverage than *All-C*, a model that fails to get even close to the target of 4000 states.

We note that higher order approximations, such as s=*6* and s=*7* provide increasingly better coverage than lesser order ones on cases with low interval expansion, *Base*, *All-A* and *Ini-A*. This suggests that the bounds on the error in these higher order approximations needs to be as small as possible.

Weighted Accumulated Error. Weighted accumulated error is a good measure of how close each experiment was to the actual behavior of the system that we want our results to enclose as narrowly as possible. The high accumulated error in the $s = 0$ experiments is expected, as this is a very simple discretization and thus prone to high approximation error on each state. The most surprising results are the ones for $s = 2$, as they are consistently low across all categories. This suggests that the $s = 2$ discretization level might be overall closer to bounding the expected solution.

Weighted Average Error and Coverage Average. With this metric, we want to compare how close each state in the system might be to the expected solution. Similar to weighted accumulated error, the results for the $s = 2$ are also good: not only does the overall error is low across the whole set of states, but each state that fails to enclose the solution does so by a smaller amount than in other discretizations. This case also does well with weighted coverage average. Surprisingly, on these two metrics the $s = 6$ discretization also does well, with its major drawback being that none of the cases run under this discretization level reached the desired final narrow state of $N = 4000$.

4 Conclusions and Future Work

Based on the presented results, we find that, for the sliding windows heuristic, increasing the complexity of the discretization does not necessarily lead to better results. The simplest types of discretization such as backwards Euler, represented by the $s = 0$ Adams-Moulton discretization, is definitely the worst; once we increased the complexity, we started getting better results. The best results we obtained were with an Adams-Moulton discretization involving two prior state variables ($s = 2$). While it may seem that increasing it even further could produce better results, for this experiment we obtained diminishing returns, as the overestimation starts occurring progressively earlier as the value of s increases. This suggests that, under these conditions, we must use discretizations beyond a single prior variable, but also avoid using too many variables.

All of these results come with the caveat that these experiments are not entirely guaranteed enclosures, due to the lack of an enclosure of the approximation error. This is a natural future work: to seek and implement a technique to bound the approximation error that is compatible with constraint-solving techniques. We are also looking into other implicit discretization schemes beyond Adams-Moulton that could have smaller error, as well as different contractor

settings and configurations that could either improve the computation time or provide better contraction with reduced overestimation. We also plan to incorporate these techniques with other dynamical systems such as boundary value problems, as well as comparing against other approaches that reduce computational complexity, such as reduced order modeling.

References

1. Berz, M., Makino, K.: Verified integration of odes and flows using differential algebraic methods on high-order taylor models. Reliable Comput. **4**, 361–369 (1998)
2. Butcher, J.C.: Numerical Methods for Ordinary Differential Equations. Wiley, Hoboken (2016)
3. Chabert, G.: Ibex - an interval-based explorer. Online slides (2007). https://agora.bourges.univ-orleans.fr/ramdani/gtmea/legacy/www2.lirmm.fr/ensemble/IMG/pdf/slides_chabert.pdf
4. Chabert, G., Jaulin, L.: Contractor programming. Artif. Intell. **173**(11), 1079–1100 (2009)
5. Collavizza, H., Delobel, F., Rueher, M.: Comparing partial consistencies. Reliable Comput. **5**(3), 213–228 (1999)
6. dit Sandretto, J.A., Chapoutot, A.: Validated explicit and implicit Runge-Kutta methods. Reliable Comput. **22**(1), 79–103 (2016)
7. Garcia Contreras, A., Throneberry, G., Olumoye, O., Valera, L., Ceberio, M., Abdelkefi, A.: Interval-based solving techniques for large-scale dynamical systems. In: Proceedings of the 2020 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC-CIE) (2020)
8. Contreras, A.F.G., Ceberio, M.: Solving dynamical systems using windows of sliding subproblems. In: Figueroa-García, J.C., Díaz-Gutierrez, Y., Gaona-García, E.E., Orjuela-Cañón, A.D. (eds.) WEA 2021. CCIS, vol. 1431, pp. 13–24. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86702-7_2
9. Goubault, E., Putot, S.: Under-approximations of computations in real numbers based on generalized affine arithmetic. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 137–152. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_9
10. Granvilliers, L., Benhamou, F.: Algorithm 852: realpaver: an interval solver using constraint satisfaction techniques. ACM Trans. Math. Software (TOMS) **32**(1), 138–156 (2006)
11. Lin, Y., Stadtherr, M.A.: Validated solutions of initial value problems for parametric odes. Appl. Numer. Math. **57**(10), 1145 (2007)
12. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. Int. J. Pure Appl. Math. **4**(4), 379–456 (2003)
13. Moore, R.E.: Methods and applications of interval analysis. SIAM (1979)
14. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to interval analysis. SIAM (2009)
15. Olumoye, O., Throneberry, G., Garcia, A., Valera, L., Abdelkefi, A., Ceberio, M.: Solving large dynamical systems by constraint sampling. In: Figueroa-García, J.C., Duarte-González, M., Jaramillo-Isaza, S., Orjuela-Cañon, A.D., Díaz-Gutierrez, Y. (eds.) WEA 2019. CCIS, vol. 1052, pp. 3–15. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31019-6_1

16. Rump, S.M., Kashiwagi, M.: Implementation and improvements of affine arithmetic. Nonlinear Theory Appl. Inst. Electron. Inf. Commun. Eng. (NOLTA-IEICE) **6**(3), 341–359 (2015)
17. Valera, L., Garcia, A., Gholamy, A., Ceberio, M., Florez, H.: Towards predictions of large dynamic systems' behavior using reduced-order modeling and interval computations. In: Proceedings for the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 345–350. IEEE (2017)