Interval Constraint-Solving Strategies For Solving Dynamical Systems

Angel Fernando Garcia Contreras

University of Texas at El Paso

Follow this and additional works at: https://scholarworks.utep.edu/open_etd

Part of the Computer Sciences Commons

Recommended Citation

https://scholarworks.utep.edu/open_etd/3411

This is brought to you for free and open access by ScholarWorks@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of ScholarWorks@UTEP. For more information, please contact lweber@utep.edu.
INTERVAL CONSTRAINT-SOLVING STRATEGIES FOR SOLVING DYNAMICAL SYSTEMS

ANGEL FERNANDO GARCIA CONTRERAS

Doctoral Program in Computer Science

APPROVED:

Martine Ceberio, Ph.D., Chair

Vladik Kreinovich, Ph.D.

Heidi A. Taboada-Jimenez, Ph.D.

Mark A. Stadtherr, Ph.D.

Abdessattar Abdelkefi, Ph.D.

Stephen Crites, Ph.D.
Dean of the Graduate School
To my Family.

I would not be here without your love and patience.
I still remember how, after finishing my Masters, my advisor Dr. Martine Ceberio kept mentioning how I was definitely Ph.D. material. I also remember refusing at first, thinking how difficult and time consuming it would be to go for a doctoral degree. Now, at that road’s end, I can say with confidence: it was very difficult, took longer than I would have expected or wanted, but you were right. Thank you for your guidance and patience, but most of all for believing in me.

The Constraint Research and Reading Group at UTEP were a sounding board to many ideas that eventually became the work of this dissertation. I am thankful to all members I had the privilege of working with. In particular I want to thank Leobardo Valera, Jonatan Contreras and Omeiza Olumoye: frequent collaborators and contributors, the best companions that listened to the ramblings that led to most of this work.

To my family: I keep moving forward only because I know you have my back. I would not have made it this far without you. Even at the worst moments, you held on to me with unwavering love and patience, and even now stand by me full of love, pride and support.

The Computer Science Department and the University of Texas at El Paso always provided me with opportunities to grow and advance in my career path. I believe in empowering people through education, and that Computer Science is for everyone. This pair of beliefs were born in our Computer Science department, from my many interactions and work with the faculty and staff. I am proud of my education at UTEP, and of the work at its Computer Science department.

NOTE: This dissertation was submitted to my Supervising Committee on October 27, 2021.
ABSTRACT

There are abundant phenomena that humans can describe through mathematical models. *Dynamical systems* are one such type of model, describing the behavior of phenomena that change over time. For example, a scientist can measure and analyze an insect’s wing parameters and movements to create a dynamical system of that behavior. We can then use this model in different applications, such as creating a nano vehicle with insect-like propulsion.

For many real life problems, there exist analytical solutions. These can represent a full description of the state of a dynamical system at any moment in *continuous time*. However, in most practical cases, there are no such analytical solutions. Instead, we use *numerical methods*: to find an *approximation* of the state of a system at a specific (future) time, we select a series of *discrete* moments in time between the start and end times, at which we will compute intermediate state approximations. This *discretization* process can turn even a dynamical system with a single equation into a system of dozens if not hundreds or thousands of equations and artificial variables representing these intermediate states over time. With the aid of computing algorithms and tools, it is possible to solve these complex systems in feasible time.

The tools commonly used to solve dynamical systems are efficient and useful, but are not well-suited to solve certain types of problems; for example, identifying changes in the parameters of an unfolding event using on-the-fly observations, or finding the parameters that minimize the power consumption of our nano vehicle. Existing tools can find one solution, but they depend on an initial guess; if multiple solutions exist, they cannot guarantee to have found *all* of them. On occasion, these methods will not find a solution, and cannot guarantee that they could not find it because no such solution exists.
In our work, we use interval methods, which are guaranteed to find solutions if any exists, and guarantee that if none is found then the system has no solution. There are two categories of methods used to solve dynamical systems: step-based methods that compute each state by using *explicit* discretizations in which each state equation uses previously-computed states; and constraint-solving techniques that work on the entire system of state equations generated using *implicit* discretization, whose state equations involve previous and future states.

The research we present in this document concerns interval constraint-solving techniques used to solve dynamical systems. There are two main drawbacks for using constraint-solving techniques: the first is the amount of time it takes to solve an implicitly discretized dynamical system, which can be impossibly long or just unfeasible. The second is the need for a discretization that is guaranteed to be accurate.

Here we present the Sliding Windows algorithm, a promising strategy that uses interval constraint-solving techniques in a novel way to solve a discretized dynamical system in a more feasible amount of time compared to traditional strategies that employ constraint-solvers. We also use this algorithm to compare discretization techniques of various levels of complexity, to identify the ones that provide better accuracy without excessively extending computation time.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xi</td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2  BACKGROUND</td>
<td>7</td>
</tr>
<tr>
<td>2.1  Dynamical systems</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1  A motivating example: the Lotka-Volterra model</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2  Definitions</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3  Examples</td>
<td>16</td>
</tr>
<tr>
<td>2.2  Interval analysis</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1  Real intervals and interval arithmetic</td>
<td>20</td>
</tr>
<tr>
<td>2.2.2  Implementation details of interval methods</td>
<td>27</td>
</tr>
<tr>
<td>2.2.3  Constraint solving</td>
<td>32</td>
</tr>
<tr>
<td>2.2.4  Interval constraint-solving techniques</td>
<td>34</td>
</tr>
<tr>
<td>2.2.5  Interval constraint solvers</td>
<td>39</td>
</tr>
<tr>
<td>2.3  Uncertainty in dynamical systems</td>
<td>42</td>
</tr>
<tr>
<td>2.3.1  Existing interval-based ODE solvers</td>
<td>45</td>
</tr>
<tr>
<td>2.3.2  Solving dynamical systems using interval constraint solvers</td>
<td>47</td>
</tr>
<tr>
<td>2.3.3  Comparison of interval techniques for dynamical systems</td>
<td>48</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.3.4</td>
<td>The challenge of solving large-scale dynamical systems with interval</td>
</tr>
<tr>
<td></td>
<td>constraint solvers</td>
</tr>
<tr>
<td>3</td>
<td>PROBLEM STATEMENT</td>
</tr>
<tr>
<td>4</td>
<td>CONTRIBUTIONS</td>
</tr>
<tr>
<td>4.1</td>
<td>The Sliding Windows Algorithm</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Motivation: an example</td>
</tr>
<tr>
<td>4.1.2</td>
<td>The Algorithm</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Complexity analysis</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Experiments, results and analysis</td>
</tr>
<tr>
<td>4.2</td>
<td>Improving enclosure accuracy in the sliding Windows algorithm</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Higher-order approximations</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Manipulating the enclosures</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Experiments, results and analysis</td>
</tr>
<tr>
<td>5</td>
<td>CONCLUSIONS</td>
</tr>
<tr>
<td>5.1</td>
<td>Conclusions</td>
</tr>
<tr>
<td>5.2</td>
<td>Future work</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
</tr>
<tr>
<td></td>
<td>CURRICULUM VITA</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Physical schematic for a nano air vehicle with flapping insect wings . . . . . 5
2.1 Illustrative example of Lotka-Volterra behavior . . . . . . . . . . . . . . . 9
2.2 Examples of Three-Species Lotka-Volterra. . . . . . . . . . . . . . . . . . . 18
2.3 Example of Bratu boundary value problem. . . . . . . . . . . . . . . . . . . 19
2.4 Illustration of the wrapping effect . . . . . . . . . . . . . . . . . . . . . . . 25
2.5 Visualization of real vs. machine-representable numbers . . . . . . . . . . . 27
2.6 Illustration of constraint $x^2 + y^2 \leq 16$ . . . . . . . . . . . . . . . . . 33
2.7 Example of domain contraction using constraints. . . . . . . . . . . . . . . 36
2.8 Illustration of Newton method . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.9 Interval Newton method contracting interval $x$ into $x'$ . . . . . . . . . . . 38
2.10 HC4Revise contractor applied on constraint $2x = z - y^2$ . . . . . . . . . . 40
2.11 Experiment 1, RealPaver results . . . . . . . . . . . . . . . . . . . . . . . . 52
2.12 Experiment 1, VSPODE results . . . . . . . . . . . . . . . . . . . . . . . . 53
2.13 Experiment 1, IbexSolve results . . . . . . . . . . . . . . . . . . . . . . . . 54
2.14 Experiment 2, RealPaver-BC5 results . . . . . . . . . . . . . . . . . . . . 57
2.15 Experiment 2, RealPaver-3B results . . . . . . . . . . . . . . . . . . . . . . 57
2.16 Experiment 2, VSPODE results . . . . . . . . . . . . . . . . . . . . . . . . 58
2.17 Experiment 2, IbexSolve results . . . . . . . . . . . . . . . . . . . . . . . . 58
4.1 Visualization of the overlap transfer . . . . . . . . . . . . . . . . . . . . . . 72
4.2 Plot of results for the flapping wing problem when $t_f = 20.0$ . . . . . . . . 79
4.3 Plot of results for the flapping wing problem when $t_f = 80.0$ . . . . . . . . . 80
4.4 Parameter values for three-species food chain . . . . . . . . . . . . . . . . . 81
4.5 Three species food chain state enclosures, VSPODE and DynIBEX . . . . . . 83
4.6 Three species food chain state enclosures, Sliding Windows with $o = 30\%$ . . 84
4.7 Three species food chain state enclosures, Sliding Windows with $o = 50\%$ . . 85
4.8 Three species food chain state enclosures, Sliding Windows with $o = 70\%$ . . 86
4.9 Sample plots for DynIBEX solution of three species food chain . . . . . . . . 88
LIST OF TABLES

2.1 Various representations of $\pi$ ............................................. 28
2.2 Model parameters for flapping wings of various insects ............... 50
2.3 Experiment 1, running times (in seconds) ............................. 51
2.4 Experiment 2, running times (in seconds) ............................. 59

4.1 Nomenclature for the Sliding Windows algorithm ........................ 69
4.2 Running times for Sliding Windows on the flapping wing experiment. 78
4.3 Table of metrics for $t_f = 40$ ............................................ 85
4.4 Table of metrics for $t_f = 100$ ............................................ 86
4.5 Comparison of time states until overestimation ........................ 94
4.6 Comparison of coverage in the experiments. .......................... 95
4.7 Comparison of total accumulated error in the experiments. .......... 95
4.8 Comparison of total average error in the experiments. ............... 95
4.9 Comparison of coverage average error in the experiments. ........... 96
4.10 Comparison of weighted coverage in the experiments. ............. 97
4.11 Comparison of weighted total accumulated error in the experiments. 97
4.12 Comparison of weighted total average error in the experiments. .... 97
4.13 Comparison of weighted coverage average error in the experiments. 98
INTRODUCTION

Historically, the human species has sought to explain and predict the world around us. Armed with scientific progress and ingenuity, humanity has developed abstract models used to represent the physical world through logic and mathematics. A dynamical system is a type of model describing the behavior of phenomena that change over time.

Phenomena that change over time, both natural entities and man-made creations, are abundant. It is no surprise, then, that the study of dynamical systems used to model chronologically evolving behavior attracts the attention of scientist and engineers alike, in both practical and theoretical problems. To give some examples: in economics, there are models for the behavior of market shares [51], and the influence wage-earning workers and profit-seeking capitalists have on each other’s income over time [73]; in medicine, the rate at which a disease spreads across individuals is used to manage potential epidemics [62, 94]; in mechanics, almost any type of motion is described by differential equations, and can be used from a practical perspective to design more efficient and resistant devices of all types and sizes, from improved automobile suspension systems [22, 29], to mobile phones and electric cars with optimized battery usage [14, 63].

Dynamical systems are used to represent and analyze the behavior of phenomena over time. These are given through differential equations, which describe how phenomena change, but not what their state will be at a given time. Solving a dynamical system means finding such a state or states. For example, with a differential equation that describes how an infectious disease spreads, we might be interested in the amount of infected people after a month, two months, or even a year after the initial outbreak; solving this system means finding those population numbers or a function of their trend over time.
In complex differential equations, this process requires creating mathematical systems with dozens, if not hundreds or even thousands of variables and equations. Even as late as the 19th century, finding the solution to a system with that number of equations was unthinkable, because of the amount of by-hand computations required. The advances in mechanical, and subsequently electronic, computations throughout the 20th and early 21st centuries have led to the development of techniques that take advantage of computational power to solve these large, complex problems that were previously considered untractable. Many computational tools have been built over the years that address different aspects of dynamical phenomena. For example:

- MATLAB [44] is a numerical computing environment and language. Its focus on matrix algebra and plotting make it a popular choice to solve problems in science and engineering. Out of the box, it features many solvers for ordinary and partial differential equations, and boundary value problems.

- Wolfram Mathematica [78] is a computing system that aims to be intuitive while providing a vast array of computing options to its developers. Differential equations can be written in a way that closely resembles mathematical notations, which can then be solved either numerically or symbolically, and plotted, all in a single line of code.

- SageMath [76] is a computing system that uses a Python-like syntax. It allows its users to interact with multiple open source numerical computation packages written for FORTRAN, C and Python seamlessly. It includes a module focused on discretized dynamical systems that allows users to solve and analyze a system’s behavior.

- COPASI [21] is a simulation and analysis software inspired by and used for biochemical systems. It embeds multiple methods to solve and simulate a system, as well as numerical analysis methods such as optimization, parameter estimation, sensitivity analysis, and others.

These tools are widely used to solve a variety of problems involving dynamical systems,
such as simulations, parameter estimation, optimization, just to name a few. However, there are types of problems for which the aforementioned tools may be unsatisfactory. What if we want to identify the parameters of an unfolding event based on on-the-fly observations? Or if we want to identify the parameters that guarantee an insect-sized flying nano-vehicle can flap its wings and fly with the least amount of electric power? The aforementioned tools can be used to address these problems, but require extra processing and may not return expected or desirable results. We will show how these tools are challenged through an example.

Let us say that a decision maker is looking for the parameter values of a dynamical system that conforms to a specific, observed behavior. It is possible to use parameter estimation techniques to find those values based on observations. These packages also require an initial guess of the parameter values sought, which then will be incrementally improved until the improved parameters cause the model’s behavior to get as close as possible to the real observed behavior.

In many cases the values obtained like this are sufficient and the decision maker is happy with them. However, the results obtained by the package will not always satisfy the decision maker. What if the parameter values found is not really close to the real behavior? The package only minimizes the difference between the models and the observed behaviors based on an initial guess of the parameter values, there might be other different “initial guesses” that lead to a model behavior closer to the observed one. What if the package does return any value? It can just mean that the solver could not find it based on the “initial guess” of the parameters. However, there is no guarantee that the model had no valid parameter values, only that they were not found given the initial guess.

Even if a value is found, it is based on observed measurements, and observations are inherently inaccurate. Even the most precise measurement tools have a margin of error. Classical parameter estimation methods use techniques that attempt to address this error by applying methods such as least squares, which minimize the discrepancy between observed and expected data according to the model being measured. However, if the expected data is very different from the observed, or even if there is no expected data, these methods may
not be able to provide a result that satisfies the decision maker, or worse, the results might mislead them into making bad decisions.

There are tools and techniques that can deal with the challenges found in the packages described above. There exist packages that provide computational guarantees, such as guaranteeing a minimal difference between computed and observed behaviors, by using interval analysis techniques. The main idea is to model quantities as ranges that contain multiple numbers, and do computations over ranges or intervals in such a way that the result also contains all possible results inside an interval.

In dynamical systems, this means that the behavior is represented by intervals as well: in the above example, we want to find parameter values such that the model behavior, given as intervals, narrowly encloses the observed behavior. If we start with very wide intervals, we can “remove” the parts of those intervals that are far away from the observed behavior, similar to how a sculptor shaves away pieces from a block of marble to create a desired shape.

Interval methods do have their drawbacks. Because we are using “numbers” that are represented by a range, there will be more computations so finding a solution can take longer. Also, while interval methods can guarantee to enclose a solution, that does not mean that the enclosure will be very narrow – it is possible that the interval enclosure is so large it does not actually help a decision maker.

When solving dynamical systems, interval methods work reasonably well, up to a point. For example, the methods will result in narrow interval enclosures if the behavior must be modeled over a short period of time or a relatively small number of discrete time states, for example real-time short-term prediction based on sensors. If the time period or the number of states increases, so does the likelihood that the interval enclosure of later states will be too wide to provide useful information; with some techniques, this also increases the required computational resources, such as memory and time, which might cause the program to reach a computer’s limits.
One example of this situation is the design of the nano air vehicle with flapping insect wings. A diagram of the schematics for this vehicle is on Figure 1.1. The dynamical system that governs the movement of its wings is based on its physical parameters and the force employed to move them. To allow the nano air vehicle to fly, the wings start in a rest state, then begin to move. They must flap periodically to ensure the vehicle is flying. Due to its size and the problem requirements, the discretization must be done with small time windows over long time periods.

A simulation of the nano air vehicle’s flapping wings is straightforward. However, solving specific problems can be more difficult when the dynamical system has uncertainty or requires a guarantee, such as when we want all solutions that do not violate physical constraints (i.e. the servos that power the wings cannot go beyond certain force level). Widely used tools (MATLAB, Wolfram, et cetera) use numerical techniques that are fast but can only provide a guarantee under certain conditions, such as a class of problem that is known to have just one solution.

As an illustration, if we wanted to minimize the power consumption of the nano vehicle, we could use a method that improves an “initial guess” of the parameters a little bit at a time. Through numerical computations, we can determine in which direction to “nudge” or change that initial guess to get it closer to the expected behavior, as if we were moving a rock down a hill. We repeatedly “nudge” the parameter values until they cannot improved reasonably enough, at which point we assume to have reached a “minimum”, the bottom of a hill. However, this is only a local minimum, the one closest to our starting point; there might be other values that lead to minima that are even lower in value. Interval techniques start by enclosing all possible values: if we find a local minimum, we “remove” all parameter
values that produce results above the current best minimum, and focus our search for a *global minimum* on the remaining interval spaces.

To summarize: we want to solve problems involving dynamical systems. There are plenty of tools that can solve certain types of problems efficiently, but real life applications are often more complex than what most tools can solve. Particularly, when the problem needs a guarantee on the solution, many existing methods cannot always provide it.

We can provide such a guarantee with interval methods. There are already techniques that use interval methods to solve dynamical systems, but when the discretized system has too many steps/equations we have observed that the interval accuracy of the final solution may not be satisfactory in an actual application. In this document, we present novel research in interval constraint-solving for dynamical systems that helps alleviate some of the observed problems.

This dissertation is structured as follows: Chapter 1 presents a brief introduction to this document and the research that informs its content. On Chapter 2, we provide the theoretical background behind our research: dynamical systems and the types of real life problems involving them; how interval methods work, how provide guarantees when solving dynamical systems problems, as well as the strengths and areas of opportunity found in existing methods. Chapter 3 states the main problem we aimed to solve. We present our contributions in Chapter 4: the Sliding Windows algorithm used to solve large-scale dynamical systems problems, followed by an exploration of discretization methods to use with this algorithm. Finally, summarize our findings and draw directions for future work in Chapter 5.
In this chapter, we show the theoretical background for our work. First, we take a look at dynamical systems, how they are defined as mathematical models, and how these are used to analyze the behavior of phenomena in the world. Second, we explain interval analysis, including motivation behind its use as well as techniques used to solve common problems. In the last section we talk about the intersection of both of the previous topics: how interval analysis is used to solve dynamical system problems with uncertainty.

2.1 Dynamical systems

Dynamical systems are abundant, both in nature and as a product of human ingenuity. To better understand these systems, let us start this section with a sample problem to show its history, definition and uses, including common solving methods.

2.1.1 A motivating example: the Lotka-Volterra model

In the early 20th century, two mathematicians, one from Italy and one from the United States, arrived to the same set of equations. Vito Volterra wanted to predict the behavior of biological predator-and-prey systems, while Alfred Lotka sought to describe certain chemical reactions. As more mathematicians learned about what is now known as the Lotka-Volterra model, they found that they describe a type of behavior found in many time-dependent problems.

The Lotka-Volterra model [12, 18] is the following system of differential equations:
The functions $x(t)$ and $y(t)$ model the population sizes of the predator and prey, respectively, as they change over time. The model uses parameters $a, b, c, d > 0$ in the following way:

- $a$ represents the growth rate of the prey population
- $b$ represents the effect of predator killing prey
- $c$ represents the death rate of the predator population
- $d$ represents the growth rate of the predator population when prey is available

In general, the equations show a behavior that is observable and relatively simple. We see an example of it in Figure 2.1: this shows the behavior between two species, represented by an owl (predator, in blue) and a mouse (prey, in red). The predators can thrive and reproduce when the prey species is large enough to support their population; however, this also reduces the numbers of prey species. At some point, the food source of the predators becomes scarce and their population diminishes, while the prey, no longer under as much of a threat from predators increase their population instead.

This system remains relevant today. Researchers expanded the model to include interactions between multiple populations that prey on each other [18]. They also apply the model outside of biology and chemistry, including control theory [35, 45] and economics [43, 95]. In the following sections, we examine dynamical systems in more detail, starting with definitions.

