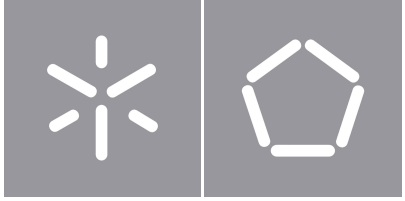
University of Minho
School of Engineering

Ricardo da Silva Correia

Simulation of Hybrid Systems Regulated by Newtonian Mechanics

november 2023



University of Minho
School of Engineering

Ricardo da Silva Correia

Simulation of Hybrid Systems Regulated by Newtonian Mechanics

Masters Dissertation
Master's in Physical Engineering

Area of physics of information
Dissertation supervised by
Renato Neves
José Proença

november 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

I couldn't help but express my deep gratitude to all the people who made this research project possible. The journey that comes to an end here was enriched by their support and dedication, so I must mention those whose contributions were essential to my growth.

Firstly, I would like to thank the University of Minho for providing the opportunity to carry out this project, allowing me to effectively apply the knowledge I have acquired throughout my academic journey.

In this regard, I wish to express my sincere appreciation to my academic supervisors, Renato Neves and José Proença, for their guidance and support throughout this journey, steering this project to success.

On this path, I cannot fail to mention my friends, girlfriend and fellow travelers who were always available to help and motivate me during my academic journey.

To my parents and sister, for believing in me and their unwavering support at every stage of my life.

To everyone, my heartfelt thanks.

This work was partially supported by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference PTDC/CCI-COM/4280/2021.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, november 2023

Ricardo da Silva Correia

Abstract

The evolution of software products that interact with the physical world has led to a greater need to simulate their behavior in order to verify their effectiveness and safety in different scenarios. This dissertation project aims to enhance a simulation tool for hybrid programs called Lince, more specifically to provide more powerful simulation capabilities to hybrid programs regulated by Newtonian mechanics. These include the addition of new language constructs (such as the division operator and the trigonometric functions), the implementation of non-linear expressions, grammar relaxation and organization, improved error detection, and the mitigation of existing tool-related issues.

Throughout this dissertation, it is discussed how the implementation of these improvements benefits the simulation of hybrid programs and are explained the key methods adopted for their conception. Finally, this new version of Lince is put to the test by handling case studies related to autonomous driving (for example, adaptive cruise control and a missile targeting a moving object) and other types of systems as well, such as purely physical systems and the so-called on-off systems. The results obtained in the treatment of these case studies attest to the enhanced capabilities of this tool and the contribution of this dissertation to the scientific community, demonstrating its relevance in simulating integrated systems in everyday life.

Keywords Lince, hybrid programming, autonomous driving, SageMath.

Resumo

A evolução de produtos de *software* que interagem com o mundo físico levou a uma maior necessidade de simular o seu comportamento como forma de verificar a sua eficácia e segurança em diferentes cenários. Este projeto de dissertação tem como objetivo melhorar uma ferramenta de simulação para programas híbridos denominada Lince, mais especificamente para fornecer capacidades de simulação mais poderosas a programas híbridos regulados pela mecânica Newtoniana. Estas incluem a adição de novas construções linguísticas (como o operador de divisão e as funções trignométricas), a implementação de expressões não-lineares, relaxamento e organização da gramática, melhoria da deteção de erros e a mitigação de problemas existentes associados à ferramenta.

Ao longo desta dissertação, é discutida a forma como a implementação dessas melhorias beneficia a simulação de programas híbridos e são explicados os principais métodos adotados para a sua conceção. Finalmente, esta nova versão do Lince é testada através do tratamento de casos de estudo relacionados com a condução autónoma (por exemplo, o *adaptative cruise control* e um míssil que visa um objeto em movimento) e também outros tipos de sistemas, como sistemas puramente físicos e os chamados sistemas *on-off*. Os resultados obtidos no tratamento destes casos de estudo aferem para as capacidades melhoradas desta ferramenta e o contributo desta dissertação para a comunidade científica, demonstrando a sua relevância na simulação de sistemas integrados no quotidiano.

Palavras-chave Lince, programação híbrida, condução autónoma, *SageMath*

Contents

List of Figures

List of Tables

Acronyms

CPS Cyber-Physical System

CPSs Cyber-Physical Systems

AEB Automatic Emergency Bracking

ACC Adaptative Cruise Control

DDL Differential Dynamic Logic

CTL Computation Tree Logic

JVM Java Virtual Machine

Part I

Introductory material

Chapter 1

Introduction

1.1 Motivation and context

Hybrid programs orchestrate classical computation with physical processes, for example movement and velocity, in order to achieve certain goals. In modern programming practice there is a growing trend of combining programs with physical processes, even though this combination is often implicit. The reason behind this trend is the need to deliver software products that can closely interact with the physical world, typically involving aspects such as velocity, movement, energy, and time. Numerous examples of such systems exist, spanning from small medical devices like pacemakers and infusion pumps to more complex systems like surgical robots, autonomous vehicles, and large-scale electric grids that serve entire districts [Nev18].

Figure 1: *Pacemaker device [Hea22].*

Figure 2: *Autonomous vehicle [Aut].*

The rapid proliferation of hybrid programs in the recent decades has led to a flurry of research on languages, semantics and tools for their design and analysis [GNP20a, GST09, Pla10, SH11]; but while great progress has been made in this area, several important challenges remain largely open. A case in point concerns the simulation of hybrid programs centred on the tool Lince [GNP20a]: it lacks certain language constructs that are important for modelling important classes of hybrid programs, particularly

those governed by Newtonian mechanics, such as autonomous vehicles. The latter are important because a myriad of autonomous systems are currently under development and being able to simulate them before implementation is useful to ensure that they will behave as intended.

1.2 Contributions

The first objective of this dissertation was to enhance the tool Lince in order to improve the simulation of hybrid programs, in particular to make it better suited to those programs that interact with physical processes governed by Newtonian mechanics. The improvements and contributions made in this regard consist of:

- **Addition of new language constructs:** this includes arithmetic operators such as division and remainder and mathematical functions such as exponentiation, power, square root, and trigonometric functions. This expansion enables a more rigorous simulation of hybrid programs that, for example, depend on calculating intersection points between two trajectories (which often involves the use of division and square roots).
- **Extension of the Lince programming language to support non-linear expressions:** the ability to use non-linear expressions, along with the new constructs, enables not only the simulation of a wider range of hybrid programs, but also provides an increased user comfort in their design.
- **Detection and implementation of potential improvements:** these improvements consist, for example, in avoiding what we call 'block assignments', which require assignments to be made atomically rather than in blocks, and the relaxation of conditional expressions.
- **Enhancement of semantic error detection:** this enhancement involves the development of semantic error detection strategies, such as mathematical indeterminacies, and the detection of non-supported mathematical operations. This contribution will facilitate the user's debugging process.
- **Mitigation of existing limitations in Lince:** this improvement consists of mitigating some of the limitations of the tool, such as the inability to handle large symbolic expressions.

The second objective of this dissertation was to explore case studies using Lince's improved version. In this regard, we put special emphasis on case-studies revolving around autonomous driving, due to their recent proliferation in the automotive industry. Specifically, we explored Automatic Emergency Bracking

(AEB) systems, Adaptative Cruise Control (ACC) and a system capable of missile targeting a moving object. Other important systems, such as classical physics systems and on-off systems, were also thoroughly studied.

In sum this dissertation develops a "next-generation" simulation tool with applications in diverse areas, offering a wide range of possibilities for precise and comprehensive simulations of hybrid programs.

1.3 Document structure

This dissertation consists of 6 chapters that address different aspects of the project. Below is a summary of each chapter:

- **Chapter 1: Introduction**

  In this initial chapter the context that motivated this dissertation project is presented and the main contributions made are highlighted. The chapter also provides an overview of the document's structure.

- **Chapter 2: State of the Art**

  The second chapter addresses the concepts of a hybrid system and of a cyber-physical system. Additionally, examples of tools capable of modelling and simulating hybrid programs will be presented, with a special focus on the tool Lince. The strengths and weaknesses of the tool will be analysed, and a brief overview of the Scala language, used in the implementation of Lince, will be provided, with emphasis on parsing.

- **Chapter 3: Lince and Newtonian Systems**

  The third chapter covers the types of differential equations required to model hybrid programs regulated by Newtonian mechanics. The main obstacles limiting the design in Lince of a wide range of hybrid programs governed by Newtonian mechanics will be identified. Moreover the implemented solutions to overcome these obstacles will be presented, enabling a more comprehensive and accurate simulation.

- **Chapter 4: Improvements to Lince**

  In this chapter each of the implemented solutions will be detailed, presenting the limitations of the previous version, the adopted method to overcome them, and the results obtained from the changes made.

- **Chapter 5: Autonomous Driving and Beyond**

  The fifth chapter discusses the importance of autonomous driving and its limitations. The utility of Lince's improved version in the simulation of systems associated to autonomous driving will be highlighted, as well as Lince's ability to simulate classical physical systems, on-off systems, and to perform numerical calculations.

- **Chapter 6: Conclusion and Future Work**

  In the final chapter, possible future work and conclusions are drawn from this dissertation and discussed.

Additionally, an Appendix is also provided with relevant functions for the improvement of the tool, the full version of the hybrid programs explored in the case studies, the grammars of Lince's old and new versions, and further exploration of some topics addressed in the aforementioned overview of the language Scala.

Chapter 2

State of the Art

2.1 Hybrid systems and cyber-physical systems

As explained in [BG11], the term "Cyber-Physical System (CPS)" refers to next-generation digital devices that closely interact with physical processes, such as velocity, time, and movement. Such a discrete-continuous interaction gives rise to new, interesting ways of managing the physical world through computation, communication and control. It is therefore a key piece in future technological developments.

On the other hand, the term "hybrid system" is used almost as a synonym to that of a CPS but such is not the case. Whilst the latter usually refers to an actual technological piece (or composite) of software (or hardware), the former is typically reserved to the *mathematical model* used to study the behaviour of a CPS, the external environment, and their interaction. Interestingly, as a modelling framework, hybrid systems go much beyond cyber-physical ones, as they avoid some restrictions concerning computability. For example, one can model impact-based physical systems, such as a bouncing ball (which can bounce *infinitely* many times in *finite* time) [HLLDS09].

As referred in [Mat21], Cyber-Physical Systems (CPSs) are used in various areas, some of which are:

- **Manufacturing** - They are useful in optimising manufacturing processes, reducing costs, labour and production times;
- **Healthcare and Medical Devices** - CPSs allow to monitor and manage the state and physical condition of patients, both remotely and in real-time. They also allow to monitor older people through intelligent sensors which *e.g.* in case of a fall send out an alert;
- **Agriculture** - CPSs facilitate a more efficient use of water and pesticides. Also, these systems can accurately collect and analyse the state of the soil, climate, etc., to improve agricultural management;

- **Automotive industry** - CPSs are used in autonomous driving of vehicles, cruise control, automatic braking, automatic parking, etc;

The architecture of a CPS, in general, is presented in Fig. 3.

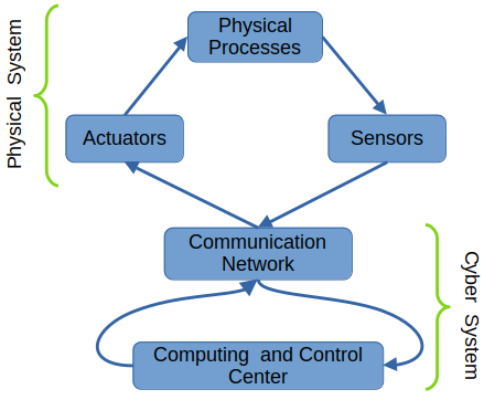


Figure 3: *The architecture of a CPS [WD22].*

This architecture consists of a cyber system and a physical system. The physical system consists of actuators (responsible for performing physical actions), sensors (responsible for acquiring data in real time) and physical processes. On the other hand, the cyber system consists of communication networks (responsible for sending data from the sensor to the control centre and sending signals from the control centre to the actuators) and computing and control centres (responsible for receiving and analysing data measured by sensors, being the control centre in charge of making the right decisions so that physical processes are correctly managed) [WD22]. [Hu14] is another interesting source of discussion about the architecture of CPSs and their role in society.

As already mentioned, a CPS typically consists of a network of devices that measure and perform physical actions whilst being controlled and monitored by a computational system and communication software. However the ability to precisely implement such an orchestration is limited by current computer technologies, particularly by the lack of system reliability and the inability to follow rigorous timing constraints. Indeed the use of precise computations to control an (apparently) unpredictable physical environment is a major challenge; and moreover any failures or safety issues arising from this control must be contained and handled in an efficient way, otherwise they might be propagated to the remaining systems. Synchronisation within a system and overcomplexity are also obstacles that must be taken into consideration for the field of CPS engineering to grow. All this stresses the importance of having rigorous tools to test CPSs before their deployment.

2.2 Modelling cyber-physical systems: tools

The task of modelling discrete-continuous interactions is a crucial step in the engineering of CPSs. In fact, there are currently several tools that can help us in this task. A brief description of some of them is provided below.

Keymaera X is a (semi-automated) theorem prover for Differential Dynamic Logic (DDL). The latter is a logic for specifying and verifying properties of hybrid systems, where (as mentioned above) a hybrid system consists of a mathematical model that mixes the continuous and the discrete worlds (and has therefore applications in the domain of CPSs). In particular, the tool includes a very simple language of "hybrid programs" which supports the usual program constructs, such as sequential composition, while loops, and conditionals. Since hybrid programs tend to be quite complex, Keymaera X supports a palette of sophisticated proof techniques and a rich interface to guide the user. Among other things, the tool allows to specify custom proof search techniques and to execute them in parallel. More details can be found in [FMQ+15], which is the basis of the current description.

Simulink is an intuitive "block-based" modelling language with simulation capabilities. It can be used to test the ideas underlying a complex system before actual deployment. This can be used for example to test an Electro-Mechanical Braking System [MNB17]. Simulink also has other functionalities that facilitate its integration in the software development cycle. For example, it has the ability to generate actual executable code from the model at hand such as C, C++, CUDA, PLC, Verilog, and VHDL. More details can be found in the tool's website [Mat23].

Uppaal is a tool based on timed automata which essentially allow to model the interaction between computation and time. In this context, arrows between states are called "edges" and states are called "locations". At each location there is the possibility of placing an "invariant" (a condition that dictates when one can stay at the current location). As for edges, these can contain guards (conditions that only allow the edge to be active if they are true), synchronisation channels (to interact with other automata), and commands for changing values in memory that register time. With all these features present in Uppaal's timed automata, one can build models to simulate real-time systems such as the adventurers problem (see reference [Nev22a]) and the correct functioning of traffic lights at an intersection (see reference [Nev22b]). Such systems were actually modelled by the author of the present dissertation in the module "Cyber-Physical Computation" (held in the second semester of the first year of the MSc in Physics Engineering in the area of Information Physics).

Further than simulating/modelling real-time systems, Uppaal also allows to check whether certain

properties are satisfied based on a logic called Computation Tree Logic (CTL). With this feature one is able to check, for example, whether the adventurers (mentioned above) can cross the bridge safely and whether the traffic lights turn green at the same time (which is unsafe). For more details on the topic see [BLL+95] and [BDL06] .

Modellica is an object-oriented language for modelling physical systems. This tool is commonly used in electrical circuits, robotics and electrical power distribution. A Modelica model is described by a set of synchronous, differential and discrete equations leading to deterministic behaviour and automatic synchronization between continuous time and discrete event [EMO01].

2.3 Lince in detail

Another tool capable of modelling CPSs is Lince, which is the focus of the current project. Given its importance, this entire subsection is dedicated to its description. As explained in [GNP19], the tool receives as input a hybrid program, written on an imperative language that supports cyclic structures ("while" and "repeat"), conditionals ("if-then-else"), wait commands, assignments and linear differential equations associated with a duration. The latter represents the duration during which a continuous dynamics (specified by the differential equation) is active. An example of a hybrid program supported by Lince is shown to the left of Fig. 4. This program starts with a discrete assignment to the variables "p" and "v", and then enters a "while" loop where at each iteration it checks if the value of "v" is less than or equal to 10. If this condition is met, it continuously updates the variables "p" and "v" for 1 second using the system of differential equations "p'=v,v'=5". If this condition is not met, it continuously updates the variables "p" and "v" for 1 second using the system of differential equations "p'=v,v'=-2". In addition to this hybrid program, others will be addressed and explained throughout this project.

The language only supports *linear* differential equations to avoid situations where the solution cannot be determined symbolically (the SageMath tool, used to solve differential equations symbolically, cannot handle a wide range of non-linear differential equations symbolically) and also because the research team initially wanted to start by treating the simplest cases. After inputting a program the tool generates a plot of the program's variables as a function of time, which are computed based on symbolic solutions of the differential equations provided by SageMath. This plot can be manipulated, allowing the user to select regions, zoom in and download, among other things.

The architecture of Lince is schematised in Fig. 4 [GNP19, GNP20b]. The *Core engine* is responsible

for parsing and evaluating hybrid programs. The *Inspector* builds the respective trajectories by requesting the evaluation of hybrid programs at different time instants and numerically computing the samples[1] between them. In other words, the *Inspector* requests the expressions for the solutions of the differential equations at the corresponding time instances based on their durations, and uses these solutions to numerically compute the samples between these instances. It additionally provides a user interface to evaluate hybrid programs at a given instant of time. The *Inspector* module in Lince also allows to adjust parameters of the plot, such as the time interval, the visibility of variable's trajectories and the display of additional trajectory information. In the *Core engine*, as mentioned earlier, SageMath is used to solve the differential equations and to evaluate the conditional expressions. The advantage of using SageMath is that it uses symbolic manipulation of the differential equations and the conditional expressions, allowing more precise results to be obtained, although the execution time is longer. In Figure 4 the arrows represent input and output relations.

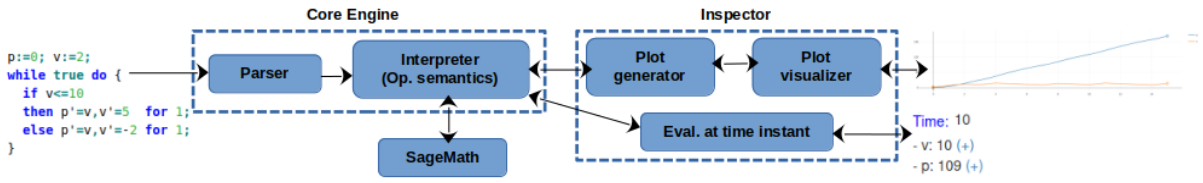


Figure 4: *Depiction of* Lince*'s architecture [GNP20b]*

Next, three examples of hybrid programs in Lince are presented.

2.3.1 Example 1: A simple displacement

The following hybrid program describes a simple displacement of a particle.

```
p:=0;
p'=2 for 2;
p'=-2 for 2;
```

First it initializes variable "p" (which represents position) to 0 meters. Then "p" evolves during the first 2 seconds according to the differential equation "p'=2" and in the 2 seconds after according to the differential equation "p'=-2". These differential equations indicate that in the first 2 seconds variable "p" will evolve with a slope equal to 2 and in the next 2 seconds it will evolve with a slope equal to -2 (the slope is equivalent to velocity). Lince presents the following plot respective to this program.

[1] Throughout this project the term "samples" refers to the numerical values collected at regular time intervals from a trajectory. These samples are used to analyse an approximation of the trajectory over time. The more frequent the samples, the more detailed the representation of the trajectory will be [DdMBJ].

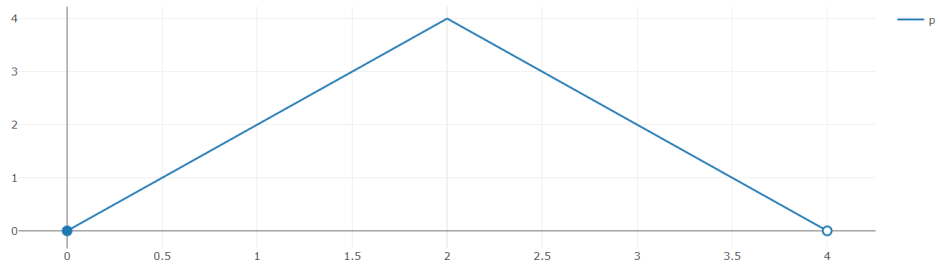


Figure 5: *Plot of Example 1: A simple displacement*

2.3.2 Example 2: Cruise control

This example concerns the simulation of a cruise control which regulates the velocity of a car so that it is always in a certain range [GNP20a]. Consider the following program in Lince:

```
p:=0; v:=0;
while true do {
  if v<=10
  then p'=v,v'=3  for 1;
  else p'=v,v'=-3 for 1;
              }
```

It does the following: assigns initial values to position "p" and velocity "v"; then performs a "while" loop that will repeatedly check after 1 second whether velocity is less than or equal to 10 m/s. If true, both variables evolve for 1 second according to the system of differential equations "p'=v, v'=3" [2] to accelerate the vehicle. If false, both variables evolve for 1 second according to the system of differential equations "p'=v, v'=-3" which effectively causes the vehicle to brake. Lince presents the following plot respective to this program.

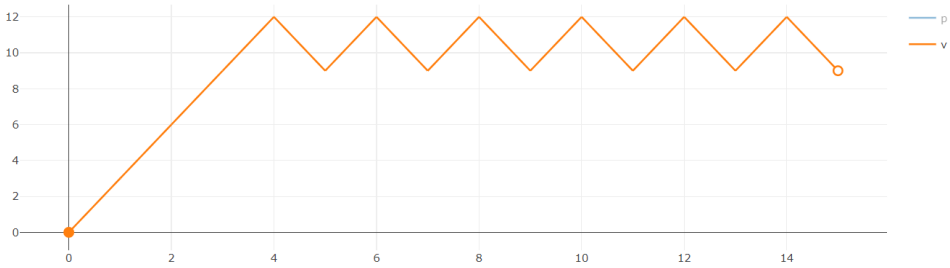


Figure 6: *Plot of Example 2: Cruise control*

[2] Recall the equations of motion from elementary physics, where the derivative of position is velocity and the derivative of velocity is acceleration.

2.3.3 Example 3: Unsupported example

This example attempts to illustrate the following system concerning autonomous driving: a vehicle needs to follow a vehicle in front as closely as possible and without colliding (it is assumed that overtaking is impossible) Consider the following program in Lince:

```
p:=0; v:=0; pl:=50; vl:=10;
np:=0; nv:=0; aux:=0;
while true do {
    np:=p+v+2.5;
    nv:=v+5;
    aux:=pow((nv-10),2)+4*(np-(pl+10));
    if (np<pl+10)/\(aux<0)
    then p'=v,v'=5,pl'=10 for 1;
    else p'=v,v'=-2,pl'=10 for 1;
            }
```

It initially assigns values to the position and velocity of the vehicle that goes behind ("p:=0, v:=0"), and then to the position and velocity of the vehicle in front (*i.e.* the leader) ("pl:=50, vl:=10"), after this it assigns initial values to variables that will be used in auxiliary calculations. Then a "while" loop is created that repeatedly does the following: assigns the value of the position of the follower vehicle after 1 second to the variable "np" , based on its current position and speed and assuming that it will accelerate with an acceleration equal to 5 m/s^2 (in essence, this is being predicted using the equations of motion); then performs an analogous routine for velocity. After this it assigns to the variable "aux" the value of the discriminating binomial resulting from the equality between the position formula of the follower in the case of braking (acceleration equal to -2) having "np" as initial position and "nv" as initial velocity, and the position formula of the leader with initial position "pl+10" and initial velocity equal to 10 m/s (it is assumed that the follower travels at a constant speed). Variable "aux" will be used to detect future possible collisions between the two vehicles. More specifically, through the Boolean condition of the "if-then-else" instruction, we check whether when the follower accelerates for 1 second, its new position ("np") is lower than the leader's position ("pl+10") and also if variable "aux" is smaller than zero (this indicates the absence of possible crashes in the future when the follower decides to accelerate). More formally, this last condition states that there are no solutions in determining intersection points between the trajectories estimated for both vehicles. Such a condition is necessary for *e.g.* avoiding the case in which the follower gains so much velocity that it can longer brake before colliding with the leader.

If the Boolean condition is true (meaning that the vehicles do not collide if they accelerate after 1 second or later), the rear vehicle will accelerate with an acceleration of 5 m/s^2 for 1 second. If the condition is false, the rear vehicle will decelerate with an acceleration of -2 m/s^2 during the same time

interval. In both cases the front vehicle maintains a constant speed.

When this program is given as input to Lince, the following parsing error message is returned:

```
Error: When parsing G§15 // maximum time in the plot§0§p:=0; v:=0; pl:=50; vl:=10;
np:=0; nv:=0; aux:=0;
while true do{
np:=p+v+2.5;
nv:=v+5;
aux:=pow((nv-10),2)+4*(np-(pl+10));
if (np<pl+10)/\(aux<0)
then p'=v,v'=5,pl'=10 for 1;
else p'=v,v'=-2,pl'=10 for 1;
} - hprog.common.ParserException: [3.1] failure: end of input expected
while true do{
^
```

Figure 7: *Error message returned by Lince*

This happens because the instruction "$\mathrm{pow}((\mathrm{nv}\text{-}10),2)$" (responsible for exponentiation) does not exist in Lince's language. A few other limitations are mentioned in the following section.

2.3.4 Strengths and limitations

Like any tool, Lince offers a number of advantages, some of which are [GNP19, GNP20a, GNP20b]:

- **Simple program constructs** – A very simple, clear language with program constructs familiar to all programmers. Contrarily to *e.g.* Simulink, the language is text-based which facilitates the development of complex systems whilst in Simulink its visual design may become quite cumbersome;
- **Well-defined semantics** – Contrarily to most tools, Lince's language has a clear, well-defined semantics which allows for a more rigorous verification of safety-critical systems;
- **Simulations capabilities** – Contrarily to Keymaera X, Lince has simulation capabilities which facilitates the analysis of complex systems. In fact in the former tool, theorem proving is a highly specialised task and resource-demanding. Simulation, on the other hand, is a relatively easy task to perform which can already reveal many possible bugs in the system under development.

Lince has also several restrictions that hinder its use in more complex hybrid programming. In detail:

- **Only linear expressions** – Assignments only support simple linear expressions which greatly restricts the specification of discrete actions. This restriction also applies to the specification of a

continuous dynamic's duration, in the expressions of the differential equations and in the relational expressions;

- **Rigid syntax** – For example the relational expressions of conditional instructions can only contain a variable on the left of the relational operator and a linear expression on the right, making it impossible, for example, to have "$\mathrm{x+1<=y+2}$". Furthermore, linear expressions are associated with a very restricted syntax, so that it is not possible to use, for example, "$\mathrm{2*(x/2)}$", but if you write "$\mathrm{2*(x*0.5)}$", there is no problem (the reader is advised to consult the Lince's grammar of the old version from the non-terminal symbol linP to the end in Appendix B.1);

- **Few supported arithmetic operators and no supported mathematical functions/constants** – The only supported operators were addition, subtraction, negation, and multiplication;

- **Unintuitive atomic assignments** – The semantic analyser updates the values of the variables at the end of each assignment block and not at each assignment necessarily. For example, if the variable "x" is assigned with the value 1 and "y" with the value 0, and within a "$\mathrm{if\text{-}then\text{-}else}$" is introduced the statement "$\mathrm{x:=10;\ y:=x+10;}$", the result of the variable assignment "x" will be 10, while the variable "y" will be assigned a value of 11, which is different from the expected value of 20. This is because variable values are not updated after each assignment, but at the end of the assignment block, and so "x" had not yet been updated when the assignment "$\mathrm{y:=x+10;}$" was performed. The previous situation also caused the initial assignments not to support the use of variables on the right-hand side, even if they had been previously assigned. This situation occurred because variables are recognised/stored at the end of the assignment block, and whenever the variables on the right hand side of the initial assignments were encountered, they had not yet been recognised/stored. As a result, Lince would return an error message indicating the presence of this variables, which might not have been expected by the user.

- **Incomplete parser of replies from SageMath**– SageMath does not always find a (simple) solution, sometimes it produces replies that Lince's parser cannot handle;

- **Error messages** – In general, the error messages produced by Lince are too vague and not useful for the user;

- **SageMath limitations**– The way in which Lince is implemented and/or the existence of SageMath limitations entails the existence of programs that cannot properly yield a symbolic plot;

The scope of this dissertation is essentially to try to overcome the previous restrictions in order to allow the creation of a simulator of hybrid programs that is better adapted to dealing with programs that interact with physical processes regulated by Newtonian mechanics.

2.4 Scala overview

To develop the Lince, its creators used the programming language Scala. In the following subsections, the key concepts of Scala will be explained, especially those that were crucial not only to develop Lince but also to modify and add new functionalities to the tool in this dissertation.

2.4.1 What is the Scala language?

As described in [tut], Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant and safe way, smoothly integrating the features of object-oriented and functional languages. This language presents certain characteristics that make it different from most conventional programming languages, such as Python, some of which were mentioned in the previous reference and will be mentioned here [tut, Overview]:

- **Scala is object-oriented** - Scala is an object-oriented language, where the types and behaviour of objects are defined by classes;
- **Scala is functional** - In a functional language a function is a value, and since in Scala a value is an object, then a function is an object;
- **Scala is statically typed** - Unlike other languages, in Scala it is not obligatory to assign the data type in the vast majority of cases – the compiler infers the type based on the data assigned;
- **Scala can execute Java Code** - Scala can be compiled into Java Byte Code which is run by the Java Virtual Machine (JVM). Futhermore Scala can import existing Java libraries;
- **Scala can compile to JavaScript** - Scala can also be compiled into JavaScript using the Scala.js plug-in [ALRF+22]. The compiled JavaScript file can be included in an HTML file and be executed in any modern internet browser.

2.4.2 Scala's most important instructions and features

Let us now discuss some of the most relevant Scala instructions and features in the context of this dissertation.

As typical features of most programming languages, Scala allows:

- To declare **mutable** and **immutable variables** (see Appendix B.3);
- To use **data types** "Int", "Float", etc (see Appendix B.3);
- To use common **operators** such as "+", "<", ">", etc (see Appendix B.4);
- To perform "While", "Do-While" and "For" **loops**, as well as the **conditional structure** "if-else" and the declaration of **functions** (see Appendix B.5);
- To work with **strings** and **arrays** (see Appendix B.6).

In addition to these "standard" features, Scala is an **object-oriented language**, where the types and behavior of **objects** are defined by **classes**, and a class can **inherit** from another class (see Appendix B.7).

2.4.3 Abstract data types in Scala

We will now focus on some concepts involving abstract datatypes in Scala. Due to the the nature of this dissertation we will only give a very brief overview. The curious reader is invited to consult Appendix B.7 to gain a better understanding of some concepts discussed in this section.

In Scala there are four types of classes that are important to deal with in the scope of our project, these being **abstract**, **trait**, **case** and **sealed** classes.

Based on the code excerpt from Fig. 8 from the "Syntax.scala" file (one of the files responsible for making the Lince *parser*), we can see that it uses at least three of the classes we intend to deal with. The class "LoopGuard" (corresponding to the loop guards) is of abstract type and of sealed type. In the context of this project, abstract type means that this class doesn't need to have its body defined and can never be instantiated [Gee19a], whereas sealed type means that this class can only have heirs in that file and as a consequence Scala's type system is able to give warnings when finding uncovered cases in case analysis [oS22].

```
sealed abstract class LoopGuard
case class Counter(i:Int) extends LoopGuard
case class Guard(c:Cond) extends LoopGuard
```

Figure 8: *"LoopGuard" sealed abstract class*

The classes "Counter" (which corresponds to guards consisting only of integers) and "Guard" (which corresponds to guards consisting of Boolean conditions) are all inherited from the "LoopGuard" class and are of the case type. In the context of this project, the case type means that classes don't need the "new" keyword to be instantiated, and more importantly, it allows for *pattern matching* [Gee19c]. *Pattern matching* happens for example in this code excerpt from Fig. 9 from the "Utils.scala" file (which contains auxiliary functions used in the implementation of Lince).

```
def getVars(guard: LoopGuard): Set[String] = guard match {
  case Counter(_) => Set()
  case Guard(c) => getVars(c)
}
```

Figure 9: *Function "getVars"*

It takes the argument "guard" (which is of type "LoopGuard") and returns the set of variables present in it, checking first whether the argument is a "Counter" case class (a "Counter" is associated with an integer, so an empty set is returned) or a "Guard" case class (a "Guard" is associated with a condition, so the "getVars" function is used to return the set of variables present in it), this verification method is only possible because these two classes are of case type.

As for the trait type, this is also used in the development of the Lince language, and this type is a variation of the abstract type with support for multiple inheritance (class that inherits from more than one class) [oS23].

2.4.4 Parsing

Following Gabriele Tomassetti [Str22], parsing is defined as: "The analysis of an input to arrange the data according to the rule of a grammar". In other words, a *parser* is used for encoding input data into a structured data type (*i.e.* based on a grammar). Programming languages are a good example of this, as programs are written in a file and understood by humans, and need to be converted into a data structure to be understood by computers and thus analysed and executed.

A very important component in parsing is *regular expressions*, which according to Gabriele Tomassetti [Str22] are defined as: "A sequence of characters that can be defined by a pattern". *Regular expressions* are widely used in lexers, becoming a very powerful tool, because a pattern of characters can identify a set of characters/words in which we are interested. For more detailed information about *regular expressions*, see the reference [Doc22].

Looking at the structure of the *parser*. It generally consists of a *lexer*, which is responsible for converting the input text into a set of *tokens*, and the *parser* itself, which takes the set of *tokens* and performs their syntactic analysis (basically, checks whether they respect the grammar established in the *parser*), returning an organised data structure that will later be processed by the *semantic analyser* to check for semantic errors and process the results [Str22].

Apart from this way of parsing, where one separates the *lexer* from the *parser*, there are situations where that is not so easy and so the *parser* ends up doing both at the same time, keeping a grammar identical to what it would be if it had the *lexer* separated [Str22].

In addition to everything seen in the previous sections, many approaches and libraries to implement *parsers* exist in Scala. A common approach, followed *e.g.* by ANTLR [Ter13], is to describe the grammar of the input language in a dedicated language and to generate code automatically that converts text into intermediate data structures. An alternative approach, followed by Lince and used in this project, is to use so-called *parsing combinators*. These are popular in functional programming languages, where *parsers* are built with the help of combinators that, given a sequence of *parsers*, produce a new *parser*. This will be further explain below. This approach facilitates the compilation of Scala to both Java and to JavaScript, since it does not rely on code generators.

Pedro Palma Ramos [Ram16] describes in detail how to use the *parser combinators* to create a *lexer* and a *parser* to make a workflow. This reference is recommended to be read so that the reader can gain a better understanding on how to create a separate *lexer* and *parser* using *parser combinators*. However, the parsing done in this project was made differently from the parsing described in reference [Ram16], due to the fact that the *lexer* is created not separately but together with the *parser*, among other changes. We thus choose to describe how this concrete parsing works and let the reader investigate how other parsing methods can be performed in Scala.

Initially, a file called "Syntax.scala" was created, which contains a set of classes that will serve as the *parser's* output. The main sealed abstract class is called "Syntax" and is associated to a program. From it four inheritance case classes were created: "Atomic" (associated to assignments and differential equations), "Seq" (associated to the composition of programs), "ITE" (associated to the conditional structure

"if-then-else") and "While" (associated to the cyclic structures "while" and "repeat"). These case classes take arguments, for example the "While" case class takes three arguments. The first is of type "Syntax" (program before the "while" loop), the second is of type "LoopGuard" (corresponds to the "while" loop guard and its implementation was shown in Fig. 8) and the third is of type "Syntax" (corresponds to the program in the body of the "while" loop). To elaborate the data types that are present in the arguments of the previous classes, it was necessary to create certain case classes and sealed abstract classes, and even after elaborating them, it may or may not be necessary to create classes that define the data types of the arguments of these new classes.

Next, the file "Parser.scala" was created. It contains the instructions for performing the parsing operations. This file begins by creating an object with the same name as the file, which inherits from "RegexParsers" to allow the use of *regular expressions*. Next, a function called "parse" is created, which receives as argument a string representing the input text, and performs the parsing of that string using the "parseAll" instruction (present in the library of *parser combinators*). This instruction only needs to receive two arguments, one is the input text, and the other is the variable "progP". The variable "progP" uses the *parser combinators* to verify whether the input text complies with the grammar intended for the Lince language and to return the corresponding parsing. This variable can be found right at the beginning of the grammar diagram in Appendix B.1, and one can see that it can only be the variable designated by "seqP". The "seqP" variable checks if the input text complies with the grammar of a "while" loop, a "repeat" loop, an atomic instruction[3] , an "if-then-else" condition and a "wait" command (via the "basicProg" variable), or any combination of these and return the corresponding parsing. To do this, this variable either returns only the "basicProg" variable, or uses recursion and returns the "basicProg" variable followed by "seqP" itself. The "basicProg" variable, as mentioned earlier, is associated with one of the aforementioned instructions and therefore requires the use of other variables, such as "whileGuard" (is responsible for parsing the guards of the "while" loops) and "linP" (is responsible for parsing the linear expressions). By creating and associating the variables in this file, the grammar of Lince was developed (the complete grammar of the old version of Lince can be found in Appendix B.1).

To describe how these variables perform parsing of the produced text, based on the *parser combinators* provided by Scala, it was decided to extract the "whileGuard" and "durP" variables from the "Parser.scala" file. The Fig. 10 contains the code excerpt that implements these two variables.

The "whileGuard" variable, as mentioned earlier, is responsible for parsing the guards of the "while"

[3] An atomic instruction consists of the most elementary instruction supported by Lince, which is a list of assignments and a list of differential equations with or without duration.

```
lazy val whileGuard: Parser[LoopGuard] = {
    condP ^^ Guard |
    intPP ^^ Counter
  }

lazy val durP: Parser[Dur] =
    "until" ~ opt(untilArgs) ~ condP ^^ {
      case _ ~ None ~ cond => Until(cond,None,None)
      case _ ~ Some(args) ~ cond => Until(cond,Some(args._1),args._2)
    } |
    "for" ~> linP ^^ For
```

Figure 10: *Variable "*whileGuard*" and variable "*durP*"*

loops, while the "durP" variable is responsible for parsing the durations (see the grammar in Appendix B.1 to understand where these variables fit in) .

Regarding the "whileGuard" variable, it has the possibility of receiving the "condP" variable (responsible for returning the parsing of conditions) or the "intPP" variable (responsible for returning the parsing of integers), this possibility of choice of more than one input parsing is due to the use of the "|" combinator (the combinators covered here come from the *parser combinators* library). If it receives the "condP" variable then the "Guard" case class will have to be returned, however if it receives the "intPP" variable it will have to return the "Counter" case class (these two case classes were covered in the Section 2.4.3). The combinator responsible for modifying the value returned by a parser is the "^^" combinator.

As for the variable "durP", it is only important to mention the other combinators that are used in its implementation, covering all the combinators used in parsing development, namely: the '~" combinator, which serves to compose parsers sequentially; the "~>" combinator, which only considers the parsing that this combinator points to; the "opt(...)" combinator, which checks whether the parsing procedure in its argument was successful or not, returning "Some(...)" (together with the corresponding result) if it was and "None" if it wasn't. The parser combinators library also provides the ability to create basic parsers for keywords (such as the "until" instruction on the variable "durP") and for regular expressions.

After obtaining the result of parsing the input text, the semantic analyzer will take this result and check for semantic errors, calculate the values assigned to variables and determine the solutions of differential equations (using SageMath), among other things.

Part II

Core of the Dissertation

Chapter 3

Lince and Newtonian Systems

In the simulation of hybrid programs regulated by Newtonian mechanics, particularly in the field of kinematics, it is of utmost importance to understand the differential equations used to simulate the temporal evolution of the variables involved. These equations establish fundamental relationships between the positions, velocities, and accelerations of moving bodies, enabling us to predict and describe their behavior over time. As referred in [Pla18a], the basic system of differential equations for 1D motion is as follows:

$$x' = v, v' = a \tag{3.1}$$

These differential equations tell that the rate of change of position (x) is velocity (v), and the rate of change of velocity (v) is acceleration (a). Given the initial conditions of the body, *i.e.* initial position and initial velocity, we obtain solutions to the differential equations (3.1) which characterize the motion of the body in 1D. An example in which the aforementioned system of differential equations is used is the simulation of programs that aim to manage a train's motion. Since trains move exclusively on a track, it is often sufficient to describe its motion in 1D.

The system of differential equations can be reformulated to describe the motion of a body in 2D and 3D. Based on reference [Pla18b], the system of differential equations for 2D motion is represented as follows:

$$x' = v, y' = u, v' = \omega * u, u' = -\omega * v \tag{3.2}$$

The body is located at position (x, y) and moves in the direction (v, u), which means that the body moves along the x-axis with a velocity v and along the y-axis with a velocity u. The direction of the body can vary, curving to the right (if ω is greater than zero), curving to the left (if ω is less than zero), or moving in a straight line with a constant velocity (if ω is equal to zero). The value of ω represents the angular velocity of the body, and thus, a larger value of ω leads to faster and tighter curves.

Additionally, the body can be accelerated with a constant acceleration in each of its axes. This can be achieved by modifying the previous system of equations so that the derivative of v and the derivative of w are both equal to a real number.

By understanding how to make the body curve in different directions and move in a straight line, it becomes possible to simulate the motion of a body in 2D. An example where the 2D systems of differential equations can be useful is in the simulation of vehicle/drone/missile motion in a plane.

On the other hand, the system of differential equations for 2D motion can be modified to represent the motion of a body in 3D:

$$x' = v, y' = w, z' = k, v' = \omega * w, w' = -\omega * v, k' = c \tag{3.3}$$

Depending on the value of ω, this system of differential equations can simulate the motion of a body that curves to the left, curves to the right, or moves in a straight line at a constant velocity in the x-y plane. Nevertheless, the z-component moves with acceleration c.

By using this system of differential equations, it is possible to simulate the motion of a body that is free to move in the plane x-y and to vary its height. This 3D system of differential equations can be useful in the simulation of the motion of airplanes and submarines.

With the use of these differential equations, the implementation of hybrid programs regulated by Newtonian mechanics in Lince should become feasible. However, to simulate more complex programs, such as adaptive cruise control (ACC), it was necessary to perform discrete calculations that required certain operations not supported by the tool. Besides this, the old version of Lince was unable to execute the differential equations of 2D and 3D motion for more than a very limited number of cycles, which restricted its use in simulating programs involving more than one dimension. Going into more detail, the main limitations that hinder the simulation of a wide range of hybrid programs regulated by Newtonian mechanics are:

- Lack of arithmetic operators and mathematical functions;
- Rigid syntax;
- Limitations in executing certain differential equations.

In the old version of Lince, linear expressions only supported the arithmetic operators of addition, subtraction, and multiplication. More specifically, the syntax of linear expressions was defined as follows[1]:

[1] The following syntax representation is analogous to the *Backus-Naur form*; a more detailed explanation of this type of representation can be found in reference [MR03].

$$
\begin{aligned}
\text{linP} =& \text{linParcelP} \\
&| \text{linParcelP “+” linP} \\
&| \text{linParcelP “-” linP} \\
\text{linParcelP} =& \text{“-” linMultP} \\
&| \text{linMultP} \\
\text{linMultP} =& \text{ realP} \\
&| \text{realP “*” linAtP} \\
&| \text{linAtP} \\
&| \text{linAtP “*” realP} \\
\text{linAtP} =& \text{identifier} \\
&| \text{“(” linP “)”}
\end{aligned}
$$

where “identifier” and “realP” are defined by the following regular expressions:

$$
\begin{aligned}
\text{identifier} =& [a-z][a-zA-Z0-9\_]* \\
\text{realP} =& -?([0-9]+)((.[0-9]+))?
\end{aligned}
$$

(**Note:** The complete syntax representation of the old version of Lince is available in Appendix B.1)

With such a syntax, limitations arose in modelling some hybrid programs, namely those that require the use of arithmetic operators and mathematical functions such as:

- The **square root** function, used, for example, to calculate the Euclidean distance between two points;
- The **power** function, used for example, to calculate the discriminating binomial in the example in Section 2.3.3;
- The **division** operator, used, for example, to calculate the average speed;
- **Trignometrical** functions, used, for example, to calculate the angle between vectors;
- The **remainder** operator, used, for example, to check whether a value is even or odd.

Besides the lack of these operations, there are also hybrid programs that need the **Pi number** (to calculate, for example, the angular velocity of a body) and the **Euler number** (used, for example, in the voltage response of an RC circuit) and due to the absence of mathematical constants that define these numbers with good accuracy, approximate real numbers were used (a less precise alternative).

We also verified that even after adding new arithemetic operators, mathematical functions and mathematical constants, the linearity restriction greatly limited the specification of assignments, differential equations, timing constraints and Boolean conditions. For example, it does not allow the multiplication of two variables and other non-linear operations that are very useful in the development of hybrid programs.

Regarding the limitations of running certain differential equations, it has been observed that the way the tool has been developed sometimes causes the symbolic result of variables to grow as the system of differential equations is iterated within the loop. This growth causes the symbolic result to increase without simplification, and sometimes this growth reaches a point where it becomes impractical to obtain the next solution of the differential equations using SageMath.

Based on the previous limitations and some others mentioned in Section 2.3.4, the necessary code files were modified to mitigate these issues, and some additional features were added to relax, organise and enrich the language, improving the user experience. Overall the following improvements have been made:

- The Lince now supports atomic instructions that receive only a single assignment instead of a list/block of assignments, ensuring that variable assignments are performed sequentially/atomically and not at the end of a block of assignments;
- The initial assignments of the variables are now required to be at the beginning;
- The arithmetic operators from Table 1, the mathematical functions from Table 2 and the mathematical constants from Table 3 are now supported (on the right is the respective instruction in Lince);

Operator	Instruction
Division	/
Multiplication	*
Addition	+
Subtraction	-
Remainder	%

Table 1: *Arithmetic operators supported by Lince*

- The tool now supports fully relaxed non-linear expressions;
- The parser of relational expressions has been changed to allow the use of a non-linear expression to the left and right of the relational operator;

Function	Instruction	Function	Instruction	Function	Instruction	Function	Instruction
Power	pow(..) or ^	**Minimum**	min(...,...)	**Arcsine**	arcsin(...)	**Hiperbolic cosine**	cosh(...)
Square root	sqrt(..)	**Sine**	sin(...)	**Arccosine**	arccos(...)	**Hiperbolic tangent**	tanh(...)
Exponen-tiation	exp(...)	**Cosine**	cos(...)	**Arctangent**	arctan(...)	**Logarithm**	log(...)
Maximum	max(...,...)	**Tangent**	tan(...)	**Hyperbolic sine**	sinh(...)	**Base-10 logarithm**	log10(...)

Table 2: *Mathematical functions supported by Lince*

Constant	Instruction
Pi	pi()
Euler	e()

Table 3: *Mathematical constants supported by Lince*

- The parser for assignments and timing constraints was changed to support non-linear expressions instead of only linear ones;

- The parser of differential equations has been modified to support non-linear expressions. Although the differential equations in Lince need to be of a linear nature, relaxing the grammar of differential equations to support non-linear expressions, rather than restricting it to linear expressions (as was done in the old version of Lince), allows for checking the linearity of differential equations in the interpreter and performing pre-processing on the differential equations to better determine the linearity. An example of the advantages of this strategy is the following program written in the Lince tool:

```
x:=1;
y:=0;
x'=x^y for 1;
```

If the parser only supported linear expressions in differential equations, the expression "x^y" would not conform to the grammar of differential equations, because raising one variable to another without knowing to which expression they are associated, it ends up being considered a non-linear operation. However, the new version of Lince has a parser that supports non-linear expressions in differential equations, allowing the expression "x^y" to conform to the grammar of differential equations. The main advantage of this method is the pre-processing performed on this expression,

which allows the interpreter to check the linearity of the differential equation. It will detect that "y" is a constant variable equal to 0, which is equivalent to changing the expression to "$\mathrm{x\widehat{}0}$" , which is exactly equal to 1. Therefore, the differential equation has a linear character and can be sent to SageMath;

- The interpreter is able to check whether the variables present in the expressions of the differential equations are constants (do not change during the execution of the differential equation) or dynamic (vary during the execution of the differential equation), replacing the constant variables with their respective expressions. This advantage is seen in the example from the previous point;
- The semantic analyser (interpreter) is now capable of:
  - Verifying whether the variables on the right hand side of the initial assignments were previously assigned;
  - Verifying the existence of variables that were not initially assigned;
  - Verifying whether there are mathematical functions and mathematical constants not supported and whether the corresponding number of arguments is correct;
  - Verifying the existence of some indeterminate forms, like for example "$5/0$";
  - Verifying the existence of inconsistent results in operations;
  - Verifying if the differential equations are linear;
  - Detect SageMath's inability to solve certain differential equations.
- Implementation of a numerical alternative to the symbolic approach adopted by Lince, preventing the excessive growth of symbolic solutions for differential equations.

In the following chapters, the limitations of the old version of Lince will be presented in more detail, along with the procedures adopted to mitigate these limitations.

Chapter 4

Extending Lince's Language

In the next sections, the changes made to the files that compose Lince will be presented in order to implement the enhancements mentioned in Chapter 3. The most important functions that have been modified or added will be showcased, along with a detailed explanation of their functionality and the reasons behind their modification or creation. Each function will be analyzed in terms of how it contributes to improve the user experience and overcoming the limitations identified in the previous chapter.

In addition, there will be presented examples that highlight the added value of each improvement to the Lince tool. These examples will demonstrate how the changes made in the functions improve the tool's ability to handle specific problems and to perform simulations of higher complexity and with greater comfort in their implementations.

Thus, the upcoming chapters will provide a comprehensive guide to the alterations made in Lince, allowing for a deep understanding of the implemented enhancements and underscoring their relevance in simulating hybrid programs based on Newtonian mechanics.

(**Note:** To consult all the files of the new version of the Lince just consult https://github.com/arcalab/lince/blob/master/src/main/scala/hprog.)

4.1 Modifications to the syntax of hybrid programs in Lince

In this section we will discuss the development of the grammar for the new version of Lince. We will start by discussing the changes made to the data structures returned by the parser. These changes include, for example, the inhibition of grouping data associated with assignments into a single atomic instruction, and the introduction of a new data structure for non-linear expressions. In addition, we will discuss the changes made to the parser that led to this new grammar, such as the recognition of non-linear expressions.

4.1.1 Updated data structures

To update the data structures returned by the parser, we need to modify the "Syntax.scala" file. This file, as mentioned in Section 2.4.4, is essentially made up of a set of classes of the case type that will serve as output from the parser, to be later processed by the interpreter (which performs semantic operations, iterating with SageMath and the Inspector, as mentioned in Section 2.3).

As such, it was necessary:

- To implement data structures referring to non-linear expressions;
- Change the data structures of assignments, relational expressions, durations and differential equations in order to admit non-linear equations;
- Ensure that pre-processing functions do not group a sequence of assignments into a single atomic instruction.

To implement the data structures referring to non-linear expressions it was necessary to alter the abstract class for linear expressions, changing its name to "Notlin" and adding inherited classes for the remaining arithmetic operators, mathematical functions and mathematical constants planned to add. The code excerpts in Fig. 11 represent the data structure of linear expressions (left), and the implementation of the data structure of non-linear expressions (right).

There are now a case class (which in other words are the data structures that will be returned by the parser) directed to a variable ("Var"), a real number ("Value"), addition of two non-linear expressions ("Add"), multiplication of two non-linear expressions ("Mult"), division of two non-linear expressions ("Div"), remainder of two non-linear expressions ("Res") and a case class "Func" directed to the mathematical functions and mathematical constants that we intend to add (see Table 2 and Table 3 of the Chapter 3).

To enable the data structures associated with assignments, relational expressions, durations and differential equations to support non-linear expressions rather than linear ones, the data type of the arguments associated with linear expressions within these case classes has been converted to the corresponding data type for non-linear expressions.

In the case of assignments, their data structure has been modified as illustrated in the code excerpt in Fig. 12 (on the left is the data structure of the assignments of the old version, on the right is the data structure of the assignments of the new version).

```
// Original data structure
sealed abstract class Lin {
  def +(other: Lin):Lin = Add(this,other)
}
case class Var(v: String) extends Lin {
  def :=(l: Lin): Assign = Assign(this,l)
  def ^=(l: Lin): DiffEq = DiffEq(this,l)
  def >(l: Lin): Cond = GT(this,l)
  def <(l: Lin): Cond = LT(this,l)
  def >=(l: Lin): Cond = GE(this,l)
  def <=(l: Lin): Cond = LE(this,l)
  def ===(l: Lin): Cond = EQ(this,l)
}
case class Value(v: Double)
extends Lin {
  def *(l: Lin): Lin = Mult(this,l)
}
case class Add(l1: Lin,l2: Lin)
extends Lin

case class Mult(v: Value,l: Lin)
extends Lin
```

```
// Updated data structure
sealed abstract class NotLin {
  def +(other:NotLin): NotLin = Add(this,other)
}
case class Var(v:String) extends NotLin {
  def :=(l: NotLin): Assign = Assign(this,l)
  def ^=(l: NotLin): DiffEq = DiffEq(this,l)
  def >(l: NotLin): Cond = GT(this,l)
  def <(l: NotLin): Cond = LT(this,l)
  def >=(l: NotLin): Cond = GE(this,l)
  def <=(l: NotLin): Cond = LE(this,l)
  def ===(l: NotLin):Cond = EQ(this,l)
}
case class Value(v:Double)
extends NotLin {
  def *(l: NotLin): NotLin = Mult(this,l)
}
case class Add(l1:NotLin,l2:NotLin)
extends NotLin

case class Mult(l1:NotLin,l2:NotLin)
extends NotLin

case class Div(l1:NotLin,l2:NotLin)
extends NotLin

case class Res(l1:NotLin,l2:NotLin)
extends NotLin

case class Func(s:String, arg:List[NotLin])
extends NotLin
```

Figure 11: *Data structure of linear and non-linear expressions*

```
// Original data structure
case class Assign(v:Var,e:Lin)
```

```
// Updated data structure
case class Assign(v:Var,e:NotLin)
```

Figure 12: *Data structure of assignments in both versions of Lince*

For relational expressions, their data structure has been changed as illustrated in the code excerpt in Fig. 13 (on the left is the data structure of the relational expressions of the old version, on the right is the data structure of the relational expressions of the new version).

(**Note**: The data types of the arguments in each case class of the relational expressions were both

```
// Original data structure
case class EQ(l1:Var,l2:Lin)
    extends Cond
case class GT(l1:Var,l2:Lin)
    extends Cond
case class LT(l1:Var,l2:Lin)
    extends Cond
case class GE(l1:Var,l2:Lin)
    extends Cond
case class LE(l1:Var,l2:Lin)
    extends Cond
```

```
// Updated data structure
case class EQ(l1:NotLin,l2:NotLin)
    extends Cond
case class GT(l1:NotLin,l2:NotLin)
    extends Cond
case class LT(l1:NotLin,l2:NotLin)
    extends Cond
case class GE(l1:NotLin,l2:NotLin)
    extends Cond
case class LE(l1:NotLin,l2:NotLin)
    extends Cond
```

Figure 13: *Data structure of the relational expressions in both versions of Lince*

converted to type "NotLin" (previously "l1" was of type "Var" and "l2" was of type "Lin") so that the relational expressions would allow relating two non-linear expressions instead of relating a variable to a linear expression, thus giving greater power and flexibility in the writing of relational expressions.)

As for durations and differential equations, the code excerpts in Fig. 14 represent the data structures of durantions and differential equations of the old version (left) and the data structures of durations and differential equations of the new version (right).

```
// Original data structure

// Differential equation
case class DiffEq(v:Var,e:Lin)

// Duration
sealed abstract class Dur
case class  For(e:Lin)  extends Dur
case class  Until(c:Cond,
eps:Option[Double],jump:Option[Double])
extends Dur
case object Forever extends Dur
```

```
// Updated data structure

// Differential equation
case class DiffEq(v:Var,e:NotLin)

// Duration
sealed abstract class Dur
case class  For(e:NotLin) extends Dur
case class  Until(c:Cond,
eps:Option[Double],jump:Option[Double])
extends Dur
case object Forever extends Dur
```

Figure 14: *Data structures of the differential equations and duration in both versions of Lince*

Finally, to ensure that the pre-processing functions did not combine sequences of assignments into a single atomic instruction[1], it was necessary to eliminate the third case from the method "$\sim$" within the "Atomic" case class, as shown in the code excerpt in Fig. 15. The third case in the method referenced in Fig. 15 was triggered when the method was applied between an atomic instruction containing only

[1] An atomic instruction consists of the most elementary instruction supported by Lince, which is a list of assignments and a list of differential equations with or without duration.

```
case class Atomic(as:List[Assign],de:DiffEqs)  extends Syntax {
  override def ~(p:Syntax): Syntax = (de.dur,p) match {
    case (_,Seq(p,q)) => Seq(this~p,q)
    case (_,While(pre, d, doP)) => While(this~pre,d,doP)
    /**case (For(Value(0)),Atomic(as2,de))=>Atomic(as++as2,de)*/
    case (_,_) => Seq(this,p)
  }
}
```

Figure 15: *Data structure of an atomic instruction and its corresponding method in the new version of Lince*

assignments (resulting in a zero duration for the differential equations) and another atomic instruction. Since the continuous component of the first atomic instruction did not exist or did not evolve in time, then it was only necessary to take into account the discrete component, which was the assignments, and join them with the discrete component of the second atomic to form a single atomic instruction. However, the interpreter was designed to update the value of the variables at the end of each atomic instruction, and if that atomic has a list/block of assignments associated with it, unexpected results can happen, as in the example referred to in Section 2.3.4 in the topic "Unintuitive atomic assignments". As such, it was necessary to remove this case so that the grouping of assignments would no longer happen, ensuring that atomic instructions are associated with only one assignment and that variable assignments are performed sequentially/atomically.

4.1.2 Updated parser

After creating a data structure for the non-linear expressions, compatibilizing the remaining data structures to handle these expressions and avoiding grouping assignments into an atomic instruction, it was necessary to modify the file "Parser.scala". This file is responsible for parsing, according to a certain grammar, the hybrid program developed in the Lince tool.

The changes made in this file were intended to modify the grammar and the output coming from the parser, so that the hybrid programs:

- Start with at least one atomic instruction to ensure that the initial assignments are made at the beginning of the program;
- Recognise non-linear expressions;
- Associate durations, assignments and differential equations with non-linear expressions;

- Have a more relaxed grammar of relational expressions, being able to relate a non-linear expression to another non-linear expression.

In order for the language grammar to require the initial assignments to be made at the beginning of the hybrid program, the changes illustrated in Fig. 16 have been implemented (on the left is the implementation of the old version and on the right is the implementation of the new version).

```
lazy val progP: Parser[Syntax] =
  seqP ^^ { stx =>
    Utils.isClosed(stx) match {
      case Left(msg) =>
      throw new ParserException(msg)
      case Right(_) => stx
    }
  }

lazy val seqP: Parser[Syntax] =
  basicProg ~ opt(seqP) ^^ {
    case p1 ~ Some(p2) => p1 ~ p2
    case p ~ None => p
  }
```

```
lazy val progP: Parser[Syntax] =
    declr ^^ { stx =>
      Utils.isClosed(stx) match {
        case Left(msg) =>
        throw new ParserException(msg)
        case Right(_) => stx
      }
    }
lazy val declr: Parser[Syntax] =
    atomP ~ opt(seqP) ^^ {
      case a ~ Some(n) => a ~ n
      case a ~ None => a
    }
lazy val seqP: Parser[Syntax] =
    basicProg ~ opt(seqP) ^^ {
      case p1 ~ Some(p2) => p1 ~ p2
      case p ~ None => p
    }
```

Figure 16: *Change made in the parser to require the hybrid program to begin with an atomic instruction*

The "progP" variable is meant to hold the parsing result (coming from the variable "seqP" in the old version and the variable "declr" in the new version), evaluate if there are certain semantic errors (by the "Utils.isClosed" function) and if the parsing contains one of these semantic errors, return it, if not, return the parsing result (we will deal with the detection of these semantic errors in the Section 4.5.1 and the Section 4.5.2). As for the variable "seqP", it expects to receive the parsing of one instruction (it may come from a "while" loop, a "repeat" loop, a "skip" instruction, a "wait" instruction, an "if-then-else" or an atomic instruction) coming from the variable "basicProg", or the composition of the parsings of more than one instruction. However, the variable "declr" was implemented in the new version and it was interspersed between the variable "progP" and the variable "seqP". This variable expects to receive the parsing of an atomic instruction (assignment and/or differential equations) or the composition of the parsing of an atomic instruction with the result of the variable "seqP", thus guaranteeing a grammar that requires at least one atomic instruction before starting other instructions.

To exemplify, consider the following hybrid program in Lince:

```
x:=1;
y:=10;
if (x<=y)
then {x:=x+1;}
else {x:=x+10;}
```

If we run the previous hybrid program on each of the Lince versions and extract the output of the parser for each of them, we get the results shown in Fig. 17.

```
Seq(Atomic(List(Assign(Var(x),Value(1.0)),
      Assign(Var(y),Value(10.0))),
      DiffEqs(List(),For(Value(0.0)))),
    ITE(LE(Var(x),Var(y)),
      Atomic(List(Assign(Var(x),
       Add(Var(x),Value(1.0)))),
       DiffEqs(List(),For(Value(0.0)))),
      Atomic(List(Assign(Var(x),
       Add(Var(x),Value(10.0)))),
       DiffEqs(List(),For(Value(0.0))))))
```

```
Seq(Seq(Atomic(List(Assign(Var(_x),Value(1.0))),
      DiffEqs(List(),For(Value(0.0)))),
     Atomic(List(Assign(Var(_y),Value(10.0))),
      DiffEqs(List(),For(Value(0.0))))),
   ITE(LE(Var(_x),Var(_y)),
    Atomic(List(Assign(Var(_x),
     Add(Var(_x),Value(1.0)))),
     DiffEqs(List(),For(Value(0.0)))),
    Atomic(List(Assign(Var(_x),
     Add(Var(_x),Value(10.0)))),
     DiffEqs(List(),For(Value(0.0))))))
```

Figure 17: *Parser output from the old version (left) and the new version (right)*

We can verify in Fig. 17 that the parser output in both versions only differs in the fact that the new version returns an atomic instructon for each assignment (so that the updating of the variables would not be done in a block, as seen in Section 4.1.1) and that the name of the variables is preceded by the character "_" (the reason of the use of this character will be explain in the Section 4.2).

Nevertheless, changing the previous example in order to remove the initial assignments (but without compromising the objective of the program):

```
if true
then {x:=1+1;}
else {x:=1+10;}
```

The results shown in Fig. 18 are obtained for each of the Lince versions.

We can verify in Fig. 18 that the new version returns an error message because, as already mentioned, it requires that the program always start with an atomic instruction[2] (whereas the old version does not), and this guarantee offers the following advantages:

[2] The atomic instruction required by Lince's grammar at the beginning of the hybrid program can be an assignment and/or one or more differential equations. However, the interpreter will verify if the variables present in the differential equations have been assigned at the beginning of the program, and will return an error message if the program starts with one or more differential equations because their variables have not been previously assigned.

```
ITE(BVal(true),
    Atomic(List(Assign(Var(x),
      Add(Value(1.0),Value(1.0)))),
      DiffEqs(List(),For(Value(0.0)))),
    Atomic(List(Assign(Var(x),
      Add(Value(1.0),Value(10.0)))),
      DiffEqs(List(),For(Value(0.0)))))
```

```
Error: When parsing G§15
// maximum time in the plot§0§if true
    then {x:=1+1;}
else {x:=1+10;}
- hprog.common.ParserException:
[1.4] failure: ''' expected but 't' found
if true then {x:=1+1;}
```

Figure 18: *Parser output from the old version (left) and error message returned by the new version (right)*

- Makes hybrid programs more grammatically structured;
- Allows the interpreter to more easily determine which assignments correspond to the initial assignments.

Next, the grammar had to be modified to recognize non-linear expressions. To do so, it was necessary to change the variables responsible for recognizing linear expressions (variable "linP" and its complementary variables), so that they would recognize non-linear expressions (variable "$\mathrm{notlinP}$" and its complementary variables).

(**Note:** Due to the dimension of the codes elaborated for both variables, it was decided not to show them in the sequence of the previous paragraph. As such, to see the grammar associated with non-linear expressions you can consult Appendix A.1.)

The grammar of non-linear expressions was designed to admit additions, subtractions, divisions, multiplications and remainders of two expressions, in addition to allowing the negation of an expression.

On the other hand, these expressions can contain pi numbers, Euler numbers, real numbers, variables, non-linear expressions between parentheses and mathematical functions such as exponentiation, power, greater, lesser, square root, logarithm, base-10 logarithm and trignometric functions.

In addition to the large number of operations that can be performed on non-linear expressions, the grammar of these expressions was designed to be as relaxed as possible, avoiding the writing limitations that existed in the grammar of linear expressions in the old version.

With the grammar of non-linear expressions already elaborated, it was necessary to make the expressions of assignments, durations and differential equations associated to non-linear expressions. In each case, it was enough to change the variable that returned the result of the parser of the linear expressions, that is the variable "linP", to the variable that returned the result of the parser of the non-linear expressions, that is the variable "$\mathrm{notLinP}$".

Finally, it was enough to relax the grammar of the relational expressions, making them relate two

non-linear expressions instead of one variable and one linear expression. To do that, the variable "bopP" (responsible for recognizing and returning the parser of booleans and relational expressions) had to be changed as illustrated in the code excerpt in Fig. 19 (on the left is the variable "bopP" in the old version and on the right the variable "bopP" in the new version).

```
lazy val bopP: Parser[Cond] =
    identifier ~ opt(bcontP) ^? ( {
      case "true" ~ None => BVal(true)
      case "false" ~ None => BVal(false)
      case e ~ Some(co) => co(Var(e))
    }: PartialFunction[String ~
    Option[Var=>Cond],Cond],
    {
      case e ~ _ => s"Not a condition: $e"
    })

lazy val bcontP: Parser[Var => Cond] =
    "<=" ~> linP ^^ (e2 => (e1: Var) =>
    e1 <= e2) |
    ">=" ~> linP ^^ (e2 => (e1: Var) =>
    e1 >= e2) |
    "<" ~> linP ^^ (e2 => (e1: Var) =>
    e1 < e2) |
    ">" ~> linP ^^ (e2 => (e1: Var) =>
    e1 > e2) |
    "==" ~> linP ^^ (e2 => (e1: Var) =>
    e1 === e2) |
    "!=" ~> linP ^^ (e2 => (e1: Var) =>
    Not(e1 === e2))
```

```
lazy val bopP: Parser[Cond] =
    "true" ^^ {
      case _ => BVal(true)
    } |
    "false" ^^ {
      case _ => BVal(false)
    } |
    notlinP ~ "<=" ~ notlinP ^^ {
      case l1 ~ _ ~ l2 => LE(l1, l2)
    } |
    notlinP ~ ">=" ~ notlinP ^^ {
      case l1 ~ _ ~ l2 => GE(l1, l2)
    } |
    notlinP ~ "<" ~ notlinP ^^ {
      case l1 ~ _ ~ l2 => LT(l1, l2)
    } |
    notlinP ~ ">" ~ notlinP ^^ {
      case l1 ~ _ ~ l2 => GT(l1, l2)
    } |
    notlinP ~ "==" ~ notlinP ^^ {
      case l1 ~ _ ~ l2 => EQ(l1, l2)
    } |
    notlinP ~ "!=" ~ notlinP ^^ {
      case l1 ~ _ ~ l2 => Not(EQ(l1, l2))
    }
```

Figure 19: *Change made in the parser to make it support relational expressions that relate two non-linear expressions*

(**Note:** The syntax schema of the new version of Lince can be found in Appendix B.2 and the syntax schema of the old version can be found in Appendix B.1)

The following examples show the ability of the new version's grammar to deal with hybrid programs containing non-linear expressions.

If we have the following hybrid program:

```
y:=1;
x:=1;
x:=sqrt(2^2+3^2)*x*y;
x'=sin(pi()/2) for exp(2);
```

The output of the parser in the new version of Lince is shown in Fig. 20.

```
Seq(Seq(Seq(Atomic(List(Assign(Var(_y),Value(1.0))),
                DiffEqs(List(),For(Value(0.0)))),
              Atomic(List(Assign(Var(_x),Value(1.0))),
                DiffEqs(List(),For(Value(0.0))))),
            Atomic(List(Assign(Var(_x),
              Mult(Func(sqrt,List(Add(Func(pow,List(Value(2.0), Value(2.0))),
              Func(pow,List(Value(3.0),Value(2.0)))))),
              Mult(Var(_x),Var(_y))))),
              DiffEqs(List(),For(Value(0.0))))),
          Atomic(List(),
            DiffEqs(List(DiffEq(Var(_x),Func(sin,List(
            Div(Func(PI,List()),Value(2.0)))))),For(Func(exp,List(Value(2.0))))))))
```

Figure 20: *Parser output from the new version of Lince for the previous hybrid program*

It can be seen that the grammar is now able to recognize non-linear expressions (with the possibility of using the operations of the Table 1, the Table 2 and the Table 3 of the Chapter 3) both in assignments, differential equations, and durations.

If we have the following hybrid program:

```
x:=1;
y:=0;
if((x^2+sin(x))<(y^2+cos(y)))
then x:=10;
else x:=-10;
```

The output of the parser in the new version of Lince is shown in Fig. 21.

```
Seq(Seq(Atomic(List(Assign(Var(_x),Value(1.0))),
              DiffEqs(List(),For(Value(0.0)))),
            Atomic(List(Assign(Var(_y),Value(0.0))),
              DiffEqs(List(),For(Value(0.0))))),
          ITE(LT(Add(Func(pow,List(Var(_x),Value(2.0))),Func(sin,List(Var(_x)))),
            Add(Func(pow,List(Var(_y),Value(2.0))),Func(cos,List(Var(_y))))),
            Atomic(List(Assign(Var(_x),Value(10.0))),
              DiffEqs(List(),For(Value(0.0)))),
            Atomic(List(Assign(Var(_x),Value(-10.0))),
              DiffEqs(List(),For(Value(0.0))))))
```

Figure 21: *Parser output from the new version of Lince for the previous hybrid program*

It can be seen that the grammar has also started to recognize relational expressions over non-linear expressions, as intended.

4.2 Adaptation of the interpreter for the treatment of non-linear expressions

Having improved the grammar with the ability to recognise non-linear expressions, the next step was to alter the interpreter so that it would be able to handle non-linear expressions.

As mentioned in Section 2.3, the interpreter is responsible for performing semantic operations. Some of the tasks performed by the interpreter include:

- Simplify symbolically an expression (using Sage);
- Calculate the approximate value of an expression;
- Find the solution of a system of differential equations (using Sage);
- Calculate symbolically the value of the variables at any given point in time (using Sage);
- Calculate approximately the value of the variables at any given point in time;
- Detection of semantic errors.

The interpreter is implemented throughout several files. The main files that constitute the interpreter are:

- Eval.scala - mainly responsible for:
  - Determining the numerical value of non-linear expressions from the parser of the hybrid program and the parser of SageMath;
  - Determining the respective boolean of the conditions boolean;
  - Converting the data type coming from the parser to the data type that is associated with the parsing of the SageMath response;
  - Updating expressions with the respective initial values.
- Distance.scala - mainly responsible for:
  - Finding the closest point that satisfies a given condition;
- Traj.scala - responsible for:

  - Evaluating hybrid programs up to a certain time or number of cycles.

- Utils.scala - file that only contains useful functions to be used in other files, such as:
  - Semantic error detection functions;
  - Variable extraction functions;
  - Etc;

- SageParser.scala - responsible for:
  - Parsing the SageMath response.

- Solver.scala - abstract class with multiple implementations:
  - LiveSageSolver (to call Sage);
  - StaticSageSolver (a layer to cache and recall computations);

In order for the interpreter to acquire the ability to handle non-linear expressions, it was necessary to make (in a wide range of functions of the files that make up the interpreter) the following changes:

- Change the type "Lin" to "NotLin" in function arguments;
- Add the treatment of new cases so that they also treat the new operations available;
- Add new functions;
- Change funtions/interpreter's task handling.

Most of the changes consisted of changing the type "Lin" to type "NotLin" in the arguments of functions so that they now expect to receive non-linear expressions rather than linear ones, and consisted of handling new cases so that the new operations available in non-linear expressions could be handled. To illustrate the two changes mentioned above, consider the "getVars" function in the "Utils.scala" file, as shown in Fig. 22. On the left we have the "getVars" function in the old version, while on the right we can see the "getVars" function in the new version of Lince.

The function of Fig. 22 was intended to extract the list of variables from a linear expression. However, it was necessary to change the data type of the argument so that it would expect to receive a non-linear expression and add the case of being a division, a remainder or a mathematical function/constant, in order to cover all possibilities. It is also important to emphasize the need to create an auxiliary function

```
def getVars(lin: Lin): Set[String] = lin
  match {
    case Var(v) => Set(v)
    case Value(_) => Set()
    case Add(l1, l2) => getVars(l1) ++
    getVars(l2)
    case Mult(_, l) => getVars(l)
  }
```

```
def getVars(notlin: NotLin): Set[String] =
    notlin
  match {
    case Var(v) => Set(v)
    case Value(_) => Set()
    case Add(l1, l2) => getVars(l1) ++
        getVars(l2)
    case Mult(l1, l2) => getVars(l1) ++
        getVars(l2)
    case Div(l1,l2) => getVars(l1) ++
        getVars(l2)
    case Res(l1,l2) => getVars(l1) ++
        getVars(l2)
    case Func(s,list) => getVarsAux(list)
  }
  def getVarsAux(list:List[NotLin]):
      Set[String] = list
  match {
    case List() => Set()
    case n::List() => getVars(n)
    case n::ns => getVars(n) ++ getVarsAux(ns)
  }
```

Figure 22: *Function "getVars" in both versions*

"getVarsAux" that had the objective of extracting the variables from the arguments of the mathematical operations associated with the case class "Func".

The "apply" function in the "Eval.scala" file was another function that had to be made compatible with non-linear expressions. This function received an argument of type "Point" (which is a Map that has the variable names as keys and their numerical values as values) and a linear expression, with the objective of returning the numerical value of this linear expression. Nevertheless, the argument of the function had to be changed to receive a non-linear expression, add the treatment of the missing cases to cover all possibilities, treat some cases for add the check of indeterminate forms and verification of non-supported mathematical operations (this error handling is analysed in the Section 4.5.3).

(**Note:** The code of the function "apply" in the new version of Lince is available in Appendix A.2 due to its high extension.)

Due to the large number of files that make up the interpreter, a large part of the functions needed to be changed, and most of them were compatibility changes as in the previous two examples. As such, each one of those changes will not be detailed in this project, but the reader can consult the https://github.com/arcalab/lince/tree/master/src/main/scala/hprog link to access the files that make up the

interpreter and check how the functions that receive non-linear expressions are implemented.

After modifying the arguments of functions that were expected to receive linear expressions, adding handling for missing cases, and reformulating/creating the necessary functions, the interpreter started to support the use of non-linear expressions based on the new range of mathematical operations. Nevertheless, after conducting some tests and designing the case studies, some bugs were identified in the tool. These bugs were related to the data communication with SageMath, as it was originally designed to handle programs with only linear expressions and a more restricted syntax.

Three of these bugs were related to the lack of robustness in the SageMath parser, the SageMath's inability to solve certain conditional expressions and its inability to handle variables and mathematical operations with the same name in the differential equations.

Regarding the previously mentioned first bug, it was observed that the SageMath parser was designed to handle simple outputs. With the enhancements made to this tool, it became capable of handling more complex case studies, which, in turn, led to more intricate solutions for differential equations. The lack of robustness in the SageMath parser and the requirement to handle these more complex solutions resulted in errors during parsing. Consequently, there was a need to modify the parser to accommodate these solutions.

About the second bug, it was found that the tool returned the error message of the Fig. 23 when trying to execute the following program:

```
v:=0; z:=0; u:=0; i:=0;
if (sqrt((v-u)^2+(z-i)^2)==0 || v==1)
then v'=1 for 2;
else v'=-1 for 2;
```

```
sage reply: 'TypeError: unsupported operand type(s) for |: 'sage.symbolic.expression.
Expression' and 'bool''
```

Figure 23: *Error message returned by Lince*

Upon further analysis of the origin of the error message of the Fig. 23, it was found that a string was sent to SageMath inside a "$\mathrm{bool}()$" function (a function existing in SageMath to calculate the boolean result of a conditional expression), containing the conversion of the conditional expression from the parser to an equivalent conditional expression that SageMath could handle. However, SageMath returned an error when trying to determine the boolean result of applying the "$\mathrm{bool}()$" function to a conditional expression with relational expressions inside it containing non-linear expressions that required a slightly more robust numerical processing. In the previous example, the first relational expression contained roots and expo-

nents that would invalidate the calculation of its respective boolean when sent in the mentioned format. Nevertheless, it was found that the "bool()" function had the capability to resolve more complex relational expressions when applied only to that relational expression. Based on this, the solution was to ensure that, in the sent string, the "bool()" function was applied to all relational expressions present in the conditional expression and also to the entire conditional expression to calculate the resulting boolean.

To perform this process, the function "apply_withbool" was created in the "Show.scala" file and is shown by the code excerpt in Fig. 24. This function is capable of converting the conditional expression from the parser to a string that represents an equivalent conditional expression that can be handled by SageMath but contains the "bool()" function applied to all relational expressions present, as desired.

```
  def apply_withbool(cond: Cond, vl:Valuation = Map()): String = cond match {
    case BVal(b)       => s"bool(${b.toString})"
    case And(And(e1,e2),e3) => apply_withbool(And(e1,And(e2,e3)),vl)
    case And(e1,e2:And)      => s"${showPP(e1,vl)} & ${showPP(e2,vl)}"
    case And(e1, e2) => s"${showPP(e1,vl)} & ${showPP(e2,vl)}"
    case Or(e1, e2)  => s"${showPP(e1,vl)} | ${showPP(e2,vl)}"
    case Not(EQ(l1,l2)) => s"bool(${apply(l1,vl)}!=${apply(l2,vl)})"
    case Not(e1)       => s"bool(!(${showPP(e1,vl)}))"
    case EQ(l1, l2)     => s"bool(${apply(l1,vl)}==${apply(l2,vl)})"
    case GT(l1, l2)     => s"bool(${apply(l1,vl)}>${apply(l2,vl)})"
    case LT(l1, l2)     => s"bool(${apply(l1,vl)}<${apply(l2,vl)})"
    case GE(l1, l2)     => s"bool(${apply(l1,vl)}>=${apply(l2,vl)})"
    case LE(l1, l2)     => s"bool(${apply(l1,vl)}<=${apply(l2,vl)})"
  }

private def showPP(exp:Cond, vl:Valuation):String = exp match {
    case BVal(b) => b.toString
    case _ => s"(${apply_withbool(exp,vl)})"
}
```

Figure 24: *Function "apply_withbool"*

Having created the "apply_withbool" function, it was then incorporated into the "askSage" function of the file "LiveSageSolver.scala", which is responsible for sending the string to SageMath. This change ensured that the string sent to SageMath consisted of a primary function "bool()" which encapsulated the new representation of the conditional expression, with the "bool()" function applied to each of the relational expressions. The function "askSage" is depicted in Fig. 25.

With these changes, this bug was mitigated, and the calculation of conditional expressions became more effective.

Regarding the third bug mentioned earlier, the way that was found to mitigate this problem was to

```
def askSage(c:Cond,vl:Valuation): Option[String] = {
    val instructions =
      "bool(" + Show.apply_withbool(c,vl) + "); \"ok\""
    debug(()=>s"expression to solve: '$instructions'")
    val rep = askSage(instructions)
    debug(()=>s"reply: '$rep'")
    rep
  }
```

Figure 25: *Function "askSage"*

change the output of the parser so that the variable names were preceded by the symbol "_". The reason why variable names are preceded by the character "_" in the parser output is because there were conflicts between the name of the variables and the name of the mathematical functions and mathematical constants in differential equations by SageMath, for example, if a variable was called "pi" and performed the following differential equation "pi'=pi()+pi for 1;", SageMath could not tell which was the variable and which was the mathematical constant pi, returning wrong results or errors. By the use of the character "_" preceding the variable name, it was possible to ensure that SageMath did not confuse variables from mathematical functions and mathematical constants, but it was still necessary to make the functions that worked with the SageMath output compatible so that they knew that variable names were preceded by a "_".

4.3 Constant variables in the differential equations

In this chapter, we will discuss constant variables in differential equations. Section 4.3.1 will illustrate the usefulness of using constant variables in the design of hybrid programs through a practical example. Section 4.3.2 will address the method adopted for implementing this new feature of differential equations.

4.3.1 Motivation for supporting constant variables in differential equations

As mentioned in Section 4.1.2 and Section 4.2, the grammar and the interpreter have been changed so that differential equations can now support non-linear expressions, allowing the user to combine variables, real values, mathematical operations, and other non-linear expressions in a more relaxed and free way.

It was also identified that it would be convenient for the user to be able to use constant variables (variables that do not vary with time during the execution of the differential equation) in differential equations,

without these contributing to non-linearity.

An example that highlights the usefulness of using constant variables is the damped harmonic motion in the underdamping regime. If the Lince did not make it possible to use constant variables without them contributing to nonlinearity, the user would have to model this example as follows [Top23]:

```
x:=2; //Initial position
v:=0; //Initial velocity
x'=v,v'=-x*2.32/1-v*0.6/1 for 20; //Differential equations
```

Although the program is short and easy to design and understand, it lacks information about the terms of the second differential equation, such as $2.32/1$ multiplied by position and $0.6/1$ multiplied by velocity. These two terms are associated with essential physical parameters in the implementation of this program, such as the spring constant and the damping constant, and bring a certain inconvenience to the user having to look for and change, for example, the real number that is associated with the constant damping for this regime to become overdamping.

However, as Lince now deals with constant variables, the previous program can be adapted to the following:

```
m:=1;     // mass of the object
k:=2.32;  // Spring constant
b:=0.6;   // Damping constant
x:=2;     // Initial position
v:=0;     // Initial velocity
x'=v,v'=-x*k/m-v*b/m for 20; //Differential equations
```

The symbolic plot resulting from the previous hybrid program is depicted in Fig. 26.

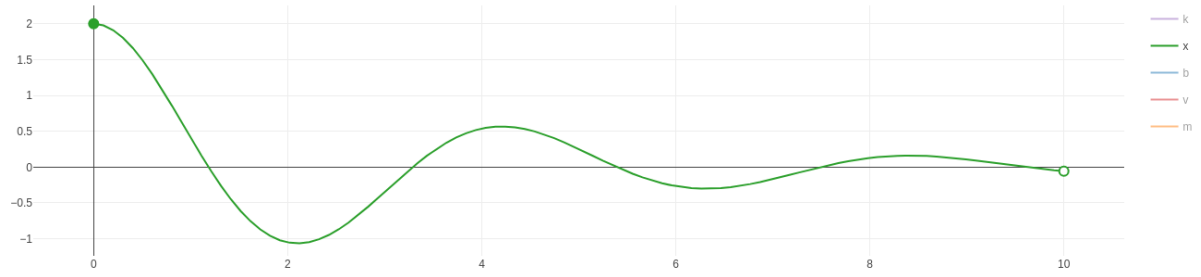


Figure 26: *Symbolic plot of the damped harmonic motion in the underdamping regime*

It is evident that the hybrid program has become more information-rich, notably improved in terms of design, and now facilitates parameter changes in the differential equations. This advantage is particularly evident in more complex programs involving multiple sets of differential equations. Without the use of constant variables, changing parameters in specific differential equations would require navigating through each one individually and adjusting the respective terms. In contrast, the use of constant variables in

differential equations allows users to streamline the process by simply adjusting the assignments of these constants, resulting in automatic updating of the terms within the differential equations.

To exemplify, based on the previous program, the ease that the use of constant variables brought when it comes to changing parameters in the differential equations, it was decided to pass the damped harmonic motion to the overdamping regime:

```
m:=1;     // mass of the object
k:=2.32;  // Spring constant
b:=3.5;   // Damping constant
x:=2;     // Initial position
v:=0;     // Initial velocity
x'=v,v'=-x*k/m-v*b/m for 20; //Differential equations
```

The symbolic plot resulting from the previous hybrid program is depicted in Fig. 27.

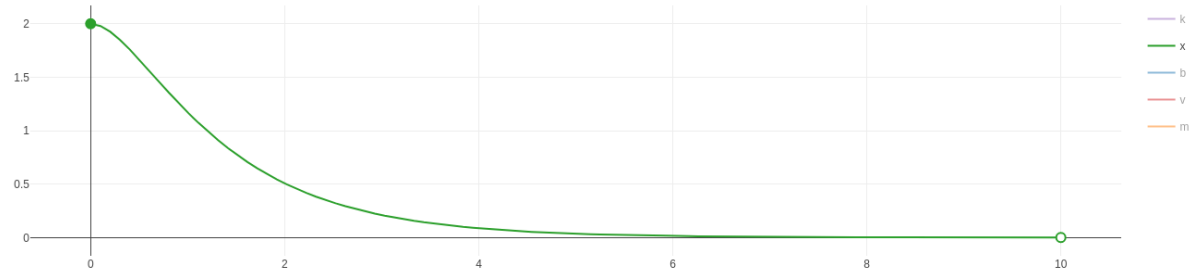


Figure 27: *Symbolic plot of the damped harmonic motion in the overdamping regime.*

For the regime to become overdamping, it is necessary that $b/(2m) > \sqrt{k/m}$, which implies that $b > 2m\sqrt{k/m} \Leftrightarrow b > 3.046$, so if the damping constant has a value of 3.5, the regime becomes overdamping. Due to all this, it was enough just to change the variable "b" to the value 3.5.

4.3.2 Implementation of constant variables

To enable the use of constant variables without them contributing to non-linearity, it was necessary to go to the file "Traj.scala", and change the functions "$\mathrm{runAtomicWithTime}$" (responsible for calculating the value of a given atomic instruction at a given time) and "$\mathrm{runAtomicWithBounds}$" (responsible for calculating the value of a given atomic instruction for a given number of cycles or a given time limit). These changes were implemented at the beginning of each of the preceding functions. The process began by saving a list of dynamic variables from the differential equations of the atomic instruction received as arguments. Next, the symbolic expressions of each variable in the program were updated based on the assignments found in the atomic instruction. In the next step, the data type of each variable in the program

(symbolic data type) was converted to the "NotLin" data type (associated with the non-linear equations in the "Syntax.scala" file).

After converting the data type of each variable in the program to the "NotLin" data type, the next phase involved examining each of the differential equations presented in the atomic instruction. This involved replacing constant variables with their corresponding expressions to ensure that the differential equations only contained dynamic variables.

Finally, we conducted a thorough examination of the differential equations to identify potential problems. First, we checked for the presence of "max" or "min" instructions/mathematical functions with dynamic variables in their arguments. This step was crucial because SageMath sometimes exhibits inconsistencies in the symbolic computation of these instructions, leading to incorrect results. In cases where such instructions were found, the system generated an error message indicating the problematic differential equation.

Next, we verified whether the differential equations exhibited any non-linear characteristics. If any of the equations exhibited, or were suspected of exhibiting, non-linearity, the system generated an error message identifying the specific differential equation in question.

Ultimately, if neither of these verifications detected any problems with the differential equations received, the system proceeded to perform the operations on the differential equations and assignments in the provided atomic instruction argument of the "Traj.scala" functions, where these modifications were implemented. The code instructions responsible for executing the procedure described are shown in Fig. 28.

Afterwards, each of the functions that were developed so that constant variables could be used without contributing to the non-linearity of the differential equations will be looked into in more detail.

As previously mentioned, the dynamic variables within the differential equations of the atomic instruction, which are passed as arguments in each of the functions where the code excerpt of the Fig. 28 was implemented, have been extracted. To achieve this, the function "extractVarsDifEqs has been implemented in the "Utils.scala" file, and you can consult its implementation in Fig. 29.

The function "extractVarsDifEqs" takes an atomic instruction and returns a list of the names of the dynamic variables[3] contained in it. To do so, the variable "listVars" was created, which will

[3] The main characteristic that distinguishes dynamic variables from constant variables is that dynamic variables vary over the execution time of their differential equation. However, it is agreed that variables are dynamic if the expression of their differential equation (after numerical simplifications) contains variables, or if the numerical result of the equation is non-zero. The reason for agreeing that the presence of variables in the expression of the differential equation would indicate dynamic behaviour was to avoid the need to use complex algorithms, such as symbolic simplification algorithms. These complex algorithms would bring a greater efficiency in obtaining variables with dynamic behavior, however the case studies for which Lince is intended, contain differential equations of low complexity and easy identification as to their evolutionary behavior over time. As such, the criteria established for the detection of dynamic variables have

```
//Extraction the dynamic variables from the diff.eq.
var extractVDE=Utils.extractVarsDifEqs(at)
//Updating the simbolic expressions of each variable
var updateValuate= x ++ Utils.toValuation(at.as,x)
//Convertion the type simbolic to "NotLin"
var newNotLin:ValuationNotLin=updateValuate.view.mapValues(e=>Eval.syExpr2notlin(e)).toMap
//Change the constant variables of the differential equations
var newListDiffEq=(at.de.eqs).map(e=>Eval.updateDiffEq(e,newNotLin,extractVDE)).toList
//Creation of the updated Atomic
var updateAtomic:Atomic=Atomic(at.as,DiffEqs(newListDiffEq,at.de.dur))

//Verification of 'max' and 'min' instructions with dynamic variables in their arguments,
//as well as the presence of non-linear expressions in the differential equations.
var linVerify=Utils.verifyLinearityEqsDiff(updateAtomic)
var min_max_check= Utils.verify_min_max(updateAtomic)
    if (min_max_check.nonEmpty) return throw new ParserException((s"It is not possible to
        apply the max
    or min instructions to expressions with dynamic variables in differential
    equations:${Show.apply(min_max_check.get)}"))
    else if (linVerify.nonEmpty) return throw new ParserException(s"There is one differential
        equation that
    is not linear or the semantic analyser suspects that it is non-linear (try simplifying
        the differential
    equation):  ${Show.apply(linVerify.get)}")
    else {
    //The code present in the else condition corresponds to symbolic/numerical calculation of
        the Atomic
     }
```

Figure 28: *Code extract responsible for enabling the use of constant variables in differential equations*

```
def extractVarsDifEqs(prog:Atomic):List[String] = {
  var listVars:List[String]=List()
  for (eqDiff <- prog.de.eqs){
     if (extractTotalVarsLinearExp(eqDiff.e)==0) {
        if (calc_doubles(eqDiff.e)!=0) listVars=listVars ++ List((eqDiff.v).v)
     } else listVars=listVars ++ List((eqDiff.v).v)
  }
  return listVars
}
```

Figure 29: *Function "*extractVarsDifEqs*"*

accumulate the names of the dynamic variables in a list. Then, a "for" cycle was created that will scan each differential equation in the atomic instruction and determine, through conditional instructions, if

proven to be adequate for the case studies already covered and seem to be effective for the whole range of case studies that can be treated.

the expression of the differential equation contains variables after numerical simplifications (through the "extractTotalVarsLinearExp" function that will be discussed in more detail in Section 4.5.5). If it has variables in the expression after numerical simplifications, the derived variable (the which is on the left side of the "=" sign of the differential equation) is saved in the "listVars" list (it is considered that the presence of variables in the expressions made the derived variable dynamic), otherwise it was checked whether the result of the equation was different from 0 (through the function "calc_doubles" which will also be discussed with more detail in Section 4.5.5). If the result of the equation was different from 0, the derived variable was dynamic and it was saved in the list "listVars", if it was 0 then the variable was constant and not stored in this list.

For example, if the differential equations present in the atomic instruction were "x'=p*x,p'=1-1,y'=x^0 for 1", the variables contained in the list returned by the previous function would be "x" and "y", because the expression of the differential equation of "x" contained variables and the expression of the differential equation of "y", after applying numerical simplifications ($x^0 = 1$), has a numerical result different from 0. However, the expression of the differential equation of "p", which contained no variables and the numerical result was 0, meeting the criteria for being a constant variable.

Next it was necessary to update the symbolic expressions of the program variables based on the assignments present in the atomic instruction. For that, it was used the function "toValuation", which had already been implemented in the file "Utils.scala".

The next step was to pass the values of the program's variables from the symbolic data type to the "NotLin" data type. This was possible by going into the "CommonTypes.scala" file (file where we find data types that are used in the other files) and adding a new data type called "ValuationNotLin", as depicted in Fig. 30.

```
type ValuationNotLin= Map[String,NotLin]
```

Figure 30: *Data type "*ValuationNotLin*"*

Unlike the "Valuation" data type (data type returned by the "toValuation" function), which maps variables to data structures of symbolic type, the "ValuationNotLin" data type maps variables to data structures of type "NotLin".

Having created the "ValuationNotLin" data type, it was necessary to create a function that converted symbolic data type to "NotLin" data type. So in the file "Eval.scala" the function "syExpr2notlin" was created. This function is shown in the code extract in Fig. 31 and receives an argument of symbolic type (more specifically of type "SyExpr") and returns the conversion of this argument to the respective

case class “NotLin”.

```
def syExpr2notlin(l:SyExpr):NotLin= l match {
  case SVal(v) => Value(v)
  case SFun(s,list)=> Func(s,list.map((l:SyExpr) => syExpr2notlin(l)))
  case SDiv(e1, e2) => Div(syExpr2notlin(e1),syExpr2notlin(e2))
  case SRes(e1, e2) => Res(syExpr2notlin(e1),syExpr2notlin(e2))
  case SMult(e1, e2)=> Mult(syExpr2notlin(e1),syExpr2notlin(e2))
  case SPow(e1, e2) => Func("pow",List(syExpr2notlin(e1),syExpr2notlin(e2)))
  case SAdd(e1, e2) => Add(syExpr2notlin(e1),syExpr2notlin(e2))
  case SSub(e1, e2) => Add(syExpr2notlin(e1),Mult(Value(-1),syExpr2notlin(e2)))
}
```

Figure 31: *Function “*syExpr2notlin*”*

Having created the “ValuationNotLin” data type and the “syExpr2notlin” function, it was enough to go to the result of updating the symbolic expressions of the variables from the code of the Fig. 28 (variable “updateValuate”) and execute the code of the Fig. 32.

```
var newNotLin:ValuationNotLin=updateValuate.view.mapValues(e=>Eval.syExpr2notlin(e)).toMap
```

Figure 32: *Line of code responsible for converting the data type of the program’s variables to the type “*NotLin*”*

What the line of code of the Fig. 32 did was store in the “newNotLin” variable (now of type “ValuationNotLin”) the result of the “updateValuate” variable, but now, instead of the variables being associated with a symbolic data structure, they are now associated with a “NotLin” data structure due to the use of the “syExpr2notlin” function.

The next step was to create a list of differential equations where constant variables were replaced by their respective expressions (of the type “NotLin”). For this, the functions of the Fig. 33 was created in the file “Eval.scala”.

The “updateDiffEq” function receives a differential equation, the association between the variables and their expression and the list of dynamic variables from the set of differential equations, returning the differential equation with the constant variables of the expression replaced by the respective expressions. However, to replace the constant variables in the expression of the differential equation, it was necessary to create the function “updateNotLin”, which receives the association between variables and their expression, the expression of the differential equation and the list of dynamic variables from the set of differential equations. This function uses recursion to go through the expression of the differential equation

```
def updateDiffEq(diffeq:DiffEq,v:ValuationNotLin,vars:List[String]):DiffEq = {
   var newNotLin= updateNotlin(v,diffeq.e,vars)
   var newdiffeq= DiffEq(diffeq.v,newNotLin)
   return newdiffeq
}
def updateNotlin(state:ValuationNotLin, notlin: NotLin,vars:List[String]): NotLin = notlin
    match {
      case Var(v) => {if (vars.contains(v)) {Var(v)} else {state(v)}}
      case Value(v) => Value(v)
      case Add(l1, l2) => Add(updateNotlin(state,l1,vars), updateNotlin(state,l2,vars))
      case Mult(l1,l2) => Mult(updateNotlin(state,l1,vars), updateNotlin(state,l2,vars))
      case Div(l1,l2) => Div(updateNotlin(state,l1,vars), updateNotlin(state,l2,vars))
      case Res(l1,l2) => Res(updateNotlin(state,l1,vars), updateNotlin(state,l2,vars))
      case Func(s,list) => Func(s,list.map(l=>updateNotlin(state,l,vars)).toList)}
```

Figure 33: *Function "*updateDiffEq*" and function "*updateNotLin*"*

and find the variables present in it, replacing the variable by its expression (using the association received in the argument) if it was not present in the list of dynamic variables.

Having these two functions implemented, it was enough to elaborate the lines of code of the Fig. 34 to create a set of differential equations equal to those present in the atomic instruction received as an argument, only with the constant variables replaced by the respective non-linear expression.

```
var newListDiffEq=(at.de.eqs).map(e=>Eval.updateDiffEq(e,newNotLin,extractVDE)).toList
var updateAtomic:Atomic=Atomic(at.as,DiffEqs(newListDiffEq,at.de.dur))
```

Figure 34: *Lines of code responsible for replacing the constant variables in the differential equations with their respective expressions*

The final step involved verifying dynamic variables within the "max" and "min" mathematical function arguments within the differential equations, as well as checking the linearity of these equations. Detailed information about the strategies and functions developed for these tasks can be found in Section 4.5.4 andSection 4.5.5, respectively.

The lines of code responsible for checking the presence of dynamic variables in the "max" and "min" instructions, as well as verifying if the differential equations exhibit non-linear behavior, is illustrated in Fig. 35.

To illustrate how the treatment of constant variables works, let's consider the following hybrid program:

```
x:=1;
y:=2/3;
x'=y*x,y'=0 for 1;
```

```
var linVerify=Utils.verifyLinearityEqsDiff(updateAtomic)
var min_max_check= Utils.verify_min_max(updateAtomic)
    if (min_max_check.nonEmpty) return throw new ParserException((s"It is not possible to
        apply the max or min instructions to expressions with dynamic variables in
        differential equations:${Show.apply(min_max_check.get)}"))
    else if (linVerify.nonEmpty) return throw new ParserException(s"There is one differential
        equation that is not linear or the semantic analyser suspects that it is non-linear
        (try simplifying the differential equation): ${Show.apply(linVerify.get)}")
    else {
    //The code present in the else condition corresponds to symbolic/numerical calculation of
        the Atomic
}
```

Figure 35: *Lines of code responsible for checking the presence of dynamic variables in the "max" and "min" instructions, as well as verifying if the differential equations exhibit non-linear behavior*

This program is parsed and then sent to the interpreter. The interpreter performs the necessary semantic operations until it reaches the processing stage of the atomic instruction containing the differential equation from the previous hybrid program. To perform this processing, it uses the function "runAtomicWithBounds" and applies the treatment of constant variables discussed in this section, with each step schematised in Fig. 36.

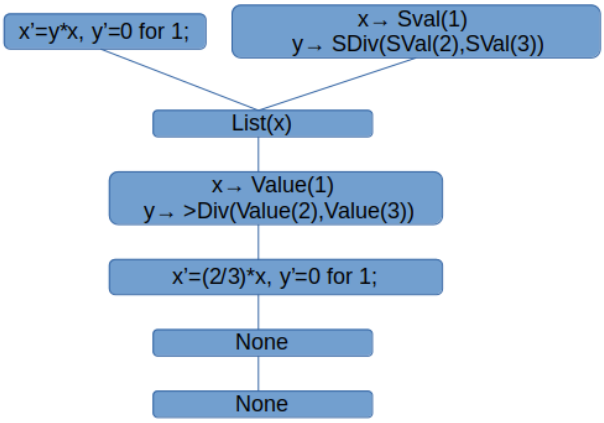


Figure 36: *Schematic example of the treatment of constant variables*

As depicted in the schematic of the Fig. 36, it was necessary to:

- Extract the list of dynamic variables based on the differential equations (which in this case only "x" is dynamic);
- Update the association between the variables and their symbolic expression and convert the symbolic expression to an expression of type "NotLin";

- Change the differential equations so that the constant variables were replaced by their respective expression;
- Check for the presence of "max" and "min" instructions with dynamic variables in their arguments in the differential equations (as there was no such instructions in this example, "None" was returned);
- Checked if the differential equations are linear or nonlinear (in this example they were linear, so "None" was returned).

In this way, all conditions were provided so that the previous set of differential equations could be evaluated, even using a constant variable.

4.4 Implementation of the numerical plot

Based on reference [GNP], Lince's semantics were designed to handle symbolic representations (or symbolic expressions) in the solutions of differential equations (obtained through SageMath), allowing for more precise solutions compared to numerically obtained ones. These symbolic representations are also present in conditional expressions, allowing the use of SageMath to perform symbolic manipulations to verify whether conditions are True or False, thereby achieving more accurate results compared to conventional conditional expression evaluation functions.

The following hybrid program illustrates the usefulness of using symbolic representations:

```
x:=1;
x'=-x for 10;
x'= x for 10;
if x==1
then x:=2;
else x:=0;
```

This program first assigns the variable x as 1 and continuously evolves this variable according to the differential equation x'=-x for 10 seconds, and then according to the differential equation x'=x for the same period. Finally, it checks whether x is equal to its initial value, returning 2 if True and 0 if False.

The symbolic plot resulting from the execution of the previous hybrid program in the Lince tool is depicted in Fig. 37.

A brief analysis of the plot of the Fig. 37 shows that the variable x evolves according to a negative exponential function until 10 seconds, and then evolves according to a positive exponential function for the next 10 seconds, ending exactly where it started.

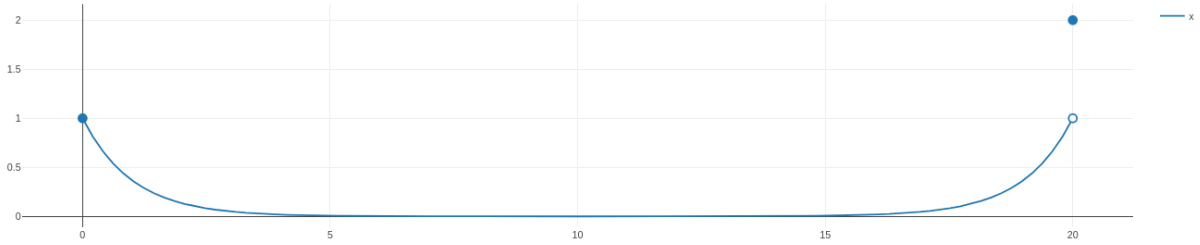


Figure 37: *Symbolic plot resulting from the hybrid program:* $\mathrm{x{:=}1;\ x'{=}{-}x\ for\ 10;\ x'{=}\ x\ for\ 10;\ if\ x{==}1\ then\ x{:=}2;\ else\ x{:=}0;}$

The variable ends exactly where it started due to the use of symbolic representations and SageMath in the solutions of the differential equations and conditional expressions.

For better understanding, you can click on the "all jumps" button at the top right of the plot to show the value of the variable x at each iteration of the hybrid program. Clicking on the value of x at $\mathrm{t{=}0}$ (initial iteration), you will see that at that moment, x is associated with the expression e^{-t} and the numerical value 1 resulting from substituting the time t with 0 in that expression. This expression represents the solution of the first differential equation based on the initial conditions and comes from the symbolic representation of the solution of this differential equation from SageMath. Additionally, the samples between time 0 and 10 come from this expression.

Clicking on the value of x at $\mathrm{t{=}10}$ (second iteration), you will see that at that moment in time, x is associated with another expression, namely $e^{-10} * e^{t}$, and the corresponding numerical value 45.39993×10^{-6} resulting from substituting the time t with 0 in that expression (since at $\mathrm{t{=}10}$ the variable x evolves according to another differential equation, the initial value corresponds to the symbolic expression of x at $\mathrm{t{=}10}$ from the previous differential equation expression). In the same way as the expression from the previous iteration, this expression represents the solution to the second differential equation. It allows the generation of samples within the time interval of 10 to 20 units. It is important to note that these samples are obtained by replacing the variable t with values between 0 and 10, since this corresponds to the evaluation of a new differential equation over a period of 10 seconds.

Finally, when clicking at $\mathrm{t{=}20}$ (third iteration), you will observe that the result of the conditional expression $\mathrm{x{==}1}$ is True, as SageMath is able to relate the symbolic expression $e^{-10} * e^{10}$ to the symbolic expression 1, returning True as expected.

With the use of symbolic solutions of differential equations, all the samples extracted from these expressions to construct the plot will be more precise than any numerical method for solving differential equations. This is because numerical methods return numerical values that are subject to rounding and propagation errors [SEN14].

In addition to the increased precision in obtaining samples for constructing the plot, the use of conditional expressions involving symbolic expressions with SageMath allows for symbolic manipulations to identify the resulting boolean value. In the previous example, SageMath will manipulate the expression $e^{-10} * e^{10}$, and since the exponentials cancel each other out, it returns 1 and the result will be True. If you were to try running the condition $e^{-10} * e^{10} == 1$ in a language like Scala, for instance, the result would be False due to approximation errors, which could critically affect the hybrid program.

Although the previous method implies higher precision in the results of hybrid programs, it is also susceptible to an error that can affect a wide range of hybrid programs governed by Newtonian mechanics, particularly when using the differential Eq. (3.2) and Eq. (3.3).

The error arises from the inability to condense/simplify the symbolic expressions of the solutions of the differential equations returned by SageMath, causing the solution to grow larger with each iteration until SageMath can no longer handle it, resulting in an error.

This error can be encountered in the simulation of the following hybrid program:

```
x:=0; y:=0;
vx:=1;vy:=1;
w:=0;
while true do {
    w:=0;
    x'=vx,y'=vy,vx'=w*vy,vy'=-w*vx for 1;
    w:=1;
    x'=vx,y'=vy,vx'=w*vy,vy'=-w*vx for 1;
    w:=-1;
    x'=vx,y'=vy,vx'=w*vy,vy'=-w*vx for 1;
            }
```

This hybrid program begins with assignment of the initial position (x,y) and initial velocity (vx,vy) of a body, as well as the assigment of the variable w which determines whether the body moves forward or turns. Them, a loop is executed that iteratively runs three differential equations of motion (see Eq. (3.2)) for 1 second each. The first differential equation is evaluated for $\mathrm{w}{:=}0$, causing the body to move forward at a constant velocity, the second equation is for $\mathrm{w}{:=}1$, causing it to turn right, and the third equation is for $\mathrm{w}{:=}\text{-}1$, which will make the body turn left.

Running the previous program resulted in the error message shown in Fig. 38.

The error message of the Fig. 38 indicates that Sage was unable to handle this expression, which occurred at $\mathrm{t}{=}19$.

Reducing the maximum simulation time to 3 seconds, the expressions for x as a function of t are the following:

```
At time 19: (Stopped waiting for Sage. This timeout is normal in the first execution of
    Lince,
because the associated systems need to be started. Please try again.)

SageMath was unable to handle the following
    expression:((-((((((((((((((((((((((cos(1)+(-1*sin(1)))*
cos(1))+((cos(1)+sin(1))*sin(1)))*cos(1))+(((((-1*cos(1))+(-1*sin(1)))*cos(1))+(((1*cos(1))+
(-1*sin(1)))*sin(1)))*sin(1)))*cos(1))+((((((cos(1)+sin(1))*cos(1))+(((-1*cos(1))+(1*sin(1)))*
sin(1)))*cos(1))+((((cos(1)+(-1*sin(1)))*cos(1))+((cos(1)+...
```

Figure 38: *Error message returned by Lince*

- t=0: x_0(t)= t;
- t=1: x_1(t)= -cos(t)+sin(t)+1+x_0(1)= -cos(t)+sin(t)+1+1;
- t=2: x_2(t)= (cos(1)-sin(1))*cos(t)+(cos(1)+sin(1))*sin(t)+2*x_1(1)= (cos(1)-sin(1))*cos(t)+(cos(1)+sin(1))*sin(t)-cos(1)+sin(1)-cos(1)+sin(1)+2;
- t=3: x_3(t)= t*((cos(1)+sin(1))*cos(1)-(cos(1)+sin(1))*sin(1))+x_2(1)= t*((cos(1)+sin(1))*cos(1)-(cos(1)+sin(1))*sin(1))+(cos(1)-sin(1))*cos(1)+(cos(1)+sin(1))*sin(1)-2*cos(1)+2*sin(1)+2;

It is possible to observe a pronounced growth in the size of the expression for x with each iteration. This growth occurs because SageMath is unable to perform significant symbolic simplifications in more complex expressions that involve some mathematical functions such as trigonometric functions, exponentials, roots, etc, and since the expression for the next iteration depends on the previous one, its size keeps increasing. As one can deduce, for $\mathrm{t}{=}19$ (iteration 19 since each iteration takes 1 second) the expression for each variable as a function of t will have a huge size, and it was found that SageMath can only handle expressions up to a certain size, leading to the appearance of the previous error.

However, differential equations like those in the previous example are quite useful in designing hybrid programs regulated by Newtonian Mechanics. Some examples of where these differential equations are helpful include the simulation of a vehicle moving in 2D that aims to follow a specific path or simulating a 2D guided missile targeting a moving object.

To mitigate this issue, another plot was implemented that calculates the solutions of the differential equations using a numerical method.

According to Senthilkumar [SEN14], the main advantages of numerical methods compared to analytical methods (such as the method used by Lince) include their easy implementation on modern computers,

the rapid attainment of solutions to differential equations, and the ability to obtain solutions for complex differential equations. On the other hand, analytical methods provide more accurate solutions, in certain cases this precision can be crucial (as demonstrated in the first hybrid program presented in this section, because if the solutions of the differential equations were derived from a numerical method, the final value of variable x would be close to 1 but not exactly 1, causing the conditional expression to be False), and are limited to simpler differential equations.

With the implementation of a new plot that obtains solutions to differential equations numerically, the issue of big growth of the solution expressions is eliminated, but at the cost of reduced precision. However, the primary focus of implementing this plot is to simulate hybrid programs governed by Newtonian Mechanics that utilize differential equations such as Eq. (3.2) and Eq. (3.3), and since these programs are generally associated with large-scale physical systems (such as vehicles, missiles, boats, etc.), some lack of precision is entirely tolerable.

For the implementation of the numerical plot the fourth-order Runge-Kutta method was used [Mai15]. This method is more accurate than some alternative methods, although it is computationally heavier and slower, and it is commonly used in the literature. The performance of the resulting implementation was very satisfactory for the examples described in this project.

The implementation of the numerical plot in Lince was based on an existing simpler implementation of a numerical plot. Instead of calling the SageMath tool to obtain a symbolic solution of the differential equations, our implementation uses the numerical Runge-Kutta method, which is evaluated directly in JavaScript without relying in external processes.

To develop the fourth-order Runge-Kutta method in Scala, it was initially necessary to understand how the method works. Based on reference [Hen16], to compute the numerical solutions of a system of first-order differential equations (which is the type of system used by Lince) using the fourth-order Runge-Kutta method for time $t = b$, with a total of N steps, and knowing the initial conditions, the following procedure is performed:

- Determine the spacing: $h = b/N$;
- Store the initial condition of each of the n differential equations in the data structure $init$: $init_i = \alpha_i$;
- Create data structures to store the values of $K1$, $K2$, $K3$, and $K4$ for each differential equation;
- Set the time b equal to the variable t: $t = b$;

- Iterate N times through the following procedure:
  - For each of the n differential equations, determine: $K1_i = h * f_i(t, init_1, ..., init_n)$;
  - For each of the n differential equations, determine: $K2_i = h * f_i(t + h/2, init_1 + K1_1/2, ..., init_n + K1_n/2)$;
  - For each of the n differential equations, determine: $K3_i = h * f_i(t + h/2, init_1 + K2_1/2, ..., init_n + K2_n/2)$;
  - For each of the n differential equations, determine: $K4_i = h * f_i(t + h, init_1 + K3_1, ..., init_n + K3_n)$;
  - Update $init$: $init_i = init_i + 1/6 * (K1_i + 2 * K2_i + 2 * K3_i + K4_i)$;
  - Update the value of t: $t = t + h$;
- Return the $init$ data structure.

Where $f_i(t, u_1, ..., u_n)$ represents the expression of the i-th differential equation.

Based on the previous procedure, the fourth-order Runge-Kutta method was implemented in Scala through the function "$\mathrm{runge\_kutta\_func}$".

(**Note:** Due to the dimension of the code, it was decided not to show the function "$\mathrm{runge\_kutta\_func}$" in the sequence of the previous paragraph. As such, to see the code you can consult Appendix A.3.)

The function "$\mathrm{runge\_kutta\_func}$" takes a set of differential equations, their initial conditions, the time t at which the numerical solution of the differential equation is to be calculated, and the name of the variable of the differential equation to be evaluated. However, unlike the previous procedure, the function only returns the numerical value of the solution of the differential equation for the variable provided in the argument at time t, due to compatibility reasons with the rest of the code.

It's also important to note that we used a fixed step size of 75 to calculate the desired solution. After some tests, this choice proved to be a good balance between accuracy and speed.

With the creation of a plot capable of numerically computing the solutions of differential equations, it was possible to overcome the issue of equations growing excessively. Additionally, hybrid programs that included linear differential equations in their design which SageMath couldn't symbolically solve (such as the example discussed in Section 4.5.6), now have the option to be simulated through the numerical plot.

To demonstrate the functionality of the numerical plot, the hybrid program that was used as an example to highlight the issue of excessively growing expressions was simulated using the numerical plot, resulting in the plot of the Fig. 39.

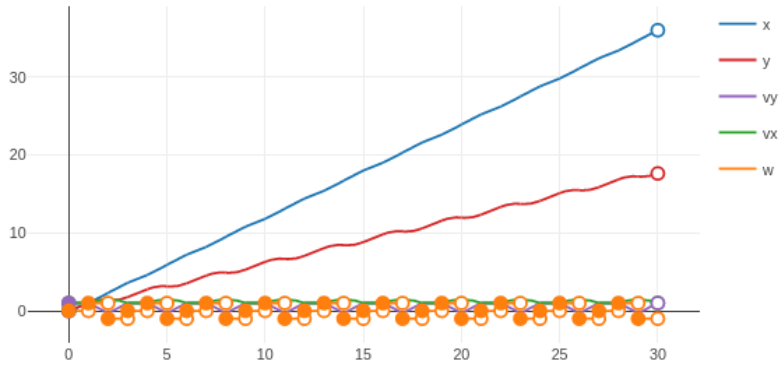


Figure 39: *Numerical plot resulting from the hybrid program above*

From the plot of the Fig. 39, it can be observed that the body iteratively moves forward for 1 second, turns left for 1 second, and then turns right in the same time interval.

Other examples where the numerical plot is used to obtain simulation results of a specific hybrid program are in the hybrid program discussed in Section 4.5.6 and in the case study of Section 5.3.

4.5 A better error-message system

As mentioned in Section 1.2 and Chapter 3, the objective of this dissertation is to detect problems and try to correct them, find possible improvements and try to implement them, extend the Lince programming language for a better capture of hybrid programs governed by Newtonian Mechanics and improve the user experience.

One of the topics that improve the user experience is the detection of semantic errors by the semantic analyzer. It is quite common for errors to be implemented in the design of any program on any platform, however a good error message can facilitate correction by the user, while its absence or lack of information implies a long and painful debugging.

In the old version of Lince, the detection of some errors by the semantic analyser had already been implemented, but some messages contained little information, some inconsistencies or did not even correspond to what was intended.

Therefore, the new version aims to improve the detection of existing errors, both at the level of the message itself and at the implementation level, and it was added the detection of other errors that were consider important.

The following chapters aim to explain the implementations and improvements made in error detection

by the semantic analyzer of the new version of Lince.

4.5.1 Detection of unassigned variables on the right hand side of the initial assignments

The semantic analyser in the old version checked if the initial assignments had variables on the right hand side or not and if they did, an error message was returned. Nevertheless, it was concluded that the use of variables on the right hand side of the initial assignments gives great comfort and utility when writing initial assignments.

In order for the semantic analyser to support the use of variables on the right hand side of the initial assignments, the file "Utils.scala" was modified. First, it was created a recursive function called "extractAssignments", which received an argument of type "Syntax", that corresponds to the result of parsing a program, and returns the list of assignments that are performed before the appearance of cyclic or conditional instructions. This function is depicted in Fig. 40.

```
def extractAssignments(prog:Syntax):List[Assign] =prog match {
  case Atomic(as,deqs) => {
    if (deqs.eqs==List()){
        as
        }
    else {
        var aux=List(Assign(Var("Stop case"),Value(-1)))
        aux
        }
                          }
  case While(pre,c,p) => {
     extractAssignments(pre)
                          }
  case Seq(p,q) =>{
        (extractAssignments(p) ++ extractAssignments(q))
                  }
  case _ =>  List()
}
```

Figure 40: *Function "extractAssignments"*

The function "extractAssignments" starts by checking if the parser output corresponds to an atomic instruction, cyclic instruction, sequence of programs or any other type of instruction. If the output is an atomic instruction associated with an assignment, it returns a list of its assignment. If it is an atomic associated with a set of differential equations, it returns a list of an assignment associated with the variable

named "Stop case" (as the variable names can never be separated by a space, this name will never conflict with the names of the assignments in the program, and can therefore be used as a marker to filter assignments that correspond to the initial assignments). In case the input corresponds to an cyclic instruction, it is enough just to return the list of initial assignments present in the "pre" program (program which precedes the cyclic instruction). If the output corresponds to a sequence of programs, it is necessary to return the list of initial assignments from both programs. Finally, if the output corresponds to some other type of instruction, such as the "if-then-else", an empty list is returned, because there is no interest in the assignments performed during and after these instructions.

Following that, we create a function named "assignmentsVerify". This function takes an argument of type "Syntax", representing the result of parsing a program, and its purpose is to return the list of variables on the right hand side of the initial assignments that have not been assigned previously. You can find this function illustrated in Fig. 41.

The function "assignmentsVerify" starts by checking if the parser output is related to a sequence of programs, an atomic instruction, a cyclic instruction or any other type of instruction.

If the parser output corresponds to a program sequencing, the strategy to return the variables on the right hand side of the initial assignments that were not previously assigned consisted of initially storing all the assignments that precede cyclic and conditional instructions in both programs through the function "extractAssignments", then verifying the existence of the assignment associated with the variable "Stop case" and if it existed, only the list of assignments that preceded the first occurrence of that assignment were retained[4]. These assignments are those associated with the initial assignments. After obtaining the initial assignments, it was verified if there were variables on the right hand side of the initial assignments that had not been previously assigned, storing them in a list that would be returned.

However, if the parser output is associated with an atomic instruction, the method for obtaining unassigned variables on the right hand side of the initial assignments is identical to the previous case, it only changes the fact that the initial assignment is the assignment itself (if any).

Finally, if the parser output is associated with a cyclic or conditional instruction, the programs associated with these instructions may contain assignments and there can be variables present in the assignment expressions. However, these variables are not variables to the right of the initial assignments because they

[4] Note that the assignments associated with the "Stop case" variable refer to the occurrence of differential equations and the initial assignments are just the assignments that appear before the first occurrence of differential equations, cyclic and conditional instructions, and since it had already retained all assignments that preceded the first occurrence of cyclic and conditional instructions, it was necessary to also retain those that preceded the first occurrence of differential equations to only obtain the assignments that correspond to the initial assignments.

```
def assignmentsVerify(prog:Syntax): Set[String] = prog match {
    case Seq(p,q) => {
      var res=extractAssignments(p) ++ extractAssignments(q)
      var indice=res.indexOf(Assign(Var("Stop case"),Value(-1)))
      var as=res
      if (indice>=0) {
        as=as.take(indice)
      }
      var declVar= as.map(_.v.v).toList
      var aux=0
      var aux2=1
      var zz:Set[String]=Set()
      for (i <- as){
        var z=getVars(i.e)

        for (j <- 0 until (aux) by 1){
          z -= declVar(j)
        }
        zz=zz++z
        aux=aux+1
        }
       return zz
    }
    case Atomic(as,_)=>{
      var declVar= as.map(_.v.v).toList
      var aux=0
      var aux2=1
      var zz:Set[String]=Set()
      for (i <- as){
        var z=getVars(i.e)
        for (j <- 0 until (aux) by 1){
          z -= declVar(j)
        }
        zz=zz++z
        aux=aux+1
        }
       return zz
    }
    case While(pre,c,p) => assignmentsVerify(pre)
    case _ => Set()
  }
```

Figure 41: *Function "assignmentsVerify"*

are not associated with initial assignments. As such, in case the output is a cyclic instruction, the recursivity of the "assignmentsVerify" function is applied in the program that precedes it and the list of variables

on the right hand side of the initial assignments that were not previously assigned from the program that precedes it, is extracted. In case the output is a conditional instruction it returns an empty list.

Having created the two previous functions, it was necessary to modify the "isClosed" function (function invoked by the "Parser.scala" file to check if the program contains unassigned variables on the right hand side of the initial assignments and in the rest of the program) so that it would be able to detect the existence of unassigned variables on the right hand side of the initial assignments and return the respective error. The function "isClosed" is shown in the Fig. 42.

```
def isClosed(prog:Syntax): Either[String,Unit] = {
  val asVerify=assignmentsVerify(prog)
  if (asVerify.nonEmpty)
    Left(s"Initial assignments have variables on the right hand side that were
        not assigned: ${asVerify.mkString(", ")}")
  else
    Right(())
}
```

Figure 42: *Function "isClosed"*

This function takes as argument a variable of type "Syntax" (i.e. the result of parsing the hybrid program) and returns a statement of type Either, which can be a Right(()) if there are no errors, or a Left(s"...") if there are errors. To check if there are variables on the right hand side of the initial assignments that were not previously assigned, this function use the function "assignmentsVerify" to return a list with these variables, then checking if this list is an empty list or not and if it is an empty list just return Right(()), but if it is a non-empty list, a Left(s"...") is returned (containing an error message indicating the variables to the right of the initial assignments that were not previously assigned in the program).

(**Note**: The "isClosed" function mentioned above is not the final version of the "isClosed" function, in the Section 4.5.2 a future version will introduce instructions to check for variables in the rest of the program that were not assigned in the initial assignments.)

To demonstrate how the detection of unassigned variables on the right hand side of the initial assignments works in the Lince tool, an example will be presented where the hybrid program presents this type of semantic error in its conception.

Having the following hybrid program:

```
x:=1+y;
y:=2+z;
z:=3;
```

```
z'=2 for 1;
```

And if this program is run on Lince, the error message depicted in Fig. 43 appears.

```
Error: When parsing G§150 // maximum time in the plot§0§
x:=1+y;
y:=2+z;
z:=3;
z'=2 for 1;
— hprog.common.ParserException: Initial assignments have variables on the right hand side
    that were not assigned: _y, _z
```

Figure 43: *Error message returned by Lince*

It can be noted, as intended, that an error was returned indicating that the variables on the right hand side of the initial assignments "y" and "z" were not assigned previously, because in the expression of the assignment of the variable "x" ("x=1+y") the variable "y" was used which had not been assigned previously, and the same happened in the expression of the assignmente of the variable "y" ("y=2+z") because the variable "z" had not been assigned before either.

However, if the previous hybrid program is changed so that the variables on the right hand side of the initial assignments are assigned before:

```
z:=3;
y:=2+z;
x:=1+y;
z'=2 for 1;
```

And if that program is run in Lince, the symbolic plot of the Fig. 44 is obtained.

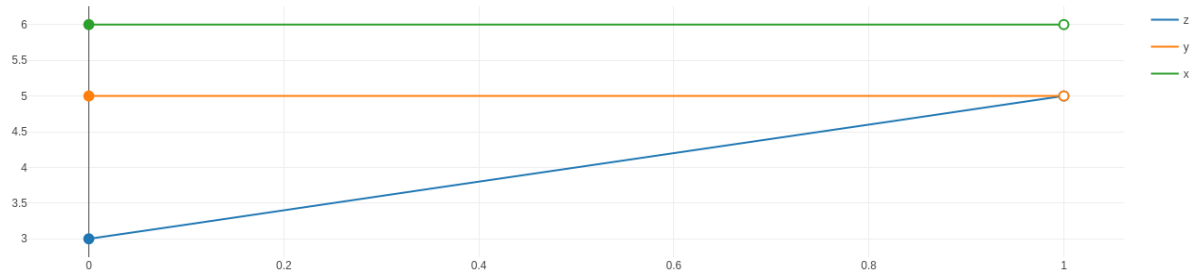


Figure 44: *Symbolic plot resulting from the hybrid program:* z:=3; y:=2+z; x:=1+y; z'=2 for 1;

The appearance of the symbolic plot, relative to variables as a function of time of the previous hybrid program, shows that the changes made previously allow the new version of Lince to use variables in the right side of the initial assignments, provided that they have been assigned before being used in the

expressions of the initial assignments. On the other hand, the old version could not support any kind of these variables, returning the following error if the previous hybrid program was running:

```
Error: When parsing G§15 // maximum time in the plot§0§
z:=3;
y:=2+z;
x:=1+y;
z'=2 for 1; — hprog.common.ParserException: Initial assignments have variables on the
    right hand side that were not assigned: z, y
```

Figure 45: *Error message returned by old version of Lince*

4.5.2 Detection of variables that were not assigned at the beginning of the program

Regarding the verification of variables that were not assigned at the beginning of the program, we found that the old version of Lince already had functions designed for this purpose. However, the verification of initially assigned and used variables exhibited some inconsistencies. The old version did not require programs to start with initial assignments, so in cases where a program started with a "if-then-else" statement, the initial assignments were considered to be placed inside the "then" branch of the "if-then-else" construct. This particular feature, which allows initial assignments to be placed exclusively within one of the program branches associated with the conditional statement, was considered inconsistent and inappropriate.

However, in the new version of Lince, the initial assignments are obligatory and always take place at the beginning of the program (see Section 4.1.2). Consequently, the function "getFstDeclVars", responsible for determining the set of initially variables, has been modified to consider only the initial set of assignments. This change allows it to extract initially variables from the program in a consistent and efficient manner. The function "getFstDeclVars" in the new version of Lince is shown in Fig. 46.

If the program is a sequence of instructions, the function of Fig. 46 starts by extracting the assignments that precede the appearance of a cyclical or conditional structure, then it only extracts the assignments that precede the appearance of the first set of differential equations (corresponding to the initial assignments, as seen in Section 4.5.1) and returns the set of variable names corresponding to those assignments. If the program were an atomic instruction, it would just return the set of the variable name of the assignment associated with it. For the case of the program being a cyclic structure, it was enough to use recursion to extract the set of variables corresponding to the initial assignments in the previous program, and finally, if

```
def getFstDeclVars(prog:Syntax): Set[String] = prog match {
  case Seq(p,q) => {
    var res=extractAssignments(Seq(p,q))
    var indice=res.indexOf(Assign(Var("Stop case"),Value(-1)))
    var as=res
    if (indice>=0) {
      as=as.take(indice)
    }
    var asSet=as.map(_.v.v).toSet
    return asSet
    }
  case Atomic(a, _)      => a.map(_.v.v).toSet
  case While(pre, _, _) => getFstDeclVars(pre)
  case _ => Set()
  }
```

Figure 46: *Function* "getFstDeclVars"

it is another type of statement (like a conditional statement) it was enough to return an empty set.

Besides the previous function, responsible for returning the set of variable names initially assigned in the program, it was necessary to alter the function "getUsedVars", responsible for returning the set of variable names used in the program. This function is represented in the code excerpt shown in Fig. 47.

```
def getUsedVars(eqs: List[DiffEq]): Set[String] =
  eqs.flatMap(eq => getVars(eq.e)+eq.v.v).toSet   //New

def getUsedVars(prog:Syntax): Set[String] = prog match {
  case Atomic(as, de) => as.toSet.flatMap((a:Assign)=>getVars(a.e)+a.v.v) ++
      getUsedVars(de)
  case Seq(p, q) => getUsedVars(p) ++ getUsedVars(q)
  case ITE(ifP, thenP, elseP) => getVars(ifP) ++ getUsedVars(thenP) ++
      getUsedVars(elseP)
  case While(pre, d, doP) => getUsedVars(pre) ++ getVars(d) ++ getUsedVars(doP)
}

def getUsedVars(eqs: DiffEqs): Set[String] =
  getUsedVars(eqs.eqs) ++ getUsedVars(eqs.dur)

def getUsedVars(dur: Dur): Set[String] = dur match {
  case Until(c,_,_) =>getVars(c)
  case For(nl) => getVars(nl) //New
  case _ => Set()
}
```

Figure 47: *Function* "getUsedVars"

The fuction "getUsedVars" has been modified only for the case where it is applied to a list of differential equations, as it now also returns the variables present on the left side of each differential equation (which was not taken into account in the old version), and also for the case where it is applied to a duration, as it did not handle the case where the duration is a "for ...".

Finally, the detection of variables that were not assigned at the beginning of the program and the return of the corresponding error, similar to the detection of the unassigned variables on the right hand side of the initial assignments, are handled by the "isClosed" function. With the introduction of this detection, we have reached the final version of the "isClosed" function, and its representation can be found in Fig. 48.

```
def isClosed(prog:Syntax): Either[String,Unit] = {
  val declVar=getFstDeclVars(prog)
  val usedVars = getUsedVars(prog)
  val asVerify=assignmentsVerify(prog)
  if (asVerify.nonEmpty)
    Left(s"Initial assignments have variables on the right hand side that were
        not assigned: ${asVerify.mkString(", ")}")
  else if (!usedVars.forall(declVar))
    Left(s"Variable(s) not assigned: ${((usedVars-- declVar)).mkString(", ")}")
  else
    Right(())
}
```

Figure 48: *Function "isClosed"*

The function "getFstDeclVars" starts by storing the set of variables that were initially assigned in the program in the variable "declVar" and the function "getUsedvars" stores the set of variables used in the program in the variable "usedVars". After adding this variables and checked if there were any variables on the right hand side of the initial assignments not previously assigned, it was verified if there were not any variables used that had not been assigned in the beginning of the program and if it was false, a "Left(s"...")" was returned with the error message identifying which variables are.

To demonstrate how the detection of variables that were not assigned at the beginning of the program works in the Lince tool, an example will be presented where the hybrid program presents this type of semantic error in its conception.

Given the following hybrid program:

```
p:=0; v:=2;
while true do {
  if v<=x
  then p'=v,v'=k  for w;
  else p'=v,v'=-k for w;
```

```
}
```

The output results in the error message shown in Fig. 49.

```
Error: When parsing G§15 // maximum time in the plot§0§// Cruise control
p:=0; v:=2;
while true do {
if v<=x
then p'=v,v'=k for w;
else p'=v,v'=-k for w;
} - hprog.common.ParserException: Variable(s) not assigned: _x, _w, _k
```

Figure 49: *Error message returned by Lince*

It can be seen that the hybrid program only assigned the variable "p" and "v" in the beginning of the program, but used the variable "x" in the "if-then-else" condition, the variable "k" in the expression of the equation differential in order to "v" and the variable "w" in the duration expression. Since the variables "x", "k" and "w" were not assigned at the beginning of the program, then the semantic analyzer is expected to detect this error and return an error message regarding it, as it happened.

4.5.3 Better errors when using mathematical functions and mathematical constants

In the old version, there were only the arithmetic operators $+$, $-$ and $*$, and whenever you used an operation that was not supported, a parsing error was returned, just like the error message in Fig. 50 resulting from running the following hybrid program:

```
v:=sqrt(25);
v'=1 for 1;
```

```
Error: When parsing G§150
v´=1 for 1;
- hprog.common.ParserException: [1.8] failure: ';' expected but '(' found
v:=sqrt(25);
^
```

Figure 50: *Error message returned by Lince*

In the previous case, the parser error does not indicate which operation is not supported, giving relatively vague information to the user, who in more complex programs may have difficulty debugging.

As for the detection of mathematical indeterminacies, its implementation was not useful, since there were no indeterminacies for the three arithmetic operations supported by the old version.

In the new version, to parse the mathematical functions (see Table 2) and mathematical constants supported by Lince (see Table 3), the instructions in Fig. 51 have been implemented in the variable "notlinOthers" of the file "Parser.scala".

```
identifier ~ "("~")" ^^ {
        case s ~ _ ~ _  => Func(s, List())
                        } |
identifier ~ opt("(" ~> argsFunction <~ ")") ^^ {
        case s ~ Some(arguments) => Func(s, arguments)
        case s ~ _ => Var("_" + s)
                                                  }
```

Figure 51: *Instructions of variable "notlinOthers" responsible for recognizing mathematical functions and mathematical constants*

The "notlinOthers" variable is responsible for receiving the parser of reals, variables, non-linear expressions between parentheses, mathematical functions applied over non-linear expressions, mathematical constants and the parser of certain mathematical functions that needed to be treated individually. However, as you can see in the Fig. 51, for the variable "notlinOthers" to receive the parsing of the mathematical constants, it expects to receive the parsing of the variable "identifier" followed by the parsing of open and closed parentheses, returned the case class "Func(s,List())", where the "s" is the result of parsing of the "identifier" variable (recalling that the "identifier" variable returns the parsing of the strings that start with a lowercase letter). As for receiving the parsing of mathematical functions applied to non-linear expressions, it can be seen in the Fig. 51 that the variable "notlinOthers" expects to receive parsing from the variable "identifier", followed by a list of the parsing of non-linear expressions between parentheses (the function "argsFunction" returns this list of parsing of non-linear expressions), returning the case class "Func(s,arguments)", where the "s" is the result of parsing the variable "identifier" and "arguments" is the parsing list of non-linear expressions.

(**Note:** The full implementation of variable "notlinOthers" can be found in Appendix A.1.)

For this reason, the way the parser has been implemented for mathematical constants and mathematical functions applied to non-linear expressions does not restrict the names given to them, so the parser will accept any name given to them as long as it conforms to the established grammar. In this way, it was necessary to introduce functions/instructions in the interpreter capable of verifying whether the names given to mathematical constants and mathematical functions are in fact supported by the tool.

As such, the solution found was to treat each of the mathematical functions and mathematical constants supported by the tool individually in the "apply" function of "Eval.scala". As mentioned in Section 4.2, this function was responsible for calculating the numerical result of a given non-linear expression, and whenever there was a case class "$\mathrm{Func}(\ )$", it was checked whether the name and the list received as an arguments of that class corresponded to one of the mathematical functions/constants supported by the tool. If they did, the respective function was applied to the numerical result of the arguments received or the mathematical constant was returned, if one of the two did not correspond, an error message was returned. The version of this "apply" function, with only detection of unsupported mathematical functions/constants, can be found in the Fig. 52.

Having implemented the previous error detection, it was found that it would be also useful to modify the previous function so that it returned an error message whenever a certain indeterminate form was encountered.

These indeterminate forms can occur because there are arithmetic operators and mathematical functions whose domain is limited to a certain range of values.

The indeterminate forms that will be treated are:

- Divide a nonlinear expression by 0;
- Compute the remainder of a nonlinear expression by 0;
- Raising 0 to a negative number (such as $0 \wedge (-1)$);
- Apply an arcsin or an arccos to a number outside its domain, as it can only be contained between -1 and 1;
- Apply a square root to a negative number;
- Apply a log or a $\mathrm{log10}$ to a number negative or equal to 0.

To identify the occurrence of these indeterminate forms, it was necessary to go to the previous function and make the following changes:

- In case of finding a division, check if the divisor is zero. If it is zero, it returns an error, if it is different from 0, it performs the division;
- In case of finding a remainder, check if the divisor is zero. If it is zero, it returns an error, if it is different from 0, it performs the remainder;

```
def apply(state:Point, notlin: NotLin): Double = {
    val res = notlin match {
            case Var(v) => state(v)
            case Value(v) => v
            case Add(l1, l2) => apply(state,l1) + apply(state,l2)
            case Mult(l1,l2)  => apply(state,l1)  * apply(state,l2)
            case Div(l1,l2)  => apply(state,l1) / apply(state,l2)
            case Res(l1,l2)  => apply(state,l1) % apply(state,l2)
            case Func(s,list) => (s,list) match {
              case ("PI",Nil) => math.Pi
              case ("E",Nil) => math.E
              case ("max",v1::v2::Nil) => math.max(apply(state,v1),
                  apply(state,v2))
              case ("min",v1::v2::Nil) => math.min(apply(state,v1),
                  apply(state,v2))
              case ("pow",v1::v2::Nil) =>
                  math.pow(apply(state,v1),apply(state,v2))
              case ("exp",v::Nil) => math.exp(apply(state,v))
              case ("sin",v::Nil) => math.sin(apply(state,v))
              case ("cos",v::Nil) => math.cos(apply(state,v))
              case ("tan",v::Nil) => math.tan(apply(state,v))
              case ("arcsin",v::Nil) =>  math.asin(apply(state,v))
              case ("arccos",v::Nil) => math.acos(apply(state,v))
              case ("arctan",v::Nil) => math.atan(apply(state,v))
              case ("sinh",v::Nil) => math.sinh(apply(state,v))
              case ("cosh",v::Nil) => math.cosh(apply(state,v))
              case ("tanh",v::Nil) => math.tanh(apply(state,v))
              case ("sqrt",v::Nil) => math.sqrt(apply(state,v))
              case ("log",v::Nil) =>  math.log(apply(state,v))
              case ("log10",v::Nil) => math.log10(apply(state,v))
              case (_,_) => throw new RuntimeException(s"Unknown function '${s}(
               ${(list.map(Show.applyV).toList).mkString(",")})', or the number
                   of arguments
               are incorrect")
            }
      }
  res
}
```

Figure 52: *The first version of the function "*apply*"of the "Eval.scala"*

- In the case of raising 0 to a negative number, check if the base is zero and de exponent is a negative number. If that is true, then it returns an error, otherwise, it performs the pow;
- In the case of finding an arcsin , an arccos or a sqrt, check if using respectively one of these three mathematical functions on the numerical result of the argument the result is possible. If the

result is possible (different from Nan) the same is returned, otherwise an error is returned;

- In the case of finding a $\log$ or a $\log 10$, it checks whether the numerical result of the argument was less than or equal to zero. If the result is less than or equal to zero, an error is returned, otherwise the operation is performed.

To implement the previous changes, it was necessary to develop two new functions: "$\mathrm{multOfPi}$" and "$\mathrm{multOfPiOn2}$".

The function "$\mathrm{multOfPi}$" receives a real number and checks if it is a multiple of Pi. Its implementation is represented in Fig. 53.

```
def multOfPi(number: Double): Boolean = {
  val eps = 1e-8 // Define a small value for tolerance
  val res = abs(number % math.Pi)
  // Check if the remainder is within the tolerance range
  return res < eps || abs(res - math.Pi) < eps
}
```

Figure 53: *Function "*$\mathrm{multOfPi}$*"*

On the other hand, the function "$\mathrm{multOfPiOn2}$" receives a real number and checks if it is a multiple of Pi/2. Its implementation is represented in Fig. 54.

```
def multOfPiOn2(number: Double): Boolean = {
  val eps = 1e-8 // Define a small value for tolerance
  val res = abs((number+math.Pi/2) % math.Pi)
  // Check if the remainder is within the tolerance range
  return res < eps || abs(res - math.Pi) < eps
}
```

Figure 54: *Function "*$\mathrm{multOfPiOn2}$*"*

The function "$\mathrm{multOfPi}$", was used in the "apply" function to check if the sine and tangent arguments were multiples of Pi (within a tolerance of $1 * 10^{-8}$) and return zero if so, because in the case of the instructions "$\mathrm{math.sin}(...)$" and "$\mathrm{math.tan}(...)$" if in the argument there was a multiple of Pi, Scala returned a number very close to zero (raised to -16), and in fact this result should be 0. In this way, it was possible to avoid inaccurate results as in the case of dividing 1 by $\sin(\mathrm{pi}())$, which previously gave a number raised to 16 and now returns an error message regarding the detection of an indeterminate form.

The function "$\mathrm{multOfPiOn2}$" was used identically and for the same reasons as the function"$\mathrm{multOfPi}$", except for the fact that the cosine argument was also checked.

(**Note:** To visualize the implementation of the previous changes in the "apply" function, just consult Appendix A.2.)

As mentioned before, the purpose of the "apply" function was to determine the numerical result of the non-linear expressions coming from the parser, and also to detect the presence of unsupported mathematical functions/constants, as well as to detect the indeterminate forms mentioned above. However, there are situations where the nonlinear expressions coming from the parser do not pass through the "apply" function before reaching the SageMath, making it impossible to detect these semantic errors in the desired way and forcing SageMath to evaluate nonlinear expressions that contain these semantic errors.

Because of this, it was decided to force this function to be used every time a nonlinear expression was evaluated before it was sent to SageMath. To do so, the "run" function of the "Traj.scala" file was accessed, which was intended to evaluate the hybrid program (already converted by the Parser) at a given instant of time or number of cycles, and change the function "runAtomicUntilEnd", "runITE" and "runWhile".

In the case of "runAtomicUntilEnd":

- If its an assignment, the "apply" function is applied to its expression;
- In case it is a set of differential equations, the "apply" function is used on both the "For" and "'Until" duration expressions, as well as on each expression within the set of differential equations.

In the case of "runITE":

- Due to the fact that the "run" function was used to evaluate the "then" and "else" programs, it was only necessary to apply the "apply" function to the "if-the-else" condition expressions.

And in the case of "runWhile":

- It was only necessary to apply the "apply" function to the cycle condition expressions because the program that precedes the While loop and the program that the While loop executes are evaluated using function "run", "runITE", and "runAtomicUntilEnd".

Having ensured that the "apply" function is executed on the non-linear expressions that appear during the execution of the program (this execution does not store the output values of the "apply" function, as there is only interest in knowing if there are unsupported mathematical constants, unsupported mathematical functions and indeterminate forms), it is also ensured that these two types of semantic errors are

checked before SageMath starts its work, ensuring a quick and efficient display of error messages for the user.

To exemplify how the detection of these two types of semantic errors works, some examples will be presented.

If we run the following hybrid program, we obtain the error message from Fig. 55.

```
v:=1;
v'=expoent(2) for 2;
```

```
# Unknown function 'expoent(2)', or the number of arguments are incorrect
```

Figure 55: *Error message returned by Lince*

The error message in Fig. 55 indicates that the instruction "$\mathrm{expoent}(2)$" does not exist or the number of arguments is incorrect, which is in line with what was intended because the mathematical function "$\mathrm{expoent}$" is not supported by the tool.

Nevertheless, if the instruction "$\mathrm{expoent}(2)$" is changed to the instruction "$\mathrm{exp}(2{,}0)$", as shown in the following hybrid program, the error message shown in Fig. 56 will occur.

```
v:=1;
v'=exp(2,0) for 2;
```

```
Unknown function 'exp(2,0)', or the number of arguments are incorrect
```

Figure 56: *Error message returned by Lince*

Similar to the error message in Fig. 55, the error message in Fig. 56 indicates that the instruction "$\mathrm{exp}(2{,}0)$" is not supported or the number of arguments is not valid, and since "exp" is a mathematical function supported by the tool and receives only one argument, so the reason this error appears is due to the number of invalid arguments.

Finally, you will get the error message shown in Fig. 57 if you change the instruction "$\mathrm{exp}(2{,}0)$" to the instruction "$\mathrm{exp}(2)/\mathrm{sin}(\mathrm{pi}())$", as shown in the following hybrid program.

```
v:=1;
v'=exp(2)/sin(pi()) for 2;
```

In the previous example, an expression is divided by another expression that has a result equal to zero, so an error message is returned indicating the occurrence of an indeterminate form.

```
Error: the divisor of the division 'exp(2)/(sin(pi))' is zero.
```

Figure 57: *Error message returned by Lince*

4.5.4 Detection of inconsistent results

In addition to detecting the semantic errors mentioned in the previous sections, it was necessary to detect the occurrence of operations that yielded inconsistent results.

These operations involved the use of the mathematical functions "max" and "min" with dynamic variables in their arguments in the differential equations. As mentioned in Section 4.3.2. SageMath is unable to consistently and accurately select the correct result when applying these mathematical functions to arguments that contain dynamic variables. Due to this issue, it was decided to detect the occurrence of these mathematical functions applied to expressions with dynamic variables in the differential equations (after numerical simplifications) and return an error message if they were found.

To detect these occurrences, the function "verify_min_max" was created, which received an atomic instruction and returned "None" if no occurrences were detected, or the differential equation where the first occurrence of these mathematical functions with dynamic variables in their arguments occurred. The Fig. 58 contains the representation of function "verify_min_max".

```
def verify_min_max(at:Atomic):Option[DiffEq]= {
    var diffeqs=at.de.eqs
    var aux:Double=1
    for (diffeq <- diffeqs){
    aux=vars_in_min_max(diffeq.e)
    if (aux>0) return Some(diffeq)
    }
    return None
}
```

Figure 58: *Function "verify_min_max"*

The function "verify_min_max" starts by storing the list of differential equations and then iterates through this list using a "for" loop. Within this loop, it is checked if the problem in question exist in the corresponding expression of the current differential equation being evaluated. If they exist (i.e., if "aux" is greater than zero), that differential equation is returned. If none of the equations contain these instructions, the loop continues until completion, and then "None" is returned.

To check if the expressions contain mathematical functions "max" and/or "min" with dynamic vari-

ables in their arguments, it was necessary to create the function "vars_ in_min_max", which receives a non linear expression and returns a real number.

(**Note:** Due to the dimension of the code, it was decided not to show the function "vars_ in_min_max" in the sequence of the previous paragraph. As such, to see the code you can consult Appendix A.4.)

The function "vars_ in_min_max" begins by determining the type of instruction in the received nonlinear expression. If the type of instruction is a terminal instruction, such as a real number, variable, or mathematical constant, it returns the value 0 since there is no presence of a mathematical function"max" or "min". If the instruction type is an operation that contains arguments (except the mathematical functions "max" and "min"), it adds up the result of applying this function to each of the arguments of the operation, taking advantage of the underlying recursion in this function, but if the instruction is a multiplication or an exponentiation, a preprocessing step is performed before applying recursion to its arguments[5]. If the instruction is a mathematical functions "max" or "min", it adds up the result of applying the "extractTotalVarsLinearExp" function to each of its arguments, returning an integer that indicates whether there are dynamic variables in the arguments (as mentioned in Section 4.5.5, the "extractTotalVarsLinearExp" function performs numerical preprocessing to simplify expressions). In this way, the "vars_in_min_max" function returns a nonzero integer if there is a mathematical function "max" or "min" with dynamic variables in its arguments (after the numerical simplification is performed), and 0 if otherwise.

The next step involved applying the function "verify_min_max" to the atomic instruction of the functions "runAtomicWithTime" and "runAtomicWithBounds", immediately after replacing constant variables with their respective expressions. The system then checked whether the result was a differential equation or "None". If it was a differential equation, an error message was sent to the user indicating the presence of these inconsistent operations in the differential equation. If the result was "None", the intended procedure was performed in the functions "runAtomicWithTime" and "runAtomicWithBounds". For further details about this step, please refer to Section 4.3.2.

To illustrate how the detection of this semantic error works, we will provide some examples.

Running the following program:

```
p:=2; v:=1;
p'=v,v'=max(p,2) for 1;
```

[5] For example, if the instruction is a multiplication of a "max(p, 2)" by 0, the result will be 0 regardless of whether "p" is a constant or dynamic variable. On the other hand, if the instruction was "max(p, 2)∧ 0", even if "p" is a dynamic variable, raising it to the power of 0 always results in 1. Hence, there is no problem if the mathematical function "max" yields inconsistent results in both cases.

The error message from Fig. 59 is obtained.

```
Error: When parsing G§15
p'=v,v'=max(p,2) for 1; — hprog.common.ParserException: It is not possible to apply the
    max or min
functions to expressions with dynamic variables in differential equations:_v'=max(_p,2)
```

Figure 59: *Error message returned by Lince*

In this example the mathematical function "$\max$" was applied to the dynamic variable "p", resulting in the error message from Fig. 59.

However, if the program is changed as it follows:

```
p:=2; v:=1;
p'=v,v'=max(p,2)^0 for 1;
```

The symbolic plot in Fig. 60 is generated.

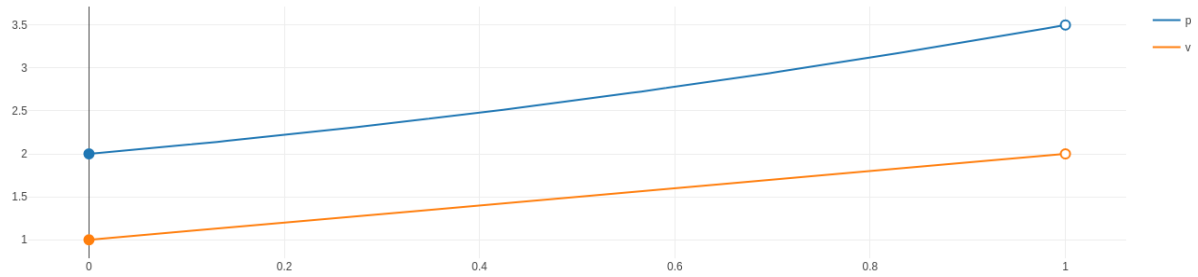


Figure 60: *Symbolic plot resulting from the hybrid program:* $\mathrm{p}{:=}2;\ \mathrm{v}{:=}1;\ \mathrm{p'}{=}\mathrm{v},\mathrm{v'}{=}max(p,2) \wedge 0$ for $1;$

When running the previous program, the pre-processing performed by the "$\mathrm{vars\_in\_min\_max}$" function means that raising an expression to 0 always results in 1, regardless of the base of the exponent. Therefore, the "$\max(\mathrm{p},\ 2)$" is ignored and the symbolic plot in Fig. 60 is obtained.

Finally, changing the program again:

```
p:=2; v:=1;
p'=0,v'=max(p,2) for 1;
```

The symbolic plot in Fig. 61 is obtained.

The only that changed from the first example is that the variable "p" became a constant variable. Therefore, applying the mathematical function "$\max$" to a constant variable does not cause any inconsistencies in obtaining the solution of the differential equation by SageMath. As a result, no error is detected, and the corresponding plot is obtained.

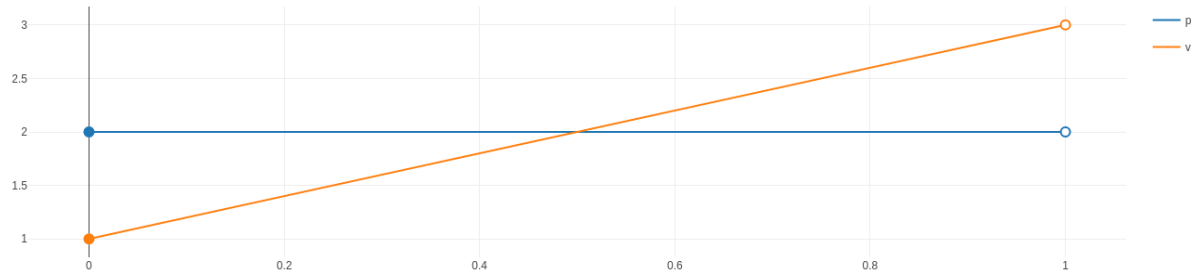


Figure 61: *Symbolic plot resulting from the hybrid program:* p:=2; v:=1; p'=0,v'=$max(p, 2)$ for 1;

4.5.5 Verification of linearity of differential equations

As mentioned in Section 4.3.2, after replacing the constant variables with their expressions and check for the presence of dynamic variables in the mathematical functions "max" and "min" within the differential equations, it was necessary to check if the resulting differential equations were indeed linear (since SageMath is unable to solve a wide range of non-linear differential equations).

For this, it was created the function "verifyLinearityEqsDiff" in the "Utils.scala" file, which receives a program and returns the first occurrence of nonlinear differential equation. The representation of this function can be found in Fig. 62.

```
def verifyLinearityEqsDiff(prog:Syntax):Option[DiffEq] ={
    var diffeqs=extractDifEqs(prog)
    var varsDifEqs=extractVarsDifEqs(prog)
    var iteration=0
    var aux=0
    for (lsteqDiff <- diffeqs){
    var aux=0
    for (eqDiff <- lsteqDiff){
     aux=extractVarsLinearExp(eqDiff.e,varsDifEqs(iteration))
     if (aux > 1 ) return Some(eqDiff)
     }
     iteration=iteration + 1
    }
    return None
}
```

Figure 62: *Function* "verifyLinearityEqsDiff"

The function of the Fig. 62 starts by saving in the variable "diffeqs" the list of lists of differential equations coming from the parser of the program (each list of differential equations refers to the set of differential equations present in each atomic instruction), then it saves in the variable "varsDiffEqs" the list of lists of dynamic variables of each set of differential equations.

After saving the set of differential equations and their respective dynamic variables, variables "aux" and "iteration" were created. The variable "aux" will be used to store the integer referring to the linearity of each set of differential equations and variable "iteration" will be used to scan the list of lists of dynamic variables for each set of differential equations.

Finally, two "for" cycles were performed in order to go to each list of differential equations (or in other words, each set) and check whether or not their expressions were linear, returning the first occurrence of nonlinear differential equation, or returning "None" if all the sets had linear expressions.

To perform this function, it was necessary to create the functions "extractDifEqs", "extractVarsDifEqs", and "extractVarsLinearExp" in the "Utils.scala" file.

As mentioned earlier, the function "extractDifEqs" takes a program and returns the list of lists of differential equations present in the program, where each list of differential equations represents the set of differential equations present in each atomic instruction. On the other hand, the function "extractVarsDifEqs" takes a program and returns the list of lists of dynamic variables present in each set of differential equations. The functions "extractDifEqs" and "extractVarsDifEqs" are represented in Fig. 63 and Fig. 64, respectively.

```
def extractDifEqs(prog:Syntax):List[List[DiffEq]] = prog match {
  case Atomic(as,de) => return List(de.eqs)
  case Seq(Atomic(as,de),q) => List(de.eqs)++extractDifEqs(q)
  case Seq(p,q) => extractDifEqs(p) ++ extractDifEqs(q)
  case While(pre,c,p) =>  extractDifEqs(pre) ++ extractDifEqs(p)
  case ITE(ifP,thenP,elseP) => extractDifEqs(thenP) ++ extractDifEqs(elseP)
}
```

Figure 63: *Function "*extractDifEqs*"*

The function "extractVarsLinearExp", on the other hand, takes a non-linear expression derived from one of the differential equations belonging to a set of differential equations, along with its corresponding list of dynamic variables from that set of differential equations. Its objective is to return an integer that indicates whether the expression is linear, non-linear, or if there is suspicion of it being non-linear (if it is greater than 1 it is non-linear or there is suspicion of it being non-linear, but if it is equal to 1 or 0 it is linear).

(**Note:** Due to the dimension of the code, it was decided not to show the function "extractVarsLinearExp" in the sequence of the previous paragraph. As such, to see the code you can consult Appendix A.5.)

Based on the operations performed within the expression, the function "extractVarsLinearExp" will return an integer indicating its linearity. For example, if the expression is a variable (class "'Var"), the

```
def extractVarsDifEqs(prog:Syntax):List[List[String]] = {
  var eqsdiff=extractDifEqs(prog)
  var listVars:List[List[String]]=List()
  for (lsteqDiff <- eqsdiff){
    var aux:List[String]=List()
    for (eqDiff <- lsteqDiff){
      if (extractTotalVarsLinearExp(eqDiff.e)==0) {
        if (calc_doubles(eqDiff.e)!=0) aux=aux ++ List((eqDiff.v).v)
        else aux=aux
      } else aux=aux ++ List((eqDiff.v).v)
    }
    listVars=listVars ++ List(aux)
  }
  return listVars
}
```

Figure 64: *Function* "extractVarsDifEqs"

function checks if the variable corresponds to one of the dynamic variables present in the argument. It returns 1 if it matches and 0 if it doesn't. However, if the expression corresponds to "2*x+x*x" (where x is a dynamic variable), the function will return the maximum value among each of the terms, which in this case would be 2 since there are two dynamic variables multiplying in the second term.

Although, some operations required special attention, such as division, where the integer corresponding to the linearity of the divisor is multiplied by 2. This is because even if it is a linear expression with one variable per term, the inclusion of the variable in the denominator invalidates the linearity.

To summarise, this function does the following:

- If the expression is a real number, it returns the value 0 because regardless of the real number, the expression remains linear;
- If the expression has a variable, it returns 1 if it is dynamic and 0 if it is constant;
- If the expression is an addition, it returns the maximum integer value obtained by applying this function to each term. The term that returns the highest integer value indicates potential non-linearity;
- If the expression is a multiplication, it checks if the term on the left-hand side contains variables. If it doesn't have, it calculates its numerical value and checks if it is equal to 0. If it is equal to 0, it returns 0 because multiplying an expression by 0 results in 0, which is linear. However, if it is different from 0, it returns the result of applying this function to the right-hand side term, as it may

not be linear. Conversely, if the term on the left-hand side contains variables but the one on the right-hand side does not, a similar process is performed. Finally, if both terms contain variables, the result of applying this function to each term is summed;

- If the expression is a division, the result of applying this function to the dividend term is added to twice the result of applying this function to the divisor term. This ensures that the presence of dynamic variables in the divisor will yield a number greater than 1, indicating that the expression is non-linear or requires further simplification to be linear;

- If the expression is a remainder, the result of applying this function to the dividend term is added to twice the result of applying this function to the divisor term. This also guarantees that the presence of dynamic variables in the divisor will yield a number greater than 1, indicating that the expression is non-linear or requires further simplification to be linear;

- If the expression is a mathematical function or a mathematical constant, the result is obtained from the function "'$\mathrm{funcextract}$", which serves the same purpose as the function "$\mathrm{extractVarsLinearExp}$" but specifically handles mathematical functions and mathematical constants.

Upon analyzing the function "'$\mathrm{funcextract}$", it is apparent that it follows the same methodology as the function "$\mathrm{extractVarsLinearExp}$". However, it is necessary to detail the method adopted for handling expressions that involve exponentiation. In this case, the approach is as follows:

- If the exponent does not contain variables, it is checked whether its numerical value is 0 or 1. If it is 0, the function immediately returns the integer 0 because raising any expression to the power of 0 results in 1, which is always linear. If the exponent is 1, the function "$\mathrm{extractVarsLinearExp}$" is applied to the base term because linearity depends solely on the base term;

- If the exponent contains variables or the numeric value is different from 0 and 1, it is necessary to check whether the base contains variables or not. If the base contains variables (when referring to variables in this context, it always means dynamic variables), the value 2 is returned because an expression with variables raised to another expression with variables (or with a numeric value different from 0 and 1) is nonlinear or requires further simplification to be linear. Even so, if the base does not contain variables, it is verified whether the base term is 1. If it is, the function returns the integer 0 because raising any value to the power of an expression equal to 1 always results in 1, which is always linear. If the base term is different from 1, the function returns the integer 2 because the exponentiation is nonlinear or requires further simplification.

(**Note:** It should be noted that in the implementation of functions "extractVarsLinearExp" and "'funcextract", it was necessary to use the functions "extractTotalVarsLinearExp" and "calc_doubles". The function "extractTotalVarsLinearExp" aims to return a non-zero integer if the expression contains variables (dynamic or constant) after numerical simplifications, and 0 if it does not contain variables. On the other hand, the function "calc_doubles" is responsible for calculating the resulting real value of an expression that does not contain variables. The implementation of both functions can be found in Appendix A.6.)

Returning to the function "verifyLinearityEqsDiff", an important aspect to highlight is how the linearity of a differential equation is determined. As mentioned earlier, each differential equation in the program is scanned using "for" loops, and the function "extractVarsLinearExp" is applied to the expression of each differential equation. Additionally, the list of dynamic variables present in the set of differential equations being analyzed is passed as an argument to the function "extractVarsLinearExp". As previously mentioned, the function "extractVarsLinearExp" returns an integer that represents whether the expression is indeed linear. Based on this integer and using conditional statements, function "verifyLinearityEqsDiff" decides whether to return the differential equation or continue iterating through the "for loops.

In the case where function "extractVarsLinearExp" returns the value 0 or 1, indicating that the expression is linear, function "verifyLinearityEqsDiff" continues iterating through the loops. However, if function "extractVarsLinearExp" returns a number greater than 1, indicating that the expression is nonlinear or requires simplification to assume linearity, the function "verifyLinearityEqsDiff" returns this differential equations. In the case where function "extractVarsLinearExp" never returns an integer greater than 1, function "verifyLinearityEqsDiff" completes the iteration through the differential equations and returns "None".

The functions "'runAtomicWithTime" and "'runAtomicWithBounds" from "Traj.scala" (as described in Section 4.3.2) will verify whether function "verifyLinearityEqsDiff" returns a differential equation or "None". If it returns a differential equation, as discussed earlier, it means that this differential equation is nonlinear or need to be simplified, and the corresponding error message is returned. If it returns "'None" it means that all the differential equations present in the atomic instruction were indeed linear, and the subsequent instructions are followed.

Next, three examples to analyze the performance of detecting nonlinear differential equations will be discussed.

The first example consists of the following program:

```
x:=1; y:=2;
x'=sin(x), y'=1 for 1;
```

By running the previous example, the error message from Fig. 65 is obtained.

```
Error: When parsing G§15
x'=sin(x), y'=1 for 1; — hprog.common.ParserException: There is one differential equation
    that is not
linear or the semantic analyser suspects that it is non—linear (try simplifying the
    differential
equation): _x'=sin(_x)
```

Figure 65: *Error message returned by Lince*

The nonlinearity detection implemented in the tool detected that a differential equation in the previous program showed nonlinear behaviour, and that's why the error message Fig. 65 was returned. It can be observed that the nonlinearity was present in the expression of the first differential equation, as applying a sine to a dynamic variable results in a nonlinear expression.

The second example consists of the following program:

```
x:=1; y:=2;
x'=sin(x)^(sin(pi())), y'=1^x for 1;
```

The symbolic plot of the Fig. 66 is obtained by running the previous example.

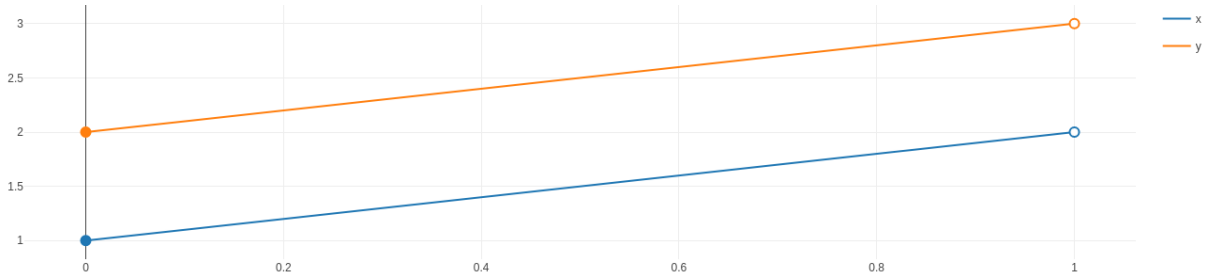


Figure 66: *Symbolic plot resulting from the hybrid program:* $\mathrm{x{:=}1;\ y{:=}2;\ x'{=}sin(x)\wedge(sin(pi())),\ y'{=}1\wedge(x)\ for\ 1;}$

It can be seen that this program is an adaptation of the first example and still contains operations with nonlinear behavior, such as "$\sin(x)$" and raising an expression to a variable. However, the nonlinear detection implemented in the tool is capable of performing numerical simplifications (as discussed earlier). Based on these numerical simplifications, the previous functions, when analyzing the nonlinearity of the expression "$\sin(x)\wedge(\sin(pi()))$", recognized that "$\sin(pi())$" evaluates to 0, and raising any expression (whether linear or not) to 0 always results in 1, which is indeed linear. Similarly, in the next expression, the

function identified that raising the value 1 to "x" is equivalent to 1, and that operation is linear. Therefore, both differential equations were considered linear and sent to SageMath, which determined their solutions. The remaining process can be carried out until obtaining the plots shown in Fig. 66.

