RICE UNIVERSITY

Acumen: An Environment for Rapid Prototyping
of Cyber-physical Systems

by

Yun Zhu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

______________________________
Dr. Walid Taha (Chair),
Assistant Professor,
Computer Science

______________________________
Dr. Robert Cartwright,
Professor,
Computer Science

______________________________
Dr. Marcia K. O’Malley,
Assistant Professor,
Mechanical Engineering

HOUSTON, TEXAS
AUGUST, 2009
Acumen: An Environment for Rapid Prototyping of Cyber-physical Systems

Yun Zhu

Abstract

Cyber-Physical Systems (CPS) combine discrete and continuous physical processes. Developing a new cyber-physical system is an iterative process that involves design, simulation, prototyping, and production. Writing simulation code using current techniques is time consuming. My thesis is that a carefully designed, domain-specific language can alleviate this difficulty. To demonstrate this thesis, I designed and implemented a prototype of a novel simulation environment called Acumen. The centerpiece of Acumen is a physical description language called PhyDL, which allows the user to directly describe dynamic equations governing the system being modeled, making the writing of simulation code much easier for engineers. A series of automatic transformations convert the high-level descriptions into a form that is directly machine executable. This thesis presents the design of PhyDL and several case studies of using PhyDL for designing CPS systems, which show that PhyDL is accurate and easy to use for domain experts.
Acknowledgments

My three years at Rice University working toward my MS degree have been a truly amazing journey. They enriched me with a wealth of experiences and knowledge, and I want to take this opportunity to express my gratitude to all who have made this journey so pleasant and enjoyable.

This thesis would not have been possible without the constant guidance and immense patience of my advisor, Prof. Walid Taha. I still remember the days when I started my study at Rice, during which he worked with me patiently and brought me up to speed in my understanding of research, comprehension of techniques, and the skills for communicating technical problems and ideas. He also encouraged me to participate in a variety of academic and non-academic activities during the three years, which in turn made my life colorful and enjoyable. I really appreciate his guidance and kindness towards me, especially when I was going through rough times in my life. His organization skills and passion for getting everything done in an efficient way are admirable. His enthusiasm for research and life taught me a great deal.

I would like to thank Dr. Robert Cartwright and Dr. Marcia K. O’Malley for serving on my thesis committee. I would also like to thank them both for their guidance and the valuable discussions we had on my research and on my classes. I truly appreciated their time and patience in carefully reading through my thesis. Dr. O’Malley contributed to Acumen design and development by providing inspiring case studies and knowledge on mechanical engineering. I am also grateful for all of her help on my technical writing. I would also like to express my sincere gratitude to Dr. Jan Hewitt and Ray Hardesty for agreeing to help me with my writing on very short notice and giving me excellent advices regarding thesis writing.
I want to give my special thanks to Schlumberger and NSF for funding this research. The two internships I have done at Schlumberger were a very valuable experience to me. Thanks are due to Michael Nguyen, Lise Hvatum, Shyam Mehta, Steve Gomez, Carol Prochnow, Hifzi Ardic, and Ashley Kishino for their administrative support. Thanks are also due to Shilpa Vaze, Kai Huang, Robin Ewan, and all of the other Schlumberger engineers I have worked with during my internship for their technical assistance.

My research would not have reached this stage without the contributions of the Acumen group. Marisa Linnea Peralta developed various examples of Acumen as supplemental materials to guide the user through the modeling process using Acumen. Kevin Hirshberg modeled the suspension of a 1989 Alfa Romeo Milano in Acumen. Alex Stoll modeled a robot arm comprising a number of segments. Joshua Langsfeld, Jun Inoue, Dr. Marcie O’Malley, and Dr. Walid Taha studied the stability issue in the context of modeling physical systems in Acumen. Laura Shepard composed the first version of the user guide. Acumen group members also made contributions to the development of the Acumen software in the areas of usability, bug-finding, and suggestions for improvement.

Besides research, all of the classes I have taken here at Rice have added immensely to my knowledge and skills. It is my honor and luck to have had a chance to take classes with such impressive teachers. I would like to give my special thanks to Dr. Luay Nakhleh, Dr. Moshe Vardi, Dr. Lydia Kavraki, Dr. T.S. Eugene Ng, Dr. Scott Rixner, and many other professors. I also would like to thank the Department of Computer Science and all of its administrative staff for all the service and help I have received on so many occasions.

It is really a pleasure to have companions and colleagues in Dr. Taha’s research group at Rice. Jun Inoue is truly gifted and can always find the most efficient and right way to solve problems. He implemented the interactive GUI in Acumen, and his comments and suggestions always inspired me to constantly improve my skills in
programming. I am grateful to Cherif Salama, Dr. Ronald Garcia, and Dr. Edwin Westbrook. They are great researchers, and their skills and passion to keep everything neat and tidy are admirable. I have received a lot of help from each of them, especially in reviewing all the technical papers I wrote and talks I gave.

There are friends and elders whom I have been very fortunate to meet, and who helped me in endless ways. Lily Lam from the Office of International Students and Scholars has provided invaluable help with immigration issues, she and her husband Pastor Brian Lam also gave me tremendous guidance in my spiritual life. I would also like to thank Xing Fang, Zhenzhen Lai, Wei Zhao, Qingyan Kong, Ming Yang, Guohui Wang, Zheng Cai, Jing Bai, Donghua Liu, Haiying Pang, and many others for all the encouragement and good times that we shared together.

It is a blessing to have people who are always with you to share the detailed stories and feelings in your life. I met my fiance Bo Zhang three years ago when I started my life at Rice. He is a friend of the kind that I can count on for any help at any time. There are times I became stressed from my research, and his understanding, tolerance, thoughtfulness, and kindness made my life much easier.

Finally, it would have been impossible without my family, especially my Mom and Dad. Thank you all for everything — your love, support, encouragement, and understanding. No words can convey how thankful I am for that.
## Contents

Abstract .................................................. ii
Acknowledgments ........................................... iii
List of Illustrations .......................................... x

1 Introduction ............................................... 1
   1.1 A Case for Simulation .................................. 2
   1.2 Developing Simulation Codes by Hand ................. 7
   1.3 Acumen: A Tool for Making Simulation Easier .......... 9
      1.3.1 PhyDL and RIDL .................................. 9
      1.3.2 Evaluation of Acumen ............................. 11
   1.4 Summary of Contributions .............................. 11

2 Related Work ............................................. 13
   2.1 Relation to Other Systems ............................. 13
      2.1.1 Tools Based on Block Diagrams ................. 13
      2.1.2 Tools Based on Symbolic Computation ............ 16
      2.1.3 Solidworks and ProEngineer .................... 18
      2.1.4 Tools Based on Functional Programming .......... 18
   2.2 Detailed Comparison with MATLAB/Simulink .......... 19
      2.2.1 Relating MATLAB/Simulink to Acumen ............ 19
      2.2.2 Comparing MATLAB/Simulink to Acumen with Small
          Examples .......................................... 20
      2.2.3 The REPTILE Project ............................. 27
      2.2.4 Simulink/SimMechanics ......................... 30
2.2.5 Summary ................................................. 31
2.3 Detailed Comparison with Modelica .......................... 32
2.4 Summary .................................................. 33

3 The Design of PhyDL ........................................ 34
  3.1 Causal and Acausal Modeling ............................... 35
  3.1.1 The RLC Electrical Circuit Example .................... 36
  3.1.2 The Acausal Model of PhyDL for the Circuit ........... 39
  3.2 Point-Free Notation ...................................... 39
  3.3 Mathematically Inspired Textual Notation .................. 41
  3.4 Module System .......................................... 42
  3.5 Bus Suspension Example in PhyDL .......................... 44

4 The Syntax and Semantics of PhyDL ........................ 45
  4.1 Syntax ................................................... 45
  4.2 Static Semantics ........................................ 48
  4.3 Interpreting Acumen Programs ............................. 51
    4.3.1 Semantics for a Simple Pendulum Example ........... 52
    4.3.2 Module Instantiation ................................ 54
    4.3.3 Topological Sorting ................................ 54
    4.3.4 Order Reduction .................................... 57
    4.3.5 Discretization ...................................... 57
  4.4 Quantitative Evaluation of Acumen Accuracy ............... 58

5 Error Models of ODE Solvers ................................. 61
  5.1 Numerical Methods and Source of Errors ................... 61
    5.1.1 Local Truncation Errors ............................. 63
    5.1.2 Propagation of the Error ............................. 64
    5.1.3 Round-off Errors .................................... 65
5.2 Error Estimation for Numerical Methods . . . . . . . . . . . . . . . . 66
  5.2.1 Estimating the Local Truncation Error . . . . . . . . . . . . . . . . 66
  5.2.2 Estimating the Global Error . . . . . . . . . . . . . . . . . . . . . . 70
  5.2.3 Effect of Round-off Errors in Error Estimation . . . . . . . . . . 71

6 Interactive Simulation 72
  6.1 Problems with the Manual Implementation of Heptic Task . . . . . . 72
  6.2 The iAcumen Environment: An Environment for Implementing
      Haptic Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
      6.2.1 Inserting Acumen into a Haptic Virtual Environment . . . . . 75
      6.2.2 A Spring-Mass Example . . . . . . . . . . . . . . . . . . . . . . . 76
  6.3 Implementing Haptic Tasks in Acumen . . . . . . . . . . . . . . . . 79
      6.3.1 A Two-ball Task . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
      6.3.2 An Inverted Pendulum Task . . . . . . . . . . . . . . . . . . . . . 82
  6.4 Conclusions and Future Work . . . . . . . . . . . . . . . . . . . . . . 83

7 Examples of Physical Modeling with Acumen 85
  7.1 Small Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
      7.1.1 A Pendulum . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
      7.1.2 A Pendulum with Interference . . . . . . . . . . . . . . . . . . . . . 89
      7.1.3 A Bouncing Ball . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
      7.1.4 A Bouncing Ball on An Oscillating Surface . . . . . . . . . . . . 93
  7.2 Alfa Romeo Suspension Model . . . . . . . . . . . . . . . . . . . . . . 96
      7.2.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
      7.2.2 One-Wheel Model and Four-Wheel Model . . . . . . . . . . . . . 100
  7.3 Stability of a Pendulum . . . . . . . . . . . . . . . . . . . . . . . . . 104
      7.3.1 Pendulum Systems and Their Stability . . . . . . . . . . . . . . . 105
      7.3.2 Pendulum Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . 106
      7.3.3 Stability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
Illustrations

1.1 The development cycle for physical systems. .......................... 1
1.2 A bus suspension system. .............................................. 3
1.3 Four different bus suspension controllers. ............................ 5
1.4 From equations to code. ................................................. 7
1.5 Acumen = PhyDL + RIDL. ............................................. 10

2.1 Car pulling trailers. ...................................................... 23
2.2 Simulink model for car pulling trailers. ............................... 27
2.3 Initial configuration of the trailers. .................................. 27
2.4 Comparing PhyDL and Simulink results. ............................. 28
2.5 Simulink code for conditionals. ...................................... 29
2.6 Design choices in PhyDL and related languages. ..................... 33

3.1 An RLC circuit. ......................................................... 37
3.2 Voltage responses of an RLC circuit. ................................ 40

4.1 A simple pendulum. .................................................... 52
4.2 Accuracy of Acumen vs. Simulink. ................................... 59

6.1 The iAcumen environment. ............................................. 75
6.2 Example: ball on a spring. ............................................. 77
6.3 The two-ball problem. .................................................. 80
Chapter 1

Introduction

Cyber-physical systems (CPSs) are integrations of digital components and physical processes. With the increase in the applications of embedded computing, scientists and engineers are building more cyber-physical systems, in which software applications operate in a physical context. Embedded computers and networks monitor and control the physical processes, usually with feedback loops in which physical processes affect computations and vice versa. Applications of CPS are ubiquitous, appearing in advanced automotive systems, airplanes, spacecraft, robots, high-confidence medical devices and systems, traffic control, manufacturing equipment, assisted living, and communication systems.

![Development Cycle](image)

Figure 1.1: The development cycle for physical systems.

Developing a new cyber-physical system is an iterative process that starts with design, then progresses to simulation, prototyping, and finally deployment (Figure 1.1). Because simulation can help reveal design flaws at a much earlier stage and with lower failure cost, it is a critical part of the development process. However, existing modeling and simulation approaches generally employ a formalism that is distant from that which engineers naturally use to design and model physical systems. This creates an artificial difference between the core of the engineer’s work and what the engineer needs to do to employ the available computing resources.
My thesis is that we can make simulation much easier by bridging the gap between the user specification and the machine executable. To investigate the viability of this approach, I have designed and built a prototype of a simulation environment called Acumen. The centerpiece of Acumen is a physical description language called PhyDL, which allows the user to directly describe dynamic equations governing the system being modeled. A series of automatic transformations convert the high-level descriptions to a form that is directly machine executable. By raising the level of abstraction for the developer, Acumen can help the engineers to be more productive by making the modeling and simulation process less time-consuming and less effortful.

This chapter presents the need for simulation, how simulation codes are developed, and my solution to making simulation easier, which is the carefully designed simulation environment called Acumen, mentioned above. Evaluation results of Acumen using various case studies are reported, followed by the contribution of this thesis. Chapter 2 discusses existing tools and languages for CPS simulation that are most closely related to our work.

1.1 A Case for Simulation

The importance of simulation is in CPS development. Design is usually represented by a mathematical model, which is a simplified representation of the actual system that is intended to promote understanding. The process of deriving this model is called modeling. Simulation is a particular tool for system analysis, and it generally refers to a computerized version of the model that is run to study the model. We view a simulation as the execution of a model, represented by a computer program that gives information about the system being investigated.