### 2.1.2 Definitions

**Definition 1.** A *continuous dynamical system* is a system that describes the evolution of a
Figure 2.1: Illustrative example of Lotka-Volterra behavior

phenomenon over time. This description is given as the tuple \((D, T, f)\) where \(D \subseteq \mathbb{R}^n\) is the state space that describes the system state, \(T \subseteq \mathbb{R}\) is a set of times, and \(f : D \times T \to \mathbb{R}^n\) is a function that describes the changes in state from pairs \((x, t)\), \(x \in D, t \in T\).

**Definition 2.** By a trajectory of a dynamical system, we mean a function \(x : [t_0, \infty) \to D\) for which

\[
\frac{dx}{dt} = \dot{x} = f(x, t)
\]

The trajectory of a dynamical system is a solution to a system of differential equations that describe how some quantity changes, usually over time. In this form, the system cannot be used to show the result of those changes: the new predator and prey population, an updated projectile position, or the temperature of a metal plate at a specific moment in time. We need functions or equations that give the actual state, not how it changes. These functions can be obtained by integrating the differential equations.

There are two ways to integrate a differential equation: analytical methods, or numerical methods. Analytical methods, such as symbolic integration, can obtain the exact integral based on the properties of the system. The drawback of the methods in this category is that not all differential systems of equations can be integrated this way: either the system cannot
be integrated symbolically, or the analytical method cannot obtain the exact integral in a feasible, or even reasonable, time.

Numerical methods create approximations to the integral. In a practical scenario, these methods provide results that are close enough to the exact equation to be useful in modeling the behavior of the system. Their drawback is that they require tweaking the parameters of the approximation to find one that can be integrated within a specific time up to a certain margin of error. In this work, we focus on cases in which we can only use numerical methods.

Numerical integration methods

To approximate the integral, most numerical methods create equations that model the state of the system over discrete moments in time. Each one of these moments is separated by a time step (And, in some cases, this separation could also be in space, in addition). A smaller time step means a more accurate approximation, but also a system with more equations. In this section, we look at the following classes of methods of discretization: Euler methods, Taylor series, Runge-Kutta methods, and the Adams-Bashfort and Adams-Moulton methods. The first two are often taught as an introduction to discretization methods, while Runge-Kutta and the Adams methods are widely studied and used in practice.

Euler methods. The idea behind the Euler methods is to use the definition of a derivative based on the limit [6, 41]:

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

When \(\Delta t\) is a non-zero value, we can substitute the right hand side into the formula for the differential equation:

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} = f(x, t)$$

In this new equation, we replace \(\Delta t = h\), \(x(t) = x_n\), and \(x(t + \Delta t) = x_{n+1}\), which results
in the common formulation of the basic (or forward) Euler method:

\[ x_{n+1} = x_n + hf(x_n, t_n) \]

Similarly, there is the backward Euler method:

\[ x_{n+1} = x_n + hf(x_{n+1}, t_{n+1}) \]

The forward Euler method is \textit{explicit}, as every \( x_{n+1} \) value depends on having a value for its predecessor \( x_n \). This makes it suitable for iterative methods that generate a point \( x_n \) for each iteration. The backward Euler method is \textit{implicit}, as both sides of the equation involve \( x_{n+1} \). It is necessary to generate the equations for all time steps. Another implicit Euler variant involves the central difference:

\[ f(x_n, t_n) = \frac{x_{n+1} - x_{n-1}}{2h} \]

\textbf{Taylor series.} We can approximate the derivative by using Taylor series [41]. The Taylor series expansion of an infinitely differentiable function \( f(x) \) at point \( a \) is the power series:

\[ f(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(a)}{i!} (x - a)^i, \]

where \( f^{(i)} \) is the \( i \)-th derivative of \( f \). In practice, this sum is not always possible: not all functions are infinitely differentiable, and it is not feasible to have an infinite sum. We obtain a \textit{Taylor polynomial} instead:

\[ f(x) = \sum_{i=0}^{k} \frac{f^{(i)}(a)}{i!} (x - a)^i + R_k(x), \]

where \( k \) is the order of the polynomial. The function \( R_k(x) \) is the \textit{remainder of order} \( k \), which represents the approximation error of the expansion. The most common form of \( R_k(x) \) is the \textit{Lagrange form}: 

11
\[ R_k(x) = \frac{f^{(k+1)}(c)}{(k+1)!} (x - a)^{k+1} \]

for some \( c \) between \( x \) and \( a \). We do not know the value of \( c \), so we can find a constant that approximates \( f^{(k+1)}(c) \) or even omit the reminder if we can assume its value will be close to 0.

In differential systems, we use Taylor polynomials to approximate the update function. By expanding around \( x_n \), replacing the corresponding terms with the ones used in differential equations, substituting \( x_{n+1} - x_n = h \), we obtain the following approximation:

\[
x_{n+1} \approx \sum_{i=0}^{k} \frac{f^{(i)}(x_n, t_n)}{i!} (h)^i
\]

**Runge-Kutta methods.** This family of methods can be understood by looking at the Euler method, as seen above. The Euler method is an approximation based on the beginning of the interval between \( x_n \) and \( x_{n+1} \). Runge-Kutta improves on this idea by making additional trial steps in between \( x_n \) and \( x_{n+1} \). For example, the second-order Runge-Kutta method is:

\[
\begin{align*}
k_1 &= h f(x_n, t_n) \\
k_2 &= h f(x_n + \frac{1}{2}k_1, t_n + \frac{1}{2}h) \\
x_{n+1} &= x_n + k_2
\end{align*}
\]

One of the most common implementations of this type of method is the fourth order Runge-Kutta method, or RK4:

\[
\begin{align*}
k_1 &= h f(x_n, t_n) \\
k_2 &= h f(x_n + \frac{1}{2}k_1, t_n + \frac{1}{2}h) \\
k_3 &= h f(x_n + \frac{1}{2}k_2, t_n + \frac{1}{2}h) \\
k_4 &= h f(x_n + k_3, t_n + h) \\
x_{n+1} &= x_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4
\end{align*}
\]

More advanced applications of the Runge-Kutta methods are adaptive: they select a
value for \( h \) on each step, such that the local truncation error stays below a user-defined value. This results in more accurate approximations, in which the user does not need to worry about selecting a unique value of \( h \).

**Adams-Bashfort and Adams-Moulton methods.** The Adams-Bashfort method [13] is an explicit *multi-step* method. This means that, unlike the Taylor and Runge-Kutta methods that create approximations by using information about the trajectory between two states, multi-step methods take advantage of multiple states in the overall computation of the dynamical system.

The general Adams-Bashfort method is:

\[
  x_{n+s} = x_{n+s-1} + h \sum_{m=0}^{s-1} b_m f(x_{n+m}, t_{n+m})
\]

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 \( b_m \) is a unique coefficient. These coefficients are independent from \( x_i \) and \( t_i \), determined only by the degree of the approximation, or the number of states involved in it, \( s \). Let us look at the Adams-Bashfort approximations involving \( s = 1, 2, 3, 4, 5 \) steps:

\[
  x_{n+1} = x_n + hf(t_n, x_n)
\]
\[
  x_{n+2} = x_{n+1} + \frac{h}{2} (3f(x_{n+1}, t_{n+1}) - f(x_n, t_n))
\]
\[
  x_{n+3} = x_{n+2} + \frac{h}{12} (23f(x_{n+2}, t_{n+2}) - 16f(x_{n+1}, t_{n+1}) + 5f(x_n, t_n))
\]
\[
  x_{n+4} = x_{n+3} + \frac{h}{24} (55f(x_{n+3}, t_{n+3}) - 59f(x_{n+2}, t_{n+2}) + 37f(x_{n+1}, t_{n+1}))
  - 9f(x_n, t_n))
\]
\[
  x_{n+5} = x_{n+4} + \frac{h}{720} (1901f(x_{n+4}, t_{n+4}) - 2774f(x_{n+3}, t_{n+3}) + 2616f(x_{n+2}, t_{n+2})
  - 1274f(x_{n+1}, t_{n+1}) + 251f(x_n, t_n))
\]
The Adams-Bashfort methods are rarely used on their own. They are used as part of predictor-corrector methods, as the initial prediction of a dynamical system’s behavior. Its corresponding corrector method is the implicit Adams-Moulton method:

\[ x_{n+s} = x_{n+s-1} + h \sum_{m=0}^{s} b_m f(x_{n+m}, t_{n+m}) \]

The main difference between Adams-Bashfort and this formula is that the Adams-Moulton method includes \( x_{n+s} \) on its calculations in both the left and right hand of the equation, which also involves a coefficient \( b_s \). This small change also means that we can have a \( s = 0 \) Adams-Moulton formula, which is the implicit Euler method. The coefficients for Adams-Moulton are computed separately and are different from the Adams-Bashfort ones.

Let us show the Adams-Moulton approximations for orders \( s = 0, 1, 2, 3, 4 \):

\[
\begin{align*}
x_{n+1} &= x_n + hf(t_{n+1}, x_{n+1}) \\
x_{n+1} &= x_n + \frac{h}{2} (f(x_{n+1}, t_{n+1}) + f(x_n, t_n)) \\
x_{n+2} &= x_{n+1} + \frac{h}{12} (5f(x_{n+2}, t_{n+2}) + 8f(x_{n+1}, t_{n+1}) - f(x_n, t_n)) \\
x_{n+3} &= x_{n+2} + \frac{h}{24} (9f(x_{n+3}, t_{n+3}) + 19f(x_{n+2}, t_{n+2}) \\
&\quad - 5f(x_{n+1}, t_{n+1}) + f(x_n, t_n)) \\
x_{n+4} &= x_{n+3} + \frac{h}{720} (251f(x_{n+4}, t_{n+4}) + 646f(x_{n+3}, t_{n+3}) - 264f(x_{n+2}, t_{n+2}) \\
&\quad + 106f(x_{n+1}, t_{n+1}) - 19f(x_n, t_n))
\end{align*}
\]

Once a value for the states is approximated using the explicit Adams-Bashfort, these are corrected using the Adams-Moulton approximations, which are solved using linear search with the value computed from the Adams-Bashfort method as an “initial point”.

**Solving dynamical systems**

So far, we looked at methods that discretize differential systems. These methods create series of equations that can be used to model dynamical systems. In practice, problems
involving dynamical systems have additional requirements, in the form of system parameters that reflect what the problem-solver knows about the system in advance. For example, the recorded populations for predator and prey, the initial position and velocity of a projectile, or the starting temperature of the metal plate. The dynamical systems that have these parameters are examples of the most basic type of problem: an initial value problem [41].

Definition 3. By an initial value problem, we mean the following problem:

**Given**: a dynamical system \((D, f)\) and an initial condition \(x_0 \in D\).

**Find**: a function \(x(t)\) for which \(\frac{dx}{dt} = f(x, t)\) and \(x(0) = x_0\).

In an initial value problem, the initial state of the system is known in advance. Dynamical systems often have additional parameters that describe certain elements of the behavior, such as the reproduction and predation rates of preys and predators, the air resistance that a projectile will encounter, or the heat conductivity of the metal plate. These parameters remain constant even after integrating the system.

While many situations expressed with dynamical systems just require an initial condition, the definition of the problem may include additional constraints, such an expected final location for the projectile, or the heat dissipation at the end of a heating metal plate. Such problems are known as boundary value problems [25]:

Definition 4. By a boundary value problem, we mean the following problem:

**Given**: a dynamical system \((D, f)\) and the boundary conditions \(x_0, x_1, \ldots, x_n \in D\).

**Find**: a function \(x(t)\) for which \(\frac{dx}{dt} = f(x, t)\) and \(x(t_0) = x_0, x(t_1) = x_1, \ldots, x(t_n) = x_n\).

To solve initial value problems, we first integrate the differential system to obtain the system of linear or non-linear equations that model the exact or approximate behavior of the problem over a desired timespan. Then, we solve this system of equations to obtain values for all the discretized states over time. For explicit systems, each state \(x_{n+1}\) can be computed from the previous state \(x_n\), starting with \(x(0) = x_0\). For implicit systems, each equation has
state dependencies that prevent solving them in such a forward way. The system of equation is solved through a method that solves non-linear systems of equations [41].

The most famous method to solve non-linear systems of equation is Newton’s method. The basic Newton’s method is used to find the roots of a function, or \( x : f(x) = 0 \). This iterative method approximates the root based on the derivative of \( f \) and an initial point \( x_0 \):

\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}
\]

The method iterates until it finds an \( x_n \) s.t. \( f(x_n) \approx 0 \) up to a certain tolerance.

For systems of equations, Newton’s method uses a similar technique, representing the system as a vector function \( F(X) \), and its derivative as the Jacobian \( J_F(X) \). For functions \( F : \mathbb{R}^k \to \mathbb{R}^k \), each iteration is computed by solving:

\[
J_F(X_n)(X_{n+1} - X_n) = -F(X_n)
\]

These solution methods work for many different dynamical systems. In the next subsection, we look at some example of dynamical systems, including some sample solutions.

### 2.1.3 Examples

#### Three species food chain Lotka-Volterra

The original Lotka-Volterra system models the interaction between two species, one a predator and the other its prey. The three species food chain Lotka-Volterra model [18] involves a low level prey \( x \) that is preyed by another species \( y \), which in turn is preyed by an apex predator \( z \). The model for this type of interaction is:

\[
\begin{align*}
\frac{dx}{dt} &= ax - bxy, \\
\frac{dy}{dt} &= -cy + dxy - eyz, \\
\frac{dz}{dt} &= -fz + gyz,
\end{align*}
\]
where \(a, b, c, d, e, fg > 0\). Parameters \(a, b, c,\) and \(d\) are the same as traditional Lotka-Volterra, while the remaining parameters are:

- \(e\) represents the effect of \(z\)’s predation on \(y\),
- \(f\) represents the death rate of \(z\), and
- \(g\) represents the growth rate of \(z\) when prey is available.

We want to be able to model the behavior over time of different species. To solve this problem, we start by integrating the differential system. As the system is non-linear, we must use a discretization method to approximate the integration. As this is just a small demonstration, for simplicity we will use forward Euler method. We replace the derivative with its corresponding finite difference, then solve for the next steps:

\[
\begin{align*}
    x_{n+1} &= x_n + h(ax_n - bx_n y_n), \\
    y_{n+1} &= y_n + h(-cy_n + dx_n y_n - ey_n z_n), \\
    z_{n+1} &= z_n + h(-f z_n + gy_n z_n)
\end{align*}
\]

In this particular example, we are using an explicit Euler method and consider this system an initial value problem. We can use this approximation to solve it iteratively without generating the entire system of equations. Figure 2.2 shows examples of the solved system, with different parameters and initial conditions.

**Bratu**

The Bratu equation [50] is a second-order differential equation. It is derived from models in thermal combustion, and has applications ranging from chemical reaction theory, heat transfer and nanotechnology, to the expansion of the universe. The Bratu 1-D equation is as follows:

\[
\frac{d^2 u}{dx^2} + r \cdot e^u = 0.
\]
Figure 2.2: Examples of Three-Species Lotka-Volterra.

We need to integrate this equation through discretization. There are various ways of discretizing it; for the example we want to show, we will be using the Bratu 1-D equation for a boundary value problem (BVP), which means there are other conditions besides the initial condition – in this case, initial and final conditions \( u(t_0) \) and \( u(t_f) \). While there are methods to solve a BVP using explicit discretizations, it is easier to visualize using implicit discretization. In this case, we use an implicit second-order central difference method [26, 27] to create a system of equations as follows:
\[
\frac{u_{n+1} - 2u_n + u_{n-1}}{h^2} + r \cdot e^{u_n} = 0.
\]

As an implicit system, with this discretization we cannot compute one state at a time. Instead, we can obtain the solution using a linear algebra search method (i.e. Newton method). Figure 2.3 shows a plot with two examples of the Bratu problem, for boundary conditions \(u(0) = u(1) = 0\), step size \(h = 0.02\), and two different values for parameter \(r = \{3.4, 3.5\}\).

![Figure 2.3: Example of Bratu boundary value problem.](image-url)
2.2 Interval analysis

All the problems show so far assume that quantities are known and computed exactly. In practice, this is not always possible: measurement instruments have an inherent error, and the arithmetic methods most widely used in electronic computations deal with an approximation of real numbers.

*Interval analysis* is a field based on representing uncertain quantities as intervals that enclose uncertainty and that carry out *interval computations* to provide reliable, guaranteed results. We start this section with definitions of the *interval arithmetic* used in this field, as well as some details as to how these techniques are implemented. Next we look at *interval constraint solving*, which uses interval techniques to solve problems in which the domains are restricted by arithmetic and relational expressions.

2.2.1 Real intervals and interval arithmetic

Let us start with some definitions.

**Definition 5.** [39, 65, 66, 91] A real *interval* $[a, b]$ is the set of all real numbers between $a$ and $b$, where $a \leq b \in \mathbb{R} \cup \{+\infty\} \cup \{-\infty\}$:

$$x = [a, b] = \{ x \in \mathbb{R} \mid a \leq x \leq b; \ a, \ b \in \mathbb{R}\}.$$  

**Definition 6.** The set of all real intervals will be denoted by $\mathbb{IR}$.

**Definition 7.** We call $[a, a], \forall a \in \mathbb{R}$ a *canonical interval* [64]

**Definition 8.** The *convex hull* of a set $\rho$ is equal to $\square \rho = [\inf(\rho), \sup(\rho)]$, where $\inf(\rho)$ and $\sup(\rho)$ are the infimum (greatest lower bound) and supremum (least upper bound) of $\rho$.

**Definition 9.** Given a tuple of intervals $(x_1, \ldots, x_n) \in \mathbb{IR}^n$, we denote by $X$, and call *box*
\( X \), the Cartesian product of all the intervals in the vector \([65, 66]\):

\[
X = x_1 \times \ldots \times x_n.
\]

**Definition 10.** Given a real interval \( x = [a, b] \), its *width* is defined as follows \([65, 66]\):

\[
w(x) = b - a.
\]

**Definition 11.** Given a real interval \( x = [a, b] \), its *midpoint* is defined as follows \([65, 66]\):

\[
m(x) = \frac{a + b}{2}.
\]

**Definition 12.** Given a box \( X = x_1 \times \ldots \times x_n \), its *width* is defined as follows \([65, 66]\):

\[
w(X) = \max_{1 \leq i \leq n} w(x_i)
\]

Given two interval boxes \( X = x_1 \times \ldots \times x_n \) and \( Y = y_1 \times \ldots \times y_n \), their *intersection* is:

\[
X \cap Y = \begin{cases} 
(x_1 \cap y_1) \times \ldots \times (x_n \cap y_n), & \text{if } \forall i \in \{1, \ldots, n\}, x_i \cap y_i \neq \emptyset \\
\emptyset, & \text{otherwise}
\end{cases}
\]

**Interval arithmetic**

We use intervals to represent uncertain quantities. To model those uncertainties into the mathematical models used to represent real world scenarios, we need to be able to do arithmetic using intervals. *Interval arithmetic* is an extension of real arithmetic applied to intervals \([65, 66]\).

**Definition 13.** For intervals \( x, y \in \mathbb{IR} \) and an operator \( \triangleright \in \{+, -, \times, \div\} \), \( x \triangleright y \) is an
interval that contains all the real values \( x \bowtie y \), where \( x \) and \( y \) are in \( x \) and \( y \), respectively:

\[
\mathbf{x} \bowtie \mathbf{y} = \{ x \bowtie y | (x, y) \in \mathbf{x} \times \mathbf{y} \}
\]

This definition works with elementary arithmetic operations and even trigonometric functions. Based on this definition, we can derive some interval arithmetic rules that we can use to carry out operations. Some of these rules are:

- **Addition:** \([a, b] + [c, d] = [a + b, c + d]\)
- **Subtraction:** \([a, b] - [c, d] = [a - d, b - c]\)
- **Multiplication:** \([a, b] \times [c, d] = [\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}]\)

Here are some examples of interval arithmetic rules in action:

- \([2, 3] + [1, 6] = [3, 9]\)
- \([4, 5] - [1, 2] = [2, 4]\)
- \([-1, 1] \times [4, 4] = [-4, 4]\)
- \([2, 4]^2 = [4, 16]\)
- \(\sin\left(\left[0, \frac{\pi}{2}\right]\right) = [0, 1]\)

The rules we have shown and the examples have an exact range. However, this does not mean that we can always compute exact ranges using interval arithmetic operations. For example, we have the following rules for division:

\[
\frac{[a, b]}{[c, d]} = \begin{cases} 
[a, b] \times \left[\frac{1}{c}, \frac{1}{d}\right] & \text{if } 0 \notin [c, d] \\
\left[-\infty, c\right] & \text{if } d = 0 \\
\left[d, \infty\right] & \text{if } c = 0 \\
\left[-\infty, c\right] \cup [d, \infty] \subseteq [-\infty, \infty] & \text{if } 0 \in [c, d]
\end{cases}
\]
The last case represents a *discontinuous function* whose returned value of $[-\infty, \infty]$ guarantees an *enclosure* of the actual range. Here are more examples in which the result is an *enclosure* of the actual range:

- $[0, 1] - [0, 1] = [-1, 1]$
- $[0, 2]^2 - [0, 2] = [-2, 4]$

In the first example, we see a substraction of an interval with itself. Normally, we could expect the result of such an operation to be 0; however, with the interval arithmetic we have detailed so far we get an *enclosure* that is wider than the expected interval. A similar situation occurs in the second example, where the exact range is $[-1/4, 2]$ but the operation returns a wider *enclosure*.