The third example consists of the following program:

```
x:=1; y:=2;
x'=sin(x-x), y'=1 for 1;
```

When running the previous example, the error message of the Fig. 67 is obtained.

```
    \begin{lstlisting}[style=error]
Error: When parsing G§150
— hprog.common.ParserException: There is one differential equation that is not
linear or the semantic analyser suspects that it is non—linear (try simplifying the
    differential
equation):
_x'=sin(_x + (-1*_x))
```

Figure 67: *Error message returned by Lince*

It can be observed that running the previous program resulted in an error indicating that the differential equation is nonlinear or need to be simplified. Since the implemented method for detecting nonlinearity only allows for numerical simplifications, symbolic simplifications are not performed. As a result, the expression "$\sin(\text{x-x})$" was considered nonlinear, and the previous error was raised. However, as indicated by the error message, if the user attempts to simplify the previous program, they will eventually eliminate the variables within the sine function, resulting in the linear equations "$\text{x'} = \sin(0)$" and "$\text{y'} = 1$", which can be solved.

4.5.6 Detection of SageMath's inability to solve differential equations

After implementing the detection of variables that were not assigned at the beginning of the program, variables on the right hand side of the initial assignments that were not previously assigned, unsupported mathematical functions/constants, indeterminate forms, inconsistent results and verification of linearity of differential equations, it was found that SageMath was unable to symbolically solve certain differential equations.

For example, if you consider a mechanical system such as a mass-spring-damper system, as shown in the following figure:

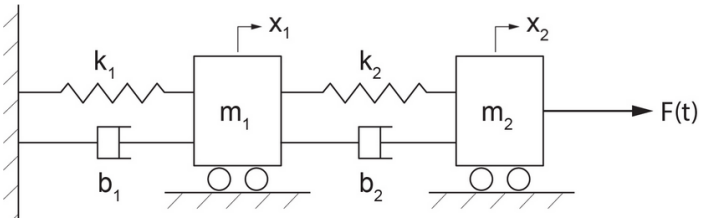


Figure 68: *Mass-spring-damper system [Tab21]*

To simulate through Lince, it is necessary to determine its differential equations. To determine the differential equations, initially it is necessary to determine the forces applied to each mass (in this example, it will be considered that the dampers function as friction applied to each individual block to simplify the analysis):

- In mass 2, the applied forces are the force $F(t)$, the elastic force exerted by spring 2 (F_{k2}), and the frictional force offered by damper 2 (F_{b2});
- In mass 1, the applied forces are the elastic force exerted by spring 1 (F_{k1}), the frictional force offered by damper 1 (F_{b1}) and the elastic force exerted by spring 2 (F_{k2}).

Based on the reference [Alv11], the elastic force applied by a spring is defined as $F_k = k * y$ and the frictional force is $F_b = b * v$, where y is the deformation of the spring, v is the velocity variation, b is the friction coefficient and k is the elasticity constant. Based on the forces applied to each mass and Newton's second law, the following equations are obtained:

- $F(t) - F_{k2} - F_{b2} = m_2 * d^2y_2/dt^2 \Leftrightarrow F(t) - k_2(y_2 - y_1) - b_2(dy_2/dt) = m_2 * d^2y_2/dt^2$
- $F_{k2} - F_{k1} - F_{b1} = m_1 * d^2y_1/dt^2 \Leftrightarrow k_2(y_2 - y_1) - k_1(y_1) - b_1(dy_1/dt) = m_1 * d^2y_1/dt^2$

Rearranging the previous differential equations so that the second derivative is on the left-hand side, the differential equations are now in the following format:

- $d^2y_2/dt^2 = F(t)/m_2 - (k_2/m_2) * (y_2 - y_1) - (b_2/m_2) * dy_2/dt$
- $d^2y_1/dt^2 = (k_2/m_1) * (y_2 - y_1) - (k_1/m_1) * y_1 - (b_1/m_1) * dy_1/dt$

Having the differential equations for each of the masses, the following hybrid program was implemented:

```
m1:=1;
m2:=2;
k1:=3;
k2:=3;
b1:=1;
b2:=1;
y1:=0;
v1:=0;
y2:=0;
v2:=0;
f:=0;

f:=10;
y2'=v2, v2'= f/m2-(k2/m2)*y2+(k2/m2)*y1-(b2/m2)*v2,
y1'=v1, v1'= (k2/m1)*y2-(k2/m1)*y1-(k1/m1)*y1-(b1/m1)*v1 for 10;

f:=0;
y2'=v2, v2'= f/m2-(k2/m2)*y2+(k2/m2)*y1-(b2/m2)*v2,
y1'=v1, v1'= (k2/m1)*y2-(k2/m1)*y1-(k1/m1)*y1-(b1/m1)*v1 for 10;
```

This program consists of pushing mass 2 with a constant force of 10N for 10 seconds and then releasing it.

Running the previous example in Lince, the error message of the Fig. 69 is obtained.

```
Error java.util.NoSuchElementException # null
```

Figure 69: *Error message returned by Lince*

This error message is not very informative, as a user receiving this error would not know what has happened. However, after debugging the outputs received from SageMath, it was found that the tool was unable to solve the systems of differential equations in this example, even though they are linear. This highlights the inability of SageMath to solve certain differential equations symbolically.

Due to SageMath's limitations in solving differential equations, it was decided to improve the error message provided to the user in order to inform him of what occurred and allow him to make his own decisions based on that information.

During the debugging process of the previous example's output, it was observed that SageMath consistently returns a string (that appears in the current version of SageMath as "g1634") whenever it cannot determine the solution to a particular set of differential equations. Therefore, the file "LiveSageSolver.scala" was accessed and the "askSage" function , depicted in the Fig. 70, was modified.

The "askSage" function is responsible for converting a list of differential equations into a format that SageMath can handle, sending it to SageMath, and receiving the result. However, the modification made

```
def askSage(eqs: List[DiffEq]): Option[String] = {
  val instructions = genSage(eqs)
  val rep = askSage(instructions)
  if (rep.get.contains("g1634")) {
    return throw new TimeoutException(s"Sage could not find the solution(s) to
        the differential equation(s): ${Show(eqs)}")
  }
  else {
    return rep
  }
}
```

Figure 70: *Function* "askSage"

was to check if the solution contains the previous string, and if it does, an error message is returned indicating that the user-provided differential equations cannot be solved by SageMath.

(**Note:** Alternative approaches to detect that no good solution was found include checking if unknown functions were used, which are used by SageMath to explain how far it could solve the equations.)

Thus, the error that resulted from running the previous program in Lince has been modified to the error message shown in Fig. 71.

```
Sage could not find the solution(s) to the differential equation(s):
    _y2'=_v2,_v2'=(10/(2)) +
((((-1*3)/(2))*_y2) + (((3/(2))*_y1) + (((-1*1)/(2))*_v2))),_y1'=_v1,_v1'=((3/(1))*_y2) +
((((-1*3)/(1))*_y1) + ((((-1*3)/(1))*_y1) + (((-1*1)/(1))*_v1)))
```

Figure 71: *Error message returned by Lince*

Providing the user with information about the problem that occurred and which differential equations cannot be solved by SageMath.

(**Note:** Even though SageMath is not able to solve the system of differential equations in the above example, this new version of Lince provides the option to obtain the solutions of the differential equations numerically using the numerical plot (as mentioned in Section 4.4). With this new capability, the user can simulate the previous hybrid program using the numerical plot and obtain the plot of the position of mass 1 and mass 2 as illustrated in the Fig. 72.

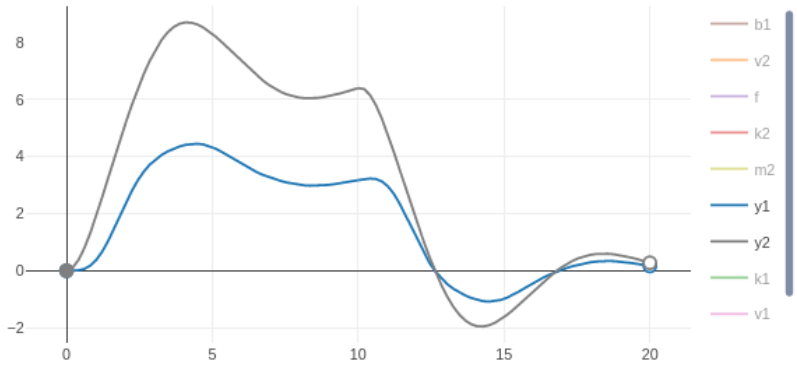


Figure 72: *Position of mass 1 (blue) and position of mass 2 (grey)*

This allows the user to visualize a graph that closely approximates how the positions of both masses vary, as well as other properties such as velocity.)

Chapter 5

Autonomous Driving and Beyond

Autonomous driving is a technology that aims to develop vehicles and transportation systems that move without the intervention of a human driver [ZF] – this includes cars, planes, boats, and even missiles. Depending on the vehicle, this technology offers the following advantages:

- **Driving Safety** – Autonomous driving aims to reduce traffic accidents, primarily caused by human errors, by replacing the human factor with more precise and reactive automated systems [PMP20, SPA23, CNN23];
- **Efficiency and sustainability** – Autonomous vehicles can optimize road/flight paths/waterways usage and improve energy efficiency by reducing congestion and saving fuel [GS15, SPA23, CNN23]. In the case of missiles, a better efficiency in the fuel consumption provides a greater range of distance that it can travel [Dun23];
- **Accessibility and mobility** – Autonomous cars can provide greater access to mobility for elderly individuals, people with disabilities, or those without a driver's license, offering safe and independent transportation options [HHMS16]. In the case of autonomous planes, they provide greater accessibility to disadvantaged communities [For23], while autonomous boats enable the transportation of goods, waste, and passengers in cities with waterways, enhancing accessibility and reducing road traffic [Pub22];
- **Productivity and comfort** – Passengers in autonomous vehicles have the opportunity to utilize travel time more productively by performing tasks, working, or simply relaxing [SBVP19, Vox22].

However, there are some challenges that the autonomous driving industry needs to address. Some of these challenges and areas of focus include:

- **Safety** – Safety is a fundamental concern in autonomous driving. Autonomous driving systems must be capable of making correct decisions and reacting to unforeseen situations in a safe manner.

To improve obstacle detection, to handle adverse weather conditions, and to minimize the possibility of system failures are key areas of research [KW17, MBZ22, Nut]. Think for example what would happen if a missile hits the wrong target such as school or an hospital;

- **Complex decision-making** – Autonomous cars need to be able to make quick and accurate decisions in various traffic situations. This involves the ability to correctly interpret traffic signals, behavior of other vehicles and pedestrians, and to take appropriate actions such as overtaking, lane changing, and negotiating intersections [HDCW+19]. In autonomous planes, decision-making occurs, for example, during challenging landings, adverse weather conditions, and in congested air traffic [For23]. In autonomous boats, decision-making takes place to avoid collisions (with obstacles or other boats), to follow specific routes, and to adapt to real-time changes [CNN23]. In the case of guided missiles, they need to make fast and precise decisions to find a route that allows them to collide with the target;

- **Legal and regulatory issues** – Autonomous driving presents significant legal and regulatory challenges. Establishing responsibilities in case of accidents, determining how autonomous vehicles should be tested and certified, and defining safety standards and norms are areas where the industry is working together with regulatory bodies and lawmakers [II17, Lex23, IIIR22];

- **Privacy and cybersecurity** – As vehicles become increasingly connected and reliant on computational systems, privacy and cybersecurity become important concerns. It is crucial to protect autonomous vehicles against cyber attacks and ensure the security of data collected by sensors and onboard systems [YGA15, Mar23, CCD]. In the case of missiles, a cyber attack can jeopardize the safety of hundreds of people and potentially lead to retaliatory actions between nations [Wir21];

- **Public acceptance** – Public acceptance of autonomous driving is another significant challenge. Convincing users that autonomous vehicles are safe, reliable, and capable of surpassing the human driving experience is essential for widespread adoption. The industry is focused on informing the public about the potentials and advantages of autonomous driving, as well as creating a positive user experience [KHdW15].

Some practical examples of each of the challenges and areas that the autonomous driving industry faces are:

- **Safety** – The use of advanced emergency braking technologies to prevent collisions [DBN+94], the development of driver monitoring systems to detect signs of distraction or drowsiness [NTB17],

implementation of collision avoidance systems for autonomous planes [LDVF+11] and the implementation of algorithms that enhance the ability of missiles to avoid obstacles [HP22];

- **Complex decision-making** – The use of data fusion algorithms from various sensors to improve the accuracy of driving decisions [SKDE23], implementation of reinforcement learning systems to enhance the learning and adaptation capabilities of autonomous cars [KST+21], the design of algorithms that model the trajectory of a missile to collide with a target [Sec12] and optimization of flight routes based on weather conditions [SH14];
- **Legal and regulatory issues** – To establish regulatory frameworks and liability frameworks to determine the obligations and responsibilities of autonomous vehicle manufacturers and users, and collaboration between technology companies and regulatory authorities to establish safety and interoperability standards for autonomous vehicles [II17];
- **Privacy and cybersecurity** – Development of encryption and authentication solutions to secure communication systems in autonomous cars [DZX+21] and the implementation of intrusion detection systems to identify and mitigate cyberattacks targeting autonomous vehicles [WLX+19];
- **Public acceptance**- To conduct pilot programs and demonstrations for the public to experience autonomous driving [FCK+22].

We will see next that the adaptation of Lince to handle Newtonian systems, makes possible to model several of the underlying systems of autonomous driving. The use of Lince to model these systems allows for safe and cost-effective testing, evaluation of different scenarios, and refinement of configurations, aiding in mitigating a considerable range of autonomous driving challenges. Through this tool, the Automatic Emergency Braking (AEB) and the Adaptive Cruise Control (ACC) were modeled, as well as the modeling of a guided missile trajectory planning system. The detailed explanation of the methods used and the program developed can be found in the following section, along with the treatment of other systems not directly associated with autonomous driving.

5.1 Automatic Emergency Braking

The AEB is primarily associated with the “Safety” challenge in autonomous driving. AEB is a safety system that uses sensors such as radars, cameras, infrared or ultrasound to detect objects in front of the vehicle and, in the event of an imminent collision, automatically activates the brakes to prevent or mitigate the impact [Acu].

Figure 73: *Representative image of a vehicle with AEB [Ack22].*

AEB aims to reduce collisions and minimize the severity of accidents. It helps to avoid or reduce the vehicle's speed in emergency situations where the driver may not have enough time to react or apply the brakes appropriately [Wes22].

In an attempt to test the performance of an AEB for various scenarios and understand how its configurations can affect the final outcome, a program was implemented in the new version of Lince with the capability of simulating a situation where an AEB is used.

In the program, a situation where two vehicles are involved was simulated, one controlled entirely by a driver and the other with autonomous driving capability and equipped with AEB. It is assumed that both vehicles start from the same location and at the same speed, and their ability to accelerate and brake is exactly the same. Additionally, it was considered that both the driver and the autonomous driving system intended to accelerate the vehicles whenever possible (an emergency situation that required more aggressive driving). The appearance of a visible obstacle at a certain distance from the vehicles was introduced, and the ability of the vehicles to avoid the collision was simulated.

During this simulation, it was assumed that the time required for the driver to assess the situation and make a decision (brake or accelerate) was 100 times slower than the time required by the AEB.

The program was started by defining the position, velocity, and acceleration of both vehicles and the obstacle (the obstacle is stationary at a distance of 40 meters):

```
p:=0; v:=10; // Initial position and velocity of the autonomous vehicle with AEB
ph:=0; vh:=10; // Initial position and velocity of the  vehicle, without AEB, controlled by a
    driver.
pl:=40; vl:=0; al:=0; // Initial position, velocity and accelaration of the obstacle.
```

Next, the value of the deceleration of the vehicles when braking (braking capacity) and the value of the acceleration of the vehicles when accelerating (acceleration capacity) were established:

```
aT:=-8; aA:=8; // Braking and accelerating acceleration of the vehicles
```

The next step consisted of defining the time required for both vehicles to make a decision (react and act), and three constants ("temp", "aux", and "c") were created that will be useful in the development of the program:

```
sampling_time:=0.01; // Time taken by the AEB to make a decision.
```

```
reaction_time:=1; //  Time taken by the human driver to make a decision.
// counter and auxiliar variables
c := 1;
temp := aA;
aux:=aA;
```

Having assigned the necessary variables, a while loop was implemented to execute a specific program until both vehicles came to a stop (specifically the velocity of both vehicles needs to be 0 m/s or less because the program performs calculations based on *samples* and not continuously, so it is considered that the loop finishes its execution only when the velocities are 0 m/s or less to avoid the risk of the velocity value always being different from 0 m/s and the loop never ending). The following code fragment consists of evaluating the decision of the vehicle without AEB.

```
// Vehicle without AEB (reaction is 100 times less)
  if (c == 100 && temp!=aT)
  then {
      c := 1;
      if ((ph + vh*reaction_time + aA/2*reaction_time^2 <
           pl+vl*reaction_time+al/2*reaction_time^2 )
      && (((vh+aA*reaction_time-vl-al*reaction_time)^2 -
           4*(ph+vh*reaction_time+aA/2*(reaction_time^2)-
      (pl +vl*reaction_time+al/2*(reaction_time^2)))*(aT/2-al/2))<0))
      then {
           temp := aA;
           }
      else {
           temp := aT;
           }
      }
  else {
       c := c + 1;
       }
```

(**Note**: In real-life scenarios, obtaining the values of position, velocity, and acceleration for both vehicles is accomplished through sensors. There are sensors designed to measure internal properties of the vehicle itself (e.g., a speedometer), as well as sensors responsible for measuring properties external to the vehicle (e.g., a parking sensor). It is upon these measurements that the AEB performs the necessary processing.)

The program executed by the loop consisted of evaluating the decision of the vehicle without AEB (controlled entirely by the driver) and the vehicle with AEB based on the distance to the obstacle, and then acting according to the decisions.

To evaluate the decision of the vehicle without AEB, the one fully controlled by the driver, the program initially checked if the counter "c" was 100 (because the time the driver takes to make a decision was 100

times longer than that of the autonomous vehicle) and if the variable "temp" was different from the value of the acceleration during braking (the variable "temp" stores the decision made by the driver, which can be to brake or accelerate). If both conditions were met, it was checked whether the vehicle needed to brake or not. If the vehicle did not need to brake, the value of the acceleration for the case of acceleration was stored in the variable "temp". But if braking was necessary, the value of the acceleration for the case of braking was stored in the variable "temp". If the value of "c" was different from 100, the driver did not make any decision, and the variable was incremented by one because it was still within the time interval to wait for the driver to make a decision. In the case of the previous decision of the driver being to brake, the variable "temp" had the value "aT" (acceleration for braking), so the counter was incremented because it was no longer necessary to reevaluate the driver's decision (after initiating the braking, the car will continue to brake until it stops, without the need to evaluate a new decision).

The verification of whether the vehicle needs to accelerate or brake to avoid colliding with the obstacle is based on the use of motion equations and the application of the discriminant binomial. The verification consists of two conditions. The first condition determines whether the vehicle's position at the end of the decision-making time (the position it will be in when the driver makes the next decision), in the case of accelerating, will be lower than the position of the obstacle. To perform this condition, the following steps were taken:

1. New position of the vehicle:

```
ph+vh*reaction_time+aA/2*reaction_time^2
```

2. New position of the obstacle (in this example, since the obstacle is stationary, the velocity "vl" and acceleration "al" are always 0):

```
pl+vl*reaction_time+al/2*reaction_time^2
```

3. Condition (new position of the vehicle < new position of the obstacle):

```
ph+vh*reaction_time+aA/2*reaction_time^2 <
    pl+vl*reaction_time+al/2*reaction_time^2
```

The second verification consists of checking whether the vehicle's position will eventually surpass the position of the obstacle (collide) if the driver decides to accelerate and then decides to brake at the end of the decision-making time. To perform this condition, the following steps were followed:

1. New position of the vehicle if it accelerates:

```
ph+vh*reaction_time+aA/2*reaction_time^2
```

2. New velocity of the vehicle if it accelerates:

```
vh+aA*reaction_time
```

3. Position at time "t" if the driver decides to brake in the next decision-making:

```
(ph+vh*reaction_time+aA/2*reaction_time^2)+(vh+aA*reaction_time)*t + aT/2*t^2
```

4. New position of the obstacle at the end of the decision-making time:

```
pl+vl*reaction_time+al/2*reaction_time^2
```

5. New velocity of the obstacle at the end of the decision-making time:

```
vl+al*reaction_time
```

6. Position at time "t" of the obstacle after the decision-making time:

```
(pl+vl*reaction_time+al/2*reaction_time^2)+(vl+al*reaction_time)*t + al/2*t^2
```

7. To check if there is any moment in time where the vehicle surpasses the obstacle, the expressions 3 and 6 are equalised, their discriminant binomial is extracted, and it is checked whether there are no solutions (i.e., if the discriminant is less than zero):

```
((vh+aA*reaction_time-vl-al*reaction_time)^2-4*(ph+vh*reaction_time+
aA/2*reaction_time^2-pl+vl*reaction_time+al/2*reaction_time^2)*(aT/2-al/2))<0
```

The next step consisted of evaluating the decision of the vehicle with AEB. To do this, it is first checked if the variable "aux" (which stores the decision of the vehicle with AEB) was different from the value of the braking acceleration. If it was different from the braking acceleration, it would mean that the vehicle had chosen to accelerate instead of braking, a similar condition to the one explained earlier was used to determine if the vehicle needed to brake or accelerate and the decision was stored in the "aux" variable. If the vehicle had chosen to brake in the previous evaluation, no new decision was evaluated, and the vehicle continued braking. The implementation of the evaluation of the decision of the vehicle with AEB is as follows:

```
// Vehicle with AEB
 if (aux!=aT)
 then {
     if ((p + v*sampling_time + aA/2*sampling_time^2 <
          pl+vl*sampling_time+al/2*sampling_time^2 )
     && (((v+aA*sampling_time-vl-al*sampling_time)^2 -
          4*(p+v*sampling_time+aA/2*(sampling_time^2)-
     (pl +vl*sampling_time+al/2*(sampling_time^2)))*(aT/2-al/2))<0))
     then {
```

```
            aux:=aA;
           }
      else {
            aux:=aT;
           }
       }
  else skip;
```

Finally, it is necessary to use systems of differential equations to simulate the trajectory of the vehicles based on their previous decisions (Eq. (3.1) was used to simulate the trajectory of both vehicles, as it is working in 1D), and as well in the simulation of the position of the object over time. For this purpose, it was checked if both, the vehicle without AEB and the vehicle with AEB, had speeds equal to or less than 0 m/s (that is, if they have already come to a stop). If their speeds are less than or equal to 0 m/s, a system of differential equations is run in which the velocity and position of both vehicles remain unchanged. If one of them has a speed less than or equal to 0 m/s, another system of equations is run where the velocity and the position of that vehicle do not change, while the velocity and position of the other vehicle will vary according to the decision it had made. If both vehicles have positive speeds, then a system of equations is run where the position and velocity of both vehicles vary according to their previous decisions. The following code represents the implementation of the systems of differential equations:

```
// Action
  if (vh<=0)
  then {
        if (v<=0)
        then  p'=0,v'=0,ph' = 0, vh' = 0, pl'=vl,vl'=al for sampling_time;
        else  p'=v,v'=aux,ph' = 0, vh' = 0, pl'=vl,vl'=al for sampling_time;
       }
  else {
        if (v<=0)
        then p'=0,v'=0,ph' = vh, vh' = temp, pl'=vl,vl'=al for sampling_time;
        else p'=v,v'=aux,ph' = vh, vh' = temp, pl'=vl,vl'=al for sampling_time;
       }
```

(**Note:** The program aimed at determining the decision-making of the AEB-equipped vehicle, as well as the program implementing the systems of differential equations, is evaluated during the "sampling_time". The reason for evaluating the decision-making of the AEB-equipped vehicle during this decision time is because it is associated with this vehicle. However, the reason for evaluating the system of equations based on the "sampling_time" is because it is the smallest of the decision times, encompassing all actions of both the AEB-equipped and non-AEB-equipped vehicles.)

(**Note:** The complete implementation of this program can be found in Appendix A.7.)

When executing the entire program in Lince, you get the plot from Fig. 74.

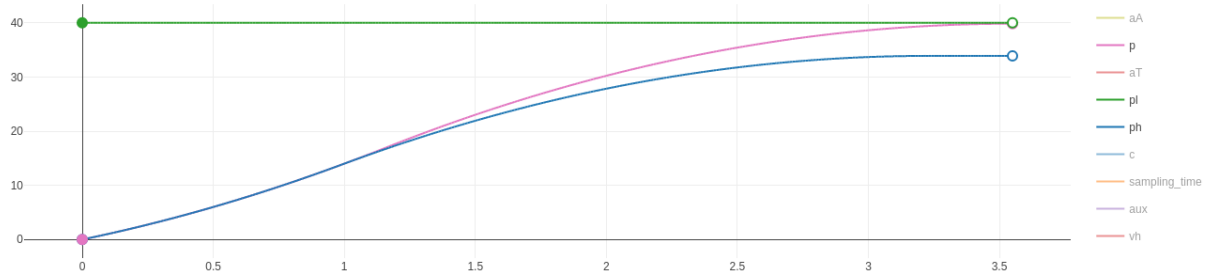


Figure 74: *Position of the vehicle with AEB (pink), position of the vehicle without AEB (blue), and object at 40 meters (green).*

It can be seen in the Fig. 74 that both vehicles manage to avoid the collision. However, the vehicle equipped with AEB is able to stop very close to the object, while the vehicle without AEB had to apply the brakes earlier to avoid the collision, stopping further away from the object

If the position of the object is changed so that it is 30 meters away from the vehicles, the result of the simulation is depicted in Fig. 75.

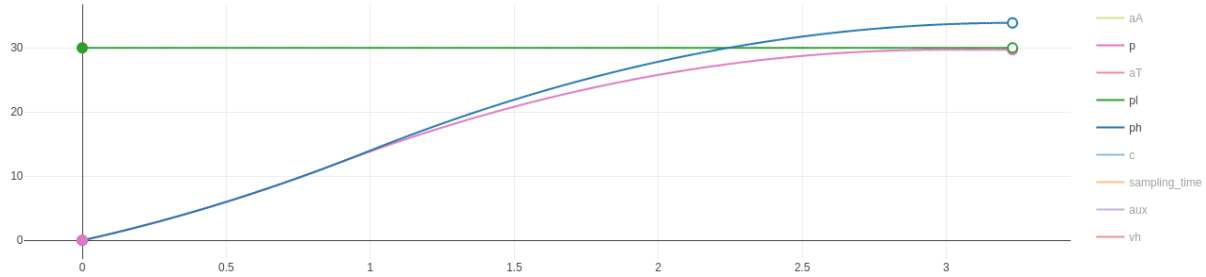


Figure 75: *Position of the vehicle with AEB (pink), position of the vehicle without AEB (blue), and object at 30 meters (green).*

The vehicle with AEB still has the ability to avoid the collision as it has a shorter decision time and therefore started braking earlier. The vehicle without AEB only started braking after 1 second, which corresponds to its decision time, which was not enough to prevent the accident.

If the initial distance to the obstacle is reduced to 5 meters, the result of the simulation becomes as shown in Fig. 76.

The collision happens simply because the vehicle starts with a too high velocity to completely stop before hitting the obstacle.

From this simulation, it is clear that AEB can significantly reduce accidents compared to conventional braking. In the given circumstances and with the specific vehicle, the AEB-enabled vehicle can prevent collisions with objects at a distance greater than 6.25 meters, while the vehicle without AEB can only prevent collisions with objects more than 33.89 metres away.

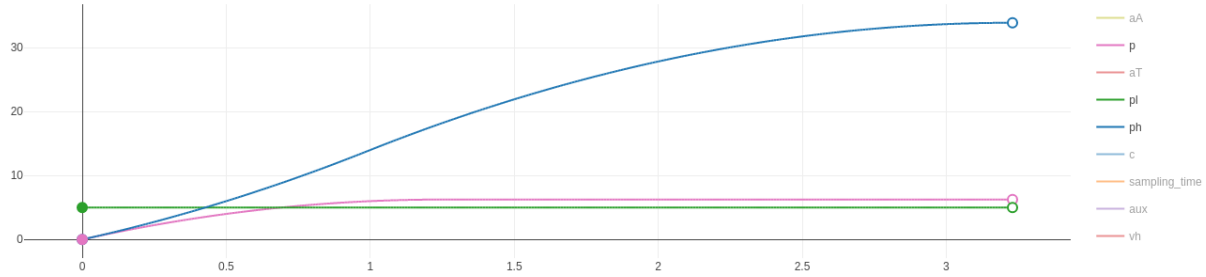


Figure 76: *Position of the vehicle with AEB (pink), position of the vehicle without AEB (blue), and object at 5 meters (green).*

(**Note:** Remember that this example is based on the assumption that the vehicle without AEB is solely controlled by the driver, who decides to either accelerate or brake in exactly the same way as the vehicle with AEB. To determine whether braking or accelerating, the current position, velocity, and acceleration of both the vehicle and the object (based on sensors present in the vehicle) must be known, and calculations and comparisons based on the equations of motion must be performed. In real life, the driver operating a car without AEB will subjectively evaluate whether braking is necessary or not based on the speedometer and distance to the object. This makes the decision entirely dependent on human perception, which can lead to more drastic outcomes than those obtained in this simulation. Therefore, it is assumed that the impact of AEB would be even greater in real life.)

Through this program, the performance of the AEB can be analyzed in other scenarios, such as:

- Varying the initial velocities of both vehicles;
- Modifying the values of acceleration and braking;
- Changing the decision times of both vehicles.

This analysis helps to understand how these factors impact the behavior of the system.

5.2 Adaptive Cruise Control

The ACC is primarily associated with the "Complex decision-making" challenge in autonomous driving. Based on [Col23], ACC is a driver assistance system that uses a combination of sensors, radars, and cameras to monitor the distance to the vehicle ahead and automatically adjust the vehicle's velocity to maintain a safe distance. The main goal of ACC is to reduce driver fatigue, enhance safety, and improve fuel efficiency on highways and long-distance traveling. It allows the vehicle to maintain a constant velocity

and automatically adjusts it according to the traffic ahead, relieving the need for the driver to manually control the accelerator and brakes in congested or variable traffic situations.

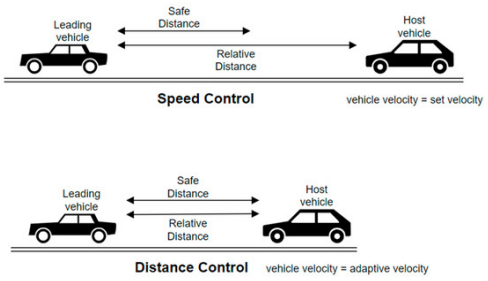


Figure 77: *Representative image of a vehicle with ACC [PP22].*

In an attempt to test the performance of an ACC for various scenarios and comprehend how its configurations can affect the outcome, a program was implemented in the new version of Lince capable of simulating a situation where an ACC is used. This program involves a vehicle with ACC that aims to maintain a constant velocity. However, another vehicle appears in front of it with constant velocity, forcing the ACC to adjust its vehicle's velocity to maintain a safe distance.

The program was started by assigning the initial position and velocity of the vehicle with ACC, as well as the initial position and velocity of the vehicle in front:

```
p:=0; v:=20; // Initial position and velocity of the vehicle with ACC.
pl:=30; vl:=10; // Initial position and velocity of the vehicle in front.
```

Next, the braking acceleration ("aT") of the vehicle with ACC, as well as the acceleration ("aL") of the vehicle in front, are assigned:

```
aT:= -8; aL:=0;
```

The next step involved assigning the desired safety distance that the vehicle should maintain from the vehicle in front ("safety_distance") and the value of the decision-making time by the ACC ("sampling_time"):

```
safety_distance:=10;
sampling_time:=0.01;
```

Finally, a while loop is executed, running the program until reaching the time limit or the maximum number of cycles allowed by Lince. The program within the while loop begins by determining whether the ACC needs to brake or maintain the velocity in order to keep a safe distance from the car in front (this verification uses the same method as in the previous example, however, in this case, it was necessary to subtract the safe distance from the position of the front vehicle and consider that the vehicle maintains its velocity instead of accelerating, replacing "aA" with 0). If the verification determines that there are conditions to maintain the velocity, a set of differential equations of motion in 1D (see the set of differential

equations 3.1) is executed with the acceleration of the ACC vehicle as 0 m/s^2, during the decision-making time. If the verification determines that braking is necessary, a set of differential equations of motion in 1D is executed with the acceleration of the ACC vehicle as "aT", during the decision-making time as well. In both cases, the set of differential equations of motion in 1D for the vehicle in front remains the same, with its acceleration being equal to "aL".

The while loop and the program executed by it are as follows:

```
while (true) do {
    if ((p + v*sampling_time < (pl-safety_distance) + vl*sampling_time + aL/2*sampling_time^2)
    && (((v-vl + (-aL)*sampling_time)^2 - 4*(p-(pl-safety_distance) + (v-vl)*sampling_time +
    (-aL)/2*sampling_time^2)*(aT-aL)/2 ) <0))
    then p'=v,v'=0,pl'=vl,vl'=aL for sampling_time;
    else p'=v,v'=aT,pl'=vl,vl'=aL for sampling_time;
                }
```

When executing the entire program in Lince, you obtain the plot from Fig. 78.

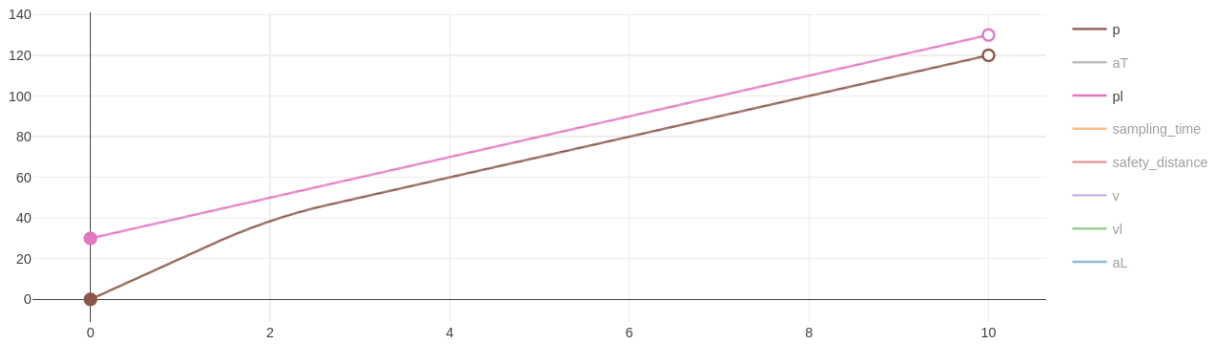


Figure 78: *Position of the vehicle with ACC (brown) and position of the vehicle in front (pink).*

(**Note:** The complete implementation of this program can be found in Appendix A.8.)

Based on this hybrid program, it can be observed that the ACC-equipped vehicle can regulate its velocity to maintain a safe distance from the front vehicle (if the vehicle with ACC and the vehicle in front have the characteristics defined in the initial conditions).

If the initial velocity of the ACC-equipped vehicle is changed to 35 m/s, the simulation result for the first 2 seconds is as depicted in Fig. 79.

In this simulation the system was unable to avoid the collision between the vehicles even though it started braking as quickly as it could. The reason for the collision is that the braking capability of the ACC-equipped vehicle is not sufficient to maintain the safe distance and avoid collision when trying to maintain an initial constant speed of 35 m/s. Simulating cases where the system fails is equally important to understand the contributing factors.

In addition to varying the initial speed of the ACC-equipped vehicle, it's important to vary other param-

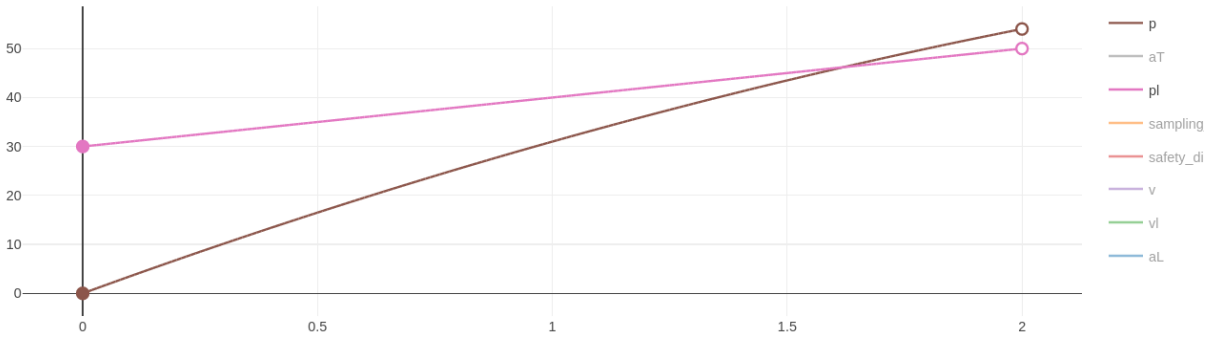


Figure 79: *Position of the vehicle with ACC (brown) and position of the vehicle in front (pink), if the initial velocity of the vehicle with ACC was changed to 35 m/s*

eters to understand their influence on this system. These changes include:

- Varying the initial position of the ACC-equipped vehicle;
- Varying the position and initial velocity of the front vehicle;
- Varying the braking capability of the ACC-equipped vehicle;
- Varying the following distance;
- Varying the decision time.

5.3 Missile vs Target

In addition to systems related to autonomous vehicle driving, we also have the trajectory planning that a guided missile must follow to intercept the target. These missiles are designed to track the location of a moving target (using, for example, radars or thermal signatures) in order to accurately hit it, without human intervention [ABC22].

A hybrid program was thus developed in Lince to simulate a system in 2D capable of computing the missile's trajectory to intercept the target, testing its behavior in different scenarios.

To model this system, the following characteristics were considered:

- The missile and the target can only move forward at a constant speed and perform turns with predefined magnitudes (to simplify);
- Besides its own parameters, the missile only has access to the target's position and velocity at each moment, without being able to predict its movement;

- The missile travels at a higher speed than the target.

With knowledge of the aforementioned characteristics, a strategy was developed to determine when the missile needs to move forward, turn right, or turn left in order to hit the moving target. Since the missile can only know the target's position and velocity at each moment, the strategy to determine the action the missile should take involves relating the target's position and velocity to its own position and velocity.

Due to this, the concept of relative position and velocity between the target and the missile was used. As mentioned in [Gee], position and velocity are concepts that involve a reference. For example, if it is said that car A is traveling at 100 km/h, it generally means that the car is traveling at 100 km/h relative to a pre-determined point at surface of the Earth. Based on this concept, and since the missile has access to its position relative to the point E (point E represents a given point on the surface of the Eart) ($\vec{P}_{ME}$) and the target's position relative to the same reference ($\vec{P}_{TE}$), the target's position relative to the missile ($\vec{P}_{TM}$) was calculated as follows:

$$\vec{P}_{TM} = \vec{P}_{TE} - \vec{P}_{ME} \tag{5.1}$$

Similarly, the target's velocity relative to the missile was calculated in a similar manner:

$$\vec{V}_{TM} = \vec{V}_{TE} - \vec{V}_{ME} \tag{5.2}$$

(**Note:** Whenever the missile or the target are mentioned to be in a certain position or velocity without specifying the reference, it is understood that the reference is point E.)

(**Note:** The graphs involving the positions and velocities of both the missile and the target will only consider scalar values in order to represent both vectors on the same graph, facilitating their analysis.)

To exemplify, if the missile (point M) is at the position $\vec{P}_{ME} = (0m, 20m)$ with velocity $\vec{V}_{ME} = (20m/s, 15m/s)$ and the target (point T) is at the position $\vec{P}_{TE} = (40m, 50m)$ with velocity $\vec{V}_{TE} = (10m/s, 7.5m/s)$, using Eq. (5.1) the target's position relative to the missile will be $\vec{P}_{TM} = (40m, 30m)$ and using Eq. (5.2) the target's velocity relative to the missile will be $\vec{V}_{TM} = (-10m/s, -7.5m/s)$, as shown in the in Fig. 80.

The direction of the missile and the target is defined by the vectors $\vec{V}_{ME}$ and $\vec{V}_{TE}$, respectively, and the vector $\vec{P}_{TM}$ represents not only the position of the target relative to the missile but also the direction that in that moment optimizes the minimization of the distance between the missile and the target.

We observe from the previous graph that the directions of the missile, the target, and the vector $\vec{P}_{TM}$ are the same, meaning that at the moment when the missile's decision-making system analyzes this situation, it will recognize that the direction in which the missile is moving is the same as the direction of the target and the direction that currently optimizes the minimization of the distance between the two

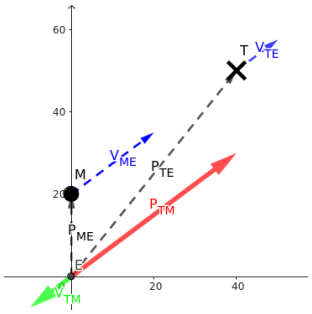


Figure 80: *Graphical representation of the vectors $\vec{P}_{ME}$, $\vec{V}_{ME}$, $\vec{P}_{TE}$, $\vec{V}_{TE}$, $\vec{P}_{TM}$, and $\vec{V}_{TM}$ [Geo22]*

objects. Therefore, a good decision by the decision-making system would be to make the missile move forward.

However, if the target were to travel with a velocity of $\vec{V}_{TE} = (15m/s, 0)$, the graph would change to that of Fig. 81.

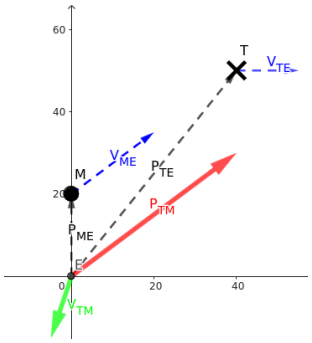


Figure 81: *Graphical representation of the vectors $\vec{P}_{ME}$, $\vec{V}_{ME}$, $\vec{P}_{TE}$, $\vec{V}_{TE}$, $\vec{P}_{TM}$, and $\vec{V}_{TM}$, with a different vector $\vec{V}_{TE}$ [Geo22]*

With the previous change, the direction of the missile remains the same as the direction of the vector $\vec{P}_{TM}$, but now it differs from the direction of the target. By analysing this situation, the missile's decision-making system could decide to let the missile move forward, since it was already in the direction that currently optimises the minimisation of the distance between itself and the target. However, by analysing the direction of the target, it becomes clear that the target is moving in a different direction from the missile and the vector $\vec{P}_{TM}$, and in the next moment the target will move in that direction or a direction close to it (if it curves). Therefore, a good decision by the system would be to make the missile curve to the right, thus shortening the distance between the missile and the target in the next iteration.

If, in addition to changing the target's velocity to $\vec{V}_{TE} = (15m/s, 0m/s)$, the missile's velocity is changed to $\vec{V}_{ME} = (30m/s, 0m/s)$, the graph would change to that of Fig. 82.

With the two previous changes, the missile's direction becomes the same as the direction of the target,

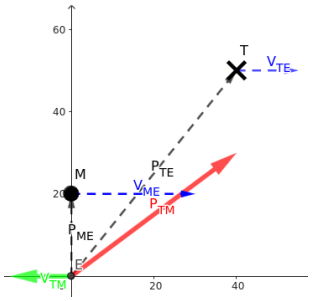
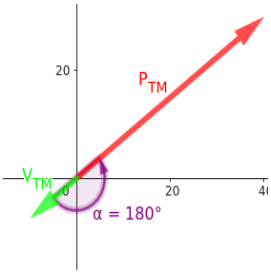


Figure 82: *Graphical representation of the vectors $\vec{P}_{ME}$, $\vec{V}_{ME}$, $\vec{P}_{TE}$, $\vec{V}_{TE}$, $\vec{P}_{TM}$, and $\vec{V}_{TM}$, with a different vector $\vec{V}_{TE}$ and $\vec{V}_{ME}$ [Geo22]*

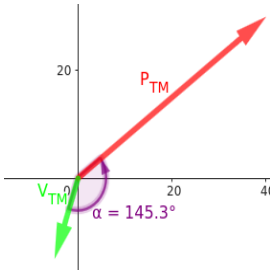
but different from the direction of the vector $\vec{P}_{TM}$. In this case, the missile's decision-making system would have to change the missile's direction to hit the target, so the best option would be to turn left.

From the conclusions drawn from the previous three examples, it is clear that the missile's decision-making system can rely on vectors $\vec{V}_{ME}$, $\vec{V}_{TE}$ and $\vec{P}_{TM}$ to determine whether the missile must move forward, turn right or turn left to successfully intercept the target. However, subtracting the vector $\vec{V}_{ME}$ from the vector $\vec{V}_{TE}$ results in the vector $\vec{V}_{TM}$ (as mentioned earlier), which is nothing but the vector of the target's velocity relative to the missile. Therefore, it's sufficient to consider the vector $\vec{V}_{TM}$ and the vector $\vec{P}_{TM}$ to determine the decision that the missile should take.

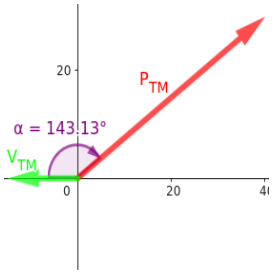
The key relationship between the vector $\vec{V}_{TM}$ and vector $\vec{P}_{TM}$ that allows to determine the missile's decision is the minimum angle α between them (going from $\vec{V}_{TM}$ to $\vec{P}_{TM}$). Adding this angle to the previous graphs, we obtain the graphs shown in Fig. 83.



(a) Angle α between the vector $\vec{V}_{TM}$ and $\vec{P}_{TM}$ in Fig. 80 [Geo22]



(b) Angle α between the vector $\vec{V}_{TM}$ and $\vec{P}_{TM}$ in Fig. 81 [Geo22]



(c) Angle α between the vector $\vec{V}_{TM}$ and $\vec{P}_{TM}$ in Fig. 82 [Geo22]

Figure 83: Angle α between the vector $\vec{V}_{TM}$ and $\vec{P}_{TM}$ in the previous examples

Based on the graphs in Fig. 83, it is observed that when the missile needs to move forward the angle α is 180 degrees, and when it needs to turn, the angle α is different from 180 degrees.

The determination of the angle α between the vector $\vec{V}_{TM}$ and vector $\vec{P}_{TM}$ can be achieved using the dot product as follows [Cue]:

$$\alpha = \arccos\left(\frac{\vec{V}_{TM} \cdot \vec{P}_{TM}}{|\vec{V}_{TM}||\vec{P}_{TM}|}\right) \tag{5.3}$$

Based on this graphs, it was also observed that when the missile needs to turn right, the orientation of the angle α is positive (counterclockwise), but when it needs to turn left, the orientation of the angle α is negative (clockwise).

The determination of the orientation of the angle α can be done using the cross product as follows [ORI]:

$$\begin{cases} \vec{V}_{TM} \times \vec{P}_{TM} >= 0, & \text{turn right;} \\ \vec{V}_{TM} \times \vec{P}_{TM} < 0, & \text{turn left.} \end{cases} \tag{5.4}$$

Based on this analysis, the strategy for determining the missile's next iteration action is as follows:

- Determine the position of the target relative to the missile ($\vec{P}_{TM}$) using Eq. (5.1);
- Determine the velocity of the target relative to the missile ($\vec{V}_{TM}$) using Eq. (5.2);
- Calculate the angle α using the Eq. (5.3);
- If the angle α is between 179.5 degrees and 180.5 degrees, the missile needs to move straight[1]. If it is outside this range, the missile needs to curve;
- The direction of the curve will be obtained from Eq. (5.4).

This strategy for executing the trajectory to intercept the target with the missile is effective regardless of the initial positions of both, but it does not necessarily model the shortest trajectory in all cases. Using the orientation of the angle α to decide which direction to curve will lead the system to organize itself in such a way that α tends towards 180 degrees, and in some cases, other decisions could potentially shorten the trajectory required for the missile to collide with the target. Although it is not the most efficient strategy, it offers a reasonable balance between effectiveness and simplicity. Therefore, it was decided to implement a program in Lince that simulates in 2D a system capable of computing the trajectory of the missile to collide with the target based on this strategy; and its behaviour was analysed for different scenarios.

The program starts with the assignment of the initial position and velocity of the missile and the target:

[1] Ideally, the missile should move straight if the angle were 180 degrees. However, as Lince can only evolve and evaluate the system at discrete time intervals rather than continuously, it was necessary to consider an angular range within which the missile should move straight.

```
// Initial position and velocity of the missile
x:=300; vx:=20;
y:=300; vy:=0;
// Initial position and velocity of the target
xl:=500; vxl:=15;
yl:=500; vyl:=0;
```

Next, the capacity of each of them to curve was assigned (this turning capacity is based on the angular velocity $\mathrm{W}=2\pi/\mathrm{T}$, where T is the time in seconds required to perform a complete turn):

```
// Angular velocity of the missile
aw:=(1/20)*2*pi();
// Angular velocity of the target
awl:=(1/40)*2*pi();
```

The missile's turning capacity was assigned to be twice that of the target, as generally they have a high ability to change direction compared to targets.

The next step was to define the counter, the missile's decision time, the minimum collision distance[2], and to define variables that store the value of the angle α, the result of the cross product, the target's decision, the missile's decision, the target's position relative to the missile, and the target's velocity relative to the missile:

```
// Counter
cont:=0;
// Decision time
sampling_time:=0.1;
// Minimum collision distance
dist_min_col:=1;
// variable that stores the alpha angle
alpha:=0;
//Variable that stores the vectorial product to decide which way to turn
vect_P:=0;
// Variables that stores the angular velocity decision to the missile and the target
w:=0;
wl:=0;
// Variables that stores the relative positions and velocities
dx:=0;
dy:=0;
vrelx:=0;
vrely:=0;
```

Subsequently, a while loop was created that runs the program inside it until the distance between the missile and the target is less than or equal to the minimum collision distance. Within the loop, the program begins creating a series of if-then-else statements that define the target's decision (move forward during the first 100 iterations, then turn left in the next 100, turn right in the subsequent 100, and move forward in

[2] Since it is a discrete system, the missile is unlikely to collide exactly with the target, so it is necessary to define a radius that considers the possibility of collision.

the remaining iterations), as well as incrementing the counter responsible for counting the iterations. The next step involves implementing the previously developed strategy. This starts by calculating the position and velocity of the target relative to the missile using Eq. (5.1) and Eq. (5.2), and the angle α using Eq. (5.3). Finally, a series of if-then-else instructions are implemented to determine which decision the missile's decision-making system should make. This set of instructions first checks whether α is between 179.5*π/180 (179.5 degrees) and 180.5*π/180 (180.5 degrees). If it is true, the missile's decision is to move forward (w=0), but if it is false, the cross product between the velocity vector and the position vector of the target relative to the missile is calculated and verified to make sure this value is positive. If this cross product value is positive, the decision is to turn right (w=aw), but if it's negative, the decision is to turn left (w=-aw), as indicated in Eq. (5.4).

The final step involves executing two systems of 2D motion differential equations (see Chapter 3, Eq. (3.2)) during the missile's decision time, resulting in a continuous evolution of the missile's and target's position and velocity according to the decisions made.

```
// Run the following program whilst the distance between the missile and the target is
    greater than the collision distance
while (sqrt((x-xl)^2+(y-yl)^2)>dist_min_col) do {
    //Conditional structures to establish the target path
    if (cont<=100)
    then wl:=0;
    else {
          if (cont<=200)
          then wl:=-awl;
          else {
               if (cont<=300)
               then wl:=awl;
               else wl:=0;
               }
         }
    // The counter is incremented
    cont:=cont+1;
    //Update distances and relative velocities
    dx:=xl-x;
    dy:=yl-y;
    vrelx:=vxl-vx;
    vrely:=vyl-vy;
    // Determine the value of the angle alpha
    alpha:=arccos((vrelx*dx + vrely*dy)/(sqrt(vrelx^2 + vrely^2)*sqrt(dx^2 + dy^2)));
    // Conditional structures to determine whether the missile needs to move forward or make
        a curve
    if (alpha>=179.5*pi()/180 && alpha<=180.5*pi()/180)
    then {
         // If the theta is  between 179.5 and 180.5 degrees, the missile follows a straight
```

```
        line at a constant velocity
    w:=0;
    }
else {
    // Determine the value of the vetorial product between the relative velocity vector
        and the relative position vector
    vect_P:=vrelx*dy-vrely*dx;
    // If the theta is not between 179.5 and 180.5 degrees, the missile needs to curve
        to the left or right
    // To decide which way to turn, simply check the sign of the vectorial product.
    if (vect_P>=0)
    then {
        // If the vectorial product is positive or zero,  it curves to the right
        w:=aw;
        }
     else {
        // If the vectorial product is negative,  it curves to the left
        w:=-aw;
        }
    }
// Differential equations
x'=vx,y'=vy,vx'=w*vy,vy'=-w*vx,
xl'=vxl,yl'=vyl,vxl'=wl*vyl,vyl'=-wl*vxl for sampling_time;
                    }
```

(**Note:** The complete implementation of this program can be found in Appendix A.9.)

By executing the previous program, you obtain the plot in Fig. 84[3].

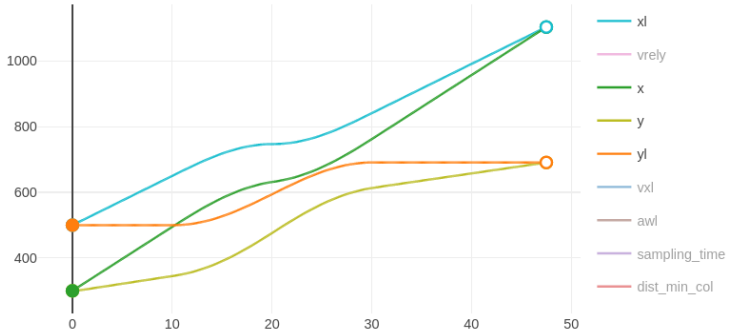


Figure 84: *Position of the missile (x,y) and the target (xl,yl) as a function of time, according to the previous hybrid program*

It can be observed from the plot of the Fig. 84 that the missile's position was adjusted to intercept the target's position at 47.5 seconds.

The plot of the Fig. 84 can be converted into a 2D representation, where the horizontal axis corresponds

[3] These plot are all generated using the numerical plot because the set of differential equations in this program leads to excessively large symbolic expressions, preventing the use of symbolic plotting, as discussed in Section 4.4.

to the x-coordinate and the vertical axis corresponds to the y-coordinate, in order to provide a more intuitive visualization of the missile and target positions. To perform this conversion, the numerical plot settings in Lince can be accessed, and by clicking on "Edit in Chart Studio," an editor is opened. Then, the values of x are set in terms of y, and the values of xl are set in terms of yl to achieve the overlay of the two trajectories. The result of this conversion is depicted in Fig. 85[4].

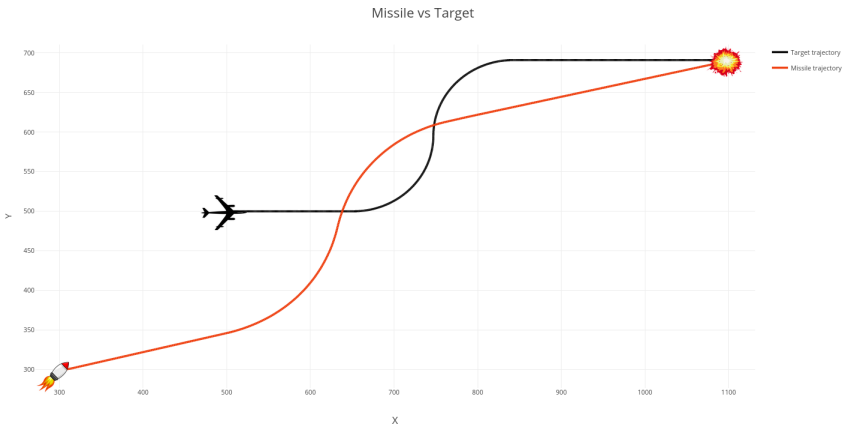


Figure 85: *2D representation of the trajectories of the missile and the target*

By altering the initial position of the missile to $(700m, 300m)$ and its initial velocity to $(20m/s, -10m/s)$, the plot of the missile's and target's positions obtained from Lince, as well as their 2D representation, are represented in Fig. 86 and Fig. 87, respectively.

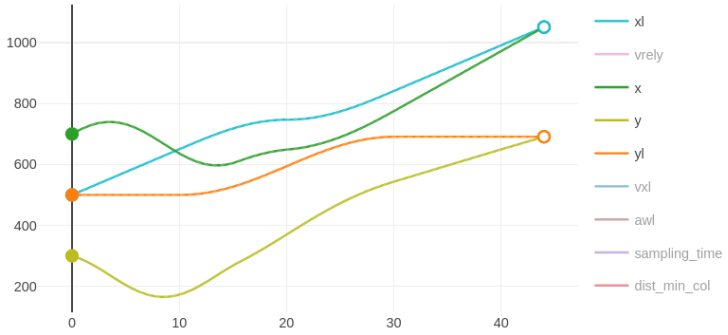


Figure 86: *Position of the missile (x,y) and the target (xl,yl) as a function of time, when the initial position of the missile is changed to* $(700m, 300m)$ *and the initial velocity of the missile is changed to* $(20m/s, -10m/s)$

Despite the missile's initial position and velocity being changed, the developed strategy was able to make the missile collide against the target after 44.1 seconds.

4 In addition to the mentioned conversion, axis colors and thickness were adjusted, labels were renamed, and some images were added to make the new plot more appealing and easy to interpret.

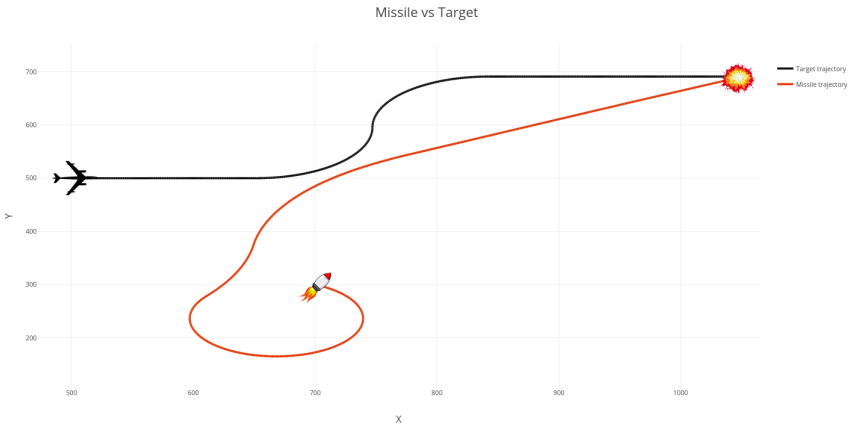


Figure 87: *2D representation of the trajectories of the missile and the target when the initial position of the missile is changed to* $(700m, 300m)$ *and the initial velocity of the missile is changed to* $(20m/s, -10m/s)$

(**Note:** If we analyse Fig. 87, we can see that the missile has decided to turn around (i.e. it did not take the shortest path). As mentioned earlier, the reason for this is that the strategy developed in the design of this hybrid program aims to make the angle tend towards 180 degrees, which does not always result in the shortest trajectory. Nevertheless, this strategy proves to be quite effective.)

Finally, if in addition to altering the missile's initial position and velocity, its turning capacity is reduced by half (meaning it completes a full turn in 40 seconds instead of 20 seconds), the plot of the missile's and target's positions from Lince, along with their 2D representation, are shown in Fig. 88 and Fig. 89, respectively.

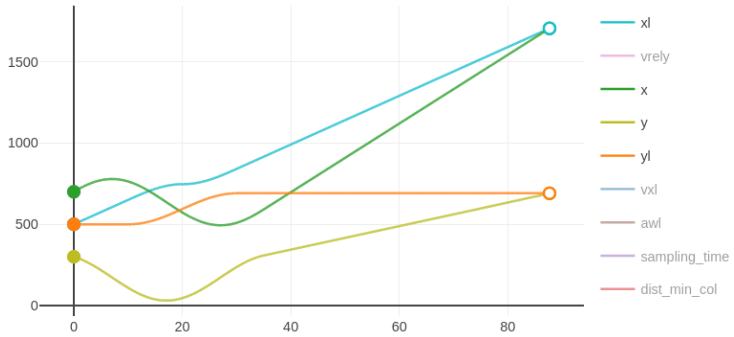


Figure 88: *Position of the missile (x,y) and the target (xl,yl) as a function of time, when the initial position of the missile is changed to* $(700m, 300m)$*, the initial velocity of the missile is changed to* $(20m/s, -10m/s)$ *and the turning capacity is changed to* $(1/40)2\pi$

Even with the reduction in the missile's turning capability, the developed strategy was able to model a trajectory that enables the missile to collide with the target, albeit requiring more time (87.7 seconds) and

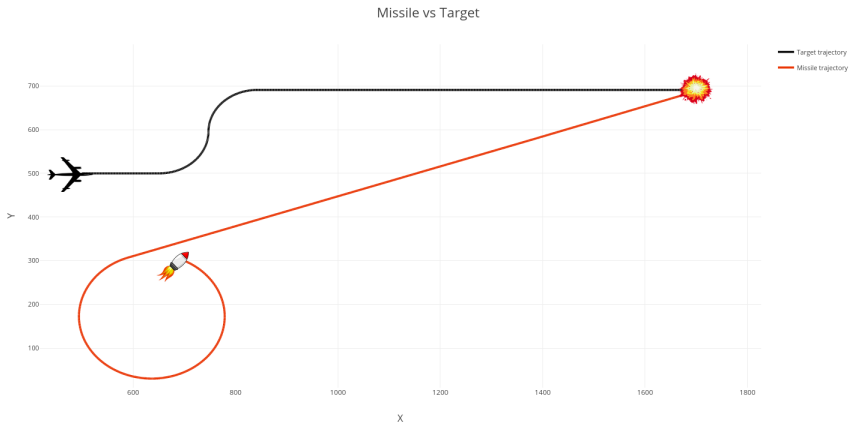


Figure 89: *2D representation of the trajectories of the missile and the target when the initial position of the missile is changed to* $(700m, 300m)$*, the initial velocity of the missile is changed to* $(20m/s, -10m/s)$ *and the turning capacity is changed to* $(1/40)2\pi$

covering a greater distance than the previous simulation.

Based on the previous simulations, it can be observed that the program (using the strategy developed in this section) is effective in executing the trajectory and allows the analysis of the behaviour of this system in different scenarios. In addition to the scenarios analysed so far, there are other very interesting ones to be simulated, such as:

- Varying the target's trajectory;
- Changing the initial position and velocity of the target;
- Modifying the target's turning capability;
- Adjusting the system's decision time.

5.4 Modeling of other types of systems

In addition to Lince being capable of simulating hybrid programs governed by Newtonian mechanics, such as the programs developed in the previous sections, it is also capable of simulating other types of systems, such as classical physical systems and On-Off systems, and to perform numerical analyses. The current section illustrates this aspect.

Specifically in what concerns classical physical systems, the following simulations were performed:

- Damped harmonic oscillator;

- Projectile motion without air resistance.

Regarding On-Off systems, the following simulations were conducted:

- RLC series electrical circuit;
- Hydraulic system;

Finally, for numerical analysis, the following simulations were performed:

- Numerical derivative;
- Numerical integration.

There are several other systems that can be modeled similarly to the examples that will be discussed in the following subsections. For instance, thermal systems use sets of differential equations very similar to hydraulic systems, and as such, they are not covered in this work.

5.4.1 Damped harmonic oscillator

The damped harmonic oscillator is a classical physical system that consists of an object subjected to a restoring force proportional to its displacement from an equilibrium position, while also experiencing a damping force proportional to its velocity [Top23].

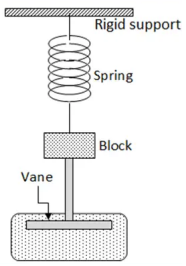


Figure 90: *Representative image of an assembly for studying damped harmonic motion [Exp21].*

The differential equation that describes the motion of a damped harmonic oscillator is given by [Top23]:

$$x'' = -\frac{k}{m} * x - \frac{c}{m} * x' \tag{5.5}$$

where m is the mass of the object, x is the displacement of the object from the equilibrium position, c is the damping coefficient, and k is the spring constant that determines the restoring force.

(**Note:** Differential Eq. (5.5) arises from the use of Newton's second law, where the restoring force is equal to $k * x$ and the damping force is equal to $c * x'$.)

Depending on the value of the damping coefficient, the damped harmonic oscillator can exhibit different regimes [Wik22]:

- **Underdamping**: In this regime, the damping coefficient (c) is less than the critical value ($2 * sqrt(m * k)$). The amplitude of the oscillations gradually decreases over time. The lower the damping coefficient relative to the critical value, the more prolonged the oscillations will be before the amplitude significantly decreases.

- **Overdamping**: In this regime, the damping coefficient (c) is greater than the critical value ($2 * sqrt(m * k)$). When the object is released under this regime, it quickly returns to its equilibrium position without oscillations.

- **Critical Damping**: In this regime, the damping coefficient (c) is equal to the critical value ($2 * sqrt(m * k)$). This is the regime where the object reaches its equilibrium position more quickly, without exhibiting oscillations around it, similar to the overdamped regime.

The critical value of the damping coefficient (c) is determined by the properties of the system, such as the mass (m) and the spring constant (k). Adjusting the damping coefficient relative to the critical value allows control over the behavior of the damped harmonic oscillator. These different regimes have applications in various fields of physics and engineering, such as designing vehicle suspension systems and RLC circuits [Ins23]. To analyze the behavior of the three regimes of the damped harmonic oscillator, a program was developed in Lince capable of simulating these regimes for a specific body oscillating under certain properties.

Initially, one starts by defining the value of the body's mass (m) and the value of the spring constant (k):

```
m:=1; // mass of the object
k:=2.32; // Spring constant
```

Next, the value of the damping coefficient, initial position, and initial velocity is defined for each of the three regimes:

```
// damping coefficient and initial values of the underdamping regime
b_sc:=1; //damping coefficient
xsc:=2; //Initial position
vsc:=0; //Initial velocity
```

```
// damping coefficient and initial values of the overdamping regime
b_Sc:=3.5 ; //damping coefficient  regime
xSc:=2; //Initial position
vSc:=0; //Initial velocity

// damping coefficient and initial values of the critical damping regime
b_c:=2*sqrt(k*m); //damping coefficient
xc:=2; //Initial position
vc:=0; //Initial velocity
```

The variable xsc will represent the position of the body undergoing underdamped harmonic oscillator since its damping coefficient (b_sc) is less than $2 * sqrt(k * m)$ (which is approximately equal to 3.046). On the other hand, the variable xSc will represent the position of the body undergoing overdamped harmonic oscillator since its damping coefficient (b_Sc) is greater than $2 * sqrt(k * m)$. Finally, the variable xc will represent the position of the body undergoing critically damped harmonic oscillator since its damping coefficient (b_c) is equal to $2 * sqrt(k * m)$.

In the end, it is only necessary to execute the set of differential equations that define the damped harmonic oscillator (Eq. (5.5)) for each of the regimes for a duration of 15 seconds:

```
// Differential equations
xsc'=vsc , vsc'=-xsc*k/m-vsc*b_sc/m,
xSc'=vSc , vSc'=-xSc*k/m-vSc*b_Sc/m,
xc'=vc , vc'=-xc*k/m-vc*b_c/m for 15;
```

Executing the previous program in Lince results in the plot shown in Fig. 91.

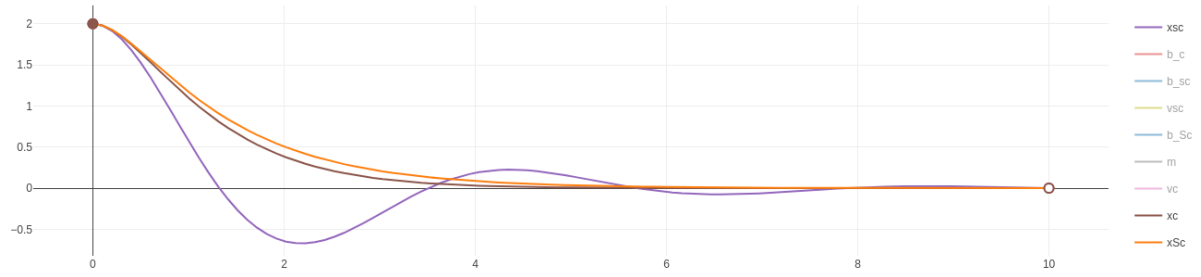


Figure 91: *Representation of the damped harmonic oscillator in the three regimes.*

In Fig. 91, one can visualize the behavior of the three regimes of the damped harmonic oscillator: the purple curve represents the underdamping regime, the orange curve represents the overdamping regime, and the brown curve represents the critical damping regime.

As mentioned earlier in this chapter, it can be observed that the underdamping regime exhibits oscillations, unlike the other regimes. On the other hand, the overdamping regime takes longer to reach equilibrium compared to the critical damping regime.

(**Note:** The complete implementation of this program can be found in Appendix A.10.)

5.4.2 Projectile motion without air resistance

The projectile motion without air resistance is a classical physical system that describes the motion of an object launched in the air without the influence of air resistance [Ope22].

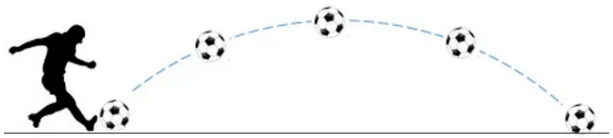


Figure 92: *Representative image of a projectile motion [Phy23b].*

The differential equations describing this 2D physical system can be found in reference [Phy23a], and based on Eq. (3.2) from Chapter 3, they can be written as follows:

$$x' = vx, y' = vy, vx' = 0, vy' = -g \tag{5.6}$$

Considering "g" as the gravitational acceleration and assuming that the projectile is launched from a given position ($x0$, $y0$) with a given initial velocity $v0$ at an angle θ, the initial velocity in the x direction is $v0 * cos(\theta)$ and the initial velocity in the y direction is $v0 * sin(\theta)$. However, since no forces act on the projectile in the x direction throughout the motion, the sum of the resulting forces in the x direction is zero ($Fr_x = 0$), leading to a constant velocity ($vx' = 0$) in that direction. On the other hand, in the y direction, the weight of the projectile is the only force acting and it points in the opposite direction. Therefore, $Fry = m * vy' \Leftrightarrow -m * g = m * vy' \Leftrightarrow vy' = -g$, meaning that the projectile travels along the y axis with a constant acceleration equal to $-g$ [Wal18].

The utility of this physical system can be found in various everyday applications, some of them are found in the following areas [Dew23]:

- **Sports**: Projectile motion is encountered in various sports such as basketball, golf, and baseball. Understanding the principles of projectile motion helps athletes calculate the optimal launch angle and velocity for achieving desired trajectories and distances;
- **Engineering**: The understanding of projectile motion without air resistance is crucial in fields like ballistics, artillery, and aerospace engineering. It is employed to determine the trajectory and range of projectiles, aiding in the design and optimization of weapons, rockets, and other projectile-based systems;

Due to the usefulness of this physical system, it was decided to develop a program in Lince capable of simulating this system.

To do so, we started by defining the initial position, the magnitude of the initial launch velocity, the gravitational acceleration, the launch angle, and the initial velocity component in each of the axes:

```
x:=2;
y:=2;
v0:=10;
g:=9.8;
theta:=pi()/4;
vx:=v0*cos(theta);
vy:=v0*sin(theta);
```

Next, the system of differential equations 5.6 was executed until the projectile reached the ground (that is, until the y-axis became less than or equal to zero):

```
x'=vx,y'=vy,vx'=0,vy'=-g until_0.01 (y<=0);
```

Running the previous program resulted in the plot shown in Fig. 93.

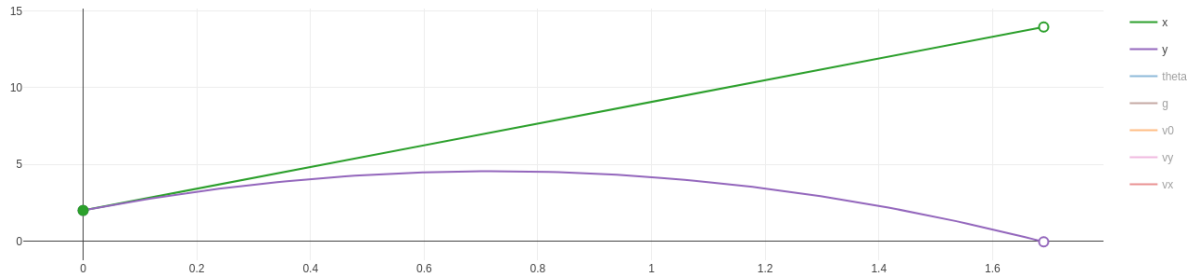


Figure 93: *Position of the projectile in the "x" and "y" coordinates*

Through this program, users can adjust the initial parameters to obtain the graph of the projectile launch according to the characteristics they have defined. This allows them to analyze how the projectile would behave with those specific characteristics and gain a better understanding of this type of physical system.

(**Note:** The complete implementation of this program can be found in Appendix A.11.)

5.4.3 RLC series electrical circuit

The RLC series electrical circuit is a system composed of a resistor (R), an inductor (L), a capacitor (C) and a voltage source connected in series. In this case, the circuit is considered as an On-Off system, which means that the voltage source is turned on and off over time.

For an RLC series circuit, the differential equation describing the variation of voltage across the capacitor over time is derived for the following laws [CC06]:

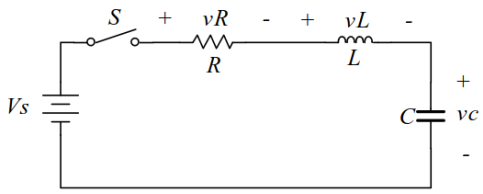


Figure 94: *Representative image of a RLC series electrical circuit [CC06].*

- Kirchhoff's second law gives $Vs = vc + vL + vR$, where Vs is the voltage at the source, vc is the voltage at the capacitor, vL is the voltage at the inductor and vR is the voltage at the resistor;
- Ohm's law gives $vR = R * i_R$, $i_c = C * vc'$, $vL = L * i'_L$ and $i_c = i_L = i_R = i$ (R is the resistance, C is the capacitance, L is the inductance, i_R is the current in the resistor, i_c is the current in the capacitor and i_L is the current in the inductor);

Based on these two laws and the expressions established by them it follows that [CC06]: $Vs = vc + vL + vR \Leftrightarrow Vs = vc + L * i' + R * i \Leftrightarrow Vs = vc + L * C * vc'' + R * C * vc' \Leftrightarrow vc'' = -vc' * (R/L) - vc/(L * C) + Vs/(L * C)$;

Converting the previous differential equation to a system of differential equations following the Lince grammar, gives the following set of differential equations:

$$vc' = dvc, dvc' = -dvc * (r/l) - vc/(l * c) + vs/(l * c) \tag{5.7}$$

On the other hand, the RLC series circuit can operate in different regimes depending on the values of the components ($\alpha = R/(2 * L)$ and $w_0 = 1/\sqrt{L * C}$) [int23]:

- **Critically damped regime:** In this regime, the circuit quickly reaches a state of equilibrium without oscillations. This occurs when $\alpha = w_0$;
- **Underdamped regime:** In this regime, the circuit exhibits oscillations and the amplitude decreases exponentially. This regime occurs when $\alpha < w_0$;
- **Overdamped regime:** In this regime, the circuit returns to state of equilibrium more slowly than in the critically damped regime without oscillations. This regime occurs when $\alpha > w_0$.

The RLC circuit has many applications, some of them involve [rlc23]:

- Oscillator circuits, radio receivers, and television sets where they are used for tuning purposes;
- Signal processing and communication systems.

Given that RLC circuits are widely used in various everyday applications, it was decided to simulate the behavior of three RLC series circuits, each operating in a different regime, where the voltage source is turned on for a certain period and then turned off (On-Off system). This simulation will allow users to analyze how the voltage across the capacitor in an RLC circuit varies in each regime when the voltage source is switched on and off.

To perform this simulation, a hybrid program was developed in Lince. This hybrid program begins by initializing the initial voltage of the capacitor in each regime ($\mathrm{vc\_rac}$ is the voltage in the critically damped regime, $\mathrm{vc\_rsa}$ is the voltage in the underdamped regime, and $\mathrm{vc\_rSa}$ is the voltage in the overdamped regime), the respective derivative of the voltage ($\mathrm{dvc\_rac}$ is the derivative of the voltage in critically damped regime, $\mathrm{dvc\_rsa}$ is the derivative of the voltage in the underdamped regime, and $\mathrm{dvc\_rSa}$ is the derivative of the voltage in the overdamped regime) and the initial voltage of the voltage source (vs):

```
vc_rac:=0;
vc_rsa:=0;
vc_rSa:=0;
dvc_rac:=0;
dvc_rsa:=0;
dvc_rSa:=0;
vs:=10;
```

Next, the value of the inductance (l), capacitance (c), and the resistance values for each circuit were defined according to their respective regimes ($\mathrm{r\_rac}$ is the resistance in the critically damped regime, $\mathrm{r\_rsa}$ is the resistance in the underdamped regime, and $\mathrm{r\_rSa}$ is the resistance overdamped regime):

```
l:=0.047;
c:=0.047;
r_rac:=2;
r_rsa:=0.5;
r_rSa:=4;
```

To choose the value of resistance for each regime, the expression of α was equated with the expression of w_0 and the value of the resistance required for the circuit to operate in the critical regime was extracted: $\alpha = w_0 \Leftrightarrow R/(2*L) = 1/\sqrt{L*C} \Leftrightarrow R = (2*L)/\sqrt{L*C} \Leftrightarrow R = (2*0.047)/\sqrt{0.047*0.047} \Leftrightarrow R = 2$. Thus to obtain the critically damped regime, the resistance value must be equal to 2 ($\mathrm{r\_rac}$=2), to obtain the underdamped regime the resistance value must be less than 2 ($\mathrm{r\_rsa}$<=2) so that α is less than w_0, and to obtain the overdamping regime the resistance value must be greater than 2 ($\mathrm{r\_rSa}$ >= 2) so that α is greater than w_0.

The next step was to run a system of differential equations that describes the voltage variation in the capacitor (see Differential equations 5.7) for each regime for a duration of 1 second while the voltage

source was connected with a value of 10:

```
vc_rac'=dvc_rac, dvc_rac'=-dvc_rac*r_rac/l-vc_rac/(l*c)+vs/(l*c),
vc_rsa'=dvc_rsa, dvc_rsa'=-dvc_rsa*r_rsa/l-vc_rsa/(l*c)+vs/(l*c),
vc_rSa'=dvc_rSa, dvc_rSa'=-dvc_rSa*r_rSa/l-vc_rSa/(l*c)+vs/(l*c) for 1;
```

The final step is to turn off the voltage source and run the aforementioned system of equations:

```
vs:=0;
vc_rac'=dvc_rac, dvc_rac'=-dvc_rac*r_rac/l-vc_rac/(l*c)+vs/(l*c),
vc_rsa'=dvc_rsa, dvc_rsa'=-dvc_rsa*r_rsa/l-vc_rsa/(l*c)+vs/(l*c),
vc_rSa'=dvc_rSa, dvc_rSa'=-dvc_rSa*r_rSa/l-vc_rSa/(l*c)+vs/(l*c) for 1;
```

By executing the aforementioned hybrid program, the plot in Fig. 95 was obtained, representing the variation of voltage across the capacitor over time.

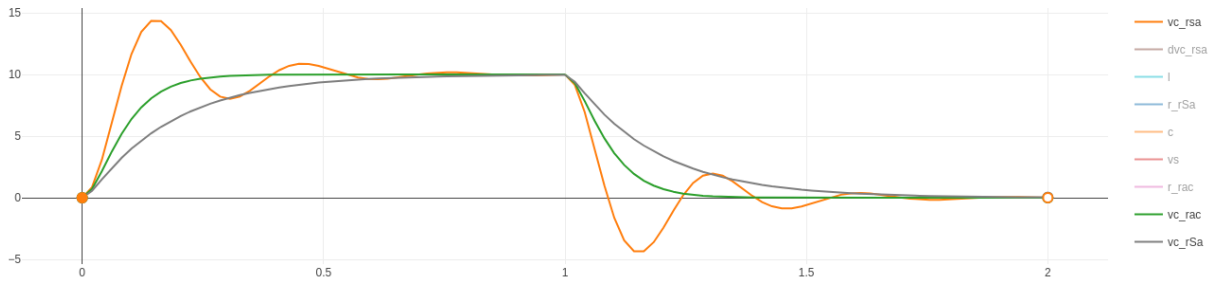


Figure 95: *Variation of voltage across the capacitor for critically damped regime (green), for underdamed regime (orange), and for overdamped regime (grey).*

Based on the simulation of this system using this hybrid program, it can be observed that the behavior of the three regimes aligns with the description provided at the beginning of this section. This allows the user to visually understand the system's behavior and explore variations in the initial parameters and the On-Off time of the voltage source, providing insights into how the system responds to these changes.

(**Note:** The complete implementation of this program can be found in Appendix A.12.)

5.4.4 Hydraulic system

Hydraulic systems are systems that use fluids, such as water or oil, to transmit energy and perform mechanical tasks [aad23]. Hydraulic systems have a wide range of practical applications in various sectors:

- **Amusement Park Rides:** Hydraulic systems are crucial for ensuring the safety of amusement park rides. These systems can handle significant forces and repetitive motions, while also providing precise control over pressure. This precision is essential for controlling the spinning, pushing, lifting, and speeding of the rides in a regulated manner, guaranteeing the riders' safety. In addition to their

role in controlling the ride's movements, hydraulic systems are used in various safety elements. For instance, they power the bars or harnesses that automatically lower and lock into position, securing passengers in their roller coaster seats. These hydraulic systems contribute to creating a safe and enjoyable experience for riders at amusement parks [Mar21];

- **Automotive industry:** Hydraulic systems are responsible for various functions in a car. For instance, they control the brakes through a hydraulic brake circuit. Hydraulics also play a role in the suspension system, particularly in shock absorbers and power steering, and tilting systems in dump trucks and cranes [Mar21];

- **Lifts:** Lifts have been popular since their invention in the 1880s. While we often use them without questioning their operation, it's worth noting that most lifts operate like a pulley system, with a heavy-duty metal rope and a counterweight to maintain balance. In addition to these traditional lifts, there are hydraulic-powered lifts. These lifts rely on a piston housed within a cylinder. An electric motor pumps oil into the cylinder, causing the piston to move and elevate the lift's cabin. To bring the lift back down, an electronic valve carefully releases the hydraulic oil, allowing the piston to return to its original position. Hydraulic lifts provide an alternative mechanism for vertical transportation, offering efficient and controlled movement [Mar21];

- **Trash compactors:** Trash compactors play a vital role in waste management as landfills globally face limited capacity. To address this issue and promote sustainability, hydraulic systems are employed to alleviate pressure. Garbage trucks and compactors utilize hydraulic force to compress large volumes of waste, reducing the space it occupies. While this is not a long-term solution, it offers a temporary measure to slow down landfill fill-up rates and contribute to a more sustainable approach to waste disposal [Mar21];

- **Water supply in tall buildings:** In tall buildings such as skyscrapers, water is usually pumped into tanks located at the top of the building. A hydraulic system to control the water level in these tanks is critical to ensure that there is sufficient pressure in the water supply system to allow water to reach all floors at the appropriate pressure [Blo23].

These are just a few of the many practical applications of hydraulic systems. They offer advantages ranging from the automotive industry to the manufacturing of elevators, compactors, water supply and safety systems.

Therefore, it was decided to develop a hybrid program in Lince capable of simulating a simple hydraulic system. This hydraulic system will consist of a water tank that receives water from a faucet and loses water

through a hole at the bottom. The goal of the program is to try to maintain the water level close to 9 by opening or closing the faucet (assuming that the faucet operates as an On-Off system), thus ensuring an approximately constant water pressure flowing out of the hole.

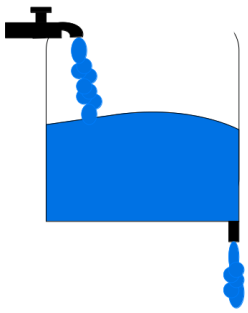


Figure 96: *Schematic representation of the hydraulic system to be simulated.*

Before developing the hybrid program, it is necessary to obtain the system of differential equations associated with this example. The derivation of these differential equations was based on reference [Alv11].

Starting from the law of conservation of mass (where the amount of mass entering a system is equal to the amount of mass leaving plus the amount of mass retained in the system), the inflow (F_{in}) is equal to the sum of the outflow (F_{out}) plus the accumulation in the tank (F_{ac}): $F_{in} = F_{out} + F_{ac}$. The accumulated flow can be defined as the change in water volume over time ($V = A * h$, where V is the volume of water in the tank, A is the base area, and h is the water height):$F_{ac} = V' = A * h'$. The outflow is given (approximately) by the ratio of the water height to the resistance offered by the hole (R): h/R.

Based on the previous expressions, the differential equation that models this system is: $F_{in} = h/R + A * h'$, which can be adjusted to obtain the differential equation that follows the grammar of Lince:

$$h' = f\_in/a - h/(r * a) \tag{5.8}$$

where f_in is the inflow rate, r is the resistance offered by the hole, h is the water height, and a is the base area. With the obtained differential equation, we begin the development of the hybrid program. To develop it, we start by assigning the value of the tank's base area, the resistance offered by the hole, the initial water inflow from the faucet, and the initial water height:

```
a:=2; //Area of tank
r:=6; //Resistance
f_in:=8; //inflow rate
h:=5; //initial water height
```

Finally, a repeat loop was created that executed 40 times (it is sufficient to visualize the intended result), containing an if-then-else statement and Differential equation 5.8. The if-then-else statement checked if

the water height was less than 9, and if so, the tap was turned on; otherwise, it was turned off. Additionally, Differential equation 5.8, which models the dynamics of this system, was executed for a duration of 0.1 seconds during each iteration. The described code is as follows:

```
repeat 40 {
    if (h<9)
    then f_in:=8;
    else f_in:=0;

    h'=f_in/a-h/(r*a) for 0.1;
        }
```

By running the developed hybrid program, he plot in Fig. 97 is obtained.

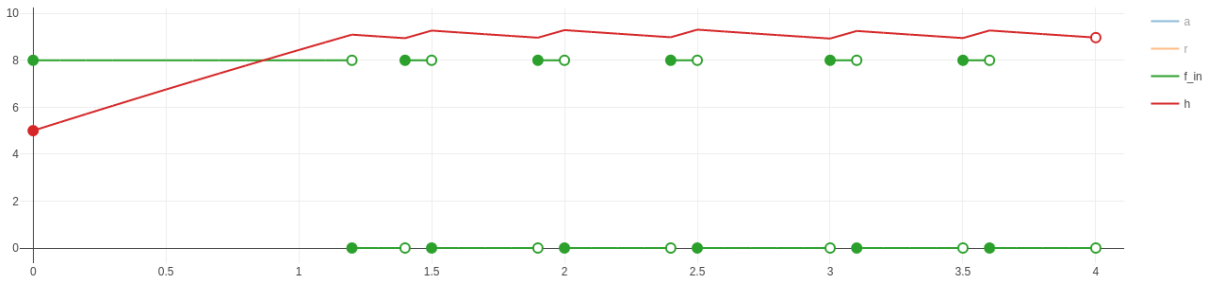


Figure 97: *Evolution of water level in the tank (red) and faucet state (green).*

Through this hybrid program, a simple hydraulic system was simulated, capable of maintaining the water level near 9 using a faucet that only turns on and off (operating in an On-Off manner). The user can also vary parameters such as the base area of the tank, the resistance offered by the hole to the flow of water, and the water flow provided by the faucet in order to analyze the operation of this system under different initial conditions.

(**Note:** The complete implementation of this program can be found in Appendix A.13.)

5.4.5 Numerical derivative and integration

The numerical derivative and integral are techniques used to approximate the value of the derivative and integral of a function. They are widely used in various fields, from physics and engineering to computing and data science [Uni23].

The numerical derivative is used to estimate the instantaneous rate of change of a function at a specific point. There are various methods to obtain an approximation of the numerical derivative, with the most common one being the finite difference method [Uni23].

The numerical integral, on the other hand, is used to calculate the area under the curve of a function over a specific interval. Similar to the numerical derivative, there are various methods to approximate

the numerical integral. The most common methods include the midpoint rule, the trapezoidal rule, and Simpson's rule [Lib23]. According to reference [Sin22], numerical differentiation and integration have various applications in everyday life. Some of them include:

- **Calculating pressure within dams:** We can use integration to calculate the force exerted on the dam when the reservoir is full and also calculate how changing water levels affect that force.
- **Automobiles:** In an automobile, we always find an odometer and a speedometer. Their gauges work in synchrony and determine the speed and distance the automobile has traveled. The electronic meters use differentiation to transform the data sent to the motherboard from the wheels (speed) and the distance (odometer).
- **Video games:** The graphic engineer uses integration and differentiation to determine the difference and change of three-dimensional models and how they will change when exposed to multiple conditions. This helps to create a very realistic environment in 3D movies or video games.
- **Medicine:** Calculus is a crucial mathematical tool for analyzing drug activity quantitatively. Differential equations are utilized to relate the concentrations of drugs in various body organs over time. In addition, integrated equations are often used to model the cumulative therapeutic or toxic outcomes of drugs in the body.

Based on the importance of numerical differentiation and numerical integration, it was decided to develop a hybrid program in Lince capable of performing the derivative (using the finite difference formula) and the numerical integration (using the trapezoidal rule) of the function x^2.

Regarding the finite difference formula, both the 2-point formula and the 3-point formula were used.

The 2-point formula is as follows [Net23]:

$$f'(x) = \frac{f(x) - f(x-h)}{h} \tag{5.9}$$

The 3-point formula is as follows [Ato23]:

$$f'(x) = \frac{3 * f(x) - 4f(x-h) + f(x-2h)}{2h} \tag{5.10}$$

Regarding the trapezoidal rule, the formula used is [ECT23]:

$$I_n = \frac{h * (f(nh) + f((n-1)h))}{2} + I_{(n-1)h} \tag{5.11}$$

Having defined the formulas to be used, the intended hybrid program was developed. To begin, the interval width (h), initial conditions of the function to be numerically differentiated and integrated (y and dy), the variables that will accumulate the value of the function at $f(x)$, $f(x-h)$, and $f(x-2h)$ (yi,yhi and y2hi, respectively), the variables that will store the value of the derivative (dyi) and integral (int_i) at each iteration, and a control variable (aux) were assigned:

```
h:=0.1; y:=0; dy:=0;
yi:=y; yhi:=yi; y2hi:=yhi;
dyi:=0; int_i:=0; aux:=0;
```

Next, a repeat loop was created with the goal of executing the program inside it 1000 times (i.e., calculating the numerical derivative and integral for 1000 iterations). Inside the loop, the program began by updating the values of the variables that would accumulate the value of the function at $f(x)$, $f(x-h)$, and $f(x-2h)$. Then, a conditional structure checked if the control variable was greater than or equal to 2 (indicating that the function had been evaluated for at least 3 points), equal to 0 (indicating that the function had been evaluated for only 1 point), or equal to 1 (indicating that the function had been evaluated for at least 2 points). If it was greater than or equal to 2, Eq. (5.10) was applied for the derivative and Eq. (5.11) was applied for the integral. The program also ran the differential equation y'=dy,dy'=2 for h (which models the function x^2 every h seconds) and incremented the control variable. If it was equal to 1, the same procedures were followed, but Eq. (5.9) was used for the derivative. Finally, if the control variable was zero, neither the derivative nor the integral were calculated (as there were not enough points), and only the differential equation was solved and the control variable was incremented. The code for the program is as follows:

```
repeat 1000 {
    y2hi:=yhi;
    yhi:=yi;
    yi:=y;

    if (aux>=2)
    then {
    dyi:=(3*yi-4*yhi+y2hi)/(2*h);
    int_i:=(h/2)*(yhi+yi)+ int_i;
    y'=dy, dy'=2 for h;
    aux:=aux+1;
        }
    else {
      if (aux==0)
      then {
      y'=dy,dy'=2 for h;
      aux:=aux+1;
          }
```

```
    else {
    dyi:=(yi-yhi)/(h);
    int_i:=(h/2)*(yhi+yi)+ int_i;
    y'=dy, dy'=2 for h;
    aux:=aux+1;}}}
```

Running the above hybrid program results in the plot shown in Fig. 98.

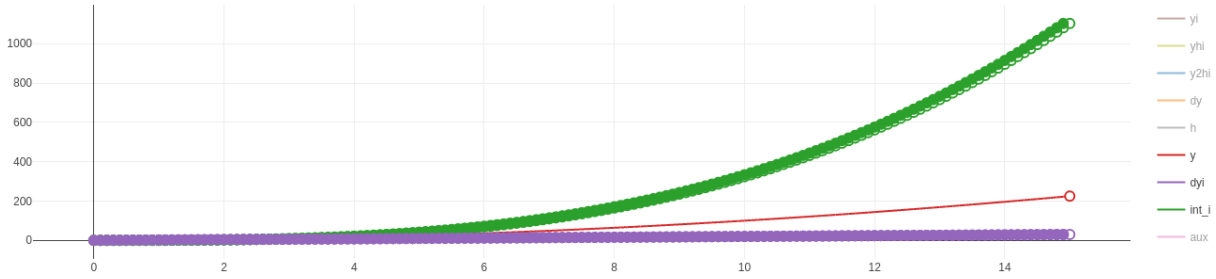


Figure 98: *Graphical representation of the numerical derivative (purple) and numerical integral (green) of the function x^2 (red)*

(**Note:** The complete implementation of this program can be found in Appendix A.14.)

It can be verified that the numerical derivative and integral are well implemented by creating a hybrid program that evaluates the functions x^2, $2x$ (derivative of x^2), and $(1/3)x^3$ (integral of x^2) over the same time interval as the previous graph:

```
h:=0.1;
y1:=0; dy1:=0;
y2:=0; dy2:=2;
y3:=0; dy3:=0;
repeat 1000 {
    y1'=dy1,dy1'=2,
    y2'=dy2,dy2'=0,
    y3'=dy3,dy3'=y2 for h;
          }
```

The plot in Fig. 99 is the result of running this hybrid program.

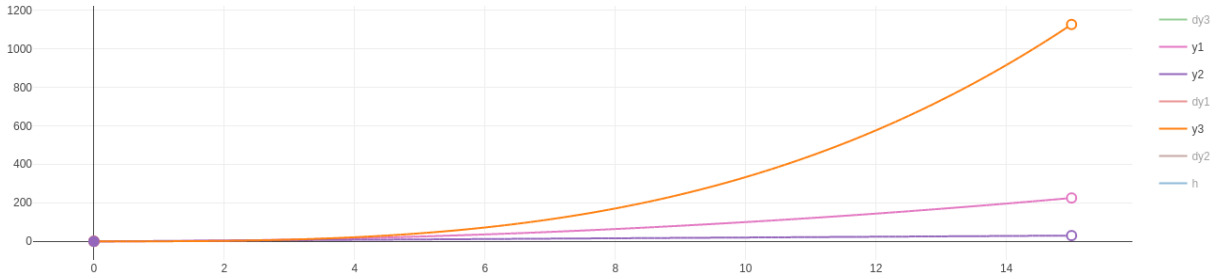


Figure 99: *Graphical representation of the function x^2 (pink), the function $2x$ (purple) and the function $(1/3)x^3$ (orange)*

Through these hybrid programs, it is concluded that the real and numerical representations are quite similar demonstrating the capability of this new version of Lince to perform numerical derivatives and integrals using Eqs. (5.9) to (5.11).

Chapter 6

Conclusions and future work

6.1 Conclusions

Through the completion of this dissertation project, the Lince tool, which previously had constraints and limitations, was improved and transformed into a more versatile hybrid program simulation tool, with greater simulation capabilities. These improvements include the addition of a wide range of linguistic constructs, the ability to use non-linear expressions, grammatical relaxation, improved error detection, grammatical structuring, bug fixes in parsers and the interpreter, and the possibility of determining the solutions of differential equations numerically.

As a consequence, Lince evolved into a more capable tool of modelling hybrid programs, especially those governed by Newtonian mechanics, something that was previously impossible to implement with the precision and simplicity achieved in the current version of Lince. The enhancement was driven by the need to have programs performing instructions with high complexity on both discrete and continuous levels.

A myriad of hybrid programs, including AEB, ACC, and the guided missile trajectory system, were designed and tested in the new version of Lince, demonstrating its applicability to the modelling of such systems. Not only this, we also presented the simulation and modelling of other systems, such as purely physical systems, on-off systems and numerical analyses.

Therefore these enhancements made to the tool exceed the original objectives of this study, which solely sought to introduce novel linguistic constructs to enable the modelling of a wider spectrum of hybrid programs, particularly those governed by Newtonian mechanics. Such enhancements have converted the tool into a simulator for hybrid programs that is not only more proficient but also more user-friendly.

This work has opened up new possibilities in simulating complex systems and it is predicted that the contributions presented will inspire further research and practical applications in this rapidly advancing field.

6.2 Prospect for future work

During the execution of a dissertation project, potential improvements or issues often arise that cannot be implemented or resolved due to time constraints. It is thus important to identify these potential improvements or issues, highlight their significance, and, if possible, provide strategies for their implementation or mitigation to encourage future contributions. As such, throughout the execution of this dissertation project, a set of potential improvements and issues were identified. How the implementation or mitigation of these could positively impact the tool was explored, and strategies for their application or mitigation were considered. This resulted in the following proposals for future work:

- **Improving trajectory generation from the symbolic graph** - As mentioned in Section 4.4, SageMath's inability to handle large symbolic expressions prevented the use of the symbolic graph in the simulation of certain hybrid programs. As discussed in that chapter, this problem was mitigated by creating a solver that solved differential equations using a numerical method, and creating the numerical graph that allowed the simulation (albeit with some inaccuracy) of programs that encountered this problem in the symbolic graph. However, the ideal approach would be to improve the method of calculating trajectories from the symbolic graph, thus avoiding the inaccuracy introduced by the numerical graph. This improvement could involve numerically calculating only the symbolic expressions that approach the limit that SageMath can handle. This way, only symbolic expressions that would cause issues due to their length would be numerically simplified, reducing the error propagation introduced by the numerical plot;

- **Configuration of Lince's Graphs** - Lince's graphical interface allows some manipulation of graphs, such as zooming in, zooming out, selecting, and determining which trajectories to visualize. However, it would be a significant enhancement to the tool if the graphical interface allowed the user to select some parameters and create an n-dimensional graph that relates n variables.

  As the first suggestion of an enhancement, it would be useful allow the user to select important parameters for the simulation of hybrid programs, such as the number of samples (Nyquist's theorem); the maximum number of loop iterations; the algorithm for obtaining numerical solutions to differential equations (assuming that other algorithms are involved); and which variables appear in the graphs, among others. In order to implement this enhancement, a procedure similar to the one used, for example, to support the configuration of the time limit for the simulation of hybrid programs, already implemented in the old version of Lince, would be necessary.

As the second suggestion, it would be useful to allow the user to visualize the behaviour of variables in a faster and more convenient way (similar to the 2D representation of missile and target trajectories in Section 5.3), instead of having to rely on the "Chart Studio" editor. A possible implementation of this improvement would be to configure a parameter in the Lince platform to select the type of graph (variables over time, 2D or 3D) and the relationship between variables (for example, in a 2D graph, variable 'a' is associated with the x-axis and variable 'b' with the y-axis, both defining a trajectory). In the case of a 2D or 3D graph, the interpreter would need to collect relevant variable values along their trajectories and then project these values onto their respective axes to create a trajectory.

- **Enabling Direct Use of Differential Equation Solutions** - In this new version of Lince, the continuous behavior of variables can only be described by differential equations. However, allowing the use of trajectory equations (i.e., solutions to differential equations) to model continuous behavior offers greater flexibility to the user, enabling them to directly write the desired trajectory equations instead of initially having to model the system's differential equations. The strategy for incorporating the use of trajectory equations involves creating syntax for them, extending the parser to support the use of these equations alongside differential equations, making the interpreter compatible with the use of trajectory equations (if the continuous behavior is described by these equations, there is no need to resort to SageMath or numerical methods to determine solutions since these expressions are the solutions themselves), and adding error handling for these equations;

- **Further Exploration of the Case Studies** - The case studies developed in this project highlighted the capabilities of this new version of Lince, as well as its applicability in various areas. However modelling other systems, such as systems based on PID controllers, would be an interesting and challenging task for this new version, as would the in-depth exploration of existing case studies (for example, modelling the trajectory of a missile that always seeks the shortest path and has the ability to decelerate). This would put Lince to the test in different scenarios, allowing the detection of possible anomalies or enriching improvements, as well as testing its applicability;

- **Collaboration with the Community** - Publicizing this tool in the academic and/or development community would provide valuable feedback and potential future collaborations, such as studying whether the tool satisfies certain properties based on CTL logic, as well as identifying bugs and potential improvements.

Bibliography

[aad23] aadityacademy. What is hydraulic: Definition, principle, properties, advantages. https://aadityacademy.com/hydraulic/, 2023.

[ABC22] Science ABC. How are missiles able to 'pursue' targets when they make evasive turns? https://www.scienceabc.com/innovation/how-guided-missiles-work-guidance-control-system-line-of-sight-pursuit-navigation.html, 2022.

[Ack22] Team Ackodrive. Autonomous emergency braking (aeb) in cars and how it works. https://ackodrive.com/car-guide/autonomous-emergency-braking/, 2022.

[Acu] Proctor Acura. Technology guide: What is an automatic braking system? https://www.proctoracura.com/automatic-braking-system-guide.

[ALRF+22] Alvin Alexander, Ben Luo, Julien Richard-Foy, Jonathan, Xhudik, and Adrien Piquerez. Scala for javascript developers. https://docs.scala-lang.org/scala3/book/scala-for-javascript-devs.html, 2022.

[Alv11] Bruno Miguel Pedro Alves. *e-Book para Controlo Digital: Teoria, Matemática, Modelos e Simulações*. PhD thesis, Minho University, 2011.

[Ato23] AtoZmath.com. 1. formula & example-1 (table data). https://atozmath.com/example/CONM/TwoPointFormula.aspx?q=3&q1=E1, 2023.

[Aut] Automate. An overview of machine vision and autonomous vehicles. https://www.automate.org/blogs/an-overview-of-machine-vision-and-autonomous-vehicles.

[BDL06] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal 4.0. *Department of computer science, Aalborg university*, 2006.

[BG11] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[BLL+95] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *International hybrid systems workshop*, pages 232–243. Springer, 1995.

[Blo23] Main Blog. Meeting the challenge of water supply systems in high-rise buildings. https://blog.bermad.com/building-and-construction/meeting-the-challenge-of-water-supply-in-high-rise-buildings, 2023.

[CC06] Chaniotakis and Cory. Examples of transient rc and rl circuits. *Spring*, 2006.

[CCD] CCDCOE. Cybersecurity considerations in autonomous ships. https://ccdcoe.org/library/publications/cybersecurity-considerations-in-autonomous-ships/.

[CNN23] Nadia Leigh-Hewitson CNN. We're still waiting for self-driving cars, but autonomous boats are already here. https://edition.cnn.com/travel/article/autonomous-boats-spc-intl/index.html, 2023.

[Col23] Collimator. What is adaptive cruise control? https://www.collimator.ai/reference-guides/what-is-adaptive-cruise-control, 2023.

[Cue] Cuemath. Angle between two vectors - formula, how to find? AngleBetweenTwoVectors-Formula,HowtoFind?

[DBN+94] Ayumu Doi, Tetsuro Butsuen, Tadayuki Niibe, Takeshi Takagi, Yasunori Yamamoto, and Hirofumi Seni. Development of a rear-end collision avoidance system with automatic brake control. *Jsae Review*, 15(4):335–340, 1994.

[DdMBJ] A Barbosa Daniele de Moraes and Yaro Burian Jr. Iniciação científica: Instrumentos digitais e teorema da amostragem.

[Dew23] DewWool. 20 examples of projectile motion. https://dewwool.com/20-examples-of-projectile-motion/, 2023.

[Doc22] Scala Documentation. Regular expression patterns. https://docs.scala-lang.org/tour/regular-expression-patterns.html, 2022.

[Dun23] Jack Dunhill. New hypersonic missile engine could have almost double the range, claim chinese researchers, 2023.

[DZX+21] Mahdi Dibaei, Xi Zheng, Youhua Xia, Xiwei Xu, Alireza Jolfaei, Ali Kashif Bashir, Usman Tariq, Dongjin Yu, and Athanasios V Vasilakos. Investigating the prospect of leveraging blockchain and machine learning to secure vehicular networks: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(2):683–700, 2021.

[ECT23] ECT/UFRN. Integração numérica - método do trapézio - ect/ufrn. https://cn.ect.ufrn.br/index.php?r=conteudo%2Finteg-trapezio, 2023.

[EMO01] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Object-oriented and hybrid modeling in modelica. *Journal Européen des systèmes automatisés*, 35(4):395–404, 2001.

[Exp21] Net Explanations. Damped oscillations. https://www.netexplanations.com/damped-oscillations/, 2021.

[FCK+22] Akimasa Fujiwara, Makoto Chikaraishi, Diana Khan, Atsufumi Ogawa, Yoshihiro Suda, Toshikazu Yamasaki, Takaharu Nishino, and Shutaro Namba. Autonomous bus pilot project testing and demonstration using light rail transit track. *International Journal of Intelligent Transportation Systems Research*, 20(2):359–378, 2022.

[FMQ+15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. Keymaera x: An axiomatic tactical theorem prover for hybrid systems. In *International Conference on Automated Deduction*, pages 527–538. Springer, 2015.

[For23] World Economic Forum. Could autonomous aircraft make make flying safer and more sustainable? https://www.weforum.org/agenda/2023/08/autonomous-aircraft-could-make-flying-safer-easier-to-access-and-more-sustainable/, 2023.

[Gee] GeeksforGeeks. Relative motion velocity in two dimensions: Formulas & examples. https://www.geeksforgeeks.org/relative-motion-in-two-dimension/.

[Gee19a] GeeksforGeeks. Scala: Abstract type members. https://www.geeksforgeeks.org/scala-abstract-type-members/, 2019.

[Gee19b] GeeksforGeeks. Scala: Arrays. https://www.geeksforgeeks.org/scala-arrays/, 2019.

[Gee19c] GeeksforGeeks. Scala: Case class and case object. https://www.geeksforgeeks.org/scala-case-class-and-case-object/, 2019.

[Gee19d] GeeksforGeeks. Scala: Functions - basics. https://www.geeksforgeeks.org/scala-functions-basics/, 2019.

[Gee21] GeeksforGeeks. Scala string. https://www.geeksforgeeks.org/scala-string/, 2021.

[Geo22] GeoGebra. Geogebra clássico. https://www.geogebra.org/classic?lang=pt-PT, 2022.

[GNP] Sergey Goncharov, Renato Neves, and José Proença. Lince: Lightweight prototyping of hybrid programs.

[GNP19] Sergey Goncharov, Renato Neves, and José Proença. Lince: Lightweight prototyping of hybrid programs (full version). 2019.

[GNP20a] Sergey Goncharov, Renato Neves, and José Proença. Implementing hybrid semantics: From functional to imperative. In *International Colloquium on Theoretical Aspects of Computing*, pages 262–282. Springer, 2020.

[GNP20b] Sergey Goncharov, Renato Neves, and José Proença. Implementing hybrid semantics: From functional to imperative (extended version). Technical report, CISTER-Research Centre in Realtime and Embedded Computing Systems, 2020.

[GS15] Jeffery B Greenblatt and Susan Shaheen. Automated vehicles, on-demand mobility, and environmental impacts. *Current sustainable/renewable energy reports*, 2:74–81, 2015.

[GST09] Rafal Goebel, Ricardo G Sanfelice, and Andrew R Teel. Hybrid dynamical systems. *IEEE Control Systems*, 29(2):28–93, 2009.

[HDCW+19] Carl-Johan Hoel, Katherine Driggs-Campbell, Krister Wolff, Leo Laine, and Mykel J Kochenderfer. Combining planning and deep reinforcement learning in tactical decision making for autonomous driving. *IEEE transactions on intelligent vehicles*, 5(2):294–305, 2019.

[Hea22] IronRod Health. Implantable device monitoring. https://www.ironrod.health/implantable-device-monitoring, 2022.

[Hen16] Marcos Luiz Henrique. Mini curso: Análise numérica e computacional aplicada a equações diferenciais. *Universidade Federal de Goiás*, Nov 2016.

[HHMS16] Corey D Harper, Chris T Hendrickson, Sonia Mangones, and Constantine Samaras. Estimating potential increases in travel with autonomous vehicles for the non-driving, elderly and people with travel-restrictive medical conditions. *Transportation research part C: emerging technologies*, 72:1–9, 2016.

[HLLDS09] WPMH Heemels, D Lehmann, J Lunze, and B De Schutter. Introduction to hybrid systems. *Handbook of Hybrid Systems Control–Theory, Tools, Applications*, 2, 2009.

[HP22] Daseon Hong and Sungsu Park. Avoiding obstacles via missile real-time inference by reinforcement learning. *Applied Sciences*, 12(9):4142, 2022.

[Hu14] Fei Hu. *Cyber-physical systems*. Taylor & Francis Group LLC, 2014.

[Il17] Viktória Ilková and Adrian Ilka. Legal aspects of autonomous vehicles—an overview. In *2017 21st international conference on process control (PC)*, pages 428–433. IEEE, 2017.

[IIIR22] Mohamad Issa, Adrian Ilinca, Hussein Ibrahim, and Patrick Rizk. Maritime autonomous surface ships: Problems and challenges facing the regulatory process. *Sustainability*, 14(23):15630, 2022.

[Ins23] What's Insight. Damped oscillation: Formula and daily life examples. https://whatsinsight.org/damped-oscillation/, 2023.

[int23] intmathcom RSS. 8. Damping and the Natural Response in RLC Circuits. https://www.intmath.com/differential-equations/8-2nd-order-de-damping-rlc.php, 2023.

[KHdW15] Miltos Kyriakidis, Riender Happee, and Joost CF de Winter. Public opinion on automated driving: Results of an international questionnaire among 5000 respondents. *Transportation research part F: traffic psychology and behaviour*, 32:127–140, 2015.

[KST+21] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2021.

[KW17] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

[LDVF+11] Salvatore Luongo, Vittorio Di Vito, Giancarmine Fasano, Domenico Accardo, Lidia Forlenza, and Antonio Moccia. Automatic collision avoidance system: design, development and flight tests. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 5C1–1. IEEE, 2011.

[Lex23] Lexology. Legal challenges in autonomous flight: Things to consider before investing in an aircraft that flies itself. https://www.lexology.com/library/detail.aspx?g=e847b797-2239-4097-9013-7bfda4c9af9e, 2023.

[Lib23] LibreTexts. 2.5: Numerical integration - midpoint, trapezoid, simpson's rule. https://math.libretexts.org/Courses/Mount_Royal_University/MATH_2200%3A_Calculus_for_Scientists_II/2%3A_Techniques_of_Integration/2.5%3A_Numerical_Integration_-_Midpoint%2C_Trapezoid%2C_Simpson%27s_rule, 2023.

[Lim22] Acervo Lima. Classe e objeto em scala – acervo lima. https://acervolima.com/classe-e-objeto-em-scala/, 2022.

[Mai15] Gabrielle Maioli. Métodos numéricos para equações diferencias ordinárias. 2015.

[Mar21] Piping Mart. 8 applications of hydraulic systems in daily life. https://blog.thepipingmart.com/other/8-applications-of-hydraulic-systems-in-daily-life/, 2021.

[Mar23] Flying Cars Market. Cybersecurity and the challenges of self-driving aircraft. https://flyingcarsmarket.com/cybersecurity-and-the-challenges-of-self-driving-aircraft/, 2023.

[Mat21] Matics. What is cps (cyber physical system). https://matics.live/glossary/cyber-physical-system/, 2021.

[Mat23] MathWorks. MATLAB Simulink. https://www.mathworks.com/products/simulink.html, 2023.

[MBZ22] MBZUAI. Would you fly in a plane piloted solely by ai? https://mbzuai.ac.ae/news/would-you-fly-in-a-plane-piloted-solely-by-ai/, 2022.

[MNB17] Reza Matinnejad, Shiva Nejati, and Lionel C Briand. Automated testing of hybrid simulink/stateflow controllers: industrial case studies. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 938–943, 2017.

[MR03] Daniel D McCracken and Edwin D Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. 2003.

[Net23] Multiphysics Learning & Networking. Finite difference method. http://www.multiphysics.us/FDM.html#, 2023.

[Nev18] Renato Neves. *Hybrid programs*. PhD thesis, Minho University, 2018.

[Nev22a] Renato Neves. The adventurers' problem. *ARCA Uminho*, 2022.

[Nev22b] Renato Neves. Modelling and analysis of a cyber-physical system, 2022.

[NTB17] Mkhuseli Ngxande, Jules-Raymond Tapamo, and Michael Burke. Driver drowsiness detection using behavioral measures and machine learning techniques: A review of state-of-art techniques. *2017 pattern recognition Association of South Africa and Robotics and mechatronics (PRASA-RobMech)*, pages 156–161, 2017.

[Nut] The Forecast By Nutanix. Autonomous ships chart future supply chains. https://www.nutanix.com/theforecastbynutanix/industry/autonomous-cargo-ships-technology.

[Ope22] OpenStax. 5.3 projectile motion - physics. https://openstax.org/books/physics/pages/5-3-projectile-motion, 2022.

[ORI] 4.2 produto vetorial. https://ctec.ufal.br/professor/enl/aurb006/apostilas/Vetores%20-%20Produto%20Vetorial%20e%20Misto.pdf.

[oS22] Baeldung on Scala. Sealed keyword in scala. https://www.baeldung.com/scala/sealed-keyword, 2022.

[oS23] Baeldung on Scala. Introduction to traits in scala. https://www.baeldung.com/scala/traits, 2023.

[Phy23a] Physics. Projectile motion | physics. =https://courses.lumenlearning.com/suny-physics/chapter/3-4-projectile-motion/, 2023.

[Phy23b] A-Level Physics. Projectile motion. https://alevelphysics.co.uk/notes/projectile-motion/, 2023.

[Pla10] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

[Pla18a] André Platzer. Logical fundations of cyber-physical systems. In *Logical Fundations of Cyber-Physical Systems*, pages 28–30. Springer, 2018.

[Pla18b] André Platzer. Logical fundations of cyber-physical systems. In *Logical Fundations of Cyber-Physical Systems*, pages 85–87. Springer, 2018.

[PMP20] Đorđe Petrović, Radomir Mijailovic, and Dalibor Pešić. Traffic accidents with autonomous vehicles: Type of collisions, manoeuvres and errors of conventional vehicles' drivers. *Transportation Research Procedia*, 45:161–168, 01 2020.

[PP22] Ana-Maria Petri and Dorin Marius Petreus. Adaptive cruise control in electric vehicles with field-oriented control. *Applied Sciences*, 12(14), 2022.

[Pub22] Ibex Publishing. Amsterdam autonomous boats revitalise urban waterways, reduce emissions. https://ibexpub.media/amsterdam-autonomous-boats-revitalise-urban-waterways-reduce-emissions/, 2022.

[Ram16] Pedro Palma Ramos. Building a lexer and parser with scala's parser combinators. https://enear.github.io/2016/03/31/parser-combinators/, 2016.

[rlc23] Guide on resonant rlc circuits working and applications. https://www.elprocus.com/guide-on-resonant-rlc-circuits-working-and-application/, 2023.

[SBVP19] Gunnar Stevens, Paul Bossauer, Stephanie Vonholdt, and Christina Pakusch. Using time and space efficiently in driverless cars: Findings of a co-design study. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2019.

[Sec12] Gorkem Secer. A path-following algorithm for missiles. In *2012 IEEE Aerospace Conference*, pages 1–7. IEEE, 2012.

[SEN14] Sukumar SENTHILKUMAR. Special issue on analytical and approximate solutions for numerical problems. *Walailak J. Sci. Technol.*, 11:1–2, 2014.

[SH11] Kohei Suenaga and Ichiro Hasuo. Programming with infinitesimals: A while-language for hybrid system modeling. In *International Colloquium on Automata, Languages, and Programming*, pages 392–403. Springer, 2011.

[SH14] Christina Schilke and Peter Hecker. Dynamic route optimization based on adverse weather data. *Fourth SESAR Innovation Days*, 11, 2014.

[Sin22] Manpreet Singh. 10 applications of integration and differentiation in real life. https://numberdyslexia.com/applications-of-integration-and-differentiation-in-real-life/, 2022.

[SKDE23] Numan Senel, Klaus Kefferpütz, Kristina Doycheva, and Gordon Elger. Multi-sensor data fusion for real-time multi-object tracking. *Processes*, 11(2):501, 2023.

[SPA23] TS2 SPACE. The role of autonomous vehicles in the aviation industry. https://ts2.space/en/the-role-of-autonomous-vehicles-in-the-aviation-industry/, 2023.

[Str22] Strumenta. A guide to parsing: Algorithms and terminology. https://tomassetti.me/guide-parsing-algorithms-terminology/, 2022.

[Tab21] Mehrzad Tabatabaian. Example: A system with energy dissipation and applied external force. In *Engineering Systems Dynamics Modelling, Simulation, and Design: Lagrangian and Bond Graph Methods*. 2021.

[Ter13] Parr Terence. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.

[Top23] Toppr. Damped simple harmonic motion - toppr. https://www.toppr.com/guides/physics/oscillations/damped-simple-harmonic-motion/, 2023.

[tut] tutorialspoint. Scala tutorial. https://www.tutorialspoint.com/scala/index.htm.

[Uni23] Oslo Univesity. Chapter 11 numerical differentiation and integrationl. https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h08/kompendiet/diffint.pdf, 2023.

[Vox22] Vox. The unsinkable potential of autonomous boats. https://www.vox.com/recode/23270179/sea-machines-mayflower-self-driving-boats-autonomous-ai, 2022.

[Wal18] Jearl Walker. *Halliday & Resnick Fundamentals of Physics*. Wiley, 2018.

[WD22] Abdullah Abusorrah Wenli Duo, MengChu Zhou. A survey of cyber attacks on cyber physical systems: Recent advances and challenges, 2022.

[Wes22] Rosalie Wessel. What is automatic emergency braking? https://www.tomtom.com/newsroom/explainers-and-insights/what-is-automatic-emergency-braking/, 2022.

[Wik22] Brilliant Math & Science Wiki. Damped harmonic oscillators. https://brilliant.org/wiki/damped-harmonic-oscillators/, 2022.

[Wir21] Wired. The dire possibility of cyberattacks on weapons systems. https://www.wired.com/story/dire-possibility-cyberattacks-weapons-systems/, 2021.

[WLX+19] Wufei Wu, Renfa Li, Guoqi Xie, Jiyao An, Yang Bai, Jia Zhou, and Keqin Li. A survey of intrusion detection for in-vehicle networks. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):919–933, 2019.

[YGA15] Eray Yağdereli, Cemal Gemci, and A Ziya Aktaş. A study on cyber-security of autonomous and unmanned vehicles. *The Journal of Defense Modeling and Simulation*, 12(4):369–381, 2015.

[ZF] ZF. Autonomous driving: What you need to know. https://www.zf.com/mobile/en/technologies/autonomous_driving/autonomous_driving.html.

Part III
Appendices

Appendix A
Scala functions and hybrid programs

A.1 Variables of the file “Parser.scala”, responsible for recognizing non-linear expressions

```
lazy val notlinP: Parser[NotLin] =
  notlinParcelP ~ opt(("+" ~> notlinP) | ("-" ~> negnotLinP)) ^^ {
    case l1 ~ Some(l2) => l1 + l2
    case l1 ~ _ => l1

  }

private lazy val negnotLinP: Parser[NotLin] =
  notlinParcelP ~ opt(("+" ~> notlinP) | ("-" ~> negnotLinP)) ^^ {
    case l1 ~ Some(l2) => invertNotLin(l1) + l2
    case l1 ~ _ => invertNotLin(l1)
  }

lazy val notlinParcelP: Parser[NotLin] =
  "-" ~> notlinMultP ^^ invertNotLin |
    notlinMultP


lazy val notlinMultP: Parser[NotLin] =
  notlinDivP ~ opt("*" ~> notlinMultP) ^^ {
    case l1 ~ Some(l2) => Mult(l1, l2)
    case l1 ~ None => l1

  }

lazy val notlinDivP: Parser[NotLin] =
  notlinResP ~ opt("/" ~> notlinDivP) ^^ {
    case l1 ~ Some(l2) => Div(l1, l2)
    case l1 ~ None => l1

  }


lazy val notlinResP: Parser[NotLin] =
  notlinAtP ~ opt("%" ~> notlinResP) ^^ {
    case l1 ~ Some(l2) => Res(l1, l2)
    case l1 ~ None => l1

  }

  lazy val notlinAtP: Parser[NotLin] =
  notlinOthers ~ "^" ~ notlinOthers ^^ {
    case l1 ~ _ ~ l2 => Func("pow",List(l1, l2))
  } |
    notlinOthers


lazy val notlinOthers: Parser[NotLin] =
  "pi" ~ "(" ~ ")" ~ opt("^" ~> notlinOthers) ^^ {
    case _ ~ _ ~ _ ~ None => Func("PI", List())
```

```
    case _ ~ _ ~ _ ~ Some(l1) => Func("pow",List(Func("PI", List()), l1))
  } |
  "e" ~ "(" ~ ")" ~ opt("^" ~> notlinOthers) ^^ {
    case _ ~ _ ~ _ ~ None =>Func("E", List())
    case _ ~ _ ~ _ ~ Some(l1) => Func("exp", List(l1))
  } |
  "pow" ~ "(" ~ notlinP ~ "," ~ notlinP ~ ")" ^^ {
    case _ ~ _ ~ l1 ~ _ ~ l2 ~ _ => Func("pow",List(l1, l2))
  } |
  realP ^^ {
    Value
  } |
  identifier ~ "("~")" ^^ {
    case s ~ _ ~ _  => Func(s, List())
  } |
  identifier ~ opt("(" ~> argsFunction <~ ")") ^^ {
    case s ~ Some(arguments) => Func(s, arguments)
    case s ~ _ => Var("_" + s)
  } |
  "(" ~> notlinP <~ ")" ^^ {
    case l => l
  }


lazy val argsFunction: Parser[List[NotLin]] =
  notlinP ~ opt("," ~> argsFunction) ^^ {
    case n ~ Some(ns) => n :: ns
    case n ~ _ => List(n)
  }
```

A.2 The “apply” function from the “Eval.scala” file

```
def apply(state:Point, notlin: NotLin): Double = {
  val res = notlin match {
          case Var(v) => state(v)
          case Value(v) => v
          case Add(l1, l2) => apply(state,l1) + apply(state,l2)
          case Mult(l1,l2)  => apply(state,l1)  * apply(state,l2)
          case Div(l1,l2)  => {if (apply(state,l2)==0) {return throw new
              RuntimeException(s"Error:
          the divisor of the division '${Show.applyV(notlin)}' is zero.")}
                               else {return (apply(state,l1) / apply(state,l2))}
                              }
          case Res(l1,l2)  => {if (apply(state,l2)==0) {return throw new
              RuntimeException(s"Error:
          the divisor of the remainder '${Show.applyV(notlin)}' is zero.")}
                               else {return (apply(state,l1) % apply(state,l2))}
                              }
          case Func(s,list) => (s,list) match {
            case ("PI",Nil) => math.Pi
            case ("E",Nil) => math.E
            case ("max",v1::v2::Nil) => math.max(apply(state,v1), apply(state,v2))
            case ("min",v1::v2::Nil) => math.min(apply(state,v1), apply(state,v2))
            case ("pow",v1::v2::Nil) => {if(apply(state,v1)==0 && apply(state,v2)<0) return
                throw new
            RuntimeException(s"Error: The power of zero is undefined for a negative
                exponent:
            '${Show.applyV(notlin)}'.")
                                        else pow(apply(state,v1),apply(state,v2))
            }
            case ("exp",v::Nil) => math.exp(apply(state,v))
            case ("sin",v::Nil) => {if (multOfPi(apply(state,v))) {return 0}

                                    else {return math.sin(apply(state,v))}

            }
            case ("cos",v::Nil) => {if (multOfPiOn2(apply(state,v))) {return 0}

                                    else {return math.cos(apply(state,v))}
```

```
                }
                case ("tan",v::Nil) =>{if (multOfPi(apply(state,v))) {return 0}

                                      else {return math.tan(apply(state,v))}

                }
                case ("arcsin",v::Nil) => {
                  if ((math.asin(apply(state,v))).isNaN) return throw new
                      RuntimeException(s"Error:
                  In the expression '${Show.applyV(notlin)}', '${Show.applyV(v)}' is outside
                      the domain of
                  arcsin (-1<=x<=1).")
                  else math.asin(apply(state,v))
                }
                case ("arccos",v::Nil) => {
                  if ((math.acos(apply(state,v))).isNaN) return throw new
                      RuntimeException(s"Error:
                  In the expression '${Show.applyV(notlin)}', '${Show.applyV(v)}' is outside
                      the domain of
                  arccos (-1<=x<=1).")
                  else math.acos(apply(state,v))
                }
                case ("arctan",v::Nil) => math.atan(apply(state,v))
                case ("sinh",v::Nil) => math.sinh(apply(state,v))
                case ("cosh",v::Nil) => math.cosh(apply(state,v))
                case ("tanh",v::Nil) => math.tanh(apply(state,v))
                case ("sqrt",v::Nil) => {
                  if ((math.sqrt(apply(state,v))).isNaN) return throw new
                      RuntimeException(s"Error:
                  In the expression '${Show.applyV(notlin)}', '${Show.applyV(v)}' is outside
                      the domain of
                  sqrt (x>=0).")
                  else math.sqrt(apply(state,v))
                }
                case ("log",v::Nil) =>  {
                  if (apply(state,v)<=0) return throw new RuntimeException(s"Error:
                  In the expression '${Show.applyV(notlin)}', '${Show.applyV(v)}' is outside
                      the domain of
                  log (x>0).")
                  else math.log(apply(state,v))
                }
                case ("log10",v::Nil) =>  {
                  if (apply(state,v)<=0) return throw new RuntimeException(s"Error:
                  In the expression '${Show.applyV(notlin)}', '${Show.applyV(v)}' is outside
                      the domain of
                  log10 (x>0).")
                  else math.log10(apply(state,v))
                }
                 case (_,_) => throw new RuntimeException(s"Unknown function
                 '${s}(${(list.map(Show.applyV).toList).mkString(",")})', or the number of
                     arguments
                 are incorrect")

              }
      }
  res
}

 def multOfPi(number: Double): Boolean = {
  val eps = 1e-8 // Define a small value for tolerance
  val res = abs(number % math.Pi) // Calculate the remainder
  // Check if the remainder is within the tolerance range
  return res < eps || abs(res - math.Pi) < eps
}

def multOfPiOn2(number: Double): Boolean = {
  val eps = 1e-8 // Define a small value for tolerance
  val res = abs((number+math.Pi/2) % math.Pi) // Calculate the remainder
  // Check if the remainder is within the tolerance range
  return res < eps || abs(res - math.Pi) < eps
}
```

A.3 The “runge_kutta_func” function from the “SimpleSolver.scala” file

```
def runge_kutta_func(input:DValuation, eqs:List[DiffEq], t: Double, key_V:String):Double = {
 var N:Int = 75
 var h:Double=t/N
 val init = scala.collection.mutable.Map.empty[String, Double]
 init ++= input
 var acum:scala.collection.mutable.Map[String,Double]=init.clone()
 var k1:scala.collection.mutable.Map[String,Double] = init.clone().map{case (key,value) =>
     key -> 0}
 var k2:scala.collection.mutable.Map[String,Double] = init.clone().map{case (key,value) =>
     key -> 0}
 var k3:scala.collection.mutable.Map[String,Double] = init.clone().map{case (key,value) =>
     key -> 0}
 var k4:scala.collection.mutable.Map[String,Double] = init.clone().map{case (key,value) =>
     key -> 0}

 for (i <- 0 until N){
      for ((key, value) <- acum) {
      acum(key) = init(key)
      }
      for (deq <- eqs){
        k1(deq.v.v)=h*(Eval.applyAux(acum,deq.e))
      }
     for ((key, value) <- acum) {
      acum(key) = init(key)+k1(key)/2
      }
      for (deq <- eqs){
        k2(deq.v.v)=h*(Eval.applyAux(acum,deq.e))
      }
      for ((key, value) <- acum) {
      acum(key) = init(key)+k2(key)/2
      }
      for (deq <- eqs){
        k3(deq.v.v)=h*(Eval.applyAux(acum,deq.e))
      }
      for ((key, value) <- acum) {
      acum(key) = init(key)+k3(key)
      }
      for (deq <- eqs){
        k4(deq.v.v)=h*(Eval.applyAux(acum,deq.e))
      }
      for ((key, value) <- init) {
      init(key) = value + (k1(key) + 2*k2(key) + 2*k3(key) + k4(key))/6
      }
   }
 return init(key_V)
}
```

A.4 The “vars_in_min_max” function from “Utils.scala”

```
def vars_in_min_max(nl:NotLin):Double= nl match {
  case Var(v) => 0
  case Value(v) => 0
  case Add(l1, l2) => vars_in_min_max(l1) + vars_in_min_max(l2)
  case Mult(l1,l2)  =>if (extractTotalVarsLinearExp(l1)==0){
                        if (calc_doubles(l1)==0) 0
                        else vars_in_min_max(l2)
                      } else if (extractTotalVarsLinearExp(l2)==0){
                         if (calc_doubles(l2)==0) 0
                         else vars_in_min_max(l1)
                      } else vars_in_min_max(l1)  + vars_in_min_max(l2)
  case Div(l1,l2)  =>vars_in_min_max(l1)  + vars_in_min_max(l2)
  case Res(l1,l2)  => vars_in_min_max(l1)  + vars_in_min_max(l2)
  case Func(s,list) => (s,list) match {
```

```
    case ("PI",Nil) => 0
    case ("E",Nil) => 0
    case ("max",v1::v2::Nil) => extractTotalVarsLinearExp(v1)+extractTotalVarsLinearExp(v2)
    case ("min",v1::v2::Nil) => extractTotalVarsLinearExp(v1)+extractTotalVarsLinearExp(v2)
    case ("pow",v1::v2::Nil) => if (extractTotalVarsLinearExp(v2)==0) {
                                  if (calc_doubles(v2)==0) 0
                                  else vars_in_min_max(v1)
                                } else if (extractTotalVarsLinearExp(v1)==0){
                                  if (calc_doubles(v1)==1) 0
                                  else vars_in_min_max(v2)
                                } else return vars_in_min_max(v1)  + vars_in_min_max(v2)
    case ("exp",v::Nil) => vars_in_min_max(v)
    case ("sin",v::Nil) => vars_in_min_max(v)
    case ("cos",v::Nil) => vars_in_min_max(v)
    case ("tan",v::Nil) => vars_in_min_max(v)
    case ("arcsin",v::Nil) => vars_in_min_max(v)
    case ("arccos",v::Nil) => vars_in_min_max(v)
    case ("arctan",v::Nil) => vars_in_min_max(v)
    case ("sinh",v::Nil) => vars_in_min_max(v)
    case ("cosh",v::Nil) => vars_in_min_max(v)
    case ("tanh",v::Nil) => vars_in_min_max(v)
    case ("sqrt",v::Nil) => vars_in_min_max(v)
    case ("log",v::Nil) => vars_in_min_max(v)
    case ("log10",v::Nil) => vars_in_min_max(v)
    case (_,_) => throw new RuntimeException(s"Unknown function
        '${s}(${(list.map(Show.applyV).
    toList).mkString(",")})', or the number of arguments are incorrect")

}
}
```

A.5 The "extractVarsLinearExp" function from the "Utils.scala" file

```
def extractVarsLinearExp(notlin:NotLin,listOfVars:List[String]):Int = notlin match {
 case Value(value) =>  0
 case Var(v) => { if (listOfVars.contains(v))  1
                  else  0
                }
 case Add(l1,l2) =>
     math.max(extractVarsLinearExp(l1,listOfVars),extractVarsLinearExp(l2,listOfVars))
 case Mult(l1,l2) => if (extractTotalVarsLinearExp(l1)==0){
                       if (calc_doubles(l1)==0) 0
                       else extractVarsLinearExp(l2,listOfVars)
                     } else if (extractTotalVarsLinearExp(l2)==0){
                       if (calc_doubles(l2)==0) 0
                       else extractVarsLinearExp(l1,listOfVars)
                     } else (extractVarsLinearExp(l1,listOfVars) +
                       extractVarsLinearExp(l2,listOfVars))
 case Div(l1,l2) => (extractVarsLinearExp(l1,listOfVars) +
     2*extractVarsLinearExp(l2,listOfVars))
 case Res(l1,l2) => (2*extractVarsLinearExp(l1,listOfVars) +
     2*extractVarsLinearExp(l2,listOfVars))
 case Func(s,list) => funcextract(s,list,listOfVars)
}


def funcextract(s:String,list:List[NotLin],listOfVars:List[String]):Int = (s,list) match {
 case ("PI",List()) => 0
 case ("E",List()) => 0
 case ("max",List(n1,n2)) =>
     math.max(extractVarsLinearExp(n1,listOfVars),extractVarsLinearExp(n2,
 listOfVars))
 case ("min",List(n1,n2)) =>
     math.max(extractVarsLinearExp(n1,listOfVars),extractVarsLinearExp(n2,
 listOfVars))
 case ("pow",List(n1,n2)) => if (extractTotalVarsLinearExp(n2)==0) {
                               if (calc_doubles(n2)==0) 0
```

```
                                    else if (calc_doubles(n2)==1)
                                        extractVarsLinearExp(n1,listOfVars)
                                    else 2*extractVarsLinearExp(n1,listOfVars)
                                } else if (extractTotalVarsLinearExp(n1)==0){
                                    if (calc_doubles(n1)==1) 0
                                    else 2
                                } else return 2
  case ("exp",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("sin",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("cos",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("tan",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("arcsin",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("arccos",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("arctan",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("sinh",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("cosh",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("tanh",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("sqrt",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("log",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case ("log10",List(n)) => 2*extractVarsLinearExp(n,listOfVars)
  case (_,_) => throw new RuntimeException(s"Unknown function
      '${s}(${(list.map(Show.applyV).toList).
  mkString(",")})', or the number of arguments are incorrect")
  }
```

A.6 The "extractTotalVarsLinearExp" and "calc_doubles" functions from the "Utils.scala" file

```
def extractTotalVarsLinearExp(notlin:NotLin):Int = notlin match {
  case Value(value) =>  0
  case Var(v) => 1
  case Add(l1,l2) => math.max(extractTotalVarsLinearExp(l1),extractTotalVarsLinearExp(l2))
  case Mult(l1,l2) =>  if (extractTotalVarsLinearExp(l1)==0){
                        if (calc_doubles(l1)==0) 0
                        else extractTotalVarsLinearExp(l2)
                      } else if (extractTotalVarsLinearExp(l2)==0){
                         if (calc_doubles(l2)==0) 0
                         else extractTotalVarsLinearExp(l1)
                      } else (extractTotalVarsLinearExp(l1) + extractTotalVarsLinearExp(l2))
  case Div(l1,l2) => (extractTotalVarsLinearExp(l1) + 2*extractTotalVarsLinearExp(l2))
  case Res(l1,l2) => (2*extractTotalVarsLinearExp(l1) + 2*extractTotalVarsLinearExp(l2))
  case Func(s,list) => funcTotalextract(s,list)

 }


 def funcTotalextract(s:String,list:List[NotLin]):Int = (s,list) match {
  case ("PI",List()) => 0
  case ("E",List()) => 0
  case ("max",List(n1,n2)) =>
      math.max(extractTotalVarsLinearExp(n1),extractTotalVarsLinearExp(n2))
  case ("min",List(n1,n2)) =>
      math.max(extractTotalVarsLinearExp(n1),extractTotalVarsLinearExp(n2))
  case ("pow",List(n1,n2)) => if (extractTotalVarsLinearExp(n2)==0) {
                                    if (calc_doubles(n2)==0) 0
                                    else if (calc_doubles(n2)==1) extractTotalVarsLinearExp(n1)
                                    else 2*extractTotalVarsLinearExp(n1)
                                } else if (extractTotalVarsLinearExp(n1)==0){
                                    if (calc_doubles(n1)==1) 0
                                    else 2
                                } else return 2
  case ("exp",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("sin",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("cos",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("tan",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("arcsin",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("arccos",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("arctan",List(n)) => 2*extractTotalVarsLinearExp(n)
```

```
  case ("sinh",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("cosh",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("tanh",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("sqrt",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("log",List(n)) => 2*extractTotalVarsLinearExp(n)
  case ("log10",List(n)) => 2*extractTotalVarsLinearExp(n)
  case (_,_) => throw new RuntimeException(s"Unknown function
      '${s}(${(list.map(Show.applyV).toList).
  mkString(",")})', or the number of arguments are incorrect")
  }
}

 def calc_doubles(notlin:NotLin):Double = notlin match {
    case Var(v) => 1
    case Value(v) => v
    case Add(l1, l2) => calc_doubles(l1) + calc_doubles(l2)
    case Mult(l1,l2)  => calc_doubles(l1)  * calc_doubles(l2)
    case Div(l1,l2)  =>calc_doubles(l1) / calc_doubles(l2)
    case Res(l1,l2)  => calc_doubles(l1) % calc_doubles(l2)
    case Func(s,list) => (s,list) match {
      case ("PI",Nil) => math.Pi
      case ("E",Nil) => math.E
      case ("max",v1::v2::Nil) => math.max(calc_doubles(v1), calc_doubles(v2))
      case ("min",v1::v2::Nil) => math.min(calc_doubles(v1), calc_doubles(v2))
      case ("pow",v1::v2::Nil) => pow(calc_doubles(v1),calc_doubles(v2))
      case ("exp",v::Nil) => math.exp(calc_doubles(v))
      case ("sin",v::Nil) => {
        if (multOfPi(calc_doubles(v))) 0
        else math.sin(calc_doubles(v))
      }
      case ("cos",v::Nil) =>{
        if (multOfPiOn2(calc_doubles(v))) 0
        else math.cos(calc_doubles(v))
      }
      case ("tan",v::Nil) => {
        if (multOfPi(calc_doubles(v))) 0
        else math.tan(calc_doubles(v))
      }
      case ("arcsin",v::Nil) => math.asin(calc_doubles(v))
      case ("arccos",v::Nil) => math.acos(calc_doubles(v))
      case ("arctan",v::Nil) => math.atan(calc_doubles(v))
      case ("sinh",v::Nil) => math.sinh(calc_doubles(v))
      case ("cosh",v::Nil) => math.cosh(calc_doubles(v))
      case ("tanh",v::Nil) => math.tanh(calc_doubles(v))
      case ("sqrt",v::Nil) => math.sqrt(calc_doubles(v))
      case ("log",v::Nil) => math.log(calc_doubles(v))
      case ("log10",v::Nil) => math.log10(calc_doubles(v))
      case (_,_) => throw new RuntimeException(s"Unknown function
          '${s}(${(list.map(Show.applyV).toList).
      mkString(",")})', or the number of arguments are incorrect")
     }
    }
```

A.7 AEB program in Lince

```
p:=0; v:=10; // Initial position and velocity of the vehicle ,with AEB, controlled by a driver
ph:=0; vh:=10; // Initial position and velocity of the vehicle ,without AEB, controlled by a
    driver
pl:=40; vl:=0; al:=0; // Initial position, velocity and accelaration of the reference

aT:=-8; aA:=8; // Braking and accelerating acceleration of the vehicle
sampling_time:=0.01; // Time taken by the AEB to make a decision.
reaction_time:=1; //  Time taken by the Human to make a decision.

// counter and auxiliar variables
c := 1;
temp := aA;
aux:=aA;
```

```
//Start
while (v>0 || vh>0) do {
    // Vehicle without AEB (reaction is 100 times less)
    if (c == 100 && temp!=aT)
    then {
      c := 1;
      if ((ph + vh*reaction_time + aA/2*reaction_time^2 <
           pl+vl*reaction_time+al/2*reaction_time^2 ) &&
      (((vh+aA*reaction_time-vl-al*reaction_time)^2 -
           4*(ph+vh*reaction_time+aA/2*(reaction_time^2)-
      (pl +vl*reaction_time+al/2*(reaction_time^2)))*(aT/2-al/2))<0))
      then {
            temp := aA;
           }
      else {
            temp := aT;
           }
       }
    else {
      c := c + 1;
       }
    // Vehicle with AEB
    if (aux!=aT)
    then {
      if ((p + v*sampling_time + aA/2*sampling_time^2 <
           pl+vl*sampling_time+al/2*sampling_time^2 ) &&
      (((v+aA*sampling_time-vl-al*sampling_time)^2 -
           4*(p+v*sampling_time+aA/2*(sampling_time^2)-
      (pl +vl*sampling_time+al/2*(sampling_time^2)))*(aT/2-al/2))<0))
      then {
            aux:=aA;
           }
      else {
            aux:=aT;
           }
       }
    else skip;
    // Action
    if (vh<=0)
    then {
    if (v<=0)
        then  p'=0,v'=0,ph' = 0, vh' = 0, pl'=vl,vl'=al for sampling_time;
        else  p'=v,v'=aux,ph' = 0, vh' = 0, pl'=vl,vl'=al for sampling_time;
       }
    else {
    if (v<=0)
        then p'=0,v'=0,ph' = vh, vh' = temp, pl'=vl,vl'=al for sampling_time;
        else p'=v,v'=aux,ph' = vh, vh' = temp, pl'=vl,vl'=al for sampling_time;
       }
                        }
```

A.8 ACC program in Lince

```
p:=0; v:=20;
pl:=30; vl:=10;

aT:=-8; aL:=0;

safety_distance:=10;
sampling_time:=0.01;


while (true) do {
    if ((p + v*sampling_time < (pl-safety_distance) + vl*sampling_time+
    aL/2*sampling_time^2) && (((v-vl + (-aL)*sampling_time)^2 - 4*(p-(pl-safety_distance) +
    (v-vl)*sampling_time + (-aL)/2*sampling_time^2)*(aT-aL)/2)<0))
        then p'=v,v'=0,pl'=vl,vl'=aL for sampling_time;
        else p'=v,v'=aT,pl'=vl,vl'=aL for sampling_time;
                }
```

A.9 Missile vs target program in Lince

```
// Initial position and velocity of the missile
x:=300; vx:=20;
y:=300; vy:=0;
// Initial position and velocity of the target
xl:=500; vxl:=15;
yl:=500; vyl:=0;

// Angular velocity of the missile
aw:=(1/20)*2*pi();
// Angular velocity of the target
awl:=(1/40)*2*pi();

// Counter
cont:=0;
// Decision time
sampling_time:=0.1;
// Minimum collision distance
dist_min_col:=1;
// variable that stores the alpha angle
alpha:=0;
//Variable that stores the vectorial product to decide which way to turn
vect_P:=0;
// Variables that stores the angular velocity decision to the missile and the target
w:=0;
wl:=0;
//Variables that stores the relative positions and velocities
dx:=0;
dy:=0;
vrelx:=0;
vrely:=0;

// Run the following programme whilst the distance between the missile and the target is
    greater than
//the collision distance
while (sqrt((x-xl)^2+(y-yl)^2)>dist_min_col) do {
    //Conditional structures to establish the target path
    if (cont<=100)
    then wl:=0;
    else {
          if (cont<=200)
          then wl:=-awl;
          else {
               if (cont<=300)
               then wl:=awl;
               else wl:=0;
               }
         }
    // The counter is incremented
    cont:=cont+1;
    //Update distances and relative velocities
    dx:=xl-x;
    dy:=yl-y;
    vrelx:=vxl-vx;
    vrely:=vyl-vy;
    // Determine the value of the angle alpha
    alpha:=arccos((vrelx*dx + vrely*dy)/(sqrt(vrelx^2 + vrely^2)*sqrt(dx^2 + dy^2)));
    // Conditional structures to determine whether the missile needs to move forward or make
        a curve
    if (alpha>=179.5*pi()/180 && alpha<=180.5*pi()/180)
    then {
         // If the theta is  between 179.5 and 180.5 degrees, the missile follows a straight
             line at a constant velocity
         w:=0;
         }
    else {
         // Determine the value of the vetorial product between the relative velocity vector
             and the relative position vector
         vect_P:=vrelx*dy-vrely*dx;
         // If the theta is not between 179.5 and 180.5 degrees, the missile needs to curve
             to the left or right
```

```
        // To decide which way to turn, simply check the sign of the vectorial product.
        if (vect_P>=0)
        then {
             // If the vectorial product is positive or zero,  it curves to the right
              w:=aw;
              }
         else {
              // If the vectorial product is negative,  it curves to the left
              w:=-aw;
              }
        }
    // Differential equations
    x'=vx,y'=vy,vx'=w*vy,vy'=-w*vx,
    xl'=vxl,yl'=vyl,vxl'=wl*vyl,vyl'=-wl*vxl for sampling_time;
                        }
```

A.10 Damped harmonic oscillator program in Lince

```
m:=1; // mass of the object
k:=2.32; // Spring constant

// damping coefficient and initial values of the underdamping regime
b_sc:=1; //damping coefficient
xsc:=2; //Initial position
vsc:=0; //Initial velocity

// damping coefficient and initial values of the overdamping regime
b_Sc:=3.5 ; //damping coefficient  regime
xSc:=2; //Initial position
vSc:=0; //Initial velocity

// damping coefficient and initial values of the critical damping regime
b_c:=2*sqrt(k*m); //damping coefficient
xc:=2; //Initial position
vc:=0; //Initial velocity

// Diferential equations
xsc'=vsc,vsc'=-xsc*k/m-vsc*b_sc/m,
xSc'=vSc,vSc'=-xSc*k/m-vSc*b_Sc/m,
xc'=vc,vc'=-xc*k/m-vc*b_c/m for 15;
```

A.11 Projectile motion program in Lince

```
x:=2;
y:=2;
v0:=10;
g:=9.8;
theta:=pi()/4;
vx:=v0*cos(theta);
vy:=v0*sin(theta);

x'=vx,y'=vy,vx'=0,vy'=-g until_0.01 (y<=0);
```

A.12 Program in Lince of the RLC series eletrical circuit in the three regimes

```
vc_rac:=0;
vc_rsa:=0;
vc_rSa:=0;
dvc_rac:=0;
```

```
dvc_rsa:=0;
dvc_rSa:=0;
vs:=10;

l:=0.047;
c:=0.047;
r_rac:=2;
r_rsa:=0.5;
r_rSa:=4;

vc_rac'=dvc_rac, dvc_rac'=-dvc_rac*r_rac/l-vc_rac/(l*c)+vs/(l*c),
vc_rsa'=dvc_rsa, dvc_rsa'=-dvc_rsa*r_rsa/l-vc_rsa/(l*c)+vs/(l*c),
vc_rSa'=dvc_rSa, dvc_rSa'=-dvc_rSa*r_rSa/l-vc_rSa/(l*c)+vs/(l*c) for 1;

vs:=0;
vc_rac'=dvc_rac, dvc_rac'=-dvc_rac*r_rac/l-vc_rac/(l*c)+vs/(l*c),
vc_rsa'=dvc_rsa, dvc_rsa'=-dvc_rsa*r_rsa/l-vc_rsa/(l*c)+vs/(l*c),
vc_rSa'=dvc_rSa, dvc_rSa'=-dvc_rSa*r_rSa/l-vc_rSa/(l*c)+vs/(l*c) for 1;
```

A.13 Program in Lince of the hydraulic system

```
a:=2; //Area of tank 1
r:=6; //Resistance
f_in:=8; //inflow rate
h:=5; //initial water height

repeat 40 {
      if (h<9)
      then f_in:=8;
      else f_in:=0;

      h'=f_in/a-h/(r*a) for 0.1;
           }
```

A.14 Program in Lince of the numerical derivative and integral

```
h:=0.1;
y:=0; dy:=0;
yi:=y; yhi:=yi; y2hi:=yhi;
dyi:=0; int_i:=0;
aux:=0;

repeat 1000 {
    y2hi:=yhi;
    yhi:=yi;
    yi:=y;

    if (aux>=2)
    then {
    dyi:=(3*yi-4*yhi+y2hi)/(2*h);
    int_i:=(h/2)*(yhi+yi)+ int_i;
    y'=dy, dy'=2 for h;
    aux:=aux+1;
         }
    else {
      if (aux==0)
      then {
      y'=dy,dy'=2 for h;
      aux:=aux+1;
           }
      else {
      dyi:=(yi-yhi)/(h);
      int_i:=(h/2)*(yhi+yi)+ int_i;
      y'=dy, dy'=2 for h;
      aux:=aux+1;
```

```
    }
}
    }
```

Appendix B
Old and new version syntax and Scala features

B.1 The Syntax of the Old Lince's Language

The following diagram represents the syntax extracted from the old version of Lince's language, where the terminal symbols are represented by regular expressions or characters between quotation marks:

```
progP =seqP
seqP =basicProg
     |basicProg seqP
basicProg ="skip;"
     |"skip for" realP ";"
     |"while" whileGuard "do{" seqP "}"
     |"repeat" intPP "{" seqP "}"
     |"if" condP "then" blockP "else" blockP
     |"wait" linP ";"
     | atomP
blockP ="{" seqP "}"
     |basicProg
whileGuard =condP
     |intPP
atomP =identifier ":=" linP ";"
     |diffEqsP ";"
     |diffEqsP durP ";"
diffEqsP =identifier"'=" linP
     |identifier "'=" linP "," diffEqsP
durP ="until" condP
     |"until" untilArgs condP
     |"for" linP
untilArgs ="_" realP
     |"_" realP "," realP
linP =linParcelP
     |linParcelP "+" linP
     |linParcelP "-" negLinP
negLinP =linParcelP
     |linParcelP "+" linP
     |linParcelP "-" negLinP
linParcelP ="-" linMultP
     |linMultP
linMultP = realP
     |realP "*" linAtP
     |linAtP
     |linAtP "*" realP
linAtP =identifier
     |"(" linP ")"
condP =disjP
     |disjP "/\" condP
disjP =equivP
     |equivP "\/" disjP
equivP =negP
     |negP "<->" equivP
negP ="!(" condP ")"
     |"(" condP ")"
     |bopP
bopP ="true"
     |"false"
     |identifier bcontP
bcontP ="<=" linP
     |">=" linP
     |"<" linP
     |">" linP
     |"==" linP
     |"!=" linP
intPP ="(" intP ")"
     |intP
```

"intP", "identifier" and "realP" are defined by the following regular expressions:

$$\begin{aligned}\text{intP} &= [0-9]+\\ \text{identifier} &= [a-z][a-zA-Z0-9\_]*\\ \text{realP} &= -?[0-9]+(\backslash.([0-9]+))?\end{aligned}$$

B.2 The Syntax of the New Lince's Language

The following diagram represents the syntax extracted from the new version of Lince's language, where the terminal symbols are represented by regular expressions or characters between quotation marks:

```
progP =declr
declr =atomP
      |atomP seqP
seqP =basicProg
      |basicProg seqP
basicProg ="skip;"
      |"skip for" realP ";"
      |"while" whileGuard "do{" seqP "}"
      |"repeat" intPP "{" seqP "}"
      |"if" condP "then" blockP "else" blockP
      |"wait" notlinP ";"
      | atomP
blockP ="{" seqP "}"
      |basicProg
whileGuard =condP
      |intPP
atomP =identifier ":=" notlinP ";"
      |diffEqsP ";"
      |diffEqsP durP ";"
diffEqsP =identifier "'=" notlinP
      |identifier "'=" notlinP "," diffEqsP
durP ="until" condP
      |"until" untilArgs condP
      |"for" notlinP
untilArgs ="_" realP
      |"_" realP "," realP
notlinP =notlinParcelP
      |notlinParcelP "+" notlinP
      |notlinParcelP "-" negnotLinP
negnotLinP =notlinParcelP
      |notlinParcelP "+" notlinP
      |notlinParcelP "-" negnotLinP
notlinParcelP ="-" notlinMultP
      |notlinMultP
notlinMultP =notlinDivP
      |notlinDivP "*" notlinMultP
notlinDivP =notlinResP
      |notlinResP "/" notlinDivP
notlinResP =notlinAtP
      |notlinAtP "%" notlinResP
notlinAtP =notlinOthers "∧" notlinOthers
      |notlinOthers
notlinOthers ="pi()"
      |"pi()∧" notlinOthers
      |"e()"
      |"e()∧" notlinOthers
      |"pow(" notlinP "," notlinP ")"
      | realP
      |identifier "()"
      |identifier
      |identifier "(" argsFunction ")"
      |"(" notlinP ")"
argsFunction =notlinP
      notlinP "," argsFunction
condP =disjP
      |disjP "&&" condP
disjP =equivP
      |equivP "||" disjP
equivP =negP
      |negP "<=>" equivP
negP ="!(" condP ")"
      |bopP
      |"(" condP ")"
      |condP
bopP ="true"
      |"false"
      |notlinP "<=" notlinP
      |notlinP ">=" notlinP
      |notlinP "<" notlinP
      |notlinP ">" notlinP
      |notlinP "==" notlinP
      |notlinP "!=" notlinP
intPP ="(" intP ")"
      |intP
```

Where "intP", "identifier" and "realP" are defined by the following regular expressions:

$$\begin{aligned} \text{intP} &= [0-9]+ \\ \text{identifier} &= [a-z][a-zA-Z0-9\_]* \\ \text{realP} &= -?[0-9]+(\backslash.([0-9]+))? \end{aligned}$$

B.3 Data types and variable declaration in Scala

The **data types** available in Scala are **Byte**, **Short**, **Int**, **Long**, **Float**, **Double**, **Char**, **String**, **Boolean**, **Unit**, **Null**, **Nothing**, **Any** and **AnyRef** [tut, Section-Data Types]. Based on the reference [tut, Section Variables], a variable is a place in memory reserved to store a certain value, and is assigned a memory space and determined what can be stored in it based on the data type of the variable. In Scala, one can declare variables that can be changed throughout the program (mutable variables) and variables that cannot be changed (immutable variables). To declare a mutable variable one uses the keyword "**var**" before the variable name:

$$\text{var variable\_name: data\_type} = ...$$

To declare a immutable variable one uses the keyword "**val**" before the variable name:

$$\text{val variable\_name: data\_type} = ...$$

Another important aspect concerning variables is that the Scala compiler has the ability to discover what is the variable's type based just on the value assigned, i.e., it is not obligatory to declare the variable's type.

B.4 Operators in Scala

As in other languages, in Scala there are **operators**. These operators are [tut, Section Operators]:

- Arithmetic operators;
- Relational operators;
- Logic Operators;
- Bitwise operators;
- Assignment Operators;

Arithmetic operators:

Operator	Description
+	Adds two operands
-	Subtracts two operands
*	Multiplies two operands
/	Divides two operands
%	Remainder of the division of two operands

Table 4: *Arihtemetic operators*

Relational operators:

Operator	Description
==	Checks whether the values of two operands are equal (true) or not (false)
!=	Checks whether the values of two operands are different (true) or not (false)
>	Checks whether the value of the left operand is greater than the right operand (true) or not (false)
<	Checks whether the value of the left operand is less than the right operand (true) or not (false)
>=	Checks whether the value of the left operand is greater than/equal to the right operand (true) or not (false)
<=	Checks whether the value of the left operand is less than/equal to the right operand (true) or not (false)

Table 5: *Relational operators*

Logic Operators:

Operator	Description
&&	AND logic operator. If both operands are non-zero it is true, otherwise it is false
||	OR logical operator. If any of the operands are non-zero it is true, otherwise it is false
!	NOT logic operator. Used to reverse the logical state of its operand. If a condition is true, it turns false and vice versa

Table 6: *Logic operators*

Bitwise operators:

p	q	p & q	p | q	p $\wedge$ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Table 7: *Bitwise operators*

Assignment Operators:

Operator	Description
=	Assigns the result of the right operand to the left operand
+=	Sums the left operand and the right operand, assigning the result to the left operand
-=	Subtracts the left operand and the right operand, assigning the result to the left operand
*=	Multiplies the left operand and the right operand, assigning the result to the left operand
/=	Divides the left operand and the right operand, assigning the result to the left operand
%=	Takes the remainder of the integer division of the left operand and the right operand, assigning the result to the left operand
«=	Shifts right of the left operand with the right integer, assigning the result to the left operand
»=	Shifts left of the left operand with the right integer, assigning the result to the left operand
&=	Performs the binary bitwise AND operator of the left operand with the right operand, assigning the result to the left operand
/\=	Performs the binary bitwise OR operator of the left operand with the right operand, assigning the result to the left operand
|=	Performs the binary bitwise XOR operator of the left operand with the right operand, assigning the result to the left operand

Table 8: *Assignment operators*

B.5 Loops, conditional structures and functions in Scala

The conditional structure **if-else** can be declared in three ways, one of which the **else** is omitted [tut, Section-IF ELSE]:

```
    if(boolean expression) {
// Instructions if boolean expression is true
}
```

Another option to declare both clauses:

```
    if(boolean expression) {
// Instructions if the Boolean statement is true
}else{
// instruction if the Boolean expression is false
}
```

And the third amounts to conditionals in chain:

```
    if(Boolean expression 1) {
// Instructions if Boolean expression 1 is true
}else if( Boolean expression 2){
// Instructions if Boolean expression 2 is true
}else if( Boolean expression 3){
// Instructions if Boolean expression 3 is true
}else {
// Instructions if none of them is true
}
```

(**Note:** The Scala language allows nested **if-else** statements, i.e. putting **if-else** statements inside another **if-else** statement.)

Regarding the **while** loop structure, it executes the code block as long as the condition is true, testing this condition before entering the loop. When this condition is false, the while statement ends [tut, Section Loop Statements].

A **while** loop is declared as follows:

```
while(condition){
    //instruction/s
}
```

The **do-while** loop structure is very similar to the previous one – it only differs in that it tests the condition at the end of the loop body, causing the body to be executed at least once [tut, Section-Loop Statements]. A **do-while** loop is declared as follows:

```
do{
   //instruction/s;
}
while(condition)
```

The cyclical **for** structure, on the other hand, executes its body while the declared range is not totally swept, assigning to the variable "x" the value of the range relative to the execution in question [tut, Section Loop Statements]. The cyclical **for** structure is declared as follows:

```
for( var x <- Range ){
   //instruction/s
}
```

To conclude this section, let us briefly look at **functions**. Based on [Gee19d], a function is a set of instructions that perform a certain task. Their main goal is to avoid writing the same code several times for different inputs, making it only necessary to invoke the function. In Scala it is common to have confusion between methods and functions – what distinguishes them is that functions are objects that can be stored in a variable and methods always belong to a class. Basically, one can say that methods are functions that are members of some class. A **function** is declared as follows:

```
def function_name ([parameter list]):[data_type_to_return] ={
    // function body
}
```

B.6 String and Arrays in Scala

In Scala, as could not be missed, there is the possibility to create **Strings** and **Arrays**. As in other languages, in Scala a **String** is a sequence of characters. Scala provides a series of access methods that can be applied to Strings in order to acquire certain outputs, to view in detail these methods see reference [Gee21]. Here is an example concerning the declaration of Strings:

```
val str: String="ola"
\\or
val str="ola"
```

Arrays are fixed size, mutable data structures which store elements of the same type. In Arrays, the index of the first element is zero, and there is the possibility to create multidimensional Arrays (matrices, for example). Elements in the Array can be accessed, altered, and added. Additionally, two Arrays can be concatenated using the .concat() method, among other operations. To find more features that can be

done with Arrays, it is recommended to read [Gee19b] in more detail. Here is an example concerning the declaration of Arrays:

```
var days = Array("Sunday", "Monday", "Tuesday","Wednesday", "Thursday",
"Friday","Saturday" )
```

B.7 Classes and Objects in Scala

One of the most important concepts in an object-oriented language, such as Scala, is that it boils down to **classes** and **objects**. **Classes** are a kind of prototype defined by the programmer from which objects are created. In classes, one can combine fields and methods, where fields are variables defined within the class and methods are functions defined within the class [Lim22]. A simple way of understanding the concept of class is that it works as a data type of the object, but a data type that can be stuffed with fields and methods to be used by a corresponding object in the future. To declare a class one must [Lim22]:

1. Insert the keyword *class*;
2. Next, put the name that we would like to give to the class (must start with a uppercase letter);
3. Write down the class attributes;
4. Then, if necessary, insert the keyword "extends" followed by the parent class to create an inherited class (a concept that is explained later);
5. Finally just put the class body inside curly braces.

To create an **object**, it is necessary to instantiate a class, and as mentioned before, a class can be thought of as a data type. To take advantage of what this data type has to offer, it is necessary to instantiate the class using the keyword *new* and form an object in which one can access fields (as long as they're preceded by the keywords *val* or *var*) and the methods of the respective class [Lim22].

An example of creating a class and instantiating it by forming an object and using the class method is as follows:

```
class Aluno(var nota:Int,var nome:String){

    // Display method
    def display(aprov:Boolean)={
        println("Student's name:"+ nome)
        println("Student´s note:"+ nota)
        println("Approved?"+ aprov)
    }
}

object student
{
    // main method
```

```
    def main(args:Array[String])
    {
        //  Object al instantiated from the Aluno class
        var al = new Aluno(16,Ricardo)
        al.display(true)
    }
}
```

In the previous example, the class "Aluno" receives two arguments, one referring to the grade that the student received and the other referring to his name, in turn the body of the class only contains the method "display" which only takes as argument the boolean that indicates if the student passed or failed and prints out the respective grade, name and boolean. Finally, the "student" object is defined, which within the "main" method instantiates the "Aluno" class with certain arguments and accesses the "display" method with a given argument to visualize the prints.

One of the most important and useful features of classes in Scala is their ability to be **inherited**. An **inheriting class** will "inherit" the non-private members of the **parent class**, as well as becoming a subclass of it, i.e. an inheriting class will be able to enjoy the non-private fields and methods of the parent class. To create an heir class, one can simply use the keyword *extends*, followed by the name of the parent class, in front of the arguments of the heir class being created. Inheriting classes can only inherit from one parent class (except for a trait class) and they can rewrite the fields and methods of the parent class if they precede them with the keyword *override*. See more details in [tut, Section Classes & Objects]. An example of the use of inherited classes, which can be found in reference [tut, Section Classes & Objects], is as follows:

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("Point x location : " + x);
      println ("Point y location : " + y);
      println ("Point z location : " + z);
   }
}

object Demo {
   def main(args: Array[String]) {
      val loc = new Location(10, 20, 15);
```

```
        // Move to a new location
        loc.move(10, 10, 5);
    }
}
```

In this example, a class called “Point” was created, which receives two arguments of type integer that refer to the x and y coordinates of a point. Inside the class “Point”, there is the method “move” that also receives two arguments of type integer that represent the displacements that the x and y coordinates must suffer, and as such this method adds these displacements to the x and y coordinates and prints the new value of these coordinates.

Next, it was created the class inheritor of the class “Point” which is given the name of “Location”. It receives three arguments that are nothing more than the rewriting of the two arguments of the parent class and the value of the integer type referring to the z coordinate. This inheritor class contains a method very similar to the parent class, only changing the fact that it adds also the displacement of the z coordinate and makes its print. At the end, the “Demo” object was created and within the “main” method the inheritor class was instantiated, assigning values to its arguments and then accessing the “move” method to add the desired displacements and print the result.

This work was partially supported by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference PTDC/CCI-COM/4280/2021.