Solving Dynamical Systems Using Windows of Sliding Subproblems

Angel Fernando 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 this article, we propose an approach that "slices" large systems into smaller, overlapping ones that are solved using constraint-solving techniques. Our goal is to reduce the computation time and control overestimation, at the expense of solving multiple smaller problems instead of a larger one. We share promising preliminary experimental results.

1 Introduction

Phenomena that change over time are abundant in nature. We model their behavior using dynamical systems, i.e., differential equations to describe how they change over time. For some real life problems, analytical solutions exist that provide an exact description of the behavior. For many other problems, such solutions do not exist, so we use numerical approximations: we *discretize* the original continuous problem over time, where each intermediate state of the system at each discrete time depends on previous and/or subsequent states. This process may result in a very large set of equations, depending on the level of granularity that is sought.

There exist many efficient and useful tools to solve dynamical systems, which might not be well-suited for some types of problems. For example, Runge-Kutta-based solution techniques do not easily handle uncertainty on the parameters, although inherent to real world measurements. Solutions are heavily reliant on an initial set of parameters. When a problem has multiple solutions, such methods do not identify how many solutions there are or whether the found solution is the best based on some criteria. When a solution cannot be found, it is not clear whether the solving technique failed to find one or none exists.

In our research, we use interval-based methods [9,10], which provide guaranteed numerical computations. These techniques guarantee that if a solution exists, it will be found, and that if none exists, it will report this with certainty. There exist two main categories of interval-based methods to solve dynamical systems: *step-based methods* that generate an explicit system of equations one discretized state at a time, and *constraint-solving techniques* that solve the entire system of implicitly discretized equations. Step-based methods are fast, but on complex systems (either because they are simulating longer times or the differential system is very non-linear), their output provides overestimated solution ranges. Constraint-solving techniques can better control the overestimation by working on the entire system at once, but will take considerably longer to report a reasonable solution range.

In this work, we introduce a heuristic approach based on the structure of a dynamical system as a constraint satisfaction problem, solving multiple smaller overlapping sub-problems. We define the parameters that determine how big the sub-problems are and how much they overlap. We test the effect that these parameters have on the quality of the interval solution and the total execution time to solve the given problem, and compare the performance against interval-based dynamical systems solvers. We conclude that our heuristic shows promise.

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. In this research, we focus on *numerical methods* that *approximate* the actual solution. A key advantage of these methods is that they can provide good results even if the exact solution cannot be found through other methods. Their drawback is that they are not perfect and always have a margin of error that must be included in the computation. Fortunately, this error can be minimized by choosing the right type of numerical method for the problem and tweaking the parameters that generate the approximation.

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. To solve this kind of problem, it is possible to simply evaluate each discretized state equation in order, to obtain the values for all states in succession, as each state equation already has the values it needs from previous states. *Implicit* methods involve past and future states in their discretization. These equations cannot be solved by simple successive evaluation. They are often solved using 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: $\boldsymbol{X} \diamond \boldsymbol{Y} = \{x \diamond y,$ 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: $\boldsymbol{X} \diamond \boldsymbol{Y} = \square \{x \diamond y,$ 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 $\square \{f(x_1, \ldots, x_n),$ 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 [9,10].

Step-Based Methods for Solving Dynamical Systems. Such algorithms use 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 [7] uses Taylor polynomials for discretization and Taylor models [1,8] for evaluation; DynIBEX [4] uses Runge-Kutta discretization and evaluates its functions using affine arithmetic [5,12].

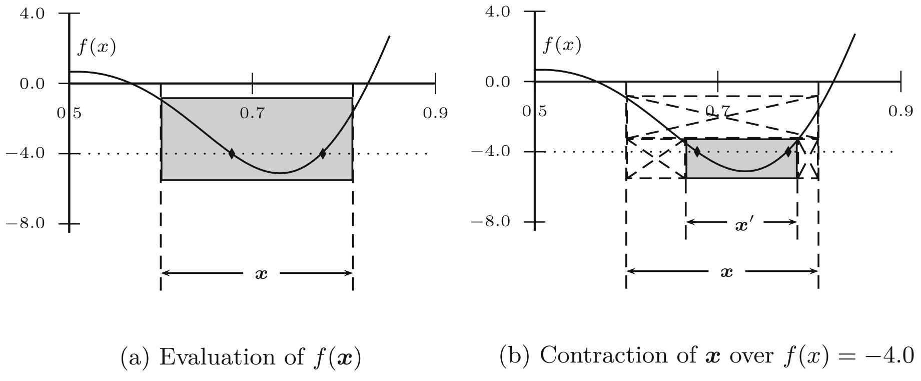
Interval Constraint-Solving Techniques. The methods used to solve a dynamical system using explicit discretization do not work for implicit discretization. We need to solve the entire system. We can do this if we treat the state equations as a system of equality constraints and the dynamical system as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

Definition 3. An interval *constraint satisfaction problem* is given by the tuple $P = (X, \boldsymbol{X}, C)$, where $X = \{x_1, \ldots, x_n\}$ is a set of n variables, with associated interval domains $\boldsymbol{X} = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\}$ and a set of m constraints $C = \{c_1, \ldots, c_m\}$

The initial interval domain $\boldsymbol{X}$ 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.

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 an algorithm called *branch-and-prune*. 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 $m(\boldsymbol{x}) = \frac{\underline{\boldsymbol{x}} + \overline{\boldsymbol{x}}}{2}$. These two new sub-boxes, $\boldsymbol{X_L} = \left\{\boldsymbol{x_0}, \ldots, \left[\underline{x_i}, m(\boldsymbol{x_i})\right], \ldots, \boldsymbol{x_n}\right\}$ and $\boldsymbol{X_U} = \left\{\boldsymbol{x_0}, \ldots, \left[m(\boldsymbol{x_i}), \overline{x_i}\right], \ldots, \boldsymbol{x_n}\right\}$, 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.



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

Fig. 1. Visual example of interval domain contraction

Interval constraint solvers such as RealPaver [6] and IbexSolve [2,3] 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 [13], and focusing on a subset of constraints at a time [11].

2.4 Step-Based or Constraint-Based?

Step-based methods (VSPODE, DynIbex) work well, up to a point. They rely on a dynamically-computed step size aimed at minimizing the error intrinsic to the approximation. As these are interval-based methods, this means they compute an enclosure of the solution that incorporates the approximation error. This error introduces a small amount of overestimation into the solution. After computing multiple states, each with their respective computed step size h, the overestimation that accumulates at every iteration can become too large to be useful in computing a new h. If the solver cannot compute a new h, the simulation stops, even before reaching the expected final state at t_f.

Solving a full system using interval constraint-solving techniques can explore multiple realizations of the system with a desired width for the enclosure. Every state equation is evaluated multiple times, potentially increasing the contraction of the initial domain for the state variables involved. Even with a static value of h for all states, implicit approximations used increase the accuracy of the approximation.

The main reason why step-based methods are often preferred is simple: interval constraint-solving techniques are slower. Interval-based domain contractors evaluate each equation multiple times and the branch-and-prune-based algorithms used within constraint solvers create subproblems exponentially based on the number of variables. The exponential amount of subproblems combined with the multiple evaluations of each equation per subproblem results in algorithms that can provide strong guarantees on the solution but at a considerably higher computation time.

So, on one hand, a family of methods that is fast but falls pray to overestimation; on the other hand, methods that reduce domains with a higher computation time. Let us compare these methods using a complex dynamical system. For this example, we have chosen a three-species food chain model with Holling II predator response functions:

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

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

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

We ran this problem with VSPODE, DynIBEX, and IBEX. We used same parameters for the system, the best settings for their respective algorithms (i.e. VSPODE and DynIBEX use dynamic step size, DynIBEX uses its most accurate Range-Kutta discretization). For IBEX, we generated a set of state equations using trapezoidal discretization, with a step size of $h = 0.01$. The dynamical system as solved up to $t_f = \{40, 100\}$ for a total of $N = 4000, 10000$ states, respectively. Figure 2 shows the plots of the solution for VSPODE and DynIBEX; there is no plot for the results of IBEX, as of the time of writing, the solver has been working for about three weeks without returning a single solution.

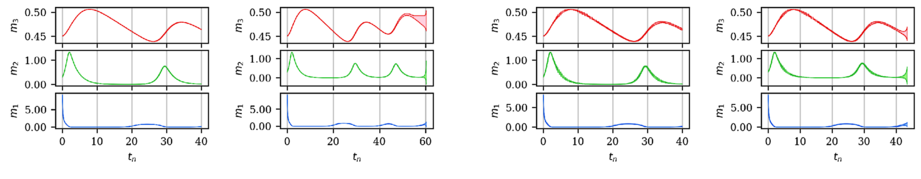


Fig. 2. State enclosures VSPODE and DynIBEX when $t_f = \{40, 100\}$

The plots for $t_f = 40$ look like a single line, but actually represent narrow ranges. This is more evident in the $t_f = 100$ plots, in which overestimation starts separating the lower and upper bounds. The results for these plots do not reach all the way to $t_f = 100$: at some point between $t = 40$ and $t = 60$, the large overestimation causes these solvers to become unable to dynamically compute a new step-size h, and the solving process stops. This shows an area of opportunity: *can we find a way to reduce that overestimation without an excessive* *cost in computation time?* How can we apply the knowledge of one type technique to the other in order to reduce their respective flaws? In this article, we explore implementing a heuristics that improves the execution time of interval constraint-solving techniques while maintaining accuracy.

3 Problem Statement and Proposed Approach

We want to reduce the computation time and increase the accuracy of interval-based dynamical system solvers. There are instances in which a decision-maker needs accuracy under uncertainty, fast. For example, if they need to recompute the parameters of a problem on-the-fly, after an event causes the state of the problem to change dramatically and with some degree of uncertainty. Existing tools can already produce good results, often at a cost: either the method is fast but has less accuracy, or its accuracy increases at the expense of additional and potentially prohibitive computation time [14].

We believe that combining ideas from step-based methods and interval constraint solvers can yield better solutions to dynamical systems in a reasonable amount of time. In this work, we outline a preliminary approach, focused on re-examining how an interval constraint solver works through the state equations of a dynamical system: a heuristic that takes advantage of the problem's structure to speed up the solving process.

The idea 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 $X_{\text{sub}} = \{x\left(j\right), \dots x\left(j + N_w\right)\}$ with domains $\{\boldsymbol{x}\left(\boldsymbol{j}\right), \dots, \boldsymbol{x}\left(\boldsymbol{j} + \boldsymbol{N_w}\right)\}$, along the following set of state equations as a system of constraints C_{sub}:

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

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 N_w. Our technique aims to speed up the computation process of interval constraint solvers by sequentially creating and solving a series of subproblems of size N_w.

The first subproblem involves the initial conditions of the problem. However, we cannot treat subsequent subproblems as smaller initial value problems, with the initial conditions taken from the last state of the previous subproblem. When doing this, we treat the new subproblem as an independent initial value problem and lose the trajectory created by the values of the states from the previous subproblem.

Our solution is to transfer multiple state values between subproblems. Solving the k-th subproblem $P_k = (X_k, C_k)$ yields a reduced domain X_k^* representing N_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}:

$$\{\boldsymbol{x_{k+1}}\left(\boldsymbol{1}\right), \dots, \boldsymbol{x_{k+1}}\left(\boldsymbol{o}\right)\} = \{\boldsymbol{x_k}\left(\boldsymbol{N_w} - \boldsymbol{o}\right), \dots, \boldsymbol{x_k}\left(\boldsymbol{N_w}\right)\}$$

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 N_w. Figure 3 shows a graphical representation of how o states are transferred from one subproblem to the next.

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. We want to find out the impact that N_w and o have in the process, both in terms of execution time, but also on the quality of said solution. Our hypothesis is that smaller sizes of N_w will be faster but with greater imation. Regarding o, we believe smaller sizes will have a similar effect.

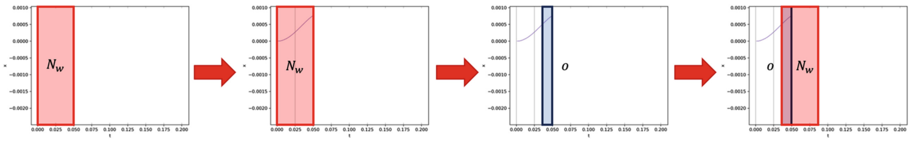


Fig. 3. Graphical plot of overlap transfer

4 Experimental Results and Analysis

4.1 Methodology

We compare the sliding windows heuristic against existing methods to solve dynamical systems using intervals (we chose VSPODE and DynIBEX), with the same three-species food chain system, with the same parameters, step size $h = 0.01$, solving up to $t_f = \{40, 100\}$. This creates two different problems to compare for: a problem with $N = 4000$ discrete times, with one state per species or a system of 12000 equations; and a problem with $N = 10000$ discrete times and 30000 equations.

We consider three metrics to compare the sliding windows heuristic against existing methods:

- *Quality of the solution (Quality).* The solution of a given dynamical system found by an interval-based solver is given as interval values that enclose the real solutions. Due to the overestimation inherent in interval computations, the boundaries of this interval might not be a perfectly narrow enclosure. This metric is the max interval width across all state values. As we want the most narrow enclosures possible, the closer this value is to 0, the better quality the solution has
- *Total execution time (Execution time).* The total computation time spent by each specific algorithm/solver, in seconds. This is a comparison metric for methods that provide similar results: if two methods provide equivalent overall quality, the faster is preferable. However, a method that provides wider enclosures but takes considerably less computation time might be preferable to a potential decision maker's problem (i.e. near real-time systems).

– *States until overestimation (S. Over).* In interval computations, overestimation is inherent. When re-using interval quantities with overestimation in interval computations, the overestimation across different interval values might be compounded. When using interval computations to solve dynamical systems, the first discretized states will be narrower than states further in the simulation's future. This metric represents the first state in the simulation at which overestimation becomes too large; we consider an interval state value to be overestimated if the supremum of its interval value is 10% above its midpoint. For this metric, if a solution has overestimation, a value that is closer to t_f is better.

We also aim to explore how the sliding windows parameters affect the solution quality and execution time. We explored the following parameters/values: the *window size*: $N_w = \{20, 50\}$ the *overlap*: expressed as a percentage of N_w, $o = \{30\%, 50\%, 70\%\}$.

We implemented the sliding heuristic using the default solver in IBEX to solve each individual subproblem. We set the default solver to contract domains into solutions of width 10^{-8}, to stop the solving process after 900 s, and finally, if multiple solutions are found, we take the hull that encloses them.

4.2 Experiments

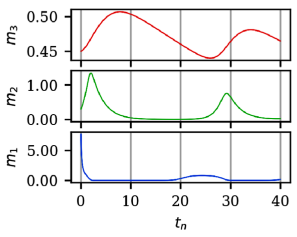
Figure 4 show the plots for all the experiments using sliding windows. Table 1 shows the comparison of metrics for DynIBEX and VSPODE against the different variations of the sliding algorithm.

4.3 Results Analysis

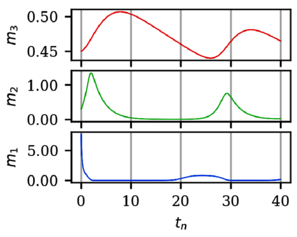
As shown in the metrics of Table 1, there is value for the "States until overestimation" metric any of the experiments using our slide heuristic on $t_f = 40$ because there is no overestimation up to that point – all the states are narrow, as seen in the "Quality" column.

Regarding the heuristic parameters, based on our experimental results, we conclude that, for the food chain problem, the size of the window N_w does not have a significant impact in the width of the obtained solution, though with a larger value for N_w, there is a slight improvement on the number of narrow states, as seen in the "States until overestimation" column for $t_f = 100$. The main drawback is that it requires a significantly longer computation time. Regarding o_w, its influence on the narrowness is similar to N_w's. In the results for up to $t_f = 40$, there is no significant difference in the quality of the solution, only on the execution time.

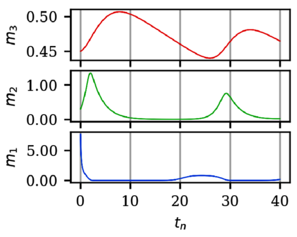
The solutions obtained by our heuristic are more relevant when compared with the solutions from other interval-based solvers. For the food chain problem, our heuristic has a later "state until overestimation" than the other two methods. This means that the slide heuristic, in all its variants, manages to return a solution that remains narrow for a longer number of states.



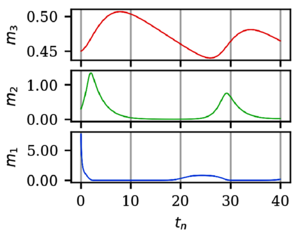
(a) $o = 30\%$



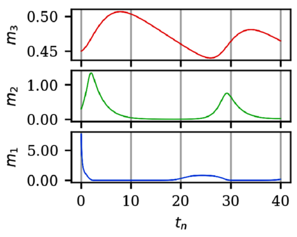
(b) $o = 50\%$



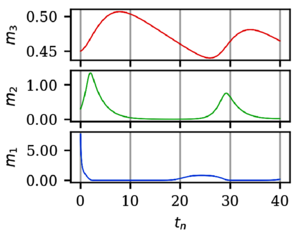
(c) $o = 70\%$



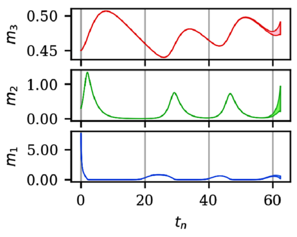
(d) $o = 30\%$



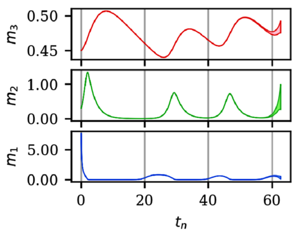
(e) $o = 50\%$



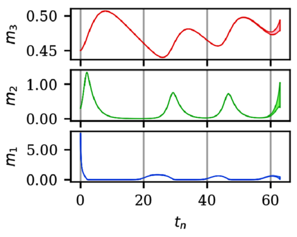
(f) $o = 70\%$



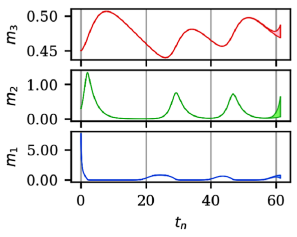
(g) $o = 30\%$



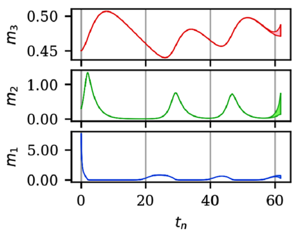
(h) $o = 50\%$



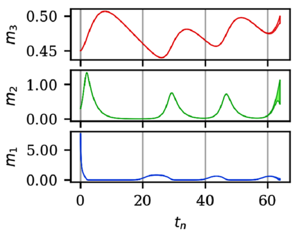
(i) $o = 70\%$



(j) $o = 30\%$



(k) $o = 50\%$



(l) $o = 70\%$

Fig. 4. (a,b,c) Plots for the sliding windows with $N_w = 20$ and $t_f = 40$. (d,e,f) Plots for the sliding windows with $N_w = 50$ and $t_f = 40$. (g,h,i) Plots for the sliding windows with $N_w = 20$ and $t_f = 100$. (j,k,l) Plots for the sliding windows with $N_w = 50$ and $t_f = 100$.

Table 1. Table of metrics

	Problem 1 ($t_f = 40$)			Problem 2($t_f = 100$)		
Solver	Quality	Exec. time	S. Over	Quality	Exec. time	S. Over
VSPODE	1.0000	3189.0619	29.09	1.5848	5351.2794	29.09
DynIBEX	0.4879	30.6470	1.50	1.1136	57.8751	1.50
$N_w = 20$, $o = 30\%$	6.1330E-04	313.6700	–	0.7437	741.1903	49.22
$N_w = 20$, $o = 50\%$	5.6379E-04	459.5741	–	0.7341	1273.4984	49.98
$N_w = 20$, $o = 70\%$	5.1991E-04	934.7888	–	0.7282	1346.8824	51.16
$N_w = 50$, $o = 30\%$	2.5693E-04	1856.3104	–	0.5853	3149.8939	50.73
$N_w = 50$, $o = 50\%$	2.1312E-04	2454.8584	–	0.6157	5124.8888	53.13
$N_w = 50$, $o = 70\%$	1.9722E-04	3258.1111	–	0.7016	5636.2485	61.64

DynIBEX is fast, but the intermediate results it reports are not as narrow, as seen in Fig. 2. Between VSPODE and our heuristic, there are two main differences: the rate at which the overestimation increases after surpassing the expected max width, and the time it takes to reach that solution. Our heuristic with a small size of N_w is faster, with a slower increase in overestimation than VSPODE. It is possible that this be due to the differences between the discretization schemes, and the current "simplified" discretization in our heuristic. Even when considering this, the similar solution quality with the faster performance shows that our approach is promising.

5 Conclusions and Future Work

Based on the results we presented, the sliding windows heuristic shows promise in providing narrower results than step-based interval solvers. It computes solutions faster than VSPODE, given the right combination of parameters N_w and o. Increasing the window size N_w produces results that will not reach overestimation until later, at the expense of much greater computational time. Increasing the overlap o has a similar effect, though not as pronounced.

We plan to further explore the potential of this technique by applying and comparing it with other challenging non-linear problems. We are looking into experimenting with different discretization schemes to reduce the error. Finally, we plan to examine how these techniques fare against other approaches that reduce the 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**(4), 361–369 (1998)
2. Chabert, G.: Ibex, an interval-based explorer (2007)

3. Chabert, G., Jaulin, L.: Contractor programming. Artif. Intell. **173**(11), 1079–1100 (2009)
4. dit Sandretto, J.A., Chapoutot, A.: Validated explicit and implicit Runge-Kutta methods. Reliable Comput. **22**(1), 79–103 (2016)
5. 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
6. Granvilliers, L., Benhamou, F.: Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques. ACM Trans. Math. Softw. (TOMS) **32**(1), 138–156 (2006)
7. Lin, Y., Stadtherr, M.A.: Validated solutions of initial value problems for parametric odes. Appl. Numer. Math. **57**(10), 1145 (2007)
8. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. Int. J. Pure Appl. Math. **4**(4), 379–456 (2003)
9. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to interval analysis. SIAM (2009)
10. Moore, R.E., Moore, R.: Methods and Applications of Interval Analysis, vol. 2. SIAM (1979)
11. 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
12. Rump, S.M., Kashiwagi, M.: Implementation and improvements of affine arithmetic. Nonlinear Theory Appl. IEICE **6**(3), 341–359 (2015)
13. 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 of the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 345–350. IEEE (2017)
14. Valera, L., Contreras, A.G., Ceberio, M.: "On-the-fly" parameter identification for dynamic systems control, using interval computations and reduced-order modeling. In: Melin, P., Castillo, O., Kacprzyk, J., Reformat, M., Melek, W. (eds.) NAFIPS 2017. AISC, vol. 648, pp. 293–299. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67137-6_33