Whether an interval arithmetic operation is continuous or not, to make sure the operations consistently return intervals, it is reasonable to consider that the result of an interval operation is not the range, but an *interval hull* of the range \([65, 66]\):  

**Definition 14.** For intervals $x, y \in \mathbb{IR}$ and an operator $\boxplus \in \{+, -, \times, \div\}$, $x \boxplus y$ is an *interval hull* that contains all the real values $x \boxplus y$, where $x$ and $y$ are in $x$ and $y$, respectively:

$$x \boxplus y = \big\{ x \boxplus y | (x, y) \in x \times y \big\}$$

When the hull matches the range, then we are happy. However, this is not always the case. The last two examples above show an interval result that is wider than expected. If we replace the interval values with variables, the first operation becomes $x - x$ with an expected range of 0; the second one becomes $x^2 - x$, a function with range $[-1/4, 2]$. In the following section, we will go more in depth about this *overestimation* found in simple interval arithmetic, and how we can use different evaluation methods to reduce this, and other, side effects of interval evaluation of functions.
Interval extensions of functions

Interval arithmetic operations and elementary functions can be combined into general functions, or *interval extensions of functions* [91].

**Definition 15.** For the real-valued function \( f : \mathbb{R}^n \to \mathbb{R} \), and the interval function \( f_I : \mathbb{I}\mathbb{R}^n \to \mathbb{I}\mathbb{R} \), we say that \( f_I \) is an interval extension of \( f \):

\[
\forall X \in \mathbb{I}\mathbb{R}^n, \{ f(x) \mid x \in X \} \subseteq f_I(X).
\]

This definition is flexible enough to allow multiple ways to define specific extensions. The most basic is the *natural extension*, in which the real-valued operations in the functions are replaced with their interval-valued counterparts and evaluated using interval arithmetic. This method is straightforward and has a low computational cost [65, 66]. However, this type of evaluation has drawbacks. First, let us recall the previous *overestimation* examples, now seen as natural extension functions:

- For function \( f(x) = x - x \), on \( x = [0, 1] \), \( f(x) = [0, 1] - [0, 1] = [-1, 1] \). However, the actual result should always be \( f(x) = [0, 0] \).

- For function \( f(x) = x^2 - x \), on \( x = [0, 2] \), \( f(x) = [0, 2]^2 - [0, 2] = [-2, 4] \). However, the actual range is \( f(x) = [-\frac{1}{4}, 2] \).

Natural interval extensions replace variables with the corresponding interval value. After the substitution, multiple occurrences of the same variable are treated as independent intervals. This *dependency problem* leads to *overestimation* in the evaluation results, an interval that contains but is wider than the actual range of the function [39, 48, 52].

Besides the dependency problem, there is another source of overestimation, known as the *wrapping effect*. This occurs when the shape of an equation or system does not fit neatly inside intervals. Figure 2.4 shows an illustration of the wrapping effect in a 2D system. The gray area represents the system’s image set, and the surrounding square its interval enclosure.
The shape of the image set prevents it from fitting neatly into the interval enclosure, which ends up having areas that represent another form of overestimation.

If overestimation is such an issue, how can we at least reduce it? One approach towards reducing overestimation with natural extensions is to use symbolic manipulation of the expression to reduce or eliminate multiple occurrences. In the first example above, the simplification is trivial ($f(x) = 0$). For the second example, we can rewrite the function to have only one occurrence of $x$ in the expression: $f(x) = (x - \frac{1}{2})^2 - \frac{1}{4}$. A single occurrence of $x$ ensures that the function will always evaluate to its actual range with no overestimation. This is also true for multivariate functions [39, 48, 52].

Not all expressions can be rewritten to ensure that there is a single occurrence for all variables. In that case, other techniques are necessary. For example, if the expression is a polynomial, rewriting it using Horner’s rule helps reduce overestimation [15].

Another attempt at reducing overestimation on interval computations is affine arithmetic [84]. Instead of representing uncertainty as an interval as defined above, each quantity is represented by an affine form $x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \ldots + x_n\epsilon_n$, where $x_0, x_1, x_2, \ldots, x_n$ are real numbers and $\epsilon_1, \epsilon_2, \ldots, \epsilon_n$ are symbolic variables whose values are in the $[-1, 1]$ range. When two quantities share an $\epsilon_j$ variable, the quantities are partially dependent, which means their uncertainty is more accurately represented as a polygon rather than a traditional interval box. The main drawback of affine arithmetic lies in choosing the right non-affine approximations for the problem to solve, and the computation time cost that those approximations may incur to ensure computational rigor [36, 81].
One of the first well-known interval extensions is the *Taylor interval extension* [91]. The main idea is to symbolically represent the function as a Taylor expansion over a single point of the interval domain.

**Definition 16.** The *Taylor interval extension* of degree $k$ of a $k + 1$ times differentiable function $f$ over an interval $x$ is defined by:

$$f(x) = f(x_m) + \sum_{i=1}^{k} \frac{(x - x_m)^i}{i!} \cdot f^{(i)}(x_m) + R_k(x, x_m),$$

where $x_m$ is a point inside $x$, $f^i$ the $i$-th order derivative of $f$. The function $R_k$ is the Taylor remainder of $f$ based on $x$ and $x_m$. Let $f^{(k+1)}$ be an interval extension of the $(k + 1)$-th order derivative of $f$, the Taylor remainder in Lagrange form is:

$$R_k(x, x_m) = \frac{(x - x_m)^{k+1}}{(k + 1)!} \cdot f^{(k+1)}(x)$$

The two factors that reduce overestimation are the value of $k$, and the choice of remainder function $R_f$. Higher values of $k$ provide better results at the cost of additional computation time for each derivative. Similarly, a good remainder function $R_f$ must guarantee that the result of the Taylor extension contains the whole image of the original function.

The most common Taylor extension is when $k = 0$. This special case is known as the *mean value extension*.

**Definition 17.** The *mean value extension* of an at least once-differentiable function $f$ over the interval $x$ is defined by:

$$f(x) = f(x_m) + f'(x) \cdot (x - x_m),$$

where $x_m$ is a point inside $x$, and $f'$ the interval extension of the first order derivative of $f$.

So far, we have defined intervals and their operations over the set of real numbers $\mathbb{R}$, which is *infinitely-sized*, in the sense that there is an infinite number of real numbers and their accurate representation requires infinitely many digits. However, a computer’s memory
is finitely-sized, so it cannot accurately perfectly represent all real numbers. In the next subsection, we take a look at how we approximate real numbers in a computer, how these approximations and their inherent error can lead to real consequences, and how we use this knowledge to implement floating-point intervals that provide computational rigor over the real numbers.

2.2.2 Implementation details of interval methods

Before getting into the specific details of how to implement intervals on a computer, first we need to know how these machines represent real numbers.

Real numbers and their machine-representation

The set of real numbers is infinitely-sized. It includes numbers that have an infinite number of decimal digits, such as $\frac{1}{3}$, $\pi$, or $e$. We can represent the real numbers on a continuous line: between any two real numbers, there is an infinite number of real numbers.

A machine’s memory is finitely-sized, so it cannot perfectly represent all real numbers. However, computers do approximate real numbers by using a finitely-sized representation. The set of machine-representable numbers is a subset of the set of reals: they can be seen as discrete points placed on the same line as the reals. We provide a visual representation of this situation in Figure 2.5: the continuous blue line represents all real numbers between two numbers $n_i$ and $n_{i+1}$, with red dots representing machine-representable numbers in between. Because discrete representations of real numbers are finitely-sized sets, the amount of numbers in the set between any two machine-representable points is also finite, and can even be zero.

We need to use real numbers in our computations. If a real number also belongs to
Table 2.1: Various representations of $\pi$.

<table>
<thead>
<tr>
<th></th>
<th>(50 digits)</th>
<th>16-bit fp</th>
<th>32-bit fp</th>
<th>64-bit fp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{R}$</td>
<td>$3.1415926535897932384626433832795028841971693993751\ldots$</td>
<td>3.140625</td>
<td>3.1415927410125732421875</td>
<td>3.141592653589793115997963468544185161590576171875</td>
</tr>
</tbody>
</table>

the set of machine-representable numbers, then we can use its machine representation in our computations, as it is an accurate conversion. However, most real numbers do not have such a representation, so they must be rounded to a nearby real that is also a machine-representable number.

Real numbers are often represented in machines as a set of numbers called floating-point numbers. Their binary structure makes them efficient for calculations under the limited memory of a computer. They are not perfect, as their discrete nature means that when a real number is stored in the machine, its floating-point representation might not be an accurate representation of its real value. Numbers such as $\pi$ are often truncated and rounded based on the size their binary floating-point representation. Table 2.1 shows various floating-point representations of $\pi$, with the underlined part representing the digits that match the digits in the real number. As we can see, the higher the size of the floating point, the more accurate it becomes, but it still remains an approximation.

Floating-point rounding is not exclusive to infinitely sized numbers. For example, let us convert the number 0.3 to a floating-point representation with a size of 32 bits. There is no 0.3 in the set of 32-bit floating-point numbers, and the nearest floating-point numbers are either 0.300000011920928955078125 or 0.2999999821186065673828125. Depending on the rounding policy, the real number 0.3 will be stored as one of those two floating-point numbers.

The inaccuracy in floating-point numbers extends to and becomes exacerbated by computations. For example, the Python programming language uses 64-bit floating-point representations. As we have seen, the number 0.3 cannot be represented accurately. The addition $0.1 + 0.1 + 0.1$ results in a value that is shown as 0.3, but its actual representation is different.
from the converted and rounded value of the original 0.3 by $5.551115e - 17$ [28]. This might not seem like a significant error, but when machine operations occur over long periods of time and without proper rounding error management, the results can be disastrous.

**Real consequences of floating-point errors**

We provide two real examples of systems that did not correctly manage floating-point computations, and the consequences of said errors:

*The patriot missile incident.* Dharan, Saudi Arabia, February 25, 1991. The middle of the Gulf War. A patriot missile battery fails to identify an incoming Iraqi Scud missile, which strikes an American Army barracks, killing 28 and wounding at least 100 others. A subsequent investigation discovered that the failure occurred due to a software error. Specifically, a floating-point error: the internal time in tenths of a second was multiplied times $\frac{1}{10}$, or 0.1, to obtain the time in seconds. The floating-point representation for the real 0.1 has a relative error of $9.5e - 08$, which went unmanaged and propagated over time. After 100 hours, the error had grown to 0.34. This time calculation was used to track incoming objects in between radar pulses, so after the second pulse detected the Scud missile, the battery expected it to have traveled the distance corresponding to an extra 0.34 seconds, approximately an extra half a kilometer. The second radar pulse showed nothing in the predicted location, so the incoming projectile was ignored [5, 11, 83].

*The Vancouver Stock Exchange index rounding error.* In 1982, the Vancouver Stock Exchange had instituted a new stock market index. The new index used floating-points with three decimal places, and started with a base value of 1,000.000. After 22 months of approximately 3,000 transactions per day, the value of the index was 524.811, way below reasonable expectations based on the transactions that had taken place. A team of analysts discovered that the result of all floating-point operations were not properly rounded, but instead were just truncated. Using proper rounding, the team recalculated the index, which resulted in a more reasonable value of 1098.892. The loss of accuracy from the truncation, combined with the propagation of the inaccuracies over the 22 months that followed the
inclusion of the new index, had led to an inaccuracy of over 500 points [42, 71].

Despite these flaws, floating-point representation of real numbers is widely used in ma-
chine computations to represent real numbers. It follows that we define floating-point inter-
vals to use in our computations.

Floating-point interval computations

By \( \mathbb{F} \) we will denote all the floating-point numbers representable on a given computer:

**Definition 18.** A floating-point interval \([a_f, b_f]\) is the set of all real numbers contained between the two floating-point bounds \(a_f\) and \(b_f\):

\[
x = [a_f, b_f] = \{ x \in \mathbb{R} | a_f \leq x \leq b_f; \ a_f, \ b_f \in \mathbb{F} \}.
\]

The floating-point number \(a_f\) is called the lower endpoint of \(x\), and \(b_f\) is called the upper endpoint of \(x\). The set of all floating-point intervals representable on a given computer will be denoted by \(\mathbb{IF}\) [39, 65, 66, 91].

Rigorous representation of real numbers using floating-point intervals. Recall how the set of real numbers is a superset of floating-point numbers, and how we can place floating-point numbers on the same line as the reals. By extension, a floating-point interval contains both the floating point and real numbers between the points in the line represented by its lower and upper bounds.

These properties can be combined. When converting to a machine representation, a real number that cannot be exactly represented is stored as an approximation, a nearby floating-point point. Strictly speaking, any time we use this number we are not using the real value but a nearby machine-representable approximation, so all computations done with it are also approximations.

If we want rigorous, guaranteed computations, we need to represent the real number in some way. We can do that with intervals:
Definition 19. For a real number \( x \in \mathbb{R} \), its floating-point interval representation is:

\[
\mathbf{x} = [x^-, x^+],
\]

where \( x^- \) is the largest floating-point number s.t. \( x^- \leq x \), and \( x^+ \) is the smallest floating-point number s.t. \( x \leq x^+ \).

This means we can represent any real number as a floating-point interval. If the real cannot be represented in floating-point, the bounds of the interval representation \( \mathbf{x} \) are \( x^- < x < x^+ \); if the real is also a floating-point, then the interval bounds are \( x^- = x = x^+ \).

Rigorous arithmetic with floating-point intervals. Interval arithmetic must ensure that the result of the operation encloses all the values resulting from applying the operation on all the numbers in the operand intervals. When doing arithmetic with floating-point numbers, it is possible that the result is a real number that is not a floating-point. In those cases, we get a floating-point approximation of the real result – a similar case to the conversion of a real into a floating-point.

This also translates into floating-point intervals: when doing floating-point interval arithmetic, the resulting bounds are rounded and might not contain all the possible real results. To ensure that the result of floating-point interval arithmetic operation is an enclosure of all possible real operation results, whenever the results of an operation over the bounds is inexact, we round upwards for the upper bound, or downwards for the lower bound. This process is called outward rounding, and is essential to provide guaranteed, reliable enclosures with interval floating-point arithmetic operations and functions.

Now that we have gone over the basics of interval arithmetic, we will take a step back and review areas that are closer to applying the mathematical concepts we have seen so far. We will start with constraint solving.
2.2.3 Constraint solving

Definition 20. A constraint is a relation between \( n \) variables \( x = (x_1, \ldots, x_n) \), \( x \in D, D \subseteq \mathbb{R}^n \) of the form

\[
f(x_1, \ldots, x_n) \gg 0,
\]

where \( \gg \) is a relation operator, \( \gg \in \{=, \leq, \geq\} \)

A constraint restricts the possible values of their variables may have within the given domain \( D \). For example:

- For a single-variable domain \( D_x \in \mathbb{R} \), the constraint \( x \geq 0 \) states that \( x \) cannot have a negative value. This means any value that is 0 or more will be consistent with the constraint, while any other value will violate the constraint.

- For domains \( D_x, D_y \in \mathbb{Z} \) (Integers), the constraint \( x + y = 3, x \geq 0 \) and \( y \geq 0 \) state that \( x \) and \( y \) are integers that can be 0 or greater, but also their sum must be 3. All non-integer and negative numbers will violate the constraints, and so will value combinations such as \( x = 1, y = 1 \).

- The constraint \( x^2 + y^2 \leq 16 \) restricts the values of \( x \) and \( y \) to the area inside a circle. Figure 2.6 shows an illustration of such a circle; the area in gray corresponds to values that are consistent with the constraint, with points \((x, y) = (1, 2)\) and \((x, y) = (-3, 1)\) given as examples of consistent values and points \((x, y) = (-4, 4)\) and \((x, y) = (2, -4)\) as examples of constraint violations.

It is possible for a set of variables to have more than one constraint. A constraint satisfaction problem (CSP) is a model designed to find any / all value assignments for a set of variables that fulfill a set of constraints \([3, 80]\). In this work, the type of CSP that interests us is real-valued problems in which the domains of the variables represented by real-valued intervals and the constraints are modeled as equalities. This problem is equivalent to a system of equations.
Definition 21. An interval constraint satisfaction problem is a tuple $P = (X, D, C)$, $X$ is a set of $n$ variables $\{x_1, \ldots, x_n\}$, $D$ is the Cartesian product of the variables’ associated interval domains $D = x_1 \times \ldots \times x_n$, and $C$ is the set of $m$ constraints $C = \{c_1, \ldots, c_m\}$. [20]

Let us have a domain $D'$. This domain $D'$ is a subdomain of $D$, or $D' \subseteq D$, if $x'_i \in D'$, $x'_i \in D'$, $\forall x'_i \subseteq x_i$. The CSP $P' = (X', D', C)$ is smaller than $P = (X, D, C)$ if $D' \subseteq D$. This relation is expressed as $P \preceq P'$.

The solution to a CSP is a set of values $A = \{a_1, \ldots, a_n\}$ s.t. $A \subseteq D$ and all constraints $C$ are satisfied with the values of $A$. We can also represent solutions using intervals, with an interval solution $D^* = \{x_1^*, \ldots, x_n^*\}$ s.t. $D^* \subseteq D$ and all the constraints $C$ are satisfied with the values of $D^*$. An interval solution is often a domain with a width $w(D^*) \leq \epsilon$, where $\epsilon$ is the desired solution accuracy. The opposite of an interval solution is a domain with no solution: $D_\emptyset$, where there is a variable with interval value $\exists x_i^* \in D_\emptyset$ s.t. said interval value is an empty set $x_i^* = \emptyset$.

Consistency is a property of CSPs. We say a CSP is consistent if no points from the domains in $D$ can be removed without compromising at least one of the constraints $C$. There exist consistency techniques designed to “enforce” consistency by removing inconsistent values from the domain $D$ in a CSP. What we call interval constraint-solving techniques is the combination of interval-based consistency techniques with algorithms that search for
solutions $D^*$ of a given accuracy $\epsilon$.

### 2.2.4 Interval constraint-solving techniques

The objective of applying interval constraint solving techniques (ICST) to a CSP $P$ is to achieve one of two possible outcomes: obtain one or more solutions $P^* \preceq P$ that are consistent with the constraints, or prove there is no solution by finding that the entire CSP is inconsistent, or $P_0 \preceq P$. The ICST algorithm that solves a CSP must generate consequently smaller problems $P' \preceq P$, check if there exists partial consistency or if the whether they are empty or not, and continue if the CSP size is larger than the desired solution. The main algorithm used to solve interval CSPs is called *branch-and-prune* (also sometimes known as branch-and-reduce) [80, 90].

**The branch-and-prune algorithm**

The general strategy behind branch-and-prune is to implement a *divide-and-conquer* approach: a *pruning* step that removes parts of the domains that are guaranteed to be inconsistent with the constraints, and a *branching* step that generates subproblems.

The general branch-and-prune method is outlined in Algorithm 1. The algorithm uses the following methods:

- **PUSH**($P, Q$): add CSP $P$ to the queue of CSPs $Q$.
- **POP**($Q$): get the next CSP $P$ from the queue $Q$.
- **ADD**($P, S$): add CSP $P$ to the set of solutions $S$.
- **SPLIT**($P'$): generates new subproblems by dividing the interval box domain $D'$ of $P'$ into two new sub domains, usually by the mid-point of one variable: $D_a = x_1 \times \ldots \times [x_i, m(x_i)] \times \ldots \times x_n$ and $D_b = x_1 \times \ldots \times [m(x_i), x_i] \times \ldots \times x_n$.
- **CONTRACT**($P$): takes the interval domain $X$ of $P$ and applies one or more *contractors* to obtain a reduced domain $D'$ of $P'$.
Algorithm 1 Branch-and-prune algorithm

1: procedure BRANCH_AND_PRUNE($P_0$) \( \triangleright \) CSP $P_0 = (X, D_0, C)$
2: \hspace{1em} PUSH($P_0$, $Q$) \( \triangleright \) $Q$ is a queue of CSPs
3: \hspace{1em} while $Q \neq \emptyset$ do
4: \hspace{2em} $P \leftarrow$ POP($Q$)
5: \hspace{2em} $P' \leftarrow$ CONTRACT($P$)
6: \hspace{2em} if $P = \emptyset$ then
7: \hspace{3em} Discard $P'$
8: \hspace{2em} else if $w(P') < \epsilon$ then
9: \hspace{3em} ADD($P', S$) \( \triangleright \) $S$ is a set of solutions
10: \hspace{2em} else
11: \hspace{3em} $P_a, P_b \leftarrow$ SPLIT($P'$)
12: \hspace{3em} PUSH($P_a$, $Q$)
13: \hspace{3em} PUSH($P_b$, $Q$)
14: \hspace{2em} end if
15: \hspace{1em} end while
16: return $S$

The contract method uses one or more contractors to reduce the domain $D$ in the CSP $P$. A contractor is an algorithm that prunes inconsistent parts of a CSP by checking its local consistency: they take a single constraint, and then check the consistency of each variable domain one at a time. If the contractor can reliably ensure that a portion of a variable’s domain can be pruned while maintaining consistency, then that portion is removed and the domain is contracted.

Figure 2.7 shows a basic example of how a contractor works: the initial domain $x$ is wholly consistent with constraint $f(x) \leq 0$; that same domain is partially consistent with the $g(x) \leq 0$ constraint. There are parts of that domain that can be safely removed while maintaining consistency. Using the image of $g(x) \leq 0$, the contractor creates a reduced domain $x' \subseteq x$.

To ensure that the domain is consistent across all constraints, the contractor takes a reduced domain and attempts to further prune it using different constraints. This constraint propagation helps remove larger inconsistent portions of an interval domain, keeping reduced domains that are at least partially consistent across the set of constraints.
Figure 2.7: Example of domain contraction using constraints.

To better understand how these contractors work, we will take a look at three different contractors: First a contractor based on interval Newton, then two based in consistency techniques: box consistency, and hull consistency. It is important to note that these are not the best interval contractors, we present them here because they are fundamental: they are still in use in current state-of-the art libraries and solvers, while also serving as the foundation for more advanced contractors and techniques [4, 47].

Newton contractor

The Newton contractor is derived from interval Newton method, an algorithm used to solve systems of linear equations using intervals [66]. That means this contractor works only on a CSP $P = (C, X)$ where there are $n$ variables in $X$ and the constraints $C$ is a set of $n$ linear equations $F(X) = 0$.

The Newton method is a well-studied algorithm for finding the roots for linear equations. Recall that we used this method to find the solution of an implicit dynamical system in Section 2.1.2. This method starts at a given point $x_0$, then iteratively computes new points $x_{i+1}$ using the transposed Jacobian of the system of equations $F$:

$$x_{i+1} = x_i - [J_f(x_i)]^{-1} F(x_i)$$

36
This equation is known as the *Newton update step*. Figure 2.8 shows how two Newton steps approach a root of a 1-D function, or a point in which \( f(x) = 0 \). We use a 1-D function as it illustrates how the Jacobian (in this case, the first derivative of \( f(x) \)) is projected into \( f(x) = 0 \) to find a point closer to the root (See Figure 2.8a). Every time this process is repeated from the new point, as in Figure 2.8b, it is possible to get closer to the solution. The general idea is that every new \( x_{i+1} \) will be closer to 0.

In Interval Newton, we start with an interval domain. It follows that the interval version must also get closer to 0 by contracting the domain using intersection. Figure 2.9 is an example of this process on a 1-D function. When \( 0 \in f(x) \) and there is only one solution in \( f(x) \), the interval Newton contractor returns a reduced domain. The function \( f \) is given in blue, with the midpoint \( m(x) \) shown as a violet dot in Figure 2.9a. As the function is strictly increasing, we also show the tangents at the endpoints of \( x \) to help illustrate how \( f' \) works. Figure 2.9b shows the actual Interval Newton step: \( f'(x) \) is projected from \( m(x) \), with the gray hashed area representing the range of values of \( f'(x) \); the points where \( f'(x) \) crosses \( y = 0 \) gives us the bounds of \( x' \). As \( x' \subseteq x \), then the contracted domain is equal to \( x' \cap x = x' \).

If \( 0 \notin F(X) \), the projection of \( f' \) returns an interval domain \( X' \) s.t. \( X \cap X' = \emptyset \). Due to this, applying the contractor returns an empty domain \( \emptyset \), which tells us there is no solution (root) in \( X \).
Box consistency contractors

Box consistency [9, 20, 87] is similar to arc consistency, a type of consistency for CSPs with discrete domains. The main intuition behind box consistency is to ensure that a single variable $X_k$ is consistent with a single constraint $c_k$: given $n$ as the number of variables in a CSP $P$, all the constraints are turned into a set of $n$ single-variable constraints by replacing all other variables with their domain variables. These single-variable constraints are then used to compute a reduced domain for variable $X_k$.

The box consistency contractor enforces box consistency across all variable-constraint pairs in the CSP $P$ to generate a contracted domain $D'$. If $D' \subset D$, meaning there was a contraction, then the box consistency contractor repeats the process until $D'$ cannot be contracted anymore. If at any point the contractor finds a domain $D' = \emptyset$, then the problem has no solution in $D$.

One of the most well-known box consistency algorithms is $BC3$. The contractor associated with this algorithm, $BC3Revise$, does a narrowing operation over the bounds of a specific variable. It combines the interval Newton contractor with bisection and recursion over either the upper or lower bound of a single variable $x_i$, until it finds an improved bound of $x_i$ that fulfills one of the above properties for box consistency. $BC3Revise$ then repeats an equivalent process for the other bound [9, 20, 87].
Hull consistency contractors

Hull consistency is a type of consistency based on the hull of the interval domain that satisfies the constraints [9].

Definition 22. Given a constraint $c$, a box $X$, and $\rho_c$ the set of all the elements in $\mathbb{R}^n$ that satisfy constraint $c$, then the constraint $c$ is said to be hull consistent w.r.t. the box $X$ if $X = \Box(\rho_c \cap X)$.

The hull consistency algorithm most relevant to this work is $HC4$, which uses the $HC4Revise$ contractor. $HC4Revise$ works by using the evaluation tree of the constraint. We illustrate this process in Figure 2.10. This tree is evaluated twice: first as a forward evaluation (Figure 2.10a) that traverses the tree from leaves to the root, which is always the constraint’s relation operation, doing the operations defined by each non-leaf node and storing the results in each node. The value stored in the root depends on the relational operator; our example has an equality, which does an intersection between its left and child node values. The second evaluation is a backward propagation (Figure 2.10b) in which the final values stored in the root are propagated down the tree, from root to leaves, using reverse operations and interval intersection. Whenever the $HC4Revise$ algorithm reaches a variable leaf, the result of the backward propagated inverse operation determined by the leaf’s parent is intersected with the variable’s value in the CSP $P$’s domain $D$ to obtain a reduced domain. The contraction is finished after traversing and propagating values through all the variable leaves [9].

2.2.5 Interval constraint solvers

All interval constraint solvers implement the branch-and-prune algorithm. There are three key differences between interval constraint solvers:

- Interval extension. The natural (naive) interval extension is rarely used, as it often leads to overestimation caused by variable dependency. The most common exten-
Figure 2.10: HC4Revise contractor applied on constraint $2x = z - y^2$
sions, though not the only ones, are Taylor-based, with different solvers using different parameters and techniques.

- **Branching strategy.** When the solver must split a particular CSP over its interval domain, it does so based on a single variable. The branching strategy determines which variable to pick, on which point (or points) to split the domain, and how to select the next problem from the queue.

- **Contractors.** Each solver implements a different set of contractors, as well as different strategies to combine them.

RealPaver [37, 38] is an interval-based system for modeling and solving systems of constraints written in C. It implements multiple contractors and splitting strategies that are parameterized, so the user can choose the ones to use. For splitting, RealPaver implements splitting into two or three subproblems, choose the next box in largest variable first, smallest variable first, or round-robin, as well as the width at which to stop splitting. RealPaver uses the following contractors: interval Newton, HC4Revise, BC3Revise, and 3B.

IBEX [16, 17, 74] (Interval-Based EXplorer) is an interval library for constraint processing. IBEX is a toolbox that contains classes that can be used to implement a custom constraint solving program. IBEX supports partition via midpoint bisection, either largest variable first, or in round robin. For problem selection, users can use either the default stack or heap, or build their own. As for contractors, IBEX implements the Newton, HC4, 3Bcid, Acid and polytope hull-based contractor. Additionally, it contains classes to build custom contractors, and for constructor algebra, which can create new contractors by combining existing ones. It also includes support for *separators*, a different kind of contractors that generate two sub-domains: one for points inside the set defined by constraints or another contractor, and one for points outside. In addition to providing all these in a library, IBEX also includes a precompiled solver IbexSolve, that solves CSPs using branch-and-prune with a composition of the HC4, Acid, Interval Newton, and a Polytope Hull of two combined linear relaxations. Some examples of the library in use include parameter and state estima-
tion [19, 55, 72], trajectory validation of autonomous vehicles [85], and robot localization [79] in both two-dimensional [40] and three-dimensional environments [49].

It is also possible to use interval constraint-solving and build custom contractors for other applications outside of just solving systems of constraints. In previous work [32, 33, 92] we built custom contractors to compare various interval-based global optimization heuristics, in both constrained and unconstrained problems.

2.3 Uncertainty in dynamical systems

Most methods to solve dynamical systems make multiple numerical assumptions about the system. Let us see how such assumptions are made with a simple “toy” example.

The predator-prey Lotka-Volterra model can be used in epidemiology to track the propagation and lethality of a disease. In order to obtain parameter values, the epidemiologist building their model must make observations of smaller populations to determine the reproduction rate of the corresponding infection, the disease propagation rate, and the death rate of both the potentially infected species and disease. The epidemiologist measures data with specialized tools, and based on such measurements determines the model’s parameters. Afterwards, the epidemiologist can then use the model to make simulations and experiment with the parameters, representing changes in the environment such as natural disasters changing the population or inoculation affecting the death rates of the species and disease propagation.

This model assumes that the measurements made to determine the parameters were perfectly accurate. However, tools used to measure phenomena in the real world always have a margin of error, which indicates that the actual phenomenon falls within a range of the reported measurement. These deviations are often small enough that the model builder ignores them without much consequence. However, even a small margin of error (uncertainty) can result in bigger errors after multiple computations. This is common in dynamical systems, where each time step of a simulation is computed based on previous
Interval analysis is one way to model uncertainty and incorporate it into the computation of a problem and its solution. Interval methods are a form of validated computing, as they guarantee that their interval results enclose the problem’s real solution set. To solve dynamical systems using intervals, there are two things that a solving algorithm must incorporate to provide a useful, guaranteed solution: proof that such solution exists, and interval extensions that reduce expansion.

The Picard operator [2, 23] is used to prove the existence and uniqueness of a solution for an ODE system. This operator is useful because many dynamical systems are represented through ODEs.

**Definition 23.** The Picard operator $\Phi$ of function $f$ on the box $D$ is:

$$\Phi(D) = D_0 + \int_0^h f([t_0, t_1], D)dt,$$

where

- $D_0$ is a box s.t. $D_0 \subseteq D$,
- $[t_0, t_1] \in \mathbb{I}$,
- $h = t_1 - t_0$,
- $O = f(t, x)$ is an ODE system,
- $f$ is continuous over $[t_0, t_1]$, and
- $f$ is an interval extension of $f$

**Proposition 1** (Picard operator as proof of uniqueness). Let $\Phi(D)$ be a Picard operator of an ODE system $O$ with a box $D_0 \subseteq D$. If $\Phi(D) \subseteq D$ then

- The ODE system $O$ with initial value $x(t_0) \in D_0$ has a unique solution over $[t_0, t_1]$;
• $\Phi(D)$ is a bounding box of the solution in $[t_0, t_1]$ w.r.t. $D_0$

This operator is useful for methods that solve the ODE system iteratively one time-step at a time. Methods that solve all the time-steps at once, such as constraint-solving techniques, are guaranteed by their properties: if any of the time steps results in an invalid domain, then the system has no solution. As for uniqueness, interval methods will perform a global search over the interval space of the state variables, so even if the solution is not unique, the interval constraint-solving techniques will find enclosures for all solutions [23].

The other issue interval dynamical system solvers encounter is interval expansion in the form of overestimation due to dependency and wrapping. We have already discussed overestimation in a general sense in Section 2.2.1; here we discuss overestimation on dynamical systems. Overestimation due to dependency is due the nature of discretization: these approximations involve state variables outside of the one that corresponds to each equation. For dynamical systems with a system of ODEs (i.e. the Lotka-Volterra equations in Section 2.1), the interaction of variables across ODEs and their subsequent discretization is another potential source of overestimation due dependency.

Overestimation due to wrapping effect is of great concern for dynamical systems. The trajectory represented by any two sequential interval states $\mathbf{x}(t_a)$ and $\mathbf{x}(t_b)$ wraps the real solution; however, because these are intervals, there is bound to be some overestimation beyond the real solution. Even a small amount of overestimation will propagate across state variables, leading to enclosures with even greater overestimation.

Interval-based dynamical system solvers must incorporate techniques that deal with both overestimation and wrapping, designed to take advantage of the unique challenges that arise when solving dynamical systems. One such technique is called Taylor models [10, 61], an interval evaluation technique designed to reduce the effects of overestimation and wrapping. The general idea of a Taylor model is to represent functional dependency as a high-order Taylor polynomial, with a small interval remainder enclosing the parts of the function that cannot be represented through the Taylor polynomial.
Definition 24. Let $f : D \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$ be a real function that is $(n)$ continuously differentiable at $a$. We define the $n$-th order Taylor polynomial of $f(x)$ around $a$ as the power series:

$$\sum_{i=0}^{n} \frac{f^{(i)}(a)}{n!}(x-a)^n,$$

where $f^{(i)}(a)$ is the $i$-th derivative of $f$ evaluated at point $a$.

Definition 25. Let $f : D \subseteq \mathbb{R}^v \rightarrow \mathbb{R}$ be a function that is $(n+1)$ continuously partially differentiable, on an open set containing the box $x$. Let $x_0$ be a point in $x$ and $P_n$ the $n$-th order Taylor polynomial of $f$ around $x_0$. Let $I$ be an interval such that

$$f(x) \in P_n(x - x_0) + I, \forall x \in D.$$

Then we can call the pair $(P_n, I)$ an $n$-th order Taylor model of $f$ around $x_0$ on $x$.

For more on Taylor models and their implementation, see [10, 61].

2.3.1 Existing interval-based ODE solvers

AWA [58] was one of the first ODE IVP solvers that used interval techniques. It was originally developed in PASCAL, with a port to FORTRAN. It was designed to solve problems where the initial value is given as an interval. It used a form of coordinate transformation called QR-factorization to deal with overestimation. For boundary value problems, it used shooting methods that replace the BVP with an equivalent IVP.

The CAPD library [46] is designed to solve dynamical systems problems and homology computation. It contains both non-validated and rigorous (interval-based) techniques.

COSY [59, 60] is a system that contains various tools used in validated computing. It validates the existence and uniqueness of solutions using Picard iteration and Schauder’s fixed-point theorem [53]. It was the first system that implements Taylor models, using automatic differentiation to obtain the required Taylor coefficients. COSY uses a shrink-wrapping technique and preconditioning to minimize the interval remainder of the Taylor
model. Newer versions incorporate Taylor model arithmetic, which helps improve the use of Taylor models by building them through elementary operations and functions.

VNODE [67, 69] is a validated package for solving IVPs written using C++. The VNODE algorithm consists of two stages. In the first stage, it proves the existence and uniqueness of the solution using Banach’s fixed point theorem [1] and a high order enclosure method. In the second stage, VNODE computes the interval enclosure of a time step using one of three methods: direct Taylor series, QR-decomposition [86] (the same algorithm used in AWA), or the interval Hermite-Obreschkoff algorithm [70]. It incorporates the PROFIL/BIAS library [54] for interval computations, and the automatic differentiation packages FADBAD [7] and TADIFF [8] to obtain Taylor coefficients and their Jacobians. An updated version, called VNODE-LP [68], written using the literate programming technique, aims to provide a stronger rigor guarantee.

VALENCIA-IVP [77] is a verified solver for IVPs for ODEs and sets of differential-algebraic equations. The main idea of this solver is the assumption that the enclosure for each state is defined by a non-validated arbitrary approximate solution and a guaranteed error bound. This interval error is derived using a Picard iteration that is further refined to improve its bounds without losing its guarantee. What distinguishes VALENCIA-IVP from other solvers is that it uses a computationally inexpensive linear interpolation as its solution, while using various interval methods to reduce overestimation: a combination of natural and mean-value interval extensions, monotonicity tests, iterative range-splitting and consistency techniques to eliminate subintervals that originate from overestimation.

VSPODE [56, 57] is a solver for ODE IVPs. It uses the traditional two-step algorithm to find each iteration: it first verifies that a solution exists and is unique, and then computes an enclosure of the next iteration. VSPODE uses Taylor series with error approximation for integration, and Taylor model arithmetic for interval evaluations. One of the differences from other methods is that it supports a variable step size \( h \): on the first part of the algorithm, VSPODE also tries to find a step size \( h_j = t_{j+1} - t_j > 0 \) that guarantees the existence and uniqueness of the solution. It uses high-order Taylor models, but instead of representing the
remainder as an interval, it is represented as a parallelepiped. Another feature in VSPODE that is not always present in other solvers is the ability to have uncertain parameters natively in addition to interval initial values. Other solvers can use uncertain parameters \( c \) in a cumbersome way: by modeling them as variables with a differential equation \( dc/dt = 0 \) and their interval parameter value given as an initial value. With VSPODE, there is no need to add any of these extra equations. In the past, we used this library to do rudimentary parameter estimation [30].

DynIBEX [24] is a plug-in for the IBEX library that can generate simulations of ODEs using validated integration methods using a variety of Runge-Kutta schemes and affine arithmetic. It is similar to other step-based methods such as VSPODE and VNODE-LP in that it computes a state-by-state simulation of a system defined by an initial value problem. It also supports differential-algebraic systems of equations (DAE). One of its main advantages is that it is built on top of the IBEX interval library, so it is compatible with its contractor, constraint solving and optimization methods.

### 2.3.2 Solving dynamical systems using interval constraint solvers

Interval-based ODE solvers perform integration at each step, generating new states over time based on previous states. Let us show a illustrative example with the following ODE:

\[
\frac{dx}{dt} = 2x - 3x^2
\]

We will solve this ODE with an initial value of \( x_0 = [5.0, 5.01] \) for \( t_0 = 0 \) to \( t_f = 1.0 \). We will make a simple discretization using Euler’s method, which yields the integral \( x_{n+1} = x_n + h(2x_n - 3x_n^2) \). Now let us set the step value to \( h = 0.001 \). A step solver would then start generating all states by replacing the value of the previous state in the equation, starting with \( x_1 \):
\[ x_1 = x_0 + h(2x_0 - 3x_0^2) \]
\[ = [5.0, 5.01] + (0.001)(2 \cdot [5.0, 5.01] - 3 \cdot [5.0, 5.01]^2) \]
\[ = [4.9346997, 4.94502] \]

The solver would then use the \( x_1 \) value obtained this way to compute \( x_2 \), then use this new value for the computation of \( x_3 \), and so on until computing \( x_{1000} \). All the values \( x_i \) represent the solution of the ODE.

It is also possible to solve dynamical systems using interval constraint solvers, by discretizing the entire problem in advance. Instead of computing one value of \( x_n \) at a time, we create a system of equations \( x_{n+1} = x_n + h(2x_n - 3x_n^2) \) for \( n = 1, 2, \ldots, 1000 \), with an interval box domain \( X = x_1 \times x_2 \times \ldots \times x_n \) with initial box domain \( x_i = [-10, 10] \). For the purpose of this method, the initial value \( x_0 \) and the step size \( h \) are constants. Then this system of 1,000 equations and variables is solved by using a branch-and-prune-like approach, which yields a series of interval \( x_i \) that constitutes an enclosure of the approximation of the real solution.

### 2.3.3 Comparison of interval techniques for dynamical systems

We have seen there are two main approaches to solving dynamical systems using interval techniques: specialized solvers that solve a dynamical system one discretization step at a time, and constraint solvers that solve the entire system with all discretization equations generated in advance. To understand their advantages and disadvantages, we choose various techniques to solve the same dynamical systems, compare their running times and plot the behavior of their simulation results.

**Selected techniques**

- **RealPaver.** This interval constraint solver has stable performance and provides output that is human-readable. In previous work (See [88, 92, 93]) we obtained good results on solving dynamical systems.
• **VSPODE.** This library’s main advantage is its efficient implementation of Taylor Models, which reduces overestimation when computing the simulation of an ODE.

• **IBEXSolve.** The basic interval constraint solver implemented in the Ibex library implements state-of-the-art interval contractors that are well-suited for general purpose problems, and shows a taste of what a solver customized for a specific application can do.

• **DynIBEX.** An IBEX plugin that solves dynamical systems step-wise using validated Runge-Kutta schemes.

**Sample problem: flapping insect wings**

In Chapter 1, we mentioned the problem of designing a nano-scale aerial drone that flies using flapping insect wings. The wings themselves display the behavior of a damped and forced oscillator:

\[ M\ddot{x} + b\dot{x}|\dot{x}| + k_1x + k_3x^3 = F_b \cos(\omega t), \]

where

\[ b = \rho r_c I_2 T^3 C_D(\alpha) \]

Table 2.2 has sets of parameters that represent different insect wings.

For our experiments, we will use the parameters for the bumblebee, with \( k_3 = 0 \) and \( \omega = 187.08 \).

**First experiment**

To solve this dynamical system, we need solving parameters: the discretization step size, the number of steps, and the time period for the simulation. For this first experiment, we want a problem on the smaller scale, so we are choosing a short time range of \( t_0 = 0 \) to \( t_f = 0.5 \) with a static step size \( h = 0.001 \), that after discretization gives us \( N = 500 \) states and equations.
Table 2.2: Model parameters for flapping wings of various insects

<table>
<thead>
<tr>
<th></th>
<th>Bumblebee</th>
<th>Honeybee</th>
<th>Fruitfly</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AR$</td>
<td>7.88</td>
<td>5.37</td>
<td>5.98</td>
</tr>
<tr>
<td>$C_D(\alpha)$</td>
<td>0.0826</td>
<td>0.0759</td>
<td>0.0780</td>
</tr>
<tr>
<td>$I_{21}$</td>
<td>0.073385 m$^4$</td>
<td>0.11657 m$^4$</td>
<td>0.10918 m$^4$</td>
</tr>
<tr>
<td>$r_{cp}$</td>
<td>0.06 m</td>
<td>0.06 m</td>
<td>0.06 m</td>
</tr>
<tr>
<td>$\rho$</td>
<td>1.225 kg/m$^3$</td>
<td>1.225 kg/m$^3$</td>
<td>1.225 kg/m$^3$</td>
</tr>
<tr>
<td>$M$</td>
<td>0.2 kg</td>
<td>0.2 kg</td>
<td>0.2 kg</td>
</tr>
<tr>
<td>$T$</td>
<td>5 1/m</td>
<td>5 1/m</td>
<td>5 1/m</td>
</tr>
<tr>
<td>$k_1$</td>
<td>7000 N/m</td>
<td>7000 N/m</td>
<td>7000 N/m</td>
</tr>
<tr>
<td>$k_3$</td>
<td>0, 1 $\times$ 10$^6$</td>
<td>0, 1 $\times$ 10$^6$</td>
<td>0, 1 $\times$ 10$^6$</td>
</tr>
<tr>
<td>$F_b$</td>
<td>1 N</td>
<td>1 N</td>
<td>1 N</td>
</tr>
<tr>
<td>$\omega$</td>
<td>185-189 rad/s</td>
<td>185-189 rad/s</td>
<td>185-189 rad/s</td>
</tr>
</tbody>
</table>

For the interval constraint solver, then initial box of $x_i = [-10, 10]$ $\forall i \in \{0, 500\}$ and a time limit of 3,600 seconds.

Each solver has its own solving parameters:

- *RealPaver* will use its default BC5 contractor, which combines box, hull and interval Newton contractors, with a round-robin bisection strategy and all other solving parameters in their default values.

- We set *VSPODE*’s step size to be *static*, so its approximation to the integral will be made over the same steps as the interval constraint solvers.

- We used the default solving parameters in IbexSolve.

As for the discretization, VSPODE provides its own scheme via high-order Taylor series, but we must write the discretized system for RealPaver and IbexSolve. We use a central-difference approximation, suitable for a second-order ODE, but unlike VSPODE we do not include the approximation error. This means the results of these two solvers can provide a good general idea of the overall behavior of the system, but cannot guarantee an enclosure of the real solution.
Table 2.3: Experiment 1, running times (in seconds)

<table>
<thead>
<tr>
<th>Solvers</th>
<th>Case A</th>
<th>Case B</th>
<th>Case C</th>
<th>Case D</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealPaver</td>
<td>3,600.378</td>
<td>3,600.465</td>
<td>3,600.619</td>
<td>13,668.568</td>
</tr>
<tr>
<td>VSPODE</td>
<td>0.396468</td>
<td>0.599752</td>
<td>0.515564</td>
<td>0.268468</td>
</tr>
<tr>
<td>IbexSolve</td>
<td>230.391</td>
<td>255.157</td>
<td>247.469</td>
<td>4421.68</td>
</tr>
</tbody>
</table>

Where is DynIBEX? The integration scheme that DynIBEX uses is validated Range-Kutta. To enforce this validation, the local truncation error of the method needs to be bound. DynIBEX computes this error through differentiation, which means the system itself needs to be differentiable. The absolute value of a variable $x$ is differentiable everywhere except at $x = 0$; this is also true for the interval extension of the absolute value of an interval variable $x$, which is non-differentiable when $0 \in x$. Coupled with needing an algebraic definition of its ODE, we concluded that, at present time, DynIBEX does not work with problems that have absolute values in their problem definition.

To show the effects of uncertainty on the system, we will run 4 different versions of the model, with varying initial values:

- **Case A**: $x_0 = [0, 0]
- **Case B**: $x_0 = [-5.0 \times 10^{-17}, 5.0 \times 10^{-17}]
- **Case C**: $x_0 = [-5.0 \times 10^{-9}, 5.0 \times 10^{-9}]
- **Case D**: $x_0 = [-5.0 \times 10^{-5}, 5.0 \times 10^{-5}]

Results and Analysis Figures 2.11, 2.12, and 2.13 show plots of the results for each solver in this initial experiment. The blue and red plots represent the upper and lower bounds, respectively; the magenta space between them represents the space enclosed by both bounds. Table 2.3 is the running times for each solver and case. Values in red represent incomplete/inaccurate solutions – we will discuss these individually.

RealPaver This interval constraint solver timed out on all cases. It effectively contracted the first few states, but left most in its initial size of $x_i = [-10, 10]$. While these
plots might look like RealPaver could not contract later states before timing out, in reality these results are hulls of all boxes that RealPaver generated and not discarded, including both boxes with potential solutions and unprocessed box domains. For cases A, B, C, and D, RealPaver returned hulls for 5217, 11737, 12873 and 7685 boxes, respectively. To further complicate matters, this result makes no distinction between solution and pending boxes, so effectively the hull “swallows” potential solutions.

To further understand why RealPaver spends so much time generating and trying to contract so many boxes, let us examine a function that RealPaver is missing, and the impact of its absence in the solving process.

RealPaver does not support the absolute value function. To make it work, we substitute the absolute with an equivalent formulation: $|x| = \sqrt{x^2}$. The main issue with this reformulation is that RealPaver uses a forward-backward contractor that uses inverse operations
and functions, and the reverse function for $|x|$ is not equivalent to the reverse of $\sqrt{x^2}$, which causes RealPaver to struggle when contracting.

**VSPODE** This solver works particularly well in Cases A and B. However, when the uncertainty increases, so does the overestimation. For illustration purposes, the plots for Cases C and D were limited to the same amplitude as the initial boxes used in interval constraint solvers. In reality, the last state for Case C was $x_{500} = [-310.588, 312.359]$; Case D stopped at $x_{341} = [-7818.7, 7819.8]$, being unable to compute an enclosure after that.

This is expected behavior: interval computations always introduce overestimation. On initial values with small uncertainty, the Taylor Model-based interval extensions in VSPODE do not introduce as much overestimation; however, as the width of the initial value interval increases, this uncertainty leads to an explosive overestimation.
As an additional observation, VSPODE also does not support absolute value as-is. However, VSPODE allows defining the differential equations as code, separate from the solver itself. This means that in the definition of the differential equations, we wrote code that could simulate the absolute value without resorting to actual reformulations of the problem.

**IbexSolve**  Looking at the plots, it looks like IbexSolve provides the best results so far, in terms of solution width. However, these improved results come at a price. Let us compare against the second best solver in this experiment. For cases A and B, the size of IbexSolve’s results are similar to VSPODE’s. If we look at the running times, we see a different story. For example, on case B, VSPODE runs in 0.599752 seconds, compared to IbexSolve’s 255.157 seconds. VSPODE takes less than a second, while IbexSolve takes 4 minutes.

VSPODE could not solve cases C and D, while IbexSolve managed to return a singular
solution for case C. Case D proved to be more difficult: IbexSolve timed out after 3,600 seconds. Note that the time in this case is longer than 3,600 seconds. This is not unusual: IbexSolve checks its current runtime after a full round of contraction; if a problem is particularly large the solver may spend a long time in this process before reaching the timeout check, exceeding the required 3,600 seconds. After timing out, IbexSolve returned 5 boxes that were marked as pending, or too big to be considered a solution and awaiting further processing. The plot shown for Case D is the hull of those 5 boxes.

**Conclusion**  These results makes it clear that all these programs/libraries can indeed solve the same problem, to various degrees of confidence, given enough time and, in some cases, a slight reformulation of the problem. IbexSolve provides smaller domains for the entire dynamical system, even after it times out. VSPODE is fast, but when the intervals of a state start getting too wide, the overestimation gets out of control after multiple iterations. RealPaver times out before it can do a more thorough exploration of the space.

This is a good initial exploration. VSPODE and IbexSolve do not have many parameters that can be tweaked, but we can test their behavior on a larger problem, with a longer timeout. Additionally, RealPaver contains multiple solving strategies, each with its own adjustable parameters, so we can compare them as well. This motivates a new experiment, focused on the same dynamical system with some differences: a longer time window, a different value for $k_3$, and tweaked strategies for RealPaver.

**The second experiment**

First, we address the new strategy for RealPaver. Its solving process is highly parameterized, and in fact has two general strategies determined by its contractor types: local and strong consistencies. Its best local consistency contractor is BC5, which was used as-is on the original experiment. Its default strong contractor is 3B, which “shaves” inconsistent portions of the domain instead of implementing a splitting strategy. RealPaver does not combine both contractor types: when running local consistency with
BC5, it follows a branch-and-prune strategy; with 3B, it generates a single box domain that is reduced until the solver times out, a desired width is achieved or it is impossible for the contractor to reduce the domain. Because these two contractors are in fact separate solving methods, for this second test we ran RealPaver as two different solvers: RealPaver-BC5, and RealPaver-3B.

Second, we set the new parameters for the dynamical system. We added forcing \( k_3 = 1 \times 10^6 \), which changes the behavior of the system. We increased the time window to \( t_f = 1.0 \) with the same value of \( h = 0.001 \), which gives a total number of states \( n = 1000 \). For the interval constraint solvers, we doubled the computation time limit, from 3600 seconds to 7200.

These are the adjustments to each solver’s solving parameters:

- **RealPaver-BC5** keeps trying to reduce the domain until contraction is less than a parameter \( q \). The default value is \( q = 10 \); we adjusted it to \( q = 15 \) for less aggressive contraction per box, but faster results. Parameter \( \varphi \) that determines the depth of the search; its default (best) value is \( \varphi = 0 \), so we changed it to \( \varphi = 0.1 \) to reduce computation time per box. The minimum size for a solution is set to \( \epsilon = 10^{-4} \), and instead of bisection we will divide the problem into 3 subproblems instead.

- The main parameter in **RealPaver-3B** is the width of a domain’s variable that might be iteratively removed; a smaller width will take longer but provide better results. We opted to use the default width value of \( w = 10^{-3} \).

- In the previous experiment, we set VSPODE’s step size to be completely static. This time we changed it to a *dynamic* step size to improve its approximation to the solution between each of the states obtained as it solves the system.

- We tweaked the bisection criteria for **IbexSolve**: it did not split boxes with a width lesser than \( 10^{-4} \), and any box with a width of less than \( 10^{-6} \) was considered a solution. With these settings, IbexSolve attempts to create fewer boxes by splitting, but also generates better solutions.
Results and Analysis  Both the size increase and the change in the forcing parameter $k_3$ had an impact in the results. This experiment further highlights the strengths and weaknesses of each method, and gives insights into how they can be improved.

RealPaver  Fine-tuning the solver’s parameters had a positive impact on the results. After the 2-hour timeout, it did not find a tight enclosure to the solution, but the tube found is definitely an improvement over the results of the first experiment. The results of RealPaver using the 3B contractor are particularly interesting, as instead of returning the hull of a set of boxes, it returns a single box. In the plots it is possible to see that there is very little difference. With the right initial boxes, the algorithms in RealPaver could provide tight
enclosures to the real solution, or at least an approximation that could assist a decision maker’s process.

**VSPODE** At first sight, the results do not look that different from the previous experiment’s, with the increased overestimation causing VSPODE to stop the process shortly after the first half of the problem. However, getting this version of the system running was not easy: adding the $k_3 x^3$ cubic term using the power function provided by VSPODE caused the solver to stop at the first state. However, when writing the equation as $k_3 \times x \times x \times x$, the system runs as shown in the plots. This reformulation seems to have an effect on the results, likely caused by the way VSPODE computes the approximation of the integral.
Table 2.4: Experiment 2, running times (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Case E</th>
<th>Case F</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealPaver-BC5</td>
<td>7203.021</td>
<td>7207.767</td>
</tr>
<tr>
<td>RealPaver-3B</td>
<td>7203.358</td>
<td>7204.226</td>
</tr>
<tr>
<td>VSPODE</td>
<td>0.744486</td>
<td>0.993238</td>
</tr>
<tr>
<td>IbexSolve</td>
<td>42212.9</td>
<td>38636</td>
</tr>
</tbody>
</table>

**IbexSolve**  The plot looks very good. What does not look so good is the time it took to get there: more than 10 hours, even after setting a timeout of 2 hours. IbexSolve checks the execution time after running an entire round of contraction; for this problem, running all the base contractors until reduction is insufficient, across the entire 1000-variable domain and all 1000 constraints, takes 10 hours. To further complicate matters, when running IbexSolve from an input file, with this problem, the solver takes 2 or more hours generating and preparing the internal constraints used by the solver; this time is not measured by IbexSolve, so it is not included in the reported solving time. In short: IbexSolve can provide great results even on large-size problems, but takes a long time to get those results and there is no way to adjust the contractor parameters to optimize the running time.

**Conclusion**

From the results of these two experiments, we can see that both step-based methods and interval constraint solving techniques can both solve dynamical systems, albeit in different conditions.

Step methods are very fast, but if the time period is too long and the number of intermediate states is too large, the overestimation can make the final state too large or simply unsolvable.

Interval constraint solving techniques can find an enclosure to the solution with a desired tightness, but the solving process can take a long time. It is possible to speed up this process by experimenting with different contractors and contractor parameters, but ultimately the problem size also has a dramatic impact onto the final running time of the algorithm.

Finally, both types of solvers have missing features that are fairly frequent in dynamical
systems, such as an absolute value function. The theory behind each solver can explain its absence, as they may require the derivative of the system.

All our sample solvers have both desirable and missing features. Curiously enough, the solvers almost complement each other: the features of one solver, such as RealPaver’s highly-expressive output, are missing in the others. This prompts the main research question of this document: *how can we combine these solvers and their features to provide tighter enclosures faster, for bigger dynamical systems?*

### 2.3.4 The challenge of solving large-scale dynamical systems with interval constraint solvers

If we want to solve large-scale dynamical systems using interval constraint solvers, we have to understand why these problems can be difficult to solve with these tools. The main algorithm behind interval constraint solvers is *branch and prune*. In this algorithm, the structure of the problem can influence its computational run time, by affecting its processes:

- *Branching bottleneck*. During branching, the interval box domain must be split to create sub-boxes that are placed on a queue of subproblems that remain to be explored. As branching only occurs when the contractor cannot reduce the current box below a certain width, the worst-case scenario for the B&P algorithm is when pruning is not possible, and the box domain must be split on every full domain from the subproblems queue. In turn, this means that the worst-case complexity for B&P during branching is *exponential* on the number of dimensions of the box; as each variable has a corresponding interval value in the box, the complexity is exponential on the number of variables in the problem. As an interval constraint solver generates new subdomains by splitting, but only process one subdomain at a time, this can cause *memory issues*: all these pending subdomains remain in memory. Generating and storing too many subdomains can cause the solver to use prohibitive amounts of memory, which can cause issues or crash the solver altogether.
Pruning bottleneck. The B&P algorithm does not apply contractors to all constraints just once. It is possible that the contraction over one variable is related to the domain of other variables on constraints that have already been processed. So the algorithm applies contractors again, until achieving some reduction (that is, the difference in domain width between the application of all contractors in the solving process) is smaller than a certain threshold. The worst-case scenario for this process is when the solver needs to re-process all of the constraints repeatedly (i.e. when the reduction is not considerable enough to branch the domain) across all contractors used in the process. Assuming a minimum reduction without splitting, the time complexity is \textit{polynomial} on the number of constraints.

Preprocessing and other issues. Constraint solvers work with the algebraic expression of a constraint. This means reading a text representation of a constraint and converting it into a data structure that can be processed by the contractors (For example, HC4 contractors use an expression tree of the constraint). This means that the constraint solver performs some pre-processing on the constraints, such as removing redundant constraints, simplify the arithmetic expression, or simply reserve enough memory for the data structures used in the contractors. In large-scale problems with thousands of constraints and variables, this preprocessing step can take a considerable amount of time or fail out-right due to lack of memory.

In dynamical systems, we have encountered these problems when we discretize a system over a long simulation time. Discretization with these parameters generates constraint systems of thousands of equations and variables. This type of system takes a prohibitive amount of time to solve, if it can be solved at all.

To show this practically, we conducted an experiment, with a dynamical system representing a three-species food chain with with Holling II predator response functions [82]. This type of model assumes that species will consume food at a decelerated rate as they must forage/hunt, and this process is heavily influenced by the density of their food. This type
of system presents non-linear growth with non-linear dependencies between the differential equations, and their consequent discrete state equations. The discretization for this experiment uses 10,000 states from $t_0 = 0$ to $t_f = 100$ seconds, which generates 30,000 equations.

We sent this problem to IbexSolve with the default solver values. After running on a dedicated machine for more than two months with no output whatsoever, the operating system killed the process.

In short: smaller problems can be solved faster by an interval constraint solver. How can we generate smaller problems in dynamical systems? One area we explored in previous work is *Reduced-Order Modeling*.

The main advantage of using reduced-order models is the drastic reduction of the number of variables in the box domain. In previous work [88, 89], we found this results in faster solution times using interval constraint solver. However, this method does have disadvantages. First we need a base used to project the full system into a reduced-order subspace; the process to generate the base can be time-consuming, depending on the algorithm used. The second drawback is that we are working on a subspace; this means that if we are attempting to find all validated solutions to the large-scale system, it is possible that not all solutions are distinctly projected into the subspace; if that is the case, then the solutions we find in the subspace are a subset of all the solutions in the original space, and we have no way of guaranteeing that the solutions found in the reduced-order model are all the solutions existing in the original full-order model.

The reduced-order model approach focuses on reducing the overall complexity of the box domain. What if instead of manipulating the state variables, we focus on changing the process through which these state variables are contracted? In particular, we look into the order in which the constraints are processed to produce a domain contraction.
3 PROBLEM STATEMENT

Dynamical systems are abundant. They have been the subject of theoretical and practical study and research even since the widespread adoption of computing devices, which has enabled more in-depth research and applications.

We want use dynamical systems to make better decisions. Dynamical systems provide useful information. We can use them to run computer simulations of complex systems and use those to make better decisions. They can tell us the measurements of a physical system that guarantees that certain undesirable conditions are not met, for example a car’s suspension not hitting the chassis; or that it works up to a certain performance, like the design of an electronic device that minimizes battery usage.

There are plenty of tools built on methods that can compute an approximate simulation with high precision. These tools incorporate search algorithms to find the state or states of the system given at least its initial state (and, depending on the problem, other intermediate and even final goal states). From a theoretical perspective, these tools can produce a good simulation given a system of ordinary differential equations. From a practical perspective, however, a simple simulation might not be enough.

Existing tools have shortcomings. Most existing methods struggle to handle certain types of practical problems. For example, optimizing the parameters of a system so it consumes energy within certain limits, or that minimizes said consumption. For example, they can find a value for the parameters that looks optimal, but for particularly complex, non-linear problems, cannot guarantee that this is the best value possible. And if the tool
does not report a solution, it cannot guarantee that this is a consequence of the problem having no solutions.

Many practical applications use real world measurements, which are inherently inexact: the real value of the measured quantity lies within a range of numbers. Most existing methods do not account for this range, which can lead to results that might not reflect reality accurately.

Interval computations can handle uncertain measurements, incorporating the ranges as sets of numbers defined by intervals and doing computations with said sets entirely. Because they work over entire sets of numbers, when implementing search algorithms with intervals, it is possible to have a global search that is guaranteed to find the all, or even the best solution values of a problem. While interval computations can provide guaranteed enclosures, they do suffer from overestimation: the enclosure might be larger than the actual results, containing elements that are not part of a solution.

There are existing solvers that handle dynamical systems using intervals. They are really good if the system is not too large: if the discretization has too many state equations, overestimation occurs in small quantities; however, these small overestimations compound and propagate across the solution, eventually leading to ranges that are too wide to provide useful information for a decision maker.

Interval constraint solvers can also be used to solve dynamical systems. They work by “contracting” or “shaving” away the parts of the system that are guaranteed to not be a solution. While they cannot eliminate overestimation entirely, this “contracting” approach can lead to narrower enclosures and a reduced overestimation, which in turn leads to longer-lasting large-scale simulations. However, because they “shave/contract” by processing all the state equations multiple times, the solution process can become prohibitively time-consuming.

Finding narrow, guaranteed enclosures of complex dynamical systems on longer time scales is difficult. The further in the future the final state is, the larger the over-
estimation found by the solver is – that is, if it is even able to finish the simulation up to that point. Trying to reduce this overestimation (for example, using constraint-solving techniques) can lead to increased, even intractable computation times. We need faster ways to find guaranteed enclosures and simultaneously reduce their overestimation as much as possible.

**We propose a new approach.** Using Interval constraint solvers to work on *all the state equations at once* is inefficient. These solvers are designed on the assumption that there is no *a priori* information about the system being solved. However, in dynamical systems we know some information. For example, in Initial Value Problems, we know the *beginning* of a simulation. This means that we can use this knowledge to restructure the constraint-solving process and make it more efficient.

We call the proposed approach the *Sliding Windows* algorithm. It focuses on solving only a subset of all the states, then using those values in subsequent “windows” of states. By reducing the number of constraints being processed entirely at any given time, we can reduce the total computation time.

To further take advantage of interval-constraint solving techniques, we use implicit discretization methods. Unlike most traditional interval methods for dynamical systems, which use explicit discretizations, we can use implicit methods with larger step sizes, which becomes systems with less equations.

However, not all discretized equations are the same. Higher-order discretizations are more accurate, but require more computations and could be more prone to overestimation. Our second proposal is an examination of discretizations with the Sliding Windows algorithm, with the objective of finding discretizations that can provide narrow enclosures with high accuracy.
4 CONTRIBUTIONS

4.1 The Sliding Windows Algorithm

We are looking for ways to solve large-scale dynamical systems using interval constraint-solving techniques, finding enclosures in a reasonable amount of time. In section 2.3.4 we looked at the different ways in which the number of state variables and equations in a discretized large-scale dynamical system made the solving process difficult if not unfeasible. We also looked at a way to reduce the problem size via model-order reduction, along its drawbacks.

Our contribution focuses not on changing the individual parts of the problem like a model order reduction those. Rather, we focus on how the constraints are processed to produce a contracted interval enclosure of the real values of the discretized system. We call this approach “Sliding Windows” [31, 34]: it solves only one portion of contiguous states at any given moment, before moving (“sliding”) into the next set (“window”) of states. In this section, we explain the motivation behind our choice of algorithm; describe the algorithm, its parameters, and its properties; analyze its time complexity in relation to using stand-alone interval constraint solvers; and provide experimental results.

4.1.1 Motivation: an example

Let us show an example of a complex system, and how we can solve it through different approaches.

We are interested in complex, non-linear problems that require a larger number of states
to find an enclosure for. Let us work on a three-species food chain model with Holling type II predator response function [82]. This kind of model is similar to the well-known Lotka-Volterra system of differential equations, with one major difference: the rate at which a predator/consumer can obtain and digest their prey decelerates over time, as they satisfy their consumption needs. These differential equations have non-linear relationships between the variables, represented by the $m_i m_k$ terms (see equation later). Their relationship are further complicated by the carrying capacity of the first species, $K_1$, which limits its total growth, and the half-saturation constants $A_1, A_2$ which restrict the rate at which species 1 and 2 are consumed. The differential equation system is:

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

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

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

Discretizing this system over 10 states would yield a system with 30 variables and 30 state equations. With a reasonable step size of $h = 0.01$, these states would cover around 0.1 time units of simulation. Not much, but solvable in a reasonable amount of time.

What happens if we want to solve it for 1 time units? The number of states becomes 100, or one order of magnitude larger in terms of the number of variables and constraints. For 10 time units, it’s another order of magnitude. For 100, we would need 10,000 states, or 30,000 variables and constraints – the example described in section 2.3.4 that we ran and was stopped by the operating system after more than two months.

As far as the constraint-solver goes, a smaller problem will be solved faster than a bigger one. However, dynamical systems require at least thousands of states and equations, with many problems requiring even larger orders of magnitude. So we asked ourselves: Is there any way to take advantage of the solving speed of smaller problems (i.e. 10 or 100 states), in a larger problem generated by discretizing a dynamical system?
An intuition: sampling constraints

An interval constraint solver reduces interval domains through contractors. These contractors process the constraints according to a heuristic that chooses which constraints to use for domain pruning at any given moment in the contraction process. For example, a common heuristic is round-robin: processing the constraints in the order in which they are declared. Random selection is also possible, as long as all the constraints are ultimately processed until no reduction is possible.

What if the order in which constraints are processed has an impact in the computation time? One idea that explores this is constraint sampling. Instead of a heuristic that covers the entire set, constraint sampling tries to contract the entire domain as much as possible using a specific subset of the constraints. The main idea is that some constraints might have more impact in the amount of contraction, so if these constraints are selected and processed first the overall process would be faster. In previous work [75], we looked into techniques for constraint selection/sampling for subsets, to focus an initial solution on those subsets before solving the overall problem.

Let us take this idea of “constraint sampling” into dynamical systems. Specifically, let us explore this with initial value problems (IVP), a type of dynamical system in which an initial system state or initial condition is known, but no other state is given in advance. Initially the most impactful constraints/state equations are the ones that involve known quantities, which for an IVP are the state equations that involve the initial state. These equations also involve the first state variables in the simulated time frame, which leads to a contraction in their initial interval values. Using the 10,000 states example, let us say we sample only 10 of those, which can be contracted relatively fast.

It follows, then, that the next set of constraints to sample is the ones that involve the states whose interval values were contracted in the previous set of samples. We can, then, repeat this selection process until we have gone through and reached the end of all the state equations, at the one representing \( t_f \). This is the main intuition behind the Sliding Windows
Algorithm: solving a series of windows of subproblems made from the full set of equations describing the original discretization.

### 4.1.2 The Algorithm

**Nomenclature** The algorithm itself is simple: in general, we solve one smaller subproblem from the bigger problem (a window) at a time, using states that have already been narrowed as a new set of values for subsequent windows. These states that are transferred from one subproblem to another overlap between windows. Table 4.1 shows the nomenclature used throughout this section to describe this algorithm and its properties. Algorithm 2 shows the pseudocode for the Sliding Windows algorithm.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>The set of all states in the discretized dynamical system</td>
</tr>
<tr>
<td>$h$</td>
<td>Step size used to generate the states in the discretization</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of states generated in the discretization</td>
</tr>
<tr>
<td>$x(i) \in X; i \in {1, \ldots, n}$</td>
<td>The $i$-th state in the discretization</td>
</tr>
<tr>
<td>$F$</td>
<td>The set of all the state equations produced by the discretization of a dynamical system</td>
</tr>
<tr>
<td>$f_i \in F; i \in {1, \ldots, n}$</td>
<td>The $i$-th state equation in the discretization</td>
</tr>
<tr>
<td>$X_{i,j} \subseteq X; X_{i,j} = {x(i), \ldots x(j)}$</td>
<td>Subset of $X$, containing contiguous states from $x(i)$ to $x(j)$</td>
</tr>
<tr>
<td>$F_{i,j} \subseteq F; F_{i,j} = {f_i, \ldots f_j}$</td>
<td>Subset of $F$, containing contiguous state equations from $f_i$ to $f_j$</td>
</tr>
<tr>
<td>$x_0$</td>
<td>Initial state. Used in initial value problem (IVP) dynamical systems</td>
</tr>
<tr>
<td>$X_0$</td>
<td>Initial interval range for all the states in $X$</td>
</tr>
<tr>
<td>$X^*$</td>
<td>Final interval range (solution) for all the states in $X$</td>
</tr>
<tr>
<td>$C(F, X, X_0)$</td>
<td>Discretized dynamical system, expressed as a system of equality constraints</td>
</tr>
<tr>
<td>$N_w; N_w \ll N$</td>
<td>Window size. Number of states in each subproblem that will be solved sequentially</td>
</tr>
<tr>
<td>$o; 1 \leq o &lt; N_w$</td>
<td>Overlap size</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Contractor parameter: The desired width for the interval enclosures of states</td>
</tr>
</tbody>
</table>
Let us describe this algorithm in more detail. Line 2 creates an interval box $X^*$ will contain all the final, narrow states of the problem. Variable $S_i$ is an index that points at the first discretized state/equation of the current window $\{t(S_i), \ldots, t(S_i + N_w)\}$. This variable gets updated at the end of every loop, to move the current window to the next set of states that need to be narrowed (Lines 8 and 14). $C_i$ and $X_i$ contain the state equations and variables, respectively, corresponding to the current window. $X_i^*$ stores the results of contracting $X_i$ based on $C_i$, with narrow interval values for the states in the current window. Not all of the states in the current window are final, though: the states corresponding to the overlap, or $\{t(S_i + N_w - o), \ldots, t(S_i + N_w)\}$, can still be reduced based on contraction on future states, so they are not initially considered part of the solution set; the GetWindowSolution method transfers the states $\{t(S_i), \ldots, t(S_i + N_w - o - 1)\}$ to $X^*$, the interval box for the solution.

A close observation of the algorithm will show that lines 4–8 correspond to lines in the loop, 10–14. This is intentional: on a higher-level perspective, these correspond to two

### Algorithm 2 Sliding Windows Algorithm

```plaintext
1: procedure SlidingWindows($C, X, \epsilon, o, N_w$)
2:     $X^* \leftarrow X$
3:     $S_i \leftarrow 0$
4:     $C_i \leftarrow \text{GetWindowEqs}(C, S_i, N_w)$
5:     $X_i \leftarrow \text{GetWindowStates}(X, S_i, N_w)$
6:     $X_i^* \leftarrow \text{Solve}(C_i, X_i, \epsilon)$
7:     $\text{GetWindowSolution}(X^*, X_i^*, S_i, N_w, o)$
8:     $S_i \leftarrow S_i + N_w - o$
9:     while $S_i \leq \text{Size}(X)$ do
10:        $C_i \leftarrow \text{GetWindowEqs}(C, S_i, N_w)$
11:        $X_i \leftarrow \text{GetWindowStates}(X, S_i, N_w)$
12:        OVERLAP($X_i^*, X_i, N_w, o$)
13:        $X_i^* \leftarrow \text{Solve}(C_i, X_i, \epsilon)$
14:        $\text{GetWindowSolution}(X^*, X_i^*, S_i, N_w, o)$
15:        $S_i \leftarrow S_i + N_w - o$
16:     end while
17:     return $X^*$
18: end procedure
```
different types of problems. The first lines generates a window using the first set of equations which involve the problem’s *initial conditions*. Any subsequent loops do not use these initial conditions, as using the overlapped states transmits the trajectory information between the windows.

- **GetWindowEqs**: Creates a subset of $C$, with the state equations for discrete times $\{t(S_i), \ldots, t(S_i + N_w)\}$.

- **GetWindowStates**: Creates a subset of $X$, with the state variables for discrete times $\{t(S_i), \ldots, t(S_i + N_w)\}$.

- **Solve**: Calls an interval constraint solver to contract the states in $X_i$ based on the set of constraints $C_i$. The solver will attempt to obtain interval values for $X_i$ such that the width of the box (that is, the width of the widest variable value) is less than $\epsilon$. If the solver finds multiple interval enclosures for the set of all states where $w(X_i) \leq \epsilon$, the Solve method creates a *hull* of all the solutions: a single set of interval state values that encloses all the interval enclosures found.

- **GetWindowSolution**: Transfers the final interval values in the current contracted window $X_i^*$ to the set $X^*$ that will hold all the state values. This method does not transfer all values from $X_i^*$ to $X_i$, only the ones that do not correspond to the overlap, that is the states corresponding to the discrete times $\{t(S_i), \ldots, t(S_i + N_w - o)\}$ (TODO: mention something about the final state needing to transfer the last states)

- **Overlap**: Method that transfers the overlapping states from the contracted set $X_i^*$ to the next window $X_i$, which, after updating the index $S_i$, correspond to the discrete times $\{t(S_i), \ldots, t(S_i + o)\}$. This method takes the last $o$ state values from $X_i^*$ and assigns them as the first $o$ state values in $X_i$. This forms a “seed” that maintains the approximated trajectory from previous windows into the new subproblem. Figure 4.1 shows a visualization of how this overlap is transferred from one “window” to the next.
Justification and Properties

The Sliding Windows algorithm finds the interval enclosure of a dynamical system by using interval constraint solving techniques on smaller subsystems of state equations instead of the entire system at once. The design of the algorithm in which smaller subsystems with overlapping states are solved sequentially takes advantage of the structure of a discretized dynamical system, and of the properties of the branch-and-bound algorithm.

Taking advantage of the problem structure  There are no general strategies for choos-ing a subset of constraints to attempt to speed up solving a general constraint satisfaction problem. Constraint-solving algorithms mostly follow a round-robin approach, applying constraints in the same order they are given by the user. This means that for a general constraint solver, the order of constraints is irrelevant: it will ultimately process all constraints.

In the case of a set of state equations generated from a discretized dynamical system, we have some prior knowledge of how the problem is structured. Discretization creates state equations that involve a specific state variable, as well as one or more additional state variables before or after that specific state variable. In constraint-solving terms, it means that the solving order is important: if we want to choose a set of constraints to solve at
any specific moment in the constraint-solving process, the best we can choose are the state equations whose states are contiguous and that involve known (solved/narrowed) state values. This last point is particularly important, as it also tells us which state equations/constraints need to be chosen first, i.e. an IVP’s initial conditions.

**Taking advantage of branch-and-bound** The branch-and-bound algorithm works by contracting interval-based domains; when this algorithm cannot contract/prune the domain, it generates/branches into two subdomains by splitting the interval box domain on a single variable. The worst case scenario for this algorithm is when there is no reduction and the algorithm must split continuously, which generates new problems exponentially based on the number of variables in the domain. It is better, then to solve smaller problems with less variables.

These two properties justify our use of windows of subproblems. Why, then, does the algorithm require a series of shared states, the overlap, between the subproblems? We will illustrate this with an example.

Let us say that we have a dynamical system that is discretized into 1,000 state equations $f_1, \ldots, f_{1000}$, with each equation involving the states right before and after, or $f_i(x_{i-1}, x_i, x_{i+1})$ (The last equation, when $i = 1000$, uses $f_{1000}(x_{999}, x_{1000})$, as there is no valid $i + 1$ state). We want to solve it using windows of subproblems, so we first create a subsystem with an initial condition $x_0$, the first 10 state variables $x_1, \ldots, x_{10}$ and the state equations that involve only them: $f_1, \ldots, f_9$. Even if we include $x_{10}$ in our set of variables, we cannot include its corresponding state equation $f_{10}(x_9, x_{10}, x_{11})$ as-is because it involves $x_{11}$, a variable outside of our first “window”. There are two things we can do about this:

(a) Include $f_{10}$ without $x_{11}$. We do this with a slightly different discretized equation $f'_{10}(x_9, x_{10})$ (Similar to $f_{1000}$ in the full problem); or

(b) Include $f_{10}$ and all its variables. This means adding $x_{11}$ to the set, even if it is only involved in one equation for this subproblem.
The first choice is not good, as this formulation will not correspond to the full problem’s. If we were to use \( f'_{10} \), we would lose the relation between the variables in the original \( f_{10} \). This means that, to preserve the original relations, we need to add \( x_{11} \) to the set of variables in this initial subproblem. Then, we use an interval constraint solver to find narrow the enclosures for all states corresponding to each one of those variables, including \( x_{11} \).

For the next subset, an initial intuition is to treat it just like the previous one, as a regular initial value problem. The new initial condition would be \( x_{10} \), with the next 10 states \( x_{11}, \ldots, x_{20} \). And, once again, if we want to include all the state equations \( f_{11}, \ldots, f_{20} \) corresponding to these states, we need the variable \( x_{21} \). However, for this second subsystem we have an additional initial enclosure besides the one for \( x_{10} \): by including \( x_{11} \) in the previous “window”, its interval value will be contracted along the previous ones, and we can use it in this second subproblem. As \( f_{11} \) also includes \( x_{10} \), we can incorporate this “initial condition”, usually an immutable constant in this type of initial value problem, as part of the set of variables whose values we will contract. For this second subproblem, then, we will find narrow enclosures for the values of state variables \( x_{10}, \ldots, x_{21} \) using the state equations \( f_{11}, \ldots, f_{20} \).

All subsequent problems follow a similar pattern, in which there are two states that are contracted in the previous subproblem, whose interval values can be used in the next one. What if instead of just two, we transfer more states from one subproblem to another? A reformulation and expansion of the idea presented above is the core behind the Sliding Windows algorithm.

### 4.1.3 Complexity analysis

To understand one of the main advantages of the sliding windows technique, we need to take a look at the computational complexity of the branch-and-prune algorithm and how dividing the state equations into windows of subproblems affects the total solution time. The branch-and-prune algorithm has two parts: branching (or splitting) and pruning (or contraction). We will take a look at the pruning part, then the splitting part, and how both
parts together define the time complexity of the branch-and-prune algorithm.

There exist multiple pruning algorithms that use constraints to remove the inconsistent parts of a domain. One thing they have in common is that they all must iterate through the constraints themselves; in general, we can say the time complexity of pruning algorithms is based on the number of constraints, which we will call $N_C$. For example, the HC4Revise contraction method applies local consistency using a single constraint over a box domain. The HC4 algorithm applies this method iterating through all the constraints of the problem. This means that the HC4 algorithm will run $N_C$ iterations of HC4Revise.

In reality, a contractor algorithm is not applied just once, as the initial contraction leads to a new domain that can potentially be further contracted by applying the same contractor again, or another contractor. This means that a single contractor can be applied multiple times, and the series of contractors can be repeated until either the box domain becomes narrow enough to be considered a solution or the entire set of contractors produces an insufficient reduction. The worst case scenario for contraction is when, on every contraction iteration, the set of contractors produces a very small contraction, but not small enough to be considered an insufficient reduction. This keeps the contractor process repeating itself for an unknown number of iterations.

Based on this, the time complexity for the entire pruning process ranges from:

- Best case scenario (i.e., given domain is already a solution): $O(N_C)$
- Average/Worst case: $O(poly(N_C))$

Branching occurs when a contractor cannot reduce the width of a domain below the expected solution width. The splitting process cuts the interval value of a single variable, and creates new domains with the rest of the variables intact. This means that the time complexity behind the splitting process can be measured based on the number of variables, which we will call $M_C$.

For now, let’s look into a case in which no contraction happens, and no subproblem gets discarded. This means that the algorithm will split at every iteration, and will only
stop creating subproblems when it has bisected all the variables enough times to reach the expected solution width. For this example, let us assume a very simple way to select the next variable to bisect on: the largest variable, or if more than one are the same size, then the next one in round-robin order.

Using largest-first splitting, the solver would need to go through all the variables once before creating a smaller subproblem. Each iteration consumes one problem and generates two new subproblems; by the time a subproblem narrower than the original problem is generated, there remain $M_C - 1$ subproblems that are still as wide as the original problem and need to be split further. Because of these existing subproblems, this type of binary splitting can be said to generate subproblems exponentially based on the number of variables, $O(2^{MC})$.

This is, of course, assuming that the solver is unable to contract the domain at any point. In reality, the contractor can speed up this process considerably, by either finding reduced domains or eliminating unfeasible subdomains. Even then, subproblem generation still happens as long as an iteration’s contractor fails to find a narrow enough problem, so the exponential growth does happen at least partially. However, this is still the algorithm’s bottleneck, so ultimately its complexity remains exponential.

The effect of the sliding windows  We will show this through an example. The worst-case complexity of an exponential algorithm such as branch-and-prune is $O(2^N)$. For a problem with 10 variables, or $N = 10$, the time complexity is an order of magnitude of $2^{10}$, or 1024. If we increase the size of the problem to $N = 100$, the magnitude becomes $1.2676506 \times 10^{30}$; with $N = 1000$, $1.071509 \times 10^{301}$. This means that an increase in the order of magnitude of the problem size dramatically increases the time it takes to solve it.

This, of course, assumes the solver attempts to solve the entire problem at once. With discretized dynamical systems, we know in advance which variables are involved in each state equation, so splitting the entire set of state equations into windows that share equations is possible. Transferring the overlap from one window to another and storing the solutions
are done in linear time, or $O(N)$, which means the biggest time roadblock remains the contraction process with exponential time.

4.1.4 Experiments, results and analysis

We compare the results of the Sliding Windows algorithm against other interval-based methods used in dynamical systems: the step-based solvers VSPODE and DynIBEX, as well as a stand-alone IBEX model without sliding windows. We did two sets of experiments, comparing different metrics: a set of experiments on a flapping wing dynamical system with a second-order differential equation [31], and on a three-species food chain with predator-response functions [34].

First experiment: Flapping wing

This experiment aimed to identify the effects of the window size $w$ and overlap $o$ on the computation time and on the quality, that is the interval width of the enclosure, in the solution. We chose the same problem presented in Section 2.3.3, the flapping wing equation:

$$M\ddot{x} + b\dot{x} |\dot{x}| + k_1x + k_3x^3 = F_b \cos(\omega t)$$

This second order differential equation is discretized using central difference method, in which we generate the state equations by replacing the second and first derivatives $\ddot{x}$ and $\dot{x}$ with the following approximations at each discrete time $t_i$:

$$\ddot{x} = f''(x, t_i) \approx \frac{x_{i+1} - 2x_i + x_i - 1}{h^2}$$

$$\dot{x} = f'(x, t_i) \approx \frac{x_{i+1} - x_i - 1}{2h}$$

We used the same initial values and parameters as the comparison in Section 2.3.3, with $h = 0.01$ and the three different final times $t_f = \{20, 40, 80\}$. For the Sliding Windows
settings, we ran it with two different window sizes $w = \{20, 200\}$ and two different overlap values $o = \{20\%, 50\\%\}$, represented as a percentage of the window size. This gave us a total of 12 problems. The initial interval enclosure for all states was $[-5.0, 5.0]$.

**Experimental results** First we will show a series of plots, depicting the results of various problems. These plots represent enclosures of the line that represents the behavior of the flapping wing described by the discretization used. We want behavior that leads to narrow enclosures, with little to no overestimation.

The plots for the shortest case of $t_f = 20$ are in Figure 4.2. These plots show the problem we encountered with window size $w = 20$ and overlap $o = 20\%$ (That is, 4 states): The overestimation in this case started happening early on (after $t_i = 6.0$s), leading to enclosures that had no reduction whatsoever, as big as the initial interval of $[-5.0, 5.0]$. The plot for this particular case looks to be of a different color because that represents the interval enclosure contained between upper and lower bounds. This enclosure also exists in the other cases, but it is not visible because the bounds are too narrow.

Figure 4.3 plots the results when $t_f = 80.0$. Outside of the overestimation in the $w = 20, o = 4$ case, explained above, all the experiments in this category have a narrow interval width (less than $10^{-4}$) and maintain the same amplitude after reaching a seemingly stable state. This shows that the Sliding Windows algorithm can obtain narrow results for problems that other solvers struggle to solve.

<table>
<thead>
<tr>
<th>$t_f$ (N)</th>
<th>$w = 20$</th>
<th>$w = 200$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.0 (2000)</td>
<td>9.7749 s</td>
<td>12.0582 s</td>
</tr>
<tr>
<td>40.0 (4000)</td>
<td>15.6876 s</td>
<td>23.4972 s</td>
</tr>
<tr>
<td>80.0 (8000)</td>
<td>29.7687 s</td>
<td>46.1260 s</td>
</tr>
</tbody>
</table>

The question is, then, how fast is the algorithm? Table 4.2 shows the computation times for all 12 problems. Results that are crossed out resulted in overestimation before the end of the simulation. Let us take a look at the twelve problems and how the parameters influence
the performance:

- **Overlap has a bigger impact in solution quality (width of interval enclosure).** We can see this in the $w = 20$ cases. The three problems for $w = 20, o = 20\%$ are fast, but they lead to excessive overestimation way too early (See Figure 4.2a) in the set of states – we have marked them in Table 4.2 by crossing over their run times. A larger overlap can lead to worse performance, but less overestimation.

- **Linear computation time complexity.** Running the same parameters on different problem sizes results in times that are linear in relation to the number of variables and equations.
Window size has a bigger impact on performance. The results for \( w = 20 \) are faster than the ones in \( w = 200 \) by multiple orders of magnitude. This suggests that smaller window sizes are preferable. Comparing the two \( w = 200 \) cases shows that while overlap does have an impact in performance, it is not as significant as the impact of \( w \).

So far, these observations apply to this flapping wing dynamical system, a second order differential equation discretized using central difference. This system is non-linear and tracks a singular variable over time: the vertical position of the wing, represented by variable \( x \). How will the algorithm perform on a system of differential equations, with more complex relationship between equations? This informed the next experiment.
Second experiment: Food chain

We are interested in complex, non-linear problems that require a larger number of states to find an enclosure for. For this second experiment, we chose a three-species food chain model with Holling type II predator response function. This kind of model is similar to the well-known Lotka-Volterra system of differential equations, with one major difference: the rate at which a predator/consumer can obtain and digest their prey decelerates over time, as they satisfy their consumption needs. The differential equation system is:

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

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

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

These differential equations have non-linear relationships between the variables, represented by the \(m_i m_k\) terms. Their relationship are further complicated by the carrying capacity of the first species, \(K_1\), which limits its total growth, and the half-saturation constants \(A_1, A_2\) which restrict the rate at which species 1 and 2 are consumed.

\[
\begin{array}{cccc}
  a_{12} = \frac{5}{3} & d_3 = 0.4 & m_1(0) = 8.3 & A_1 = \frac{1}{3} \\
  a_{23} = 0.05 & d_3 = 0.01 & m_2(0) = 0.3 & A_2 = 0.5 \\
  a_{21} = \frac{5}{3} & r_1 = 1.03 & m_3(0) = 0.45 & \\
  a_{32} = 0.05 & K_1 = 0.875 & \\
\end{array}
\]

Figure 4.4: Parameter values for three-species food chain

We ran this model with VSPODE, DynIBEX, IBEX, and the Sliding windows algorithm using IBEX as its constraint solver. The model parameters are given in Figure 4.4. For each step-based solver we used either the best settings for their algorithm (i.e. VSPODE and DynIBEX use a dynamically computed step size, DynIBEX using an order 4 Runge-Kutta discretization). For IBEX and Sliding Windows, we generated a set of state equations using
implicit backwards Euler discretization:

\[ \frac{dm}{dt} = f'(m, t_i) \approx \frac{m_{i+1} - m_i}{h} \]

We applied this scheme on the three equations. We used a step size of \( h = 0.01 \), with two different final states \( t_f = \{40, 100\} \) with a total of \( N = 4000, 10000 \) states per species, respectively.

**Experimental results**  First things first: there are no results for running IBEX alone. The \( N = 4000 \) system contains a total of 12000 constraints, and IBEX spent around 4 months trying to process this system before the operating system killed the process.

Figure 4.5 shows the plots of the solution for VSPODE and DynIBEX. For \( t_f = 40 \), VSPODE manages to find narrow enclosures for all states, which is why these results look like a single line. DynIBEX automatically computes its own value for \( h \), while we get the interval bounds at times that do not match the step size used in their algorithm; this means that for the values of \( t_i \) that we are using to compare across algorithms, the DynIBEX solution returns a nearby enclosure, which results in the “jagged boxes” look, seen in the plot.

The plots for \( t_f = 100 \) do not reach all the way to that value: VSPODE and DynIBEX stop running before reaching the expected final time. In both solvers, 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.

When comparing these algorithms with the Sliding Windows, we consider three metrics as our basis:

1. **Quality of the solution.** All states in the solution are intervals. The quality is the max width across all states and species. This metric is useful for those test cases in which the solvers manage to reach the expected final state at \( t_f \).

2. **Total execution time.** Same as the previous metric, it is more useful when the solver
Figure 4.5: Three species food chain state enclosures, VSPODE and DynIBEX

3. *Time states until overestimation.* This metric was designed after we found that the solvers did not always reach the expected / requested end of the simulation. This metric is the first state that begins overestimation. We consider a state to be overestimating, when the supremum of a state is 10% above its midpoint.

For the Sliding Windows heuristic, we also sought to compare the effects different heuristic parameter values had in the solution process, using the same metrics. These values are *window sizes* of $N_w = \{20, 50\}$ and *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 seconds, and finally, if multiple solutions are found, or if the solving process ends before a solution, we take the hull that encloses all found potential solutions and pending boxes.

Figure 4.6: Three species food chain state enclosures, Sliding Windows with \( o = 30\% \)

Table 4.3 shows the results of our experiments when \( t_f = 40 \). In this table we can identify the following:
Figure 4.7: Three species food chain state enclosures, Sliding Windows with $o = 50\%$

Table 4.3: Table of metrics for $t_f = 40$

<table>
<thead>
<tr>
<th>Solver</th>
<th>Quality</th>
<th>Execution time</th>
<th>T. until Overest.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSPODE</td>
<td>1.0000</td>
<td>3189.0619</td>
<td>29.09</td>
</tr>
<tr>
<td>DynIBEX</td>
<td>0.4879</td>
<td>30.6470</td>
<td>1.50</td>
</tr>
<tr>
<td>$N_w = 20, o = 30%$</td>
<td>6.1330E-04</td>
<td>313.6700</td>
<td>-</td>
</tr>
<tr>
<td>$N_w = 20, o = 50%$</td>
<td>5.6379E-04</td>
<td>459.5741</td>
<td>-</td>
</tr>
<tr>
<td>$N_w = 20, o = 70%$</td>
<td>5.1991E-04</td>
<td>934.7888</td>
<td>-</td>
</tr>
<tr>
<td>$N_w = 50, o = 30%$</td>
<td>2.5693E-04</td>
<td>1856.3104</td>
<td>-</td>
</tr>
<tr>
<td>$N_w = 50, o = 50%$</td>
<td>2.1312E-04</td>
<td>2454.8584</td>
<td>-</td>
</tr>
<tr>
<td>$N_w = 50, o = 70%$</td>
<td>1.9722E-04</td>
<td>3258.1111</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 4.8: Three species food chain state enclosures, Sliding Windows with $o = 70\%$

<table>
<thead>
<tr>
<th>Solver</th>
<th>Quality</th>
<th>Execution time</th>
<th>T. until Overest.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSPODE</td>
<td>1.5848</td>
<td>5351.2794</td>
<td>29.09</td>
</tr>
<tr>
<td>DynIBEX</td>
<td>1.1136</td>
<td>57.8751</td>
<td>1.50</td>
</tr>
<tr>
<td>$w = 20, o = 30%$</td>
<td>0.7437</td>
<td>741.1903</td>
<td>49.22</td>
</tr>
<tr>
<td>$w = 20, o = 50%$</td>
<td>0.7341</td>
<td>1273.4984</td>
<td>49.98</td>
</tr>
<tr>
<td>$w = 20, o = 70%$</td>
<td>0.7282</td>
<td>1346.8824</td>
<td>51.16</td>
</tr>
<tr>
<td>$w = 50, o = 30%$</td>
<td>0.5853</td>
<td>3149.8939</td>
<td>50.73</td>
</tr>
<tr>
<td>$w = 50, o = 50%$</td>
<td>0.6157</td>
<td>5124.8888</td>
<td>53.13</td>
</tr>
<tr>
<td>$w = 50, o = 70%$</td>
<td>0.7016</td>
<td>5636.2485</td>
<td>61.64</td>
</tr>
</tbody>
</table>
• Both VSPODE and DynIBEX start showing overestimation already. VSPODE’s starts at \( t = 29.09 \), and increasing up to 1.0 by \( t = 40 \). DynIBEX first overestimating box occurs much earlier, at \( t = 1.5 \), but due to how the algorithm is designed (Namely, its use of an interval value for \( t_i \) in each discretized state), it does not return a “tube-like” enclosure like VSPODE does. This means the overestimation as we defined in our metrics happens much earlier, but it is increases and decreases over time. Figure 4.5 shows this behavior in more detail.

• The Sliding Windows algorithm obtains narrower enclosures all around (See the Quality column in Table 4.4).

• DynIBEX has much better time performance overall, while Sliding Windows outperforms VSPODE by an order of magnitude when \( N_w \) and \( o \) are small.

Table 4.4 shows the results on a larger experiment, for up to \( t_f = 100 \). For this experiments, all of the algorithms had conditions that stopped the execution: for VSPODE and DynIBEX, the algorithms stopped when they could not dynamically compute a new value for \( h \), usually because the previously computed state had too much overestimation to find a value for \( h \) that maintained a narrow enclosure. As Sliding Windows does not compute \( h \) dynamically, its stopping condition was when the enclosure computed in the latest window is \( w(X_i) \geq 1.0 \).

Overall, the Sliding Windows algorithm obtains narrow results for a further \( t_i \), at the expense of computation time. DynIBEX can obtain enclosures faster, but these results are not always tight: DynIBEX solver assumes there is also uncertainty in time and represents it as an interval, which creates a “jagged” enclosure. We show samples of this behavior in Figure 4.9. The Sliding Windows algorithm with a window size of \( w = 20 \) outperforms VSPODE on all metrics, while the experiments with \( w = 50 \) are better in most metrics, except when \( o = 70\% \) with a longer execution time, a difference of little less than 300 seconds.
Figure 4.9: Sample plots for DynIBEX solution of three species food chain

Remark While these results seem remarkable, it is important to note that these experiments with Sliding Windows are carried out with an approximation, the backwards Euler method, a very simple and inaccurate approximation. Existing solvers, such as VSPODE and DynIBEX, use more complex approximations (interval Runge-Kutta and high-order Taylor models, respectively). These solvers that also incorporate the approximation error (Local truncation error) in its computations. As this error cannot be accurately computed without an exact solution, these solvers compute the bounds that delimit the local truncation error range (usually using derivatives of the state equation), and add it to the discretized equation as an additional interval term. This guarantees that these methods will enclose the real solution.

Without any changes, the backwards Euler discretization used in the Sliding Windows algorithm is not a perfectly accurate enclosure. This motivates our next research question: how can we improve the enclosures of the approximations in the Sliding Windows algorithm?

4.2 Improving enclosure accuracy in the sliding Windows algorithm

Existing solvers 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 compute 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 will focus on experiments in two main areas: higher-order approximations and artificially-inflated enclosures.

### 4.2.1 Higher-order approximations

We want to use higher-order implicit discretization schemes, similar to the ones found in existing solvers. In general, adding more states to a state equation increases the accuracy of the approximation. Both Taylor polynomials and Runge-Kutta (used in VSPODE and DynIBEX) create intermediate sub-states in the simulation of the dynamical system, dynamically computing the value of $h$ at each step; for Sliding Windows, we use a static $h$, and computing intermediate substates might be inefficient and overcomplicated: 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.

What if, instead of adding intermediate substates to our discretized set of states, we use other multiple existing states as those intermediate ones? The methods that involve discretizations using multiple steps are known as linear multistep methods. These rely on the structure of our problem as a system of constraints: unlike the single-step methods, which only involve one state in the computation of the current state, a discretized state equation generated using linear multistep methods can involve multiple states from before or after the current equation’s state. For this contribution, we explored the effects of an implicit linear multistep scheme: the Adams-Moulton method.
Implicit method: Adams-Moulton

We looked at the Adams-Moulton method in Section 2.1. This discretization is used along with its explicit version, the Adams-Bashfort method, as a predictor-corrector method: By finding a solution candidate with the explicit method first, they can use these state values as an “initial guess” for the implicit method, which needs to be solved using a search algorithm (i.e. Newton method). However, for IBEX and the Sliding Windows algorithm, our “initial guess” is a wide enclosure that can be contracted into a solution. Thanks to this, we can use the implicit discretization method without the explicit version.

We believe that this type of implicit multi-step discretization method is a good fit for solving dynamical systems using interval constraint-solving techniques. Each discretized state equation involves multiple state values, which are maintained naturally in an interval box that represents the values of the entire set of states. Like Taylor and Runge-Kutta, an approximation of higher order produces more accurate discretizations, at the expense of requiring computations on each generated equation. As the number of states involved in a specific discretization increases, so does the number of calls to \( f(x, t) \), as well as the complexity in their coefficients.

The increased complexity can impact the contraction process in an interval constraint solver. The Sliding Windows algorithm generates “windows” of subproblems generated from an initial set of states and state equations, then finds enclosures for each “window” using interval constraint solving sequentially. Once again, this does not include bounding the error in any way. This means that we know in advance that the approximations will not be a perfect enclosure of the solution, but it can help us identify 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 how close the interval approximation was to enclosing the real solution. For each state enclosure \( x_i = [\underline{x}_i, \overline{x}_i] \) and real solution \( x^*_i \), the error in \( x_i \) is:
\[ \tau(x_i) = 0 \quad \text{if } x^*_i \in x_i \]
\[ \tau(x_i) = x_i - x^*_i \quad \text{if } x^*_i < x_i \]
\[ \tau(x_i) = x^*_i - x_i \quad \text{if } x^*_i > x_i \]

The objective of the experiments centered around higher-order discretizations is to explore their accuracy using the above metrics. These take into account that there is no bounding of the approximation error to analyze the accuracy of each approximation within the Sliding Windows algorithm. However, just because there is no bounding of the approximation error, it does not mean we cannot explore the impact of such a bounding can have. To examine this, we need to manipulate the enclosures by introducing artificial intervals into the models generated through the Adams-Moulton approximations.

4.2.2 Manipulating the enclosures

For this work, we do not have any method to bound the error in the Adams-Moulton approximations. However, we do know that the error-boundi ng methods used in existing solver introduce an additional term that changes the bounds of each state. This means we can conduct experiments that involve the addition of interval bounds to further analyze the viability of each discretization level.

We explored two different types of enclosure manipulations:

- **Initial value inflation.** For all experiments so far, we have used single-number initial values. By running experiments with this inflated initial value, we can compare its impact against the uninflated values.

- **State equation inflation.** This works as a "naive" simulation of a bound of the error. The objective of this manipulation is to measure how much can we improve the approximation without doing any additional complex calculations while also examining the impact of these added enclosures.
We want to compare how these methods would work with different enclosure changes and discretization levels. The objective of comparing the different approximations under changes in the enclosures for the initial states and for all states, is to identify which discretization type provides the most accurate approximation (both in closeness to the solution and interval width) even without bounding the approximation error. Next, we will outline the experiments we carried out, their results and our analysis.

4.2.3 Experiments, results and analysis

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 an interval constant \( \delta = [-10^{-16}, 10^{-16}] \)
- **Case All-B**: Add to all state equations an interval constant \( \delta = [-10^{-8}, 10^{-8}] \)
- **Case All-C**: Add to all state equations an interval constant \( \delta = [-10^{-4}, 10^{-4}] \)
- **Case Ini-A**: Add to all initial states an interval constant \( \delta = [-10^{-16}, 10^{-16}] \)
- **Case Ini-B**: Add to all initial states an interval constant \( \delta = [-10^{-8}, 10^{-8}] \)
- **Case Ini-C**: Add to all initial states an interval constant \( \delta = [-10^{-4}, 10^{-4}] \)

We ran each of the models with each of the enclosure manipulation types, for a total of 63 experiments using the same three-species food chain described in Section 4.1.4. For the discretization, we kept the same value 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. The comparison metrics in this section are designed to measure each experiment against a solution in the real domain from a method without interval uncertainty. This solution was generated using Runge-Kutta method of order 5 with error control from the order 4 method, with a relative and absolute tolerances of $10^{-12}$ and $10^{-14}$, respectively, which provides an approximation with a high degree of accuracy.

We want to find the discretization that provides the model closest to the real-based solution using the Sliding Windows algorithm without explicitly bounding the approximation error. We used the following metrics to examine and 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. For these experiments, we set a goal of $t_f = 40.0$, or $N = 4000$ states.

- **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 enclose 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.

We present each metric as a table, with one discretization per row, and each column representing a different type of enclosure manipulation. In general, we want discretizations that last as close as possible to $N = 4000$ before reaching too much overestimation, while also maintaining high coverage and low error in the accuracy metrics described above.

**Results**

Tables 4.5–4.9 show the experiment results for the states until overestimation, coverage, total accumulated error, total average error, and coverage error when running the three-species food chain models with systems of state equations generated using Adams-Moulton discretization and enclosures obtained using Sliding Windows. In the solution accuracy metrics, or Tables 4.6–4.9, the 10 best values in each table are highlighted.
Table 4.6: Comparison of coverage in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.7630</td>
<td>0.8744</td>
<td>0.0000</td>
<td>0.0007</td>
<td>0.0815</td>
</tr>
<tr>
<td>s=1</td>
<td>0.0000</td>
<td>0.0037</td>
<td>0.8505</td>
<td><strong>0.9694</strong></td>
<td>0.0000</td>
<td>0.0868</td>
<td>0.9562</td>
</tr>
<tr>
<td>s=2</td>
<td>0.0327</td>
<td>0.2460</td>
<td><strong>0.9603</strong></td>
<td><strong>0.9769</strong></td>
<td>0.0345</td>
<td>0.4197</td>
<td><strong>0.9929</strong></td>
</tr>
<tr>
<td>s=3</td>
<td>0.0330</td>
<td>0.2177</td>
<td>0.9184</td>
<td><strong>0.9600</strong></td>
<td>0.0342</td>
<td>0.2650</td>
<td><strong>0.9780</strong></td>
</tr>
<tr>
<td>s=4</td>
<td>0.2380</td>
<td>0.2557</td>
<td>0.9187</td>
<td><strong>0.9625</strong></td>
<td>0.2382</td>
<td>0.2942</td>
<td><strong>0.9781</strong></td>
</tr>
<tr>
<td>s=5</td>
<td>0.2855</td>
<td>0.2832</td>
<td>0.9164</td>
<td>0.9500</td>
<td>0.2877</td>
<td>0.4610</td>
<td><strong>0.9730</strong></td>
</tr>
<tr>
<td>s=6</td>
<td>0.3655</td>
<td>0.3687</td>
<td>0.9128</td>
<td>0.9444</td>
<td>0.3732</td>
<td>0.8272</td>
<td><strong>0.9712</strong></td>
</tr>
<tr>
<td>s=7</td>
<td>0.7813</td>
<td>0.7922</td>
<td>0.9049</td>
<td>0.9286</td>
<td>0.7824</td>
<td>0.8607</td>
<td>0.8304</td>
</tr>
<tr>
<td>s=8</td>
<td>0.7845</td>
<td>0.7845</td>
<td>0.7234</td>
<td>0.9091</td>
<td>0.7828</td>
<td>0.8238</td>
<td>0.8636</td>
</tr>
</tbody>
</table>

Table 4.7: Comparison of total accumulated error in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=1</td>
<td>0.5927</td>
<td>0.5896</td>
<td>0.1628</td>
<td>0.0666</td>
<td>0.5927</td>
<td>0.5482</td>
<td>0.1471</td>
</tr>
<tr>
<td>s=2</td>
<td>0.0159</td>
<td>0.0148</td>
<td>0.0099</td>
<td>0.0059</td>
<td>0.0159</td>
<td>0.0113</td>
<td><strong>0.0086</strong></td>
</tr>
<tr>
<td>s=3</td>
<td>0.0910</td>
<td>0.0850</td>
<td>0.0266</td>
<td><strong>0.0089</strong></td>
<td>0.0090</td>
<td>0.0705</td>
<td>0.0212</td>
</tr>
<tr>
<td>s=4</td>
<td>0.0691</td>
<td>0.0646</td>
<td>0.0233</td>
<td><strong>0.0082</strong></td>
<td>0.0691</td>
<td>0.0481</td>
<td>0.0183</td>
</tr>
<tr>
<td>s=5</td>
<td>0.0571</td>
<td>0.0575</td>
<td>0.0242</td>
<td><strong>0.0084</strong></td>
<td>0.0568</td>
<td>0.0264</td>
<td>0.0188</td>
</tr>
<tr>
<td>s=6</td>
<td>0.0386</td>
<td>0.0383</td>
<td>0.0238</td>
<td><strong>0.0083</strong></td>
<td>0.0384</td>
<td>0.0242</td>
<td>0.0180</td>
</tr>
<tr>
<td>s=7</td>
<td>0.0244</td>
<td>0.0244</td>
<td>0.0239</td>
<td><strong>0.0083</strong></td>
<td>0.0244</td>
<td>0.0244</td>
<td>0.0176</td>
</tr>
<tr>
<td>s=8</td>
<td>0.0242</td>
<td>0.0242</td>
<td>0.0233</td>
<td><strong>0.0083</strong></td>
<td>0.0242</td>
<td>0.0241</td>
<td>0.0167</td>
</tr>
</tbody>
</table>

Table 4.8: Comparison of total average error in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=0</td>
<td>6.64e-03</td>
<td>6.64e-03</td>
<td>2.67e-03</td>
<td>5.51e-03</td>
<td>6.64e-03</td>
<td>6.63e-03</td>
<td>3.69e-03</td>
</tr>
<tr>
<td>s=1</td>
<td>1.48e-04</td>
<td>1.47e-04</td>
<td>7.40e-05</td>
<td>7.84e-05</td>
<td>1.48e-04</td>
<td>1.37e-04</td>
<td>4.95e-05</td>
</tr>
<tr>
<td>s=2</td>
<td><strong>3.97e-06</strong></td>
<td><strong>3.70e-06</strong></td>
<td><strong>4.60e-06</strong></td>
<td>2.24e-05</td>
<td><strong>3.97e-06</strong></td>
<td><strong>2.82e-06</strong></td>
<td><strong>2.93e-06</strong></td>
</tr>
<tr>
<td>s=3</td>
<td>2.27e-05</td>
<td>2.12e-05</td>
<td>1.25e-05</td>
<td>3.58e-05</td>
<td>2.27e-05</td>
<td>1.76e-05</td>
<td><strong>7.52e-06</strong></td>
</tr>
<tr>
<td>s=4</td>
<td>1.73e-05</td>
<td>1.61e-05</td>
<td>1.12e-05</td>
<td>3.41e-05</td>
<td>1.73e-05</td>
<td>1.20e-05</td>
<td><strong>7.17e-06</strong></td>
</tr>
<tr>
<td>s=5</td>
<td>1.43e-05</td>
<td>1.44e-05</td>
<td>1.21e-05</td>
<td>4.19e-05</td>
<td>1.42e-05</td>
<td><strong>6.61e-06</strong></td>
<td>9.23e-06</td>
</tr>
<tr>
<td>s=6</td>
<td>1.00e-05</td>
<td>9.89e-06</td>
<td>1.27e-05</td>
<td>4.61e-05</td>
<td>9.90e-06</td>
<td><strong>9.12e-06</strong></td>
<td>1.11e-05</td>
</tr>
<tr>
<td>s=7</td>
<td>1.04e-05</td>
<td>1.06e-05</td>
<td>1.46e-05</td>
<td>5.97e-05</td>
<td>1.05e-05</td>
<td>1.28e-05</td>
<td>7.64e-05</td>
</tr>
<tr>
<td>s=8</td>
<td>1.87e-05</td>
<td>1.87e-05</td>
<td>4.96e-05</td>
<td>7.56e-05</td>
<td>1.89e-05</td>
<td>2.29e-05</td>
<td>7.60e-05</td>
</tr>
</tbody>
</table>
Table 4.9: Comparison of coverage average error in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=0</td>
<td>6.64e-03</td>
<td>6.64e-03</td>
<td>1.13e-02</td>
<td>4.39e-02</td>
<td>6.64e-03</td>
<td>6.63e-03</td>
<td>4.02e-03</td>
</tr>
<tr>
<td>s=1</td>
<td>1.48e-04</td>
<td>1.48e-04</td>
<td>5.30e-04</td>
<td>2.56e-03</td>
<td>1.48e-04</td>
<td>1.50e-04</td>
<td>1.13e-03</td>
</tr>
<tr>
<td>s=2</td>
<td><strong>4.11e-06</strong></td>
<td><strong>4.90e-06</strong></td>
<td>1.16e-04</td>
<td>9.72e-04</td>
<td><strong>4.11e-06</strong></td>
<td><strong>4.85e-06</strong></td>
<td>4.10e-04</td>
</tr>
<tr>
<td>s=3</td>
<td>2.35e-05</td>
<td>2.72e-05</td>
<td>1.54e-04</td>
<td>8.95e-04</td>
<td>2.35e-05</td>
<td>2.40e-05</td>
<td>3.42e-04</td>
</tr>
<tr>
<td>s=4</td>
<td>2.27e-05</td>
<td>2.17e-05</td>
<td>1.38e-04</td>
<td>9.10e-04</td>
<td>2.27e-05</td>
<td><strong>1.70e-05</strong></td>
<td>3.28e-04</td>
</tr>
<tr>
<td>s=5</td>
<td>2.00e-05</td>
<td>2.01e-05</td>
<td>1.44e-04</td>
<td>8.39e-04</td>
<td><strong>1.99e-05</strong></td>
<td><strong>1.23e-05</strong></td>
<td>3.42e-04</td>
</tr>
<tr>
<td>s=6</td>
<td><strong>1.58e-05</strong></td>
<td><strong>1.57e-05</strong></td>
<td>1.45e-04</td>
<td>8.30e-04</td>
<td><strong>1.58e-05</strong></td>
<td>5.27e-05</td>
<td>3.84e-04</td>
</tr>
<tr>
<td>s=7</td>
<td>4.76e-05</td>
<td>5.11e-05</td>
<td>1.53e-04</td>
<td>8.35e-04</td>
<td>4.82e-05</td>
<td>9.17e-05</td>
<td>4.51e-04</td>
</tr>
<tr>
<td>s=8</td>
<td>8.70e-05</td>
<td>8.70e-05</td>
<td>1.79e-04</td>
<td>8.31e-04</td>
<td>8.70e-05</td>
<td>1.30e-04</td>
<td>5.57e-04</td>
</tr>
</tbody>
</table>

All the error metrics are obtained as-is. This means that some of these metrics can be misleading. For example, the coverage in the All-C case is pretty high, with 8 out of 9 experiments with over 90% coverage; however, one look at the states until overestimation paints a different picture, as none of these experiments reached even 1000 states before generating too much overestimation. To account for these differences, we have 4 weighted metrics, obtained by multiplying the original metric times a weight based on how close each respective “states of overestimation” metric was to the goal of 4000 states:

\[
\text{<weighted metric>} = \frac{\text{<states until overest.}>}{4000}
\]

This creates four new metrics: weighted coverage, weighted accumulated error, weighted average error, and weighted coverage average error. Tables 4.10–4.13 show these weighted metrics.

Analysis of results

States until overestimation. The behavior in Table 4.5 is as expected. Giving wider initial states (Cases Ini-A, Ini-B and Ini-C) leads to overestimation sooner, the wider the initial states are. This effect is considerably more noticeable in the All Cases, which have a drastically reduced states until overestimation as the width of the added interval increases. The most interesting observations come from the other axis of this table: increasing the
Table 4.10: Comparison of weighted coverage in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.4178</td>
<td>0.1880</td>
<td>0.0000</td>
<td>0.0008</td>
<td>0.0608</td>
</tr>
<tr>
<td>s=1</td>
<td>0.0000</td>
<td>0.0038</td>
<td><strong>0.4733</strong></td>
<td>0.2060</td>
<td>0.0000</td>
<td>0.0868</td>
<td><strong>0.7100</strong></td>
</tr>
<tr>
<td>s=2</td>
<td>0.0327</td>
<td>0.2460</td>
<td><strong>0.5138</strong></td>
<td>0.0635</td>
<td>0.0345</td>
<td>0.4198</td>
<td><strong>0.7298</strong></td>
</tr>
<tr>
<td>s=3</td>
<td>0.0330</td>
<td>0.2178</td>
<td><strong>0.4868</strong></td>
<td>0.0600</td>
<td>0.0343</td>
<td>0.2650</td>
<td><strong>0.6895</strong></td>
</tr>
<tr>
<td>s=4</td>
<td>0.2380</td>
<td>0.2558</td>
<td><strong>0.4778</strong></td>
<td>0.0578</td>
<td>0.2383</td>
<td>0.2943</td>
<td><strong>0.6260</strong></td>
</tr>
<tr>
<td>s=5</td>
<td>0.3518</td>
<td>0.3568</td>
<td>0.4290</td>
<td>0.0475</td>
<td>0.2878</td>
<td>0.4610</td>
<td><strong>0.4963</strong></td>
</tr>
<tr>
<td>s=6</td>
<td>0.4590</td>
<td>0.4555</td>
<td>0.3710</td>
<td>0.0325</td>
<td>0.3620</td>
<td>0.5480</td>
<td><strong>0.3958</strong></td>
</tr>
<tr>
<td>s=7</td>
<td>0.2530</td>
<td>0.2530</td>
<td>0.0850</td>
<td>0.0250</td>
<td>0.2505</td>
<td>0.2163</td>
<td>0.0475</td>
</tr>
</tbody>
</table>

Table 4.11: Comparison of weighted total accumulated error in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=1</td>
<td>0.5927</td>
<td>0.5896</td>
<td>0.2961</td>
<td>0.3136</td>
<td>0.5927</td>
<td>0.5482</td>
<td>0.1982</td>
</tr>
<tr>
<td>s=2</td>
<td><strong>0.0159</strong></td>
<td>0.0148</td>
<td>0.0184</td>
<td>0.0898</td>
<td><strong>0.0159</strong></td>
<td>0.0113</td>
<td>0.0117</td>
</tr>
<tr>
<td>s=3</td>
<td>0.0910</td>
<td>0.0850</td>
<td>0.0502</td>
<td>0.1432</td>
<td>0.0909</td>
<td>0.0705</td>
<td><strong>0.0301</strong></td>
</tr>
<tr>
<td>s=4</td>
<td>0.0691</td>
<td>0.0646</td>
<td>0.0449</td>
<td>0.1365</td>
<td>0.0691</td>
<td>0.0481</td>
<td><strong>0.0287</strong></td>
</tr>
<tr>
<td>s=5</td>
<td>0.0571</td>
<td>0.0575</td>
<td>0.0483</td>
<td>0.1677</td>
<td>0.0568</td>
<td><strong>0.0265</strong></td>
<td>0.0369</td>
</tr>
<tr>
<td>s=6</td>
<td>0.0401</td>
<td>0.0396</td>
<td>0.0506</td>
<td>0.1844</td>
<td>0.0396</td>
<td><strong>0.0365</strong></td>
<td>0.0443</td>
</tr>
<tr>
<td>s=7</td>
<td>0.0416</td>
<td>0.0425</td>
<td>0.0584</td>
<td>0.2386</td>
<td>0.0420</td>
<td>0.0511</td>
<td>0.3057</td>
</tr>
<tr>
<td>s=8</td>
<td>0.0750</td>
<td>0.0750</td>
<td>0.1983</td>
<td>0.3022</td>
<td>0.0756</td>
<td>0.0917</td>
<td>0.3039</td>
</tr>
</tbody>
</table>

Table 4.12: Comparison of weighted total average error in the experiments.

<table>
<thead>
<tr>
<th>Disc.</th>
<th>Base</th>
<th>All-A</th>
<th>All-B</th>
<th>All-C</th>
<th>Ini-A</th>
<th>Ini-B</th>
<th>Ini-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=0</td>
<td>6.64e-03</td>
<td>6.64e-03</td>
<td>4.88e-03</td>
<td>2.56e-02</td>
<td>6.64e-03</td>
<td>6.63e-03</td>
<td>4.95e-03</td>
</tr>
<tr>
<td>s=1</td>
<td>1.48e-04</td>
<td>1.47e-04</td>
<td>1.35e-04</td>
<td>3.69e-04</td>
<td>1.48e-04</td>
<td>1.37e-04</td>
<td>6.67e-05</td>
</tr>
<tr>
<td>s=2</td>
<td><strong>3.97e-06</strong></td>
<td><strong>3.70e-06</strong></td>
<td>8.60e-06</td>
<td>3.45e-04</td>
<td><strong>3.97e-06</strong></td>
<td><strong>2.82e-06</strong></td>
<td><strong>3.98e-06</strong></td>
</tr>
<tr>
<td>s=3</td>
<td>2.27e-05</td>
<td>2.12e-05</td>
<td>2.37e-05</td>
<td>5.73e-04</td>
<td>2.27e-05</td>
<td>1.76e-05</td>
<td>1.07e-05</td>
</tr>
<tr>
<td>s=4</td>
<td>1.73e-05</td>
<td>1.61e-05</td>
<td>2.16e-05</td>
<td>5.69e-04</td>
<td>1.73e-05</td>
<td>1.20e-05</td>
<td>1.12e-05</td>
</tr>
<tr>
<td>s=5</td>
<td>1.43e-05</td>
<td>1.44e-05</td>
<td>2.40e-05</td>
<td>8.39e-04</td>
<td>1.42e-05</td>
<td><strong>6.61e-06</strong></td>
<td>1.81e-05</td>
</tr>
<tr>
<td>s=6</td>
<td><strong>1.04e-05</strong></td>
<td><strong>1.02e-05</strong></td>
<td>2.69e-05</td>
<td>1.02e-03</td>
<td><strong>1.02e-05</strong></td>
<td><strong>1.02e-05</strong></td>
<td>2.72e-05</td>
</tr>
<tr>
<td>s=7</td>
<td>1.77e-05</td>
<td>1.85e-05</td>
<td>3.56e-05</td>
<td>1.70e-03</td>
<td>1.80e-05</td>
<td>2.68e-05</td>
<td>1.33e-03</td>
</tr>
<tr>
<td>s=8</td>
<td>5.81e-05</td>
<td>5.81e-05</td>
<td>4.22e-04</td>
<td>2.75e-03</td>
<td>5.90e-05</td>
<td>8.74e-05</td>
<td>1.38e-03</td>
</tr>
</tbody>
</table>
Table 4.13: Comparison of weighted coverage average error in the experiments.

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

The complexity of the discretization also leads to faster overestimation. 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.

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 cases All-C and Ini-C. Even the weighted version, which accounts for a specific model’s not meeting the expected number of states before overestimation, supports this, with Ini-C maintaining good coverage, while All-B gets better weighted coverage as All-C 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 of the error in these higher order approximations needs to be as small as possible.

Accumulated and average error. Next to coverage, accumulated error is a good measure of how close each experiment was to the actual behavior. 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 good values for the \textit{All} $- C$ case experiments is also justified: these experiments last much shorter before overestimation and tend to have much wider enclosures. Once we weighted these results based on the states until overestimation, these results stopped being as good.

The most surprising results, both weighted and not, are the ones for $s = 2$. They are consistently low, for both accumulated and average errors. This suggests that the $s = 2$ discretization level is overall closer to bounding the expected solution. When including the coverage average errors, there’s a second discretization that gets closer: $s = 6$. Its only drawback is that it did not reach the expected final state of 4000. However, it is important to point that that this is under the conditions of this experiment; there is always the possibility of finding tighter enclosures overall, which would extend the time states until overestimation, by tweaking the contractor’s settings to have more aggressive contraction in exchange of additional computation time.
5 CONCLUSIONS

In the previous chapters, we described the motivation of our work: the abundance of dynamical systems, and the need to solve dynamical systems problems with rigorous computations, which can be provided by interval analysis techniques. We then reviewed the theory behind dynamical systems, interval computations, and the methods frequently used to solve dynamical systems using intervals. We then described the problem that interests us the most: solving complex dynamical systems that require large-scale discretizations, which makes them prone to reaching state enclosures that are too wide to provide useful information to a decision maker.

For the next chapter, we presented our contributions. The first contribution is the Sliding Windows algorithm, a technique that uses implicit discretizations of dynamical systems, solving a series of successive state “windows” taken from the whole system that will provide a full view of the system’s solution once put together. The second contribution is an exploration of the discretization complexity and how it affects the Sliding Windows algorithm. The objective of this exploration is to better understand how complex a discretization must be to provide good results, even when lacking a term that bounds the error in approximation.

In this chapter we present our conclusions and remarks concerning the methods and experiments presented in this work, followed potential future work to refine the techniques presented here and how they could be used to provide improved results in large-scale discretized dynamical systems.
5.1 Conclusions

In this work we present the Sliding Windows algorithm, a technique that takes advantage of the structure in a discretized dynamical system to find solution enclosures using interval constraint solving techniques. Its main idea is to not solve the entire problem at once, but focus instead on much smaller subproblems built from successive states and their equations, which we call “windows”. After each window is solved using an interval constraint solver, we take the last states and equations from that “window” as a seed for a new one of the same size, as if we were moving a sliding window but leaving a part that “overlaps” between the previous and current “window”.

For this algorithm, we carried experiments to analyze its performance against existing methods, using simple discretization to generate the necessary systems of state equations. We then measured the performance of the Sliding Windows against the existing interval solvers DynIBEX and VSPODE, measuring performance in computation time, interval width of the enclosure, and number of states before the system reached an undesirable amount of overestimation.

Our results are promising, with comparable and sometimes better performance overall than existing methods. This, to us, highlights the potential of this technique, with one major caveat: these experiments were carried out on a discretized system that does not include a term that bounds the approximation error generated by the discretization, which means that the discretization that was used is not guaranteed to be a reliable enclosure of the real solution.

As we were unable to find an adequate bounding function for the approximation error in our discretization, we instead chose to explore the accuracy and flexibility of implicit discretization methods that involve multiple states in each state equation, when using them with the Sliding Windows algorithm. We measured the distance between the results obtained from a given discretization and a point-based solution. We also explored the flexibility of each discretization by artificially increasing the width of the solution and measuring that
same distance.

We found that increasingly complex discretizations are not necessarily better. The simple discretization used in the first experiment was definitely the worst, but once we started using more complex discretization (meaning, involving more state variables per equation), we quickly started getting better results, with discretizations involving 2 prior states producing enclosures closer to the real value. However, involving more than 2 variables produces diminishing returns: the system starts having earlier overestimation overall. As our objective is to solve complex dynamical systems with large-scale discretizations, these results show that we must avoid the simplest discretization levels yet limit ourselves to a reasonable number of states per state equation. All of this, of course, with the same caveat as before: these are not guaranteed enclosures until we find a function to bound the approximation error in each state equation.

5.2 Future work

The first venue of future work is evident: as mentioned earlier, our experiments lack discretized equations with bounded approximation error. The discretization process employed throughout this work creates a series of symbolic equations, which our algorithm gathers and divides into their respective windows. Because of this, we would need a symbolic error-bounding function that can be included in these discretized equations. Having such an equation opens more research possibilities.

The second venue involves other methods to improve the constraint solving process itself. The Sliding Windows algorithm takes advantage of how we know in advance which states are used on each state equation to divide the problem into a series of overlapping subproblems, each being solved through an interval constraint solver that evaluates each constraint, given as a symbolic expression, and applies a series of contractors over a given domain based on one or more constraints. Within this venue, here are some potential topics:

- Improve the interval evaluation process, either by an a priori symbolic manipulation of
the constraints or by evaluating the constraint through a different interval extension.

• Experiment with the contraction process itself, to find a mixture of contractors that can provide either more reduction or in less time than the default one in IbexSolve used throughout this work.

• Build a custom contractor based on the structure of dynamical systems.

The third venue is a consequence of the other two. Once we have bound the approximation error and adapted the constraint solving process to the needs of dynamical systems, we can then focus on exploring specific applications and how the explicit methods used in other solvers can be used to work in tandem with the implicit methods employed in this work. We discretized our system of equations using the implicit Adams-Moulton method; in non-interval problems, this type of discretization which is used in tandem with the explicit Adams-Bashfort method as a *predictor-corrector* pair, with the explicit method providing a starting point for the more accurate implicit method. With interval computations, this process would work differently, as the corrector part given by the implicit method (in the interval case, solved by interval constraint contraction) requires an initial domain. In this case, it would mean obtaining an initial enclosure by using interval techniques with explicit discretization, then using those results in an implicit system solved using interval constraint solving techniques, which can potentially result in bigger contraction.


Angel Fernando García Contreras was born in Mexico City in 1982. He graduated with honors from the Tecnologico de Monterrey in Ciudad Juárez, Chihuahua, Mexico in 2005 with a Bachelors in Computing Systems Engineering degree. He worked in the IT industry developing management information systems for private businesses and the Mexican government until 2010.

In spring 2010, Angel entered the Graduate School at the University of Texas at El Paso. During this time, his research focused on global optimization and continuous constraint solving using interval analysis techniques. He completed his Masters in 2014, with the Thesis titled “Contributions to global optimization using interval methods and speculation”. He started his Doctoral studies right afterwards, eventually shifting his research into solving dynamical systems using interval constraint-solving techniques.

During his studies at UTEP, Angel worked as a Teaching Assistant for undergraduate courses of all levels. From 2019-2021 he taught an introductory programming course for non-computer science students. He has a keen interest in language and human computer interaction, explainability, and computer science education.

Contact information: afgarciacontreras@google.com

This thesis was typed by Angel Fernando Garcia Contreras.