Modeling and simulation are commonly used to develop a level of understanding of the system behavior under design. They reveal the interaction of the parts of a system with one another and the working of the system as a whole. Modeling and simulation are performed at the very beginning of the design flow of a CPS, and they only
involve mathematical and programming processes. Thus, modeling and simulation can help reveal design flaws at a much earlier stage and with lower failure cost. Usually, the designer will iterate through the modeling and simulation steps several times to converge on a mathematical model that satisfies the system specifications before going to system prototyping. After system prototyping, if further changes need to be made to the design, the designer will go back and repeat the modeling and simulation process to revise the design. The same type of iteration happens even after the production of the system.

To illustrate the critical role of simulation in CPS design, we show an example of controller design enabled by simulation. Figure 1.2 shows a bus suspension system, and our aim is to control the suspension system, so that the bus can oscillate as little as possible when running on a rough road. We focus on one of the four bus wheels, and assume that all wheels experience the road in the same way and at the same time. This simplifies the problem to a one-dimensional spring-damper system as shown in the Figure 1.2.

![Figure 1.2: A bus suspension system.](image)

We assume that the initial compressions of the springs compensate the gravity in the model and make the length of both springs zero. The force equations governing
the behavior of two masses would be as follows:

\[ m_1 x_1'' = -b_1(x_1' - x_2') - k_1(x_1 - x_2) + u \]
\[ m_2 x_2'' = b_1(x_1' - x_2') + k_1(x_1 - x_2) + b_2(w' - x_2') + k_2(w - x_2) - u. \]

Here, \( w \) describes the shape of the road surface, which is a function of current bus position. Because the bus position is a function of time, so \( w \) is also a function of time. We assume that the bus is traveling with a constant speed unit.

Now, the control force to be applied to this system is modeled as variable \( u \), which is to be controlled by a digital micro-processor.

We simulate the bus suspension system with four different controllers:

1. An open-loop controller defined by the following equation:

   \[ u = 0. \]

2. A proportional-integral-differential (PID) running on a “high-speed” micro controller (100 Hz):

   \[ u = u_{-PID} = -K_d(x_1' - x_2') - K_p(x_1 - x_2) - K_i \int (x_1 - x_2) dt. \]

   where \( u \) is recomputed every 0.01 seconds, and \( K_d = 10,000; K_p = 1,000; \) and \( K_i = 100. \)

3. A PID control running on a “slow-speed” micro controller (10 Hz):

   \[ u = u_{-PID}. \]

   where \( u \) is recomputed every 0.1 second.

4. A robust controller:

   \[ u = \text{if } E > E_0 \text{ then } 0 \text{ else } u_{-PID}. \]

   The energy \( E \) is computed as the product of the previous value of \( u \) and the change in the spring length \( x_1 - x_2 \). The threshold value \( E_0 = 19. \) The intelligent controller computes the recent energy applied to the system, which is
the product of recent force and the length change of the spring. If the energy applied to the system is greater than the threshold value $E_0$, $u = 0$; otherwise $u = u_{PID}$.

Figure 1.3: Four different bus suspension controllers.

Figure 1.3 shows the results of running the bus on a bumpy road with surface height described as

$$w = 0.1 \sin(t \cdot 2\pi).$$

Road height is shown at the bottom of Figure 1.3. Above it, each column shows the simulation results when using each of the four different controllers. For each controller, we plot the control force $u$, spring length $dx$, and bus height $x_1$. 
For the open-loop controller, the control force $u$ is always zero. Notice that the spring length $\Delta x = x_{1} - x_{2}$ and bus varying height $x_{1}$ both oscillate with increasing amplitude at the beginning, but stabilize after the frequency of the road passes the system's resonance frequency.

For the PID @ 100 Hz controller, the maximum varying height of the bus is much smaller than using the first controller. The bus oscillates significantly less than in the first column, which means that the PID controller described in PhyDL is well behaved at a high frequency of 100 Hz. The sub-plot for the control force shows that we are applying an increasing control force into the system, which is not desired in reality.

For the PID @ 10 Hz controller, the control force is computed using the same equation as for the PID @ 100 Hz controller, but the clock rate is changed from 0.01 to 0.1, which means that the control force is computed every 0.1 second. This models the impact of change of clock rate on the controller, and shows what can happen if 100 Hz is not guaranteed. At the beginning, when the road height frequency is low, the controller is as well behaved as when using the second controller. But when the road height frequency becomes higher and the clock rate is not fast enough to capture the changes in the road and bus, the applied control force makes the bus vibrate much more than with the no-control-force case.

For the robust @ 10 Hz controller, control is active only as long as it is safe. In particular, the controller keeps track of the energy it applies to the system, and if the energy goes beyond a certain threshold, the control force is shut down. From the plot of the varying height of the bus, we can see that the controller has much better performance than the third controller when used on the slow processor.

This example indicates two benefits of using simulation, as follows. (1) It helps us to discover controller flaws early. The second and third controllers have the same design but with different clock speeds. The results from using those two controllers are very different: one is well-behaved and the other creates an artificial singularity. This
tells us that a good design might still fail to produce good performance, if controller
frequency is not taken into consideration. (2) Simulation helps us to develop and
validate solutions. The third and fourth controllers use the same basic equation for
the control force and are applied with the same clock frequency, but the fourth one is
applied conditionally. The simulation results indicate the latter can be a good design
in practice.

Next, we discuss how simulation code is written given the mathematical descrip-
tion of a system.

1.2 Developing Simulation Codes by Hand

Today, a significant portion of the mapping from mathematical equations to coding
is done by hand. Figure 1.4 shows steps that a typical engineer needs to do. Using
existing simulation tools, the engineers need to perform most of these steps by hand.
In what follows, we look closely into each of these steps.

![Diagram](image)

Figure 1.4 : From equations to code.

Directing equations
A mathematical model is a simplified representation of the actual system, which is a series of mathematical equations used to describe the system dynamics and kinematics as well as constant parameters (e.g., mass, length, etc.). Depending upon whether the equations have explicit data flow, a model can be classified as causal or acausal (non-causal). In a causal model, the order of the equations agrees with the order of how to compute the value of each variable. The same order can be used to derive a sequential program to compute the variables, but the burden of keeping the data flow is on the people who builds the model. On the other hand, in an acausal model, only constraints between variables are given. It is easier to derive the mathematical model, but equations must be directed before converting to a sequential program.

**Identifying and classifying ODEs**

System dynamics and kinematics are ordinary differential equations (ODEs). To simulate a system, one needs to solve the ODEs to yield each variable represented independently as a function of time. Techniques for finding solutions to linear and nonlinear ODEs are different. Thus, the system ODEs are usually classified as linear or nonlinear ODEs and treated differently.

**Solving linear and non-linear ODEs**

There are analytical methods for solving linear ODEs. However, there are no general methods for solving ODEs of nonlinear systems. For nonlinear ODEs, numerical methods are usually used to find an approximate solution. The key idea of numerical methods is to discretize the continuous time and to approximate a continuous behavior as a finite set of pairs of time instance and behavior value at the time instance. Depending upon the highest-order derivative appearing in the ODEs, a system can be classified as first-order or higher-order. Numerical methods can only be directly applied to first-order ODEs. Thus, a step to reduce the order of the ODEs needs to be performed before solving them numerically.

It is time consuming to perform most of those steps by hand. I propose Acumen to automate those steps for the user so that it will be much easier for the engineers to
write simulation code. Before the design of Acumen is introduced, the next section presents some related work and existing tools. I examine how simulation is done with these tools and examine the strengths and limits of each tool.

1.3 Acumen: A Tool for Making Simulation Easier

Existing modeling and simulation techniques can only automate a small portion of work during those steps. Thus, it becomes the engineer’s work to employ the available computing resources and to convert the mathematical model to a specific form that can be executed on a machine. In other words, there is a gap between what engineers naturally use to model a system and what a machine can execute. Artificial work is created on top of the core of the engineer’s work, which makes it time consuming to write simulation code for a cyber-physical system.

To make simulation easier, I have surveyed existing tools for simulation, and studied what can be automated to bridge the gap between a mathematical model and a simulation program. I have designed a language that allows engineers to write their mathematical model in mathematical form, and I have developed a simulation environment that translates a program in this language to machine executable and runs the simulation. In the next two subsections, I will introduce the simulation environment Acumen and the physical description language PhyDL.

1.3.1 PhyDL and RIDL

Figure 1.5 illustrates the two key components of Acumen: PhyDL and RIDL. The physical description language, PhyDL, can describe dynamic equations governing the system being modeled, and the reactive/interrupt description language, RIDL describes model discrete events and control algorithms in an event-driven programming paradigm. RIDL was previously designed by Kaiabachev [31]. The focus of my work is the design of PhyDL.

The two language components in Acumen corresponds to the physical elements
and computational elements in a cyber-physical system. Modeling of the physical elements takes place in PhyDL, and modeling of the computational elements takes place in RIDL. For example, for the bus suspension system, the ODEs that govern the dynamics and kinematics are easily modeled in PhyDL. Different controllers are modeled in RIDL. The clock speed of the controllers and the communication between PhyDL and RIDL are also described in PhyDL. Therefore, a RIDL program describes the control information of a system, and is treated as an external section of PhyDL. However, there are also other external sections that can interact with PhyDL, such as an external section for interactive GUI or MATLAB.

After modeling a system in Acumen, a series of automatic transformations will convert the high-level PhyDL description into a form that is directly machine executable. Thus, the user is relieved from the burden of taking all the steps needed to convert a mathematical model to a simulation program and thus can focus on the real design work. By raising the level of abstraction for the developer, Acumen can accelerate the modeling and simulation process of cyber-physical systems.

The Acumen environment has a convenient user interface for the execution of the PhyDL program and all of its external sections. The interface consists of two parts: a
control panel for starting, pausing, and terminating a simulation, and a display panel for showing the final and intermediate results. Detailed usage of the user interface is introduced in Section A.6.

The design details of the language PhyDL is presented in Chapter 3. Numerical methods are used in PhyDL to solve ODEs. Chapter 5 introduces error models of ODE solvers. The purpose is to understand the source of errors in numerical approximation and how error estimation methods can be used in Acumen.

1.3.2 Evaluation of Acumen

The usability and accuracy of Acumen is examined using some case studies. In this section, I will present two important applications of Acumen. One is using Acumen for interactive simulation, and the other is using Acumen for industrial applications.

To run interactive simulation, an interface for interactive input/output is needed. Also, the simulated system needs to be synchronized with the user clock, which means that the simulation must run in real time. Acumen is extended to iAcumen to support interactive simulation (Chapter 6).

In the industrial application, a complicated tractor system has been modeled in both Acumen and MATLAB/Simulink. We compared the code quality of the Acumen and Simulink models, and found that the Acumen code is more concise and easier to understand. We also compared the simulation results of the two models, and the simulation results were very close to each other. MATLAB/Simulink is known to have acceptable accuracy, and the result indicates that the accuracy of Acumen is at least as good as MATLAB/Simulink.

1.4 Summary of Contributions

The contribution of this thesis may be summarized as follows:

- The integration of the reactive/interrupt description language RIDL in the Acu-
men environment. Discrete events and controllers can be described in an event-driven paradigm in Acumen in a natural way. (Section 1.3.1)

• The design of PhyDL, a language that can directly describe the dynamic equations governing the system being modeled. Systems can be described acausally, and both nonlinear and higher-order ODEs can be expressed. Notation is carefully designed to be close to what engineers already use, and it contains a module system to avoid code duplication. (Chapter 3).

• A type system for PhyDL (Section 4.2).

• Semantics for PhyDL, including numerical solvers for ODEs that are used for program execution (Section 4.3).

• Evaluation of Accuracy by comparing Acumen to MATLAB/Simulink. (Section 4.4).

• A survey of error models of these ODE solvers is presented to explain how to bound the simulation errors in PhyDL (Chapter 5).

• An interface for coupling with external devices. The extension of Acumen with external interface is called interactive Acumen, or iAcumen for short (Chapter 6). The interface allows conceptually clear access to the low-level codes needed to inject these external devices into the real virtual world.

• Several examples and case studies developed by various people in the Acumen group have been presented in Chapter 7.
Chapter 2

Related Work

This chapter reviews software packages and languages with features similar to those of Acumen. Because mechanical simulation is highly important in many industrial fields, it is not surprising that numerous computer tools have been developed to cater to this need. Successful programs tend to find niche markets where they are far superior to the competition rather than attempting to function as an end-all solution to any conceivable problem. Therefore, it is helpful to survey a wide field of software on the market and languages widely used in order to see what is already available and what Acumen can do better for physical system simulation.

2.1 Relation to Other Systems

In this section we briefly discuss some related software and languages that serve a similar purpose as Acumen but are based on different technologies.

2.1.1 Tools Based on Block Diagrams

In this section, we introduce several tools with graphical syntax. Graphical syntax is very intuitive, and all of these tools come with a rich library of components that can be used to compose a system. Each library component is represented as a block diagram.

2.1.1.1 Imagine.Lab AMESim

Imagine.Lab AMESim [11] is a professional one-dimensional (1-D), multi-domain system-simulation platform produced by LMS International and available for Win-
dows and UNIX/Linux. The components of the model are described by analytical models representing the hydraulic, pneumatic, electric, or mechanical behavior of the system via an interactive graphical interface. The program eliminates the need to develop numerical models or custom code by focusing on physics-based building blocks.

Model development in AMESim is greatly facilitated by the availability of a large library of pre-defined components from diverse physical domains and engineering applications, including fluid, thermal, mechanical, electromechanical, power train, and control. For example, vehicle dynamics can be easily modeled through the Vehicle System Dynamics Solutions, which includes vehicle dynamics, vehicle dynamics controls, braking systems, power steering, and suspension and anti-roll.

AMESim features extensive interfaces with third-party software tools for control, real-time simulation, multi-body simulation, process integration, and design optimization. For example, for control, AMESim can interface with Mathworks Simulink and NI System Build.

A strength of AMESim is the ability to simulate systems before the exact physical geometry is known. If the geometry is known, AMESim can be easily integrated with LMS Virtual.Lab, a unified, 3-D simulation environment. This allows the simulation of the performance of mechanical systems in terms of structural integrity, noise and vibration, durability, system dynamics, ride and handling, etc.

2.1.1.2 VisSim

VisSim [27] is a visual block-diagram language for modeling, simulating, and analyzing dynamic systems produced by Visual Solutions available for Windows. The graphical interface features drag-and-drop block diagram construction, with a library of over 110 linear and nonlinear blocks, eliminating traditional programming. The simulation engine provides fast and accurate solutions for linear, nonlinear, continuous-time, discrete-time, time-varying, SISO, MIMO, and hybrid system designs.
Toolbox functions are available for control, electromechanical design, hydraulics, signal processing, process, chemical, thermal, and turbines; the algorithms available include Euler, trapezoidal, Runge Kutta 2nd and 4th orders, adaptive Bulirsh-Stoer and Runge Kutta 5th order, stiff backward Euler, and Lawrence Livermore integration algorithms. Simulation outputs may be graphed within the user interface with interactive XY, time domain, FFT, and discrete plots and strip charts. The VisSim/Motion software allows accurate simulation of motion and motor control systems, and it includes a comprehensive motion-control block library comprising over 40 motion and motor-control blocks.

Custom blocks can be constructed in C, C++, Fortran, and Pascal, and integration is possible with MATLAB, Mathcad, and Maple. For example, A MATLAB Expression Block enables MATLAB scripts to be edited and executed directly from the VisSim environment. In addition, VisSim features efficient C code generation.

2.1.1.3 Working Model 2-D

Working Model 2-D [32] is a graphical motion simulator and mechanical system analysis software produced by Design Simulation Technologies that is available for Windows and Mac OS. The program allows the user to construct two-dimensional mechanical systems comprising stiff or flexible solid objects, springs, dampers, pin joints, slots, motors, etc. Once a system is constructed, the user can run the simulation and watch the results in real time, and the resulting animation can be exported in AVI format. Environmental parameters, such as gravity and air resistance, can also be modeled.

All system variables, such as spring heights, force vector components, displacements, torques, etc., can be recorded and easily graphed in the simulation window. The software can also produce shear and bending moment diagrams. Every system variable can be expressed as a mathematical formula using the built-in formula language. For example, an active suspension system can be modeled by representing the damping constant as a function of the suspension spring length or the road height,
or by adding an input force, the magnitude of which is a function of other system variables.

For expanded functionality, Working Model 2-D can be integrated with Excel or MATLAB. For modeling more complicated controllers, the controller algorithms can be programmed into MATLAB, which communicates in real time with Working Model 2-D. Controller equations can also be written in Excel, which links dynamically to Working Model 2-D.

2.1.2 Tools Based on Symbolic Computation

Some tools for solving equations have been used for modeling and simulation, such as Mathematica [21], Maple [26], and SciLab [10]. These tools are capable of equation solving. In general, using such solutions becomes very clumsy if the virtual environment to be modeled involves both continuous behaviors and discrete events. Additionally, these tools do not provide an interface for interactive user input and output.

2.1.2.1 Mathematica

Mathematica is a computational software program used in scientific, engineering, and mathematical fields and other areas of technical computing. Mathematica is split into two parts: the “kernel” and the “front end.” The kernel interprets expressions (Mathematica code) and returns result expressions. The Mathematica Front End provides a GUI that allows the creation and editing of Notebook documents that can contain program code with pretty printing and formatted text, together with results including typeset mathematics, graphics, GUI components, tables, and sounds.

The features related to physical system simulation in Mathematica include: libraries of mathematical functions; solvers for systems of equations, ordinary differential equations (ODEs), partial differential equations (PDEs), and differential algebraic equations (DAEs); numeric and symbolic tools for discrete and continuous calculus;
and a programming language supporting procedural, functional, and object-oriented constructs.

Communication with other applications occurs through a protocol called MathLink, which allows communication between the Mathematica kernel and front-end and also provides a general interface between the kernel and other applications. Wolfram Research freely distributes a developer kit for linking applications written in the C programming language to the Mathematica kernel through MathLink. Two other components of Mathematica, whose underlying protocol is MathLink, allow developers to establish communication between the kernel and a .NET or Java program: .NET/Link and J/Link.

### 2.1.2.2 Maple

Maple is a general-purpose commercial computer algebra system. Users can enter mathematics in traditional mathematical notation. Custom user interfaces can also be created. There is extensive support for numeric computations, to arbitrary precision, as well as symbolic computation and visualization.

Maple is based around a small kernel, written in C, that provides the Maple language. Most functionality is provided by libraries, which come from a variety of sources, and many numerical computations are performed by the NAG numerical libraries, ATLAS libraries, or GMP libraries. Most of the libraries are written in the Maple language; those have viewable source code. Features of Maple include: integration, root finding, solving equations and inequalities, plotting of functions, and solving of ODEs and PDEs.

### 2.1.2.3 Scilab

Scilab is a numerical computational package developed and maintained by the INRIA. Scilab is a high-level, numerically oriented programming language that provides an interpreted programming environment, with matrices as the main data type. By
utilizing matrix-based computation, dynamic typing, and automatic memory management, many numerical problems may be expressed in a reduced number of code lines, as compared with similar solutions using traditional languages, such as Fortran, C, or C++. This allows users to rapidly construct models for a range of mathematical problems. While the language provides simple matrix operations such as multiplication, the Scilab package also provides a library of high-level operations such as correlation and complex multidimensional arithmetic. The software can be used for signal processing, statistical analysis, image enhancement, fluid dynamics simulations, and numerical optimization. Scilab also includes a package called Scicos for modeling and simulation of explicit and implicit dynamical systems, including both continuous and discrete sub-systems.

2.1.3 Solidworks and ProEngineer

Solidworks [18, 16] and ProEngineer [5] are 3-D mechanical CAD programs aimed at design automation and optimization. These tools are widely used by product designers and mechanical engineers for drafting their products. They provide a systematic way for constructing solid objects and assemblies. However, Solidworks and ProEngineer only provide limited support for modeling system dynamics and object interactions. For example, system dynamic equations cannot be easily modeled using these tools.

2.1.4 Tools Based on Functional Programming

Functional Reactive Programming (FRP) [30] is a functional reactive programming framework that is highly suited for causal hybrid modeling. The framework is embodied in a language called Yampa as an extension of Haskell. FRP permits dynamic hybrid systems to be described in a high-level, declarative manner. The key ideas in FRP are its notions of behaviors and events. Behaviors are time-varying, reactive values, while events are time-ordered sequences of discrete-time event occurrences.
Every dynamic variable in FRP is implicitly a function of time. FRP provides a family of functions called “lift,” which can lift an ordinary value or function to an analogous behavior. The same idea is used in PhyDL.

While acausal modeling is essential to modeling dynamic physical systems, there was an attempt to combine the strengths of functional programming in FRP and acausal modeling. How functional programming techniques would play a pivotal role in improving dynamic system modeling was articulated in [17], but no further results have been reported.

### 2.2 Detailed Comparison with MATLAB/Simulink

MATLAB/Simulink[23] is an environment for multidomain simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let the user design, simulate, implement, and test a variety of time-varying systems.

#### 2.2.1 Relating MATLAB/Simulink to Acumen

I have studied MATLAB/Simulink in detail in my thesis for the following reasons:

1. MATLAB/Simulink has great power and is widely used for modeling and simulation. However, to convert a mathematical model to MATLAB/Simulink code can be time consuming. Acumen is designed as a tool that bridges the gap.

2. MATLAB/Simulink lacks a clear and formal semantics. The user needs to have a deep knowledge of Simulink to correctly interpret the results of its simulations. Fully understanding what takes place inside the tools would be important for preventing unpleasant surprises. The lack of formal semantics of the models used inside Simulink has been considered a serious drawback, especially in academic circles. In contrast, Acumen is designed with a formal semantics.
3. MATLAB/Simulink is an integrated environment in which some parts can be used as external sections or references of Acumen. For example, currently Acumen uses MATLAB display functionalities for displaying simulation results. In addition, MATLAB/Simulink is known to have good precision for numerical computation. Therefore, we are using Simulink as a reference to evaluate the accuracy of Acumen simulation results. Finally, Simulink has a powerful computation/analysis back-end. Thus, one very practical direction for our future work is to generate Simulink code to do the back-end computation of the Acumen program.

2.2.2 Comparing MATLAB/Simulink to Acumen with Small Examples

In the rest of this section, we will show some small examples that compare the code quality of Acumen and MATLAB/Simulink.

2.2.2.1 Example 1: Bus Suspension System

The first example uses the same problem description as in Section 1.1.

MATLAB/Simulink and PhyDL Code

Table 2.1 shows the MATLAB/Simulink code and the Acumen code for the bus suspension system.

- For MATLAB/Simulink Frequency Domain

\[
\Delta = \det \begin{bmatrix}
    m_1 s^2 + b_1 s + k_1 & -(b_1 s + k_1) \\
    -(b_1 s + k_1) & (m_2 s^2 + (b_1 + b_2) s + (k_1 + K_2))
\end{bmatrix}
\]

\[
G_2(s) = \frac{x_1(s) - x_2(s)}{W(s)} = \frac{-m_1 b_2 s^3 - m_1 k_2 s^2}{\Delta}.
\]
Table 2.1: MATLAB/Simulink and PhyDL Code for the Bus Suspension System.

| MATLAB /Simulink | m₁=2500; m₂=320; k₁=80000; k₂=500000; b₁=350; b₂=15020; |
| Frequency Domain | num=[-(m₁*b₂) -(m₁*k₂) 0 0]; |
| Frequency Domain | den=[m₁*m₂ (m₁*(b₁+b₂))+(m₂*b₁) |
| Frequency Domain | (m₁*(k₁+k₂))+(m₂*k₁)+(b₁*b₂) (b₁*k₂)+(b₂*k₁) k₁*k₂]; |
| Frequency Domain | 'G(s)2' |
| Frequency Domain | printsys(0.1*num, den) |
| Frequency Domain | %step(0.1*num, den). |

| MATLAB /Simulink | m₁=2500; m₂=320; k₁=80000; k₂=500000; b₁=350; b₂=15020; |
| Time Domain | A=[0 1 0 0; |
| Time Domain | -(b₁*b₂)/(m₁*m₂) 0 |
| Time Domain | ((b₁/m₁)*((b₁/m₁)+(b₁/m₂)+(b₂/m₂)))-(k₁/m₁) -(b₁/m₁); |
| Time Domain | b₂/m₂ 0 0 1; k₂/m₂ 0 0 0]; |
| Time Domain | B=[0 0; 1/m₁ (b₁*b₂)/(m₁*m₂); 0 -(b₂/m₂); |
| Time Domain | (1/m₁)+(1/m₂) -(k₂/m₂)]; |
| Time Domain | C=[0 0 1 0]; |
| Time Domain | D=[0 0]; |
| Time Domain | %step(0.1*num, den). |

| PhyDL Time Domain | (* some set up for simulation *) |
| PhyDL Time Domain | boundary |
| PhyDL Time Domain | x₁ with x₁(0)=0, x₁'(0)= 0; x₂ with x₂(0)=0, x₂'(0)= 0; |
| PhyDL Time Domain | system |
| PhyDL Time Domain | m₁=2500; m₂=320; k₁=80000; k₂=500000; b₁=350; b₂=15020; |
| PhyDL Time Domain | x₁'' = 1/m₁ * (-b₁*(x₁'-x₂')-k₁*(x₁-x₂)+u); |
| PhyDL Time Domain | x₂'' = 1/m₂ * (b₁*(x₁'-x₂')+k₁*(x₁-x₂)+ |
| PhyDL Time Domain | b₂*(w'-x₂')+k₂*(w-x₂)-u);. |
• For MATLAB/Simulink State Space Equation

\[
Y_1 = -\left(\frac{b_1}{m_1} + \frac{b_1}{m_2}\right)Y_1 - \frac{b_2}{m_2}(w - x_2) + \int Y_2 dt
\]

\[
Y_1 = -\left(\frac{k_1}{m_1} + \frac{k_1}{m_2}\right)Y_1 - \frac{k_2}{m_2}(w - x_2) + \left(\frac{1}{m_1} + \frac{1}{m_2}\right)u
\]

\[
x' = Ax + Bw
\]

\[
y' = Cx + Dw
\]

\[
Y = [0 0 1 0][x_1 \ x_2 \ Y_1 \ Y_2]^T + [0 0][u \ w]^T = x_1 - x_2.
\]

Remarks

• Both the state space method and the transform function method in MATLAB/Simulink have equations very different from the original model.

• At the beginning, I can only run MATLAB very slowly because of the remote access license is not adequate. But PhyDL was even slower. Then we figured out the problem was that we interpreted the PhyDL program in a non-efficient, recursive way. After changing the implementation of PhyDL semantics to an imperative way, PhyDL became much faster.

• PhyDL is implemented using OCaml; some issues about OCaml include:

  – Converting from the time-equations to OCaml code can be quite mechanical. It is just a Laplace transform.

  – In converting to OCaml, we have to be careful about numerical accuracy issues (continuous model and discrete model).

  – The OCaml implementation will require some tweaking for larger problem sizes. (Now a simulation run of 30*1000 steps is fast for the bus suspension system model, but for a more complex system a simulation run of 30*1000 steps would be very slow.)

  – PhyDL can also integrate PID a controller description with the physical system. All that is needed is to add some code in RIDL such as the
following: $u_{\text{PID}}$ is a PID controller which can be described in PhyDL as:
\[
\text{control} = \text{init} 0 \text{ in } \{ \text{clock } \Rightarrow u_{\text{PID}} \}.
\]

- Simulation preciseness of PhyDL compared with MATLAB/Simulink (for this example only). Using the following settings, we compared the results from PhyDL and MATLAB/Simulink and found that the resulting data are very close to each other.
  - Both approximate the implicit integration using a time step of 0.001 s.
  - Simulation time is 50 s.

2.2.2.2 Example 2: Car Pulling Trailers

Problem Description

Figure 2.1 shows a car pulling several trailers, with each trailer hitched between the rear wheels of the body in front. The first car is used to steer and provide speed. Let $F_i$ be the vector of the front of $i$-th car/trailer, with $B_i$ being the body of $i$th. Input is $U$.

![Figure 2.1: Car pulling trailers.](image)

From the rigid body constraints, we can obtain the dynamic equations as the
following:

\[
F_{n+1} = R_n \\
F_n = R_n + B_n \\
F'_0 = U \\
F'_{n+1} = \text{proj}(F'_n, B_n) \\
\text{abs}(B'_n) = 0 \\
\text{abs}(B_n(0)) = l_n.
\]

**MATLAB/Simulink and PhyDL Code**

- MATLAB/Simulink code
  
  ```matlab
  function [sys,x0,str,ts] = trailer(t,x,u,flag,len)
  switch flag,
  case 0, [sys,x0,str,ts] = mdlInitializeSizes(len);
  case 1, sys = mdlDerivatives(t,x,u,len);
  case 3, sys = mdlOutputs(t,x,u,len);
  case 9 sys = [] ; % do nothing
  end
  function [sys, x0,str,ts] = mdlInitializeSizes(len)
  sizes = simsizes;
  sizes.NumContStates = 6;
  sizes.NumDiscStates = 0;
  sizes.NumOutputs = 8;
  sizes.NumInputs = 1;
  sizes.DirFeedthrough = 1;
  sizes.NumSampleTimes = 1;
  sys = simsizes(sizes);
  x0 = [0 0 (pi/6) pi (-pi/2) (pi/2)];
  str = [];
  ts = [0 0]; % continuous sample time: [period, offset]
  function sys = mdlDerivatives(t,x,u, len)
  sys(1) = u*cos(x(3)+x(4));
  sys(2) = u*sin(x(3)+x(4));
  ```
sys(3) = 0;
sys(4) = u*tan(x(3))/len;
sys(5) = u*(tan(x(3))-cos(x(3))*tan(x(5)))/len;
sys(6) = u*(cos(x(3))*tan(x(5)) -
    cos(x(3))*cos(x(5))*tan(x(6)))/len;

function sys = mdlOutputs(t,x,u,len)
sys(1) = x(1);
sys(2) = x(2);
sys(3) = x(1)-len*cos(x(4));
sys(4) = x(2)-len*sin(x(4));
sys(5) = x(1)-len*cos(x(4))-len*cos(x(4-x(5)));
sys(6) = x(2)-len*sin(x(4))-len*sin(x(4-x(5)));
sys(7) = x(1)-len*cos(x(4))-len*cos(x(4-x(5)) -
    len*cos(x(4)-x(5)-x(6));
sys(8) = x(2)-len*sin(x(4))-len*sin(x(4-x(5)) -
    len*sin(x(4)-x(5)-x(6));
\vspace{-1cm}

• PhyDL code

(* some set up for simulation *)

boundary
    f0 with f0(0)=[0, 0];
f1 with f1(0)=[0.1, 0];
f2 with f2(0)=[0.1, 0.1];
f3 with f3(0)=[0.2, 0.1];
equation
    len = 0.1;
u = v*unit(rotate(b0, dir*pi/6));
f0' = u;
f1' = proj(f0', b0);
f2' = proj(f1', b1);
f3' = proj(f2', b2);
b0 = f0 - f1;
b1 = f1 - f2;
b2 = f2 - f3;.
Remarks

- The MATLAB/Simulink code here is not using vector because it is not easy to do so. In S-function, if we want to declare input/output signals to be vectors, we must use ssSetVectorMode. However, this ssSetVectorMode is not supported universally, and it can only be available if the S-function is defined in C. We can see from the MATLAB/Simulink code that using scalar representation in this example is very annoying and error prone.

- The MATLAB/Simulink code above is just an m-file. To make the MATLAB/Simulink code work, the following extra things need to be done:
  
  - Write some auxiliary m-file to support vector operations like \texttt{proj}.
  
  - Instantiating a Level-2 m-file S-function. First create an instance of the Level-2 m-file S-function block in the model. Then open the block’s parameter dialog box and enter the name of the m-file that implements your S-function in the dialog box’s m-file name field. The final model is shown in Figure 2.2.

- Comparing the simulation results (Figure 2.4).
  
  - Simulation time 10 s, step size 0.001 s.
  
  - Each sub-plot is x-y trace of $F_0$, $F_1$, $F_2$, $F_3$. It started at position $[0 \ 0]$ $[0.1 \ 0]$ $[0.1 \ 0.1]$ $[0.2 \ 0.1]$, with the front car always steering $\pi/6$ to its left (Figure 2.3).
  
  - There is still some difference shown in the two implementations.
  
  - Trace from two different implementations:
2.2.3 The REPTILE Project

The REPTILE project is a collaboration between Schlumberger and Rice on applying “Resource-aware Programming for Oil Exploration.” The goal of the project is to explore how advanced programming language techniques can be used to accelerate the development of the first successful horizontal drilling tractor. The focus of the project will be to develop appropriate simulation tools. This is expected to help Schlumberger engineers in both the design and the implementation of intelligent tractor controllers.

The design of Acumen was partially driven by the need for modeling the tractor,
which is a real-world robot. Tests and examples for Acumen were guided by pristine specifications of the physical model and the controllers of the tractor. I have also participated in mathematical modeling of the tractor system. After that, the tractor model was simulated in Acumen as well as in MATLAB/Simulink.

MATLAB/Simulink code for the tractor model and its simulation results are used as reference points in the Acumen evaluation, and the evaluation is two-fold. First, we want to examine the quality of the Acumen code, and the efforts needed to put the mathematical model into Acumen compared with Simulink. Second, we want to look closely into the accuracy of Acumen, and thus we compared the simulation results from both tools.

Simulation in Acumen turns out to be much easier than in Simulink. We compared the code in Acumen and Simulink and found that the Acumen code is more concise and much closer to the mathematical model of the system.
2.2.3.1 Control Flow is Easier to Express in Acumen

Simulink programs are graphical, and a Simulink model has blocks connected by lines. Each line represents a signal or a vector of signals. Some very simple mathematical models may require a complicated Simulink diagram code. For example, to represent a conditional statement in Simulink, we will have a graphical program as shown in Figure 2.5.

![Figure 2.5: Simulink code for conditionals.](image)

On the other hand, PhyDL has a very natural way to describe mathematical and logical equations. Thus, in PhyDL, the model in Figure 2.5 can be modeled by just two lines of an equation:

\[
T = \text{if } ((T_e >= 0 \land T_l >= 0 \land T_e < T_l) \lor (T_e <= 0 \land T_l <= 0 \land T_e > T_l)) \land (\text{speed} == 0) \text{ then } 0 \text{ else } T_e - T_l.;
\]
2.2.3.2 Acumen is More Concise

It is very important to exploit reusability in physical system modeling, especially when the system is complicated and has components with the same pattern. The module system of the PhyDL language in Acumen allows the user to define a component once and use the definition multiple times by referring to the name of the component definition. This also enhances the modularity of the system and makes the program more readable.

Reusability in Simulink is achieved by block hierarchy. A Simulink model is built up from basic library blocks, and Simulink lets you organize your model into levels of hierarchy by using subsystems. Subsystems encapsulate a group of blocks and signals as a higher-level single block. Every subsystem has a custom user interface called a mask, which hides the subsystem’s contents, making it appear to the user as an atomic block with its own icon and parameter dialog box. A component can be defined as a subsystem in Simulink. When it is used for multiple times, the user can copy this higher-level block for the subsystem and “paste” it wherever needed. Thus, the code for components with the same definition is still duplicated, although the code is “hidden” under the mask. Moreover, when the user needs to make a change to the component definition, the change must be made several times in each instance of the component. In PhyDL, changes to the component definition only need to be made once.

2.2.4 Simulink/SimMechanics

SimMechanics [22] is an extension of Simulink environment that provides extensive physical modeling capability, especially for 3-D mechanical systems. Instead of deriving and programming equations, SimMechanics provides the user with a block/module library that allows the user to build a model from bodies, joints, constraints, and force elements that reflects the structure of the system. Alternatively, it is possible to import physical data from a CAD model, with only a need to define additional inter-
actions as required by the software. Once the model is defined, all of the governing
equations can be handled by MATLAB/Simulink, enabling the user to instantly see
how any given model will behave.

In practice, SimMechanics is not always convenient to use. The block library
of SimMechanics is highly customized, and objects that are not in the library are
usually hard to define. The modeling procedure is also very time consuming and
requires great expertise. Finally, only limited constraints can be represented easily
in the model. It is not clear how to represent useful information such as collision
detection and energy-conservation in SimMechanics.

The CAD-to-SimMechanics translators for converting CAD models to SimMechanics
models are not working as well as expected. Experiences show that many
constraints in CAD models can be lost during the translation, and it is very hard to
identify the loss of constraints and make corrections in the translated SimMechanics
model.

2.2.5 Summary

In this section, we have examined a simulation environment called MATLAB/Simulink that is widely used for modeling and simulation. The tool is very
powerful but has many drawbacks. The most significant ones are that it is not very
easy to use and that it lacks a clear semantics. Acumen is designed to make sim-
ulation easier by allowing the user to describe a physical system using high-level
equations. The compilation process generates executable from the equations, and a
clear semantics is given based on the compilation.

Three examples that compare Acumen code and MATLAB/Simulink code for the
same physical systems are given side by side. It is very clear from the comparison
that Acumen code is much easier to derive and that its simulation results are easier
to understand and interpret.
2.3 Detailed Comparison with Modelica

Modelica [6, 8] is an equation-based, object-oriented language for modeling physical systems. Mathematical equations can also be easily transferred into a Modelica program. Modelica is a specification language only and thus requires an external computational environment to actually perform simulation. A small number of computational environments support Modelica simulation, such as OpenModelica [7], Dymola [8], and MathModelica [9]. All of these environments have complex compilation processes and demand that the user have a profound understanding of Modelica semantics. A more detailed discussion of these environments and their drawbacks can be found in [29].

To our knowledge, Modelica is a language that is closest to PhyDL, except that Modelica does not allow higher-order derivatives to be written directly. However, this can be addressed by introducing new variables to represent derivatives of variables, as we will explain in Section 4.3.4.

At the same time, Modelica is quite different from PhyDL from a higher-level point of view. PhyDL is designed as equation-based. A physical system is modeled as variables and relations between variables. Variables are functions of time that exposes the state changes of a physical system, and relations in the form of equations are constraints on the values those variables can take at any time. Thus, there is a direct mapping between the system’s mathematical equations and the PhyDL program.

Modelica uses object-oriented (OO) mathematical modeling. In Modelica, classes are blueprints for creating objects, and both variables and equations can be inherited between classes. Function definitions can also be inherited. A Modelica program is structured by hierarchies, component-connections, and inheritance. In contrast to ordinary OO languages, specifying behavior in Modelica is primarily done through equations (as in PhyDL) instead of via methods or dynamic message passing.

Although both Modelica and PhyDL have acausal module systems, the view of
a component is very different in these two languages. In Modelica, components are instances of Modelica classes, which have interfaces. Interfaces in PhyDL are called ports, and they are shared variables between the component and its environment. Interfaces in Modelica are called connectors, and they are instances of connector classes.

The object-oriented mathematical modeling gives the Modelica language the power of OO design, such as inheritances, but it also requires the user to have some expertise in the OO design principle to use the language properly. The design principle of PhyDL is to be exoteric, so we choose to make the language as equation-based only.

### 2.4 Summary

Table 2.4 summarizes the distinctive features of the various environments and programming languages for physical system modeling and simulation that we have discussed in the previous subsections. Because tools and languages in the same section have similar features, we only choose one from each section as a representative of others.

<table>
<thead>
<tr>
<th></th>
<th>Derivatives</th>
<th>Direct HO Integration</th>
<th>Point-free Notation</th>
<th>Acausal Modeling</th>
<th>Module System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solidworks</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Simulink</td>
<td>■</td>
<td>☐</td>
<td>■</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Mathematica</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>☐</td>
<td>■</td>
</tr>
<tr>
<td>Modelica</td>
<td>■</td>
<td>☐</td>
<td>■</td>
<td>■</td>
<td>■</td>
</tr>
<tr>
<td>FRP</td>
<td>☐</td>
<td>☐</td>
<td>■</td>
<td>■</td>
<td>☐</td>
</tr>
<tr>
<td>PhyDL</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
</tr>
</tbody>
</table>

Figure 2.6 : Design choices in PhyDL and related languages.
Chapter 3

The Design of PhyDL

PhyDL is designed to make simulation easier for the engineers. Many different choices need to be made when designing a language, including syntactic choices, semantic choices, and implementation choices. With the goal of making PhyDL as close as possible to the notations used in modeling dynamic systems, we made the following design choices in creating PhyDL.

1. **Textual syntax.** A language may have textual syntax or graphical syntax. PhyDL has textual syntax, which has the advantage of being flexible and close to mathematical equations. Language with graphical syntax usually has a graphical library, and the programming has a drag-and-draw paradigm. Graphical language requires the user be familiar with the library blocks. When the model has a part that is not in the library, the programming becomes very clumsy.

2. **Stand-alone language.** A language for a specific domain can be designed as a stand-alone language, or as one that is embedded in and extended from an existing general-purpose language. An embedded language has the advantage that all primitive functions in the base language can be used directly, while every function in a stand-alone language needs to implement. However, stand-alone languages have clearer semantics, and they have more flexibility in the design decisions.

3. **Both causal and acausal modeling.** PhyDL allows both causal and acausal modeling. To support acausal modeling, the language needs to sort the dependencies between variables and equations for the user, so that the user needs
only to write all constraints as equations without worrying about the causality between variables. For example, equation \( a'' + g \times l \times \sin(a) = 0 \) is allowed in PhyDL language.

4. **Point-free notation.** In a physical system, every variable is a function of time, so PhyDL makes time \( t \) as an implicit parameter of each variable.

5. **Integration and higher-order derivatives.** Both derivatives and integration are supported, and higher-order derivatives can be written directly.

6. **Module system.** PhyDL has a module system. A component in a physical system can be defined independently as a module and be used in the system definition by referring to the name of the module definition.

In the rest of this chapter, we will explain some of these design choices in more detail.

### 3.1 Causal and Acausal Modeling

There are two ways to model a physical system. One is called *acausal modeling* and the other is *causal modeling*.

In acausal modeling, a model is composed of variables and relations between variables. The variables in a model are functions of time to observable quantities, and they implicitly expose changes inside models. The relations in a model act as constraints between the values that variables take at each instant. Models interact through constraints between some of their variables.

In causal modeling, a model is composed of

- Inputs
- Outputs
- Variables
• State variables (variables whose derivatives appear in the system)
• Relations between inputs and (state) variables constraining the value of the outputs, variables, and derivatives of state variables.

The inputs of causal models handle data coming from the environment, and outputs of causal models handle data to be exported to the environment. (State) variables of causal models are used to compute observable quantities. The key point is, data flow in a causal model is explicit. It is possible to simulate a causal model using value propagation first and then to do the integration.

Acausal modeling is a convenient way to express specifications, and thus acausal models are easy to build and modify. But highly elaborated tools are necessary to handle them efficiently. Equation-solving techniques are examples of such tools. On the other hand, causal modeling expresses only explicit computations, and thus causal models are more difficult to build and extremely hard to modify. But the computations for causal models generally are much easier.

3.1.1 The RLC Electrical Circuit Example

This problem description was used in [28].

Physical modeling involves specifying relationships between various quantities such as voltage, pressure, mass, etc. In this example, we will build a model of an RLC electrical circuit [28]. An RLC circuit contains a resistor, a capacitor, and inductor and exhibits oscillatory behavior in response to voltage disturbances.

Figure 3.1 shows the schematic of an RLC circuit.

First, let us assume that the voltage source, \( V \), jumps from 0 to 1 after one second of simulation. We can then write an explicit expression for the voltage at node \( a \) as follows:

\[
V_a = \begin{cases} 
0 & 0 \leq t < 1 \\
1 & t \geq 1.
\end{cases}
\]
Next, we consider the inductor model. The equation for the current through the inductor (from node $a$ to node $b$) is:

$$L \cdot i_L' = (V_a - V_b).$$

Likewise, using Ohm’s law, the current through the resistor (from node $b$ to node $c$) can be expressed as:

$$R \cdot i_R = V_b - V_c.$$

Finally, the current through the capacitor leaving node $c$ and going to ground can be expressed as

$$i_c = C \cdot V_c'.$$

But using Kirchhoff’s current law, we know that the sum of the currents going into each node must be zero. This gives us:
\[ i_V - i_L = 0 \]
\[ i_L - i_R = 0 \]
\[ i_R - i_C = 0. \]

Putting this all together, we have the following unknowns:

\[ \{ V_a, V_b, V_c, i_V, i_R, i_C, i_L \} \]

and the following equations

\[
V_a = \begin{cases} 
0 & 0 \leq t < 1 \\
1 & t \geq 1 
\end{cases}
\]

\[ L \cdot i_L' = (V_a - V_b) \]

\[ R \cdot i_R = V_b - V_c \]

\[ i_c = C \cdot V_c' \]

\[ i_V - i_L = 0 \]

\[ i_L - i_R = 0 \]

\[ i_R - i_C = 0. \]

Note that we could have simplified these equations further. For example, from the last three equations we know that the current through all of the components must be equal to \( i_V \). This would have eliminated the need to solve for \( i_R, i_C, \) and \( i_L \) altogether. For this example, we use all seven equations and all seven unknowns to demonstrate that \textit{a priori} manipulation of the equations is not necessary. Instead, the information given in the model is sufficient for such manipulations to be performed automatically by the simulator.
3.1.2 The Acausal Model of PhyDL for the Circuit

The PhyDL description of the RLC model is shown below, and the results of simulating this circuit can be seen in Figure 3.2.

```
simulation
    starting_time = 0; ending_time = 1.2;
    step_size = 0.001;
external matlab
    reads Va, Vb, Vc;
    observes event plot rate t = 0.001;
boundary
    iL with iL(0)= 0;
    Vc with Vc(0)= 0;
system
    R = 15; L = 0.1; C = 0.0001;
    Va = if t>=1 then 1 else 0;
    L * iL' = Va - Vb;
    R * iR = Vb - Vc;
    iC = C * Vc';
    iC = iL;
    iL = iR;.
```

3.2 Point-Free Notation

A physical system is an entity that can be separated from its environment but that also interacts with its environment, which results in observable changes over time. In other words, in a physical system, every variable is a function of time. We take this fact into the consideration of the design of language syntax. For example, if we write:

\[ F = m \cdot a; \]
then variables $F$, $m$, and $a$ will each be a function of time. For any time $t$, $F(t) = m(t) \times a(t)$. Syntactically, this gives the engineers the convenience to omit unnecessary parameters. More importantly, the language carries operators on numbers to functions of time $t$ for the user. The operator $\times$ takes two numbers and produces another number. With implicit-time notation, and every variable being a function of time, now the operator $\times$ takes two functions of time and produces another function of time. This happens for all other operators in the language. Discussion of the types of carried operators can be found in Section 4.2.

Some other languages also have implicit-time notation, including FRP and Modelica.
3.3 Mathematically Inspired Textual Notation

A language for modeling physical systems must support derivative and/or integration operations.

- Derivatives

For derivatives, we need second-order derivatives for system dynamics — for example, \( F = m \cdot p'' \). Some languages do not allow expressing higher-order derivatives directly. For example, in MATLAB, the ODE solvers accept only first-order differential equations. To use such solvers with higher-order ODEs, the user must rewrite each equation as an equivalent system of first-order equations of the form

\[
y' = f(t, y).
\]

Similarly, Modelica cannot express higher-order derivatives directly. Modelica uses operation \texttt{der(x)} to represent the time derivative of the variable \( x \). Notice that \( x \) must be a variable and cannot be an expression, and that the \texttt{der} cannot be used recursively to express higher-order derivatives. In other words, the following is not a legal way to represent the second derivative:

\[
\text{alpha} = \text{der(der(theta))}; \quad // \text{Illegal in modelica.}
\]

In order to represent the second derivative of a variable, the first derivative must be assigned to a variable. For example:

\[
\text{omega} = \text{der(theta)}; \quad // \text{First derivative.}
\]
\[
\text{alpha} = \text{der(omega)}; \quad // \text{Second derivative.}
\]

For the same example, we can write in Acumen as follows:

\[
\text{alpha} = \text{theta''}; \quad // \text{PhyDL.}
\]

In PhyDL, we use the prime notation for derivative over time. This is similar to what engineers use in their modeling. Mathematica has similar notations as PhyDL:
\[ \alpha[t] = \theta''[t];, \]

with the difference that time \( t \) must be explicitly written.

- **Integration**

  We also have integration operation in PhyDL. For example, if \( p \) is the integration of \( \nu \), we write

  \[ p = \text{integral}(v);. \]

Integration is from time \( t = 0 \) by default.

### 3.4 Module System

PhyDL has a module system for keeping the physical system model better structured. An important aspect of physical system modeling is to exploit the reusability gained by using acausal modeling formulations. In general, acausal models are easier to reuse than block diagrams because each component model can be formulated independently without knowledge of the equations or causality assumptions used in other parts of the system. This is because the causality for physical component models changes depending upon the context in which the model is used.

A module definition is used to describe a component in a physical system. A component is modeled independently of the environment in which it is used, which is essential for its reusability. This means that in the definition of the component/module that includes its equations, only local variables and port variables can be used. No means of communication between a component and the rest of the system, apart from going via a port, should be allowed.

Thus, we have a module definition that includes the *name* of the module, the *ports* of the module, and the system *equations* of the module. The name of the module is used to refer to the module definition. Ports are variables used to communicate between the inside and outside of the module. They are interfaces between a physical
component and the rest of a physical system. The dynamics and behavior of the component are described as equations of the module, and module equations have the same form as general system equations. A module definition in PhyDL looks as follows:

```phydl
module spring = {
    ports: x, f;
    k=1;
    f=k*x;
}.
```

Here, module and ports are keywords. The variable name spring is the name of the module, and variables x and f are ports of the module. Equations k=1; and f=k*x; are equations of the module.

When using this module definition in the system model, we write

```phydl
spring(x1, f1),
```

where variables x1 and f1 are variables that appear in the system model. By building the connection between x and x1, f and f1, these two pairs of variables will have identical values. If the same module definition is used multiple times in the system model, each one will represent a different instance of the physical component. The syntax of module definition and application can be found in Section 4.1, and an example of using module systems can be found in Section A.2.6.2.

Complex systems usually consist of large numbers of connected components, of which many components can be hierarchically decomposed into other components through several levels. Thus, the following definition is valid in PhyDL:

```phydl
module spring = {
    ports: x, f;
    k=1;
}.
```
\[ f = k \times x; \]

}`

module spring_damper = {
    ports: x, f;
    b=1;
    fb = b\times x';
    spring(x, fk);
    f=fb+fk;
};

The first module definition defines a component with the name \textit{spring}, which is used in the component defined in the second module definition. A component cannot be a part of itself. Also, two components cannot both be part of the other. Thus, modules cannot be defined recursively or mutually recursively.

### 3.5 Bus Suspension Example in PhyDL

As an example of the PhyDL program, the bus suspension modeled by mathematical equations

\[
\begin{align*}
    m_1 \times x_1'' &= -b_1 \times (x_1' - x_2') - k_1 \times (x_1 - x_2) + u \\
    m_2 \times x_2'' &= b_1 \times (x_1' - x_2') + k_1 \times (x_1 - x_2) + b_2 \times (w' - x_2') + k_2 \times (w - x_2) - u
\end{align*}
\]

has the PhyDL code as follows:

\[
\begin{align*}
    m_1 \times x_{1}'' &= -b_1 \times (x_{1}' - x_{2}') - k_1 \times (x_{1} - x_{2}) + u; \\
    m_2 \times x_{2}'' &= b_1 \times (x_{1}' - x_{2}') + k_1 \times (x_{1} - x_{2}) + b_2 \times (w' - x_{2}') + k_2 \times (w - x_{2}) - u;
\end{align*}
\]

The example shows that the syntax of PhyDL is very close to the mathematical model of the system.
Chapter 4

The Syntax and Semantics of PhyDL

In this chapter, we formalize the syntax and semantics of PhyDL.

4.1 Syntax

Physical systems are modeled as variables and relations in PhyDL. Each variable is a function of time and has type $\mathbb{R} \rightarrow \alpha$, where $\alpha$ is an arbitrary type. A PhyDL program is then a set of equations composed of those variables.

The basic types and program environment in PhyDL can be represented by the following BNF:

$$
\text{Type} \quad t \ := \ \mathbb{R} \mid t \times \cdots \times t \mid t \rightarrow t
$$

$$
\text{Program Environment} \quad \Gamma \ := \ \Box \mid c : n \mid x : t :: \Gamma.
$$

Here, $\mathbb{R}$ is the set of real numbers, and the type of time is “Real.” In PhyDL, real numbers are approximately represented by rationals (floating points); therefore, $\mathbb{Q}$ is used as the basic type for reals in PhyDL. The product type for array and tuple is denoted as $t \times \cdots \times t$, and function type is denoted as $t \rightarrow t$. A PhyDL variable is a function of time, so a variable in PhyDL or its derivative has type $\mathbb{Q}^n$, which means the product of $n \ \mathbb{Q}$s.

The syntax for core PhyDL is defined by the following BNF. We use $a|b$ or $[a,b]$ to denote $a$ or $b$, and use $[a]$ to denote that $a$ is optional (may or may not appear). In the definition of arithmetic expression, we write $\langle b,d_1,d_2 \rangle$ to denote a sequence and $\langle d_i \rangle$ to denote a sequence of $d$’s. We use $\{a_i\}$ to denote a set of $a$’s.
The syntax of PhyDL with interactive mode and module system is as follows.

Variable names  \( x \in \mathcal{X} \)

Variables  \( v ::= x \mid v' \)

Constants  \( c \in \mathbb{Q}^n \)

Primitive operators  \( f ::= + \mid - \mid * \mid / \mid \sin \mid \cos \mid \text{power} \mid \text{inner} \mid \text{cross} \)
\hspace{1cm} \mid \text{abs} \mid \text{index} \mid \text{direction} \mid \text{rotate} \mid \text{proj} \\

Logical operators  \( \oplus ::= > \mid = \mid < \mid \geq \mid \leq \mid \neq \)

Arith expressions  \( d ::= c \mid v \mid f(d) \mid \text{if}\{b,d_1,d_2\} \mid \text{integration}(d) \)

Boolean expressions  \( b ::= d \oplus d \mid -b \mid b \lor b \land b \)

External names  \( E \in \mathcal{E} \supseteq \{\text{ridl, matlab, GUI}\} \)

Module names  \( S \in \mathcal{S} \)

Timings  \( T ::= \text{starting.time} = c \mid \text{ending.time} = c \mid \text{step.size} = c \)

Equations  \( e ::= d = d \mid S\langle x_i \rangle \)

System modules  \( m ::= [\text{boundary} \{v_i(c_i) = d_i\}] \mid \text{system} \{e_i\} \)

Modules  \( s ::= \text{module} S = (\text{ports} \{x_i\} \mid m) \)

Interface commands  \( C ::= \text{reads} \{v_i\} \mid \text{writes} \{v_i\} \mid \{[\text{when}, \text{rate}] t = c_i\} \)

PhyDL programs  \( p ::= \{s_i\} \mid \text{simulation} T \{\text{external} E \{C_i\}\} \mid m. \)

Each of these syntactic productions represents a standard mathematical concept. In what follows, I explain each of these productions in the order in which they appear in the BNF definition.

\( \mathcal{X} \) is the set of variable names, and each variable name consists of characters in 0-9, _-_, a-z, and A-Z, with the first character being an English letter. A variable is represented by a variable name or a variable name followed by one or more “’” to denote the derivatives.

\( \mathbb{Q} \) is the set of fractions, or in this case floating-point numbers. Thus, a constant can be a number in \( \mathbb{Q} \) or a vector of fractions.

Primitive functions are used to compose arithmetic expressions. Operators +, −, *, /, and abs can be used both as scalar operations and vector operations. They have their usual mathematical meaning when applied to scalar. When applied
to vectors, the two operators $+$ and $-$ must have the same dimension. One of the operators of $*$ must be a scalar, and the second operator of $/$ must be a scalar. The expression $\text{abs}(a)$ computes the absolute value of $a$ if $a$ is a scalar and computes the length of $a$ if $a$ is a vector. $\sin$, $\cos$, and $\text{power}$ are scalar operators. The expression $\text{power}(a,b)$ computes value $a^b$ when both $a$ and $b$ are scalars. Operators $\text{inner}$ and $\text{cross}$ computes the inner product and cross product of two vectors. The expression $\text{index}(x,n)$ requires $x$ to be a vector and $n$ to be a natural number, and it is used to get the $n$-th component of vector $x$. The expression $\text{direction}(x)$ computes the unit vector with the same direction as vector $x$. The expression $\text{rotate}(x,a)$ requires $x$ to be a vector and $a$ to be a fraction. The result is to rotate a 2-D vector $x$ counter-clockwisely of angle $a$. The expression $\text{proj}(x,y)$ requires both $x$ and $y$ to be vectors, and the result gives the component of vector $x$ along the direction of $y$.

When a logical operator is applied to two numbers, it has its usual meaning. When it is applied to two vectors, they must have the same dimension, and the result is the conjunction of comparing each pair of corresponding components. For example, $(2, 3) > (1, 1)$ would be true, but $(2, 3) > (1, 4)$ would be false. The result of applying a logical operator to two operands is a Boolean expression. A Boolean expression can also be built from other Boolean expressions by using $\neg$, $\|$, and $\&\&$.

An arithmetic expression can be a constant, a variable, a function application, an if statement, or an integration of an arithmetic expression. Integration always starts at $t=0$ as default, if the statement is used to handle discontinuities in a physical model.

An equation in PhyDL can be an equation with two arithmetic expressions on both sides of the equality sign, or a module instantiation in the form of $S(x_i)$. The definition of the module $S$ appears at the beginning of a PhyDL program.

A PhyDL program $p$ has four sections, in which module definitions and external modules are optional. Current external names in PhyDL include “ridl,” “matlab,” and “GUI.” See Appendix A.2 for details of program composition.
4.2 Static Semantics

A PhyDL program that is not well typed does not have a valid denotational semantics. The typing rules formalize the following requirements:

1. A program.

   A program $p$ is well typed when the program is composed of a well-typed simulation section, a set of well-typed external sections, and when the set of equations from the boundary section and the system section is well typed. The simulation section and external sections require that $c_i$ must be a fraction:

   \[
   \begin{align*}
   \text{starting\_time} &= c_1 \\
   \text{ending\_time} &= c_2 \\
   \text{step\_size} &= c_3 
   \end{align*}
   \] (T-timing)

   \[
   \text{reads } \{v_i\} | \text{writes } \{v_i\} | \{\text{whenever } t = c_i\} \] (T-interface)

   \[
   \Gamma \vdash \{v_i(c_i) = d_i\} \cup \{e_j\}
   \]

   \[
   \Gamma \vdash \{\text{boundary } v_i(c_i) = d_i\} \; \text{system } \{e_j\} \] (T-system)

   \[
   \Gamma \vdash m
   \]

   \[
   \Gamma \vdash \text{module } S = (\text{ports } \{x_i\} \; m) \] (T-module)

   \[
   \{s_i\}, \; T, \; \{C_j\}, \; \Gamma \vdash m
   \]

   \[
   \Gamma \vdash \{s_i\} \; \text{simulation } T \; \{\text{external } E\{C_j\}\} \; m \] (T-program).

2. An equation.

   We then give rules for the judgments $\Gamma \vdash \{v_i(c_i) = d_i\} \cup \{e_i\}$ that the set of equations from the boundary section and the system section is well typed. We denote a set of equations $e_i$ as $E$. 

3. Arithmetic expressions.

In dynamic systems, a variable is always a function of time. A constant is also a function of time, with the function value being invariant. Thus, the type of a constant or a dynamic variable can be seen as $T \rightarrow t$, where $T$ is type for time, which is $\mathbb{R}$. We use fractions to approximate real numbers in a digital computer, so the type actually used for $T$ of time is $\mathbb{Q}$.

The following set of rules was first introduced in Functional Reactive Programming (FRP) [14] to lift primitive values and functions to dynamic values (function of time) and functions of dynamic values:

$$\text{lift0} :: t \rightarrow (T \rightarrow t)$$

$$\text{lift1} :: (t_1 \rightarrow t_2) \rightarrow ((T \rightarrow t_1) \rightarrow (T \rightarrow t_2))$$

$$\text{lift2} :: (t_1 \rightarrow t_2 \rightarrow t_3) \rightarrow ((T \rightarrow t_1) \rightarrow (T \rightarrow t_2) \rightarrow (T \rightarrow t_3)).$$

The following set of rules is used to define a well-typed arithmetic expression in PhyDL:
\[ \Gamma(x) = t \]
\[ \Gamma \vdash x : t \] (T-var)

\[ \Gamma(c) = \mathbb{Q}^n \]
\[ \Gamma \vdash c : T \rightarrow \mathbb{Q}^n \] (T-constant)

\[ \Gamma \vdash e_i : t_i \]
\[ \Gamma \vdash (e_1, \cdots, e_n) : t_1 \times \cdots \times t_n \] (T-array)

\[ \Gamma \vdash x : t \]
\[ \Gamma \vdash \sin(x) : t, \cos(x) : t \] (T-sin,cos)

\[ \Gamma \vdash x : bool, y_1 : t, y_2 : t \]
\[ \Gamma \vdash if \ (x, y_1, y_2) : t \] (T-if-stmt)

\[ \Gamma \vdash x : t^n, y : \mathbb{N} \]
\[ \Gamma \vdash index(x, y) : t \] (T-index)

\[ \Gamma \vdash x : t^n, \Gamma \vdash y : t \]
\[ \Gamma \vdash \text{rotate}(x, y) : t^2 \] (T-rotate)

\[ \Gamma \vdash x : t^n, \Gamma \vdash y : t \]
\[ \Gamma \vdash x \times y : t^n, y \times x : t^n \] (T-*)

\[ \Gamma \vdash x \ast y : t^n, y \ast x : t^n \] (T-*)

\[ \Gamma \vdash x/y : t^n, \text{pow}(x, y) : t^n \] (T-/power)

\[ \Gamma \vdash x : t^n \]
\[ \Gamma \vdash \text{abs}(x) : t, \text{direction}(x) : t^n \] (T-abs,direction)

\[ \Gamma \vdash x : t^n, y : t^n \]
\[ \Gamma \vdash \text{inner}(x, y) : t, \text{proj}(x, y) : t^n \] (T-inner,proj)

\[ \Gamma \vdash x : T \rightarrow t \]
\[ \Gamma \vdash x' : T \rightarrow t, \text{integration}(x) : T \rightarrow t \] (T-deriv,integration).

4. Boolean expressions.
\[
\Gamma \vdash x : T \to \mathbb{Q}^n, \ y : T \to \mathbb{Q}^n \\
\Gamma \vdash x \oplus y : T \to \text{bool} \quad \text{(T-comparison)} \\
\Gamma \vdash x : T \to \text{bool} \\
\Gamma \vdash \neg x : T \to \text{bool} \quad \text{(T-negation)}
\]

\[
\Gamma \vdash x : T \to \text{bool}, \ y : T \to \text{bool} \\
\Gamma \vdash x||y : T \to \text{bool}, \ x\&\&y : T \to \text{bool} \quad \text{(T-and, or)}.
\]

### 4.3 Interpreting Acumen Programs

PhyDL allows the user to directly describe dynamic equations governing the system models. For these equations,

- Each equation allows expressions on both sides of the equal sign. Thus, each equation specifies a constraint rather than an assignment as in most mainstream programming languages.

- The order of each equation in a PhyDL program is not related to the computation order of variables.

Therefore, a PhyDL program is a set of constraints on variables that have appeared in the program. Each variable in PhyDL is a function of time satisfying the constraints. So a semantic function should take the set of constraints and generate a function of time \((t)\) for each variable, from which the value of the variable at a given instance of time is directly computable.

This section defines the semantics for PhyDL, where we define a PhyDL program by a series of transformations that lead to a causal sequence of equations with a specified form. Each variable in the original program or its derivatives is uniquely defined in the result sequence of equations, and the order of the equations agrees with the data dependencies of the variables.

The transformations are as follows:

1. Module instantiation.
2. Topological sorting.

3. Order reduction.

4. Discretization.

4.3.1 Semantics for a Simple Pendulum Example

Figure 4.1 shows a simple pendulum. The dynamics of the system are defined by the following equation:

\[ I \cdot a'' - m \cdot g \cdot \sin(a) = 0, \]

where \( a \) is the angle between the rod and the vertical line and \( I \) is the moment of inertia of the system. Let the 2-D position of the ball be \( (x, y) \). The center piece of PhyDL code for describing this system is

```phydl
module pendulum = {
    ports: a, m, l;
    I = m*l*l;
    I*a'' - m*g*sin(a) = 0;
}
```
system
m = 1;
l = 1;

pendulum(a, m, l);
x = l*sin(a);
y = l*cos(a);

The transformation process for the PhyDL code is shown in Table 4.1

<table>
<thead>
<tr>
<th>(1) PhyDL code after module instantiation</th>
<th>(2) Topologically sorted PhyDL code</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 1; l = 1; I = m<em>l</em>l; x = l<em>sin(a); y = l</em>cos(a); I<em>a'' - m</em>g*sin(a) = 0;</td>
<td>m = 1; l = 1; I = m<em>l</em>l; a'' = m<em>g</em>sin(a)/I; x = l<em>sin(a); y = l</em>cos(a);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(3) Order-reduction PhyDL code</th>
<th>(4) Discretized code</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 1; l = 1; I = m<em>l</em>l; a1' = a2; a2' = m<em>g</em>sin(a1)/I; x = l<em>sin(a1); y = l</em>cos(a1);</td>
<td>m = 1; l = 1; I = m<em>l</em>l; a1(n+1) = a1(n)+dt<em>a2(n); a2(n+1) = a2(n)+dt</em>(m<em>g</em>sin(a1(n))/I); x(n) = l<em>sin(a1(n)); y(n) = l</em>cos(a1(n));</td>
</tr>
</tbody>
</table>
4.3.2 Module Instantiation

We use the simple pendulum example, and the steps of its module instantiation are as follows:

- **Matching parameters.** When calling the `pendulum` module in the system section by `pendulum(a, m, l)`, replace the ports variables `(a, m, l)` in the `pendulum` definition with `(a, m, l)` from the calling instance.

- **Temporaries renaming.** A module definition might be instantiated for multiple times. Variables that are not ports have the same name in a different instantiation. The compilation renames all temporaries in a different instantiation by adding `#` followed by a number. After one instantiation is done, the count of the number is incremented. Since `#` is not part of PhyDL syntax for naming variables, this makes sure that there is no name clash between renamed variables and existing variables. Different numbers generated for a different instantiation ensures no name clash between renamed variables.

- **Recursive instantiation.** A module definition can also refer to other module definitions. Check all generated equations see if there are module references in them. If so, instantiate the module reference. In order to avoid circular references in module definitions, a stack is used in module instantiation to keep track of module definitions used. If a circle is detected, return an error message.

- **Putting equations together.** Put all generated equations back to the system section.

4.3.3 Topological Sorting

Topological sorting takes an acausal model in PhyDL and converts it to a causal model. In other words, we want to translate a PhyDL program to a list of assignments, in which the data flow is explicit. The topological sorting process is partial. In other
words, there are acausal equations that cannot be directed. To redirect a set of acausal equations, the following procedure is used:

- We construct a graph whose nodes are the equations and variables of the problem.
- For each equation node, we add an edge between that node and each variable that occurs in the equation.
- We use Tarjan’s strongly connected components algorithm [24] to sort the equations and find the strongly connected components.

We say a variable $v$ is “solved,” if there is an assignment $v = \text{exp}$ such that each variable in $\text{exp}$ either (1) is a constant, (2) is “solved”, or (3) is a variable of a lower-order derivative of the same variable as $v$ (e.g., $p'' = 0.5 \times p$).

In the last step of the sorting procedure, we first identify equation nodes that each have only one variable node connected to them. This means that the only variable $v$ connected to an equation may be solved from the equation, and we derive a new causal equation $v = \text{exp}$ for it. We mark the variable node as “solved” and mark the equation node as “used.” We remove all edges connected to variables that are marked as “solved” and repeat the same process from identifying equation nodes that only connect to one variable. If, in the end, all of the variable nodes are marked as “solved,” we have converted the acausal model to a set of causal equations in the form of $v = \text{exp}$.

The sorting procedure fails if one of the following cases happens:

1. If only one unmarked variable $v$ appears in an equation, but the variable $v$ appears in two function applications, PhyDL will not be able to solve $v$ from the equation. For example, PhyDL cannot solve variable $a$ from the equation $\sin(a) + \cos(a) = 1$.

2. If there is an equation that has no edge connected to it and the equation is not marked as “used,” then there is an over-constraint in the model, and PhyDL
will stop the sorting and report an error for the program.

3. If some variable nodes are still unmarked, but every un-marked equation has more than one variable node connected to it, we find all variables whose derivatives also appear and mark those variable nodes as “numerically solvable.” Repeat the same process by treating those variables as “solved.” If there are still variable nodes left as unmarked, the sorting procedure fails to direct the equations.

We use a simple example to show the whole procedure. Suppose we have the following acausal equations:

\[ p' - m \cdot p = 0 \]  \hspace{1cm} (4.1)
\[ 5 \cdot p = y \] \hspace{1cm} (4.2)
\[ m = 1. \] \hspace{1cm} (4.3)

The sorting procedure can be described using the following graphs:

The graph has three equation nodes and three variable nodes, with the edges between them (Fig. (a)). At the beginning, equation (4.2) is connected to only one variable node \( m \), so we can solve \( m \) by \( m = 1 \). We mark \( m \) as “solved” and mark (4.2) as “used” (Figure (b)). Then, no more equations are connecting to only one variable, so we identify \( p \) as “numerically solvable” because implicitly we have \( p = \int p' \). Now, equations (4.3) and (4.4) only have one variable connected (Fig. (d)), and we get \( p' = m \cdot p \) and \( y = 5 \cdot p \). We mark \( p' \) and \( y \) as “solved” and (4.3) and (4.4) as “used”
as in Fig. (e). The equations have been directed successfully, and the following causal equations have been generated:

\[ m = 1 \]
\[ p' = mp \]
\[ y = 5p. \]

4.3.4 Order Reduction

Order reduction replaces any derivative \( x^{(n+1)} \) by \( y' \), where \( y = x^{(n)} \) and \( y \) is free in the program.

4.3.5 Discretization

The denotational semantics of a PhyDL program is a causal sequence of equations, each of which define a variable or the derivative of a variable. In the first case, it is straightforward to compute the value of the variable, and the second case gives an ODE. We need to make the denotational semantics into an actual computer program that computes solutions to the ODEs, but there is no general solution to an arbitrary ODE.

Numerical methods [12, 2] can be used to find approximate solutions for almost all ODEs, and the error is usually satisfiable in most non-stiff systems. The basic idea there is that we first discretize the continuous time into discrete time instances and assume that system state only has value at those time instances. Then, we rewrite the original differential equation in some form of difference equations in which integral operation can be eliminated. According to the way we translate the differential equations into difference equations, numerical methods can be classified as one-step methods or multi-step methods, and depending upon the way we discretize continuous time, numerical methods can be classified as fixed-step methods or varying-step methods. Among different numerical methods, there are trade-offs between complexity, stability, preciseness, etc. However, regarding implementation, they are very close to
each other. In this section, we explain our implementation of ODE solver semantics based on a very natural numerical method called the \textit{Euler Method}. It is a simple but efficient numerical method. Several other numerical methods have also been implemented in Acumen.

The Euler Method for solving ODEs can be described as follows. Assume that $f(t,x)$ is continuous and satisfies a Lipschitz condition\footnote{A function $f(x)$ satisfies the Lipschitz condition if there exists a constant $K \geq 0$ such that for all $x_1, x_2$ in its domain, $|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$} in the variable $x$, and consider the initial value problem

$$x' = f(t,x),$$

with $x(a) = x_0$, over the interval $a \leq t \leq b$. Euler’s method uses the formulas

$$t_{k+1} = t_k + h,$$

and

$$x_{k+1} = x_k + h \cdot f(t_k, x_k) \quad (4.4)$$

for $k = 0, 1, 2, \ldots, m - 1$ as an approximate solution to the differential equation using the discrete set of points $\{t_k, y_k\}_{k=0}^{(m-1)}$, with $t_0 = a$ and $h = t_{k+1} - t_k$.

The solving process is a discretization that transforms continuous differential equations into discrete time-difference equations. Different numerical methods can be applied in this process. Currently, three numerical solvers are implemented in Acumen. They use the Euler [3], RK4 [13], and BS3 [1] methods. The step size of the solver is defined in the simulation section.

Numerical methods inevitably produce inexact solutions. In Chapter 5, we discuss those errors in detail.

\section*{4.4 Quantitative Evaluation of Acumen Accuracy}

We have used Acumen in modeling and simulating a down-hole tractor for Schlumberger. The purpose of the simulation is to observe the system behaviors under different possible control algorithms. We are interested in the simulation results of
four system variables: grip position, motor speed, motor torque, and tractor position. The same system has been modeled using MATLAB/Simulink. We compared the simulation results between Acumen and Simulink (Figure 4.2), and obtained the following results:

- The maximum differences of each variable after $2 \times 10^5$ simulation steps are:
  - less than 0.26% for grip position,
  - less than 13.06% for motor speed,
  - less than 4.34% for motor torque,
  - less than 0.00132% for tractor position.

- The values of these variables consistently converge to the same state.

Figure 4.2: Accuracy of Acumen vs. Simulink.
The Acumen code and Simulink code come from exactly the same mathematical model, but the modeling process (from mathematical model to computer program) is different. System modeling in PhyDL uses a higher-level declarative manner, which has less possibility of introducing modeling errors.
Chapter 5

Error Models of ODE Solvers

PhyDL allows both linear and non-linear Ordinary Differential Equations (ODEs) in the program. Existing analytical methods can solve linear ODEs exactly, but there is no general method for finding closed-form solutions for nonlinear ODEs. A set of ODEs with initial conditions is called the Initial Value Problem (IVP). Numerical methods are used in PhyDL to solve ODEs with initial conditions, but they inevitably introduce errors. It is important to understand the errors and their effect in simulation results.

In this chapter, I will introduce some numerical methods for computing approximated solutions to ODEs and their corresponding error estimation mechanisms.

5.1 Numerical Methods and Source of Errors

The basic idea of numerical methods is to discretize a continuous function, so that a differential equation is approximated by a difference equation.

Definition 5.1 (Grid, discretization and approximation). Consider the standard IVP

\[ y' = f(t, y), \quad y(t_0) = y_0. \]  

(5.1)

Let \( y : \mathbb{R} \rightarrow \mathbb{R}^n \) be an exact solution to (5.1). Let \( \{t_0, t_1, \ldots, t_N\} \) be a set of time points with increasing order, and assume we would like to approximate the solution values \( y(t_i), i = 1, 2, \ldots, N \) by \( \tilde{y}_i, i = 1, 2, \ldots, N \). The set \( \{t_i\}_{i=0}^N \) is called the grid, the set \( \{y(t_i)\}_{i=0}^N \) is called a discretization of the solution \( y \) on the grid \( \{t_i\}_{i=0}^N \), and the set \( \{\tilde{y}_i\}_{i=0}^N \) is called an approximation of the solution \( y \) on the grid \( \{t_i\}_{i=0}^N \).
**Definition 5.2 (Step function).** A numerical method to approximate the solution to IVP (5.1) is given by

\[ y(t_{i+1}) \approx y(t_i) + h_i \cdot \Phi(\cdot), \]

where \( h_i = t_{i+1} - t_i \). \( \Phi \) is called a **step function**. If \( \Phi \) is dependent upon \( \{y(t_{i-j})\}_{j \leq 1, j \in \mathbb{N}} \), we have a **multi-step** method; if \( \Phi \) is not dependent upon \( \{y(t_{i-j})\}_{j \leq 1, j \in \mathbb{N}} \), we have a **one-step** method; if \( \Phi \) is dependent upon \( y(t_{i+1}) \), we have an **implicit** method; and if \( \Phi \) is not dependent upon \( y(t_{i+1}) \), we have an **explicit** method. An explicit, one-step method can be written as

\[ y(t_{i+1}) \approx y(t_i) + h_i \cdot \Phi(t_i, y(t_i), h_i). \]

Explicit, one-step methods are mathematically natural, and do not complicate the computation and error estimation when the step size is changed. Thus, they are widely used in current IVP solvers.

**Example 5.1 (Euler forward method).** The simplest way to approximate the solution \( y \) to the IVP (5.1) by a step function \( \Phi \) is to approximate \( y \) on the interval \([t_i, t_{i+1}]\) by its value at \( t_i \). This yields the **Euler forward formula**:

\[ y(t_{i+1}) \approx y(t_i) + h_i \cdot f(t_i, y(t_i)). \]

**Example 5.2 (Runge Kutta Method with order 4 (RK4)).** The classical formula of Runge and Kutta can be expressed as

\[ y(t_{i+1}) \approx y(t_i) + \frac{1}{6}h_i \cdot (k_1 + 2k_2 + 2k_3 + k_4), \]

where

\[
\begin{align*}
sty{k}_1 &:= f(t_i, y(t_i)) \\
sty{k}_2 &:= f(t_i + \frac{1}{2}h_i, y(t_i) + \frac{1}{2}hk_1) \\
sty{k}_3 &:= f(t_i + \frac{1}{2}h_i, y(t_i) + \frac{1}{2}hk_2) \\
sty{k}_4 &:= f(t_i + h_i, y(t_i) + hk_3).
\end{align*}
\]
The “order 4” in its name is related to the error estimation of this method, which will be explained in the following.

5.1.1 Local Truncation Errors

**Definition 5.3** (Local truncation error and consistency order). The normalized local truncation error $\delta(\cdot)$ of step function $\Phi$ (explicit, one-step method) is defined by

$$\delta(y(t_i), h_i) := \frac{y(t_i + h_i) - y(t_i)}{h_i} - \Phi(t_i, y(t_i), h_i).$$

We call $p$ the rate of convergence (or order) if $\delta(y(t_i), h_i) = O(h^{p+1})$, for $p$ as large as possible.

Usually, if the solution is sufficiently smooth\(^1\), we can compute the local truncation error from Taylor expansion of $y$ around $t_i$:

$$y(t_i + h) = y(t_i) + y'(t_i)h + \frac{y''(t_i)}{2!}h^2 + \frac{y'''(t_i)}{3!}h^3 + \ldots + \frac{y^{(n)}(t_i)}{n!}h^n + R_n(\xi),$$

where the remainder $R_n(\xi) = \frac{y^{(n+1)}(\xi)}{(n + 1)!}h^{n+1}$ for some $\xi \in [t_i, t_{i+1}]$.

**Example 5.3** (Local truncation error of the Euler forward method). The Euler forward method has

$$\Phi(t_i, y(t_i), h_i) = f(t_i, y(t_i)) = y'(t_i).$$

Using Taylor expansion of $y$ at $t_i$, we have

$$\delta(y(t_i), h_i) := \frac{y(t_i + h_i) - y(t_i)}{h_i} - \Phi(t_i, y(t_i), h_i)$$

$$= \frac{1}{h}(y'(t_i)h + \frac{y''(\xi)}{2!}h^2) - y'(t_i)$$

$$= \frac{y''(\xi)}{2}h = O(h).$$

---

\(^1\)A function as solution is smooth, if the derivative and higher-order derivatives of the function exist.
Thus, the Euler forward method has consistency order 1.

**Example 5.4** (Local truncation error of the RK4 method). *Similar to example 5.3, we can prove that the local truncation error of RK4 is*

\[ \delta(y(t_i), h_i) = \frac{y^{(5)}(\xi)}{5!} h^4 = O(h^4). \]

*Thus, the RK4 method has consistency order 4, and the order is indicated in the name of this method.*

### 5.1.2 Propagation of the Error

The local truncation error computed at grid point \( t_i \) as in 5.3 assumes that exact \( y(t_i) \) was known in \( \Phi(\cdot) \). In practice, this is not the case. We thus define the global error as follows.

**Definition 5.4** (Global error). *The global error of a numerical method at grid point \( t_i \) is defined as*

\[ e(t_i; t_0, y_0) := y(t_i) - \tilde{y}_i, \]

*assuming the grid is \( \{t_i\}_{i=0}^N \) and \( \tilde{y}_0 = y(t_0) \).*

Notice that

\[
e(t_{i+1}; t_0, y_0) := y(t_{i+1}) - \tilde{y}_{i+1} \\
= (y(t_{i+1}) - \Phi(t_i, y(t_i), h_i)) + (\Phi(t_i, y(t_i), h_i) - \tilde{y}_{i+1}) \\
= \delta(y(t_{i+1}, h_{i+1})) + (\Phi(t_i, y(t_i), h_i) - \Phi(t_i, \tilde{y}_i, h_i)).
\]

The first term in the summation is the local error at grid point \( t_{i+1} \). The second term is approximately (\( h_i \) is small):

\[
(1 + h_i \cdot J_i) \cdot e(t_i; t_0, y_0).
\]

where \( J_i \) is the Jacobian of the \( f \) at \( \tilde{y}_i \). The eigenvalues of \( J_i \) determine the stability of the differential equation in the IVP. Thus, the second term measures the sensitivity of the problem itself and not the method being used.
We can also describe the propagation effect of local truncation error by providing the ODE with a perturbation “source” term $r$, as

\[
\begin{aligned}
\dot{y}' &= f(t, y) + r(t, y) \quad r(t, y) = O(h^p) \\
\dot{y}(t_0) &= y_0
\end{aligned}
\]

Again, the method being used only affects the perturbation term $r$. How the solution differs from original IVP still depends upon the stability of the differential equation in the original IVP.

Theorems prove that, if $f$ in IVP (5.1) is locally Lipschitz continuous on $[t_0, t_N]$ with constant $L$, the global error satisfies $e(t_i; t_0, y_0) = O(h^p)$, where $p$ is the consistency order of the numerical method being used. However, in practice, this formula is usually useless due to the fact that the constant $C$ in the asymptotic upper bound $e(t_i; t_0, y_0) \leq C \cdot h^p$ can be extremely big. This constant $C$ has factor $(e^{TL} - 1)/L$, where $L$ is the Lipschitz constant and $T = t_N - t_0$.

### 5.1.3 Round-off Errors

From the above discussion we see that when a numerical method with consistency order $p$ is used, the global error has asymptotic upper bound $O(h^p)$. It appears that if we can choose an $h$ that is sufficiently small, we can still achieve a good approximation no matter what the constant $C$ is for the asymptotic upper bound.

However, it is not the case if we take round-off errors into account. The total round-off error in a computation depends upon the number of evaluations in the computation. When $h$ is too small, the total number of evaluations needed to complete the computation will be extremely large, which means that the error from the numerical method is much less than the round-off errors. Thus the upper bound estimated on the global error does not hold anymore.
5.2 Error Estimation for Numerical Methods

Local error estimation is usually used in step-size control when applying a numerical method. If a tolerance for local error is set and the step size $h_i = t_{i+1} - t_i$ is changeable, we can estimate the local error and decide whether it is less than the tolerance. If the local error is too large, we can reduce the step size and get an approximation with smaller local error. This process is repeated until the local error is smaller than the tolerance.

Global error estimation is usually used to measure the quality of the approximation we obtain from a numerical method. We are more interested in global error because it gives the difference between the solution of the IVP problem and the numerical result returned by a numerical method.

In general, local errors are easier to estimate than global errors due to error propagation. However, as we will show, even estimating local errors is not straightforward.

5.2.1 Estimating the Local Truncation Error

As we showed in the previous section, a numerical method with consistency order $p$ has its local truncation error as $O(h_i^p)$, where $h = t_{i+1} - t_i$ is the step size used. To get an estimate rather than the asymptotic upper bound, there are three methods that are normally used: Taylor expansion, extrapolation, and embedding. We now discuss these three methods.

Estimating the Taylor remainder

In Section 5.1.1 we got the asymptotic upper bound of local truncation error by using Taylor expansion. Usually, a method of order $p$ has its local error as

$$\delta(y(t_i), h_i) = \frac{y^{(p+1)}(\xi)}{(p + 1)!} h_i^p \quad \xi \in [t_i, t_i + h_i].$$

(5.2)

This error term is not easily evaluated. First, we do not know the exact solution $y$. Second, we do not know which $\xi$ in $[t_i, t_{i+1}]$ should be used here.
One way to evaluate (5.2) is to form an enclosure \([y_i]\), such that
\[
\forall t \in [t_n, t_{n+1}], y(t) \in [y_i].
\]
Then from
\[
y^{(p+1)} = \frac{d^p f}{dt^p} = \frac{d^p f}{dy^p}y' = \frac{d^p f}{dy^p}f
\]
we have
\[
y^{(p+1)}(\xi) = \left(\frac{d^p f}{dy^p}f\right)(\xi) \in \left(\frac{d^p f}{dy^p}f\right)[y_i].
\]
Thus,
\[
\delta(y(t_i), h_i) \leq \frac{h^p}{(p + 1)!} \max\{\left(\frac{d^p f}{dy^p}f\right)[y_i]\}.
\]
It remains to show how to compute this enclosure \([y_i]\). Notice that this enclosure should satisfy:
\[
y(t_n) + [0, h_i]f[y_i] \subseteq [y_i]. \tag{5.3}
\]
If we start from an initial guess of \([y_i]\), if it satisfies equation (5.3), we find such an enclosure; otherwise, make a “bigger” guess or reduce \(h_i\), until equation (5.3) is satisfied. Notice that this method involves interval arithmetic.

**Example 5.5** (Local error estimation for the Euler method using Taylor expansion). Given IVP,
\[
y' = 2y + 1, \quad y(0) = 0,
\]
and we suppose we know that \([y_i] = [e^{2t_i} - 1, e^{2(t_i+h)} - 1]\) is a closure that for all \(t \in [t_i, t_i + h]\), \(y(t) \in [y_i]\) holds. The local error of using the Euler forward method with step size \(h\) at \(t = t_i\) would be
\[
\delta(y(t_i), h) = \frac{y''(\xi)}{2}h, \quad \xi \in [t_i, t_{i+1}]
\]
\[= \frac{h}{2}(4y(\xi) + 2)\]
\[\leq \frac{h}{2}(4e^{2(t_i+h)} - 2).\]
Local error estimation using extrapolation

Suppose a numerical method has step function $\Phi(\cdot)$ and consistency order $p$. Also suppose that beside an approximation

$$\tilde{y}_{i+1} = y(t_i) + h_i \Phi(t_i, y(t_i), h_i)$$
on grid \{t_i\}_{i=0}^N$, we compute

$$\tilde{z}_{i+1/2} = y(t_i) + \frac{h_i}{2} \Phi(t_i, y(t_i), \frac{h_i}{2})$$

$$\tilde{z}_{i+1} = \tilde{z}_{i+1/2} + \frac{h_i}{2} \Phi(t_i + \frac{h_i}{2}, \tilde{z}_{i+1/2}, \frac{h_i}{2}).$$

We have

$$h_i \delta_y(y(t_i), h_i) = \frac{y(t_i + h_i) - y(t_i)}{h_i} - (\tilde{y}_{i+1} - y(t_i))$$

$$= c \cdot h^{p+1} + O(h^{p+2})$$

$$h_i \delta_z(y(t_i), h_i) = \frac{y(t_i + h_i) - y(t_i)}{h_i} - (\tilde{z}_{i+1} - y(t_i))$$

$$= 2(c \cdot \left(\frac{h_i}{2}\right)^{p+1}) + O(h^{p+2})$$

$$= 2^{-p} \cdot c \cdot h^{p+1} + O(h^{p+2}).$$

Thus,

$$y(t_i + h_i) = \tilde{y}_{i+1} + h_i \delta_y(y(t_i), h_i)$$

$$= \tilde{z}_{i+1} + h_i \delta_z(y(t_i), h_i)$$

$$= \tilde{z}_{i+1} + 2^{-p} \cdot h_i \delta_y(y(t_i), h_i).$$

This gives

$$\delta_y(y(t_i), h_i) = \frac{1}{h} (\tilde{z}_{i+1} - \tilde{y}_{i+1})/(1 - 2^{-p}) + O(h^{p+1}).$$

**Example 5.6** (Local error estimation for the Euler method using extrapolation).

*Given IVP*

$$y' = 2y + 1, y(0) = 0.$$

*Suppose at time $t_i$ the approximated value for $y(t_i)$ is $\tilde{y}_i$. Applying step function $\Phi$ with step size $h$, we compute that the approximated value for $y(t_i + h)$ is $\alpha$; applying*
step function \( \Phi \) twice with step size \( h/2 \), we compute that the approximated value for \( y(t_i + h) \) is \( b \). Then the local truncation error of computed value \( a \) is

\[
\frac{1}{h} (b - a) / (1 - 2^{-1}) + O(h^2) \approx \frac{2}{h} (b - a).
\]

**Local error estimation using embedding**

Let numerical method 1 with step function \( \Phi_1 \) have consistency order \( p_1 \), and numerical method 2 with step function \( \Phi_2 \) have consistency order \( p_2 \geq p_1 + 1 \). Assume \( y(t) \) is the exact solution to IVP \( y' = f(y), \ y(t_0) = y_0 \). We have:

\[
\hat{y}_{i+1} = y(t_i) + h_i \Phi_1(t_i, y(t_i), h_i)
\]

\[
\hat{z}_{i+1} = y(t_i) + h_i \Phi_2(t_i, y(t_i), h_i).
\]

Then

\[
h\delta_1 = y(t_i + h_i) - \hat{y}_{i+1} = O(h^{p_1+1})
\]

\[
h\delta_2 = y(t_i + h_i) - \hat{z}_{i+1} = O(h^{p_2+1}).
\]

Thus,

\[
\hat{z}_{i+1} - \hat{y}_{i+1} = -h(\delta_2 - \delta_1) = h\delta_1 + O(h^{p+2}).
\]

So

\[
\delta_1 = (\hat{z}_{i+1} - \hat{y}_{i+1})/h + O(h^{p+1}).
\]

**Example 5.7** (Local error estimation using embedding). Given IVP,

\[
y' = 2y + 1, \ y(0) = 0.
\]

Suppose that at time \( t_i \) the approximated value for \( y(t_i) \) is \( \hat{y}_i \). Apply step function \( \Phi_1 \) of Euler forward method. With step size \( h \), and we compute that the approximated value for \( y(t_i + h) \) is \( a \). Apply step function \( \Phi_2 \) of the RK4 method with step size \( h \), we compute that the approximated value for \( y(t_i + h) \) is \( b \). Then the local truncation error of computed value \( a \) is

\[
\frac{1}{h} (b - a) + O(h^2) \approx \frac{1}{h} (b - a).
\]
5.2.2 Estimating the Global Error

Estimating global error using an analysis similar to the discussion in Section 5.1.2 turns out to be very complicated. In practice, people use extrapolation and embedding in local error estimation. However, to apply these methods, we need to first get an asymptotic upper bound for the global error. The following theorem is a stepping stone for applying extrapolation and embedding methods in global error estimation.

**Theorem 5.5** (Global principle error function). Let \( f \) in IVP \( y' = f(y), y(t_0) = y_0 \) be sufficiently smooth. Then there exists a so-called global principle error function

\[
d : t, y(t) \mapsto d(t, y(t)).
\]

such that the global error is given by

\[
e(t_i; t_0, y_0) = d(t_i, y(t_i))h^p + O(h^{p+1}),
\]

where \( p \) is the consistency order of the numerical method being used.

**Example 5.8** (Global error estimation using embedding). Given IVP,

\[
y' = 2y + 1, y(0) = 0.
\]

Using the Euler method, at time \( t_N \) the approximated value for \( y(t_N) \) is \( a \), and the error is \( e_a \). Using the RK4 method, at time \( t_N \), the approximated value for \( y(t_N) \) is \( b \), and the error is \( e_b \). From theorem 5.5, we know that:

\[
e_a = y(t_N) - a = d(t_N, y(t_N))h^1 + O(h^1)
\]

\[
e_b = y(t_N) - b = d(t_N, y(t_N))h^4 + O(h^4).
\]

Thus

\[
b - a = -(e_b - e_a) = e_a + O(h).
\]

So, global error of \( a \) using the Euler method is

\[
e_a = b - a + O(h).
\]
5.2.3 Effect of Round-off Errors in Error Estimation

Notice that almost all of the error estimation methods discussed above have an asymptotic term or upper bound in the form of $O(h^n)$, where $h$ is the step size used and $n$ is a number related to the consistency order of the numerical method being used. We can only assume that an asymptotic term like this is very small in case a method has a big consistency order, or the step size is very small. Usually, a decrease in the step size also means an increase in the number of total evaluations needed. A method of higher consistency order can yield a smaller error for the same step size. In other words, a method of higher consistency order can yield the same error with a bigger step size. A large step size reduces the number of evaluations needed, and thus results in less round-off errors in general.

Theoretically, if an error has an asymptotic bound $O(h^n)$, we can make the error close to 0 by reducing $h$. But when taking round-off error into account, this is not true. Unfortunately, numerical methods do not have an effective way to estimate or control round-off errors during the computation.
Chapter 6

Interactive Simulation

So far, I have introduced the simulation environment Acumen and its physical description language PhyDL. I have discussed the design of PhyDL as well as its types, syntax, and semantics. In the rest of the thesis, Acumen is evaluated using some case studies. My experience with Acumen regarding its usability and accuracy will be reported.

One application of Acumen is to use it for interactive simulation, which can be used for implementing virtual environments with haptic feedback. It is necessary that the simulation run in real time and that it be interactive with the user. In this chapter, I will introduce the motivation for this application, the simulation infrastructure, and some case studies.

6.1 Problems with the Manual Implementation of Haptic Task

Haptic feedback systems are an important class of virtual environments. Haptic feedback is force feedback in a human-computer interface [20], relaying realistic, tangible sensations to a user. In a virtual environment, haptic feedback augments visual feedback to allow the user to push, pull, feel, and manipulate virtual objects. The application of haptic interfaces in areas such as computer-aided design and manufacturing (CAD/CAM), design prototyping, and production evaluation allows users to interact with and visualize virtual objects before manufacturing them. Along the same lines, the users of simulators for training in surgical procedures, control panel operations, and hostile work environments benefit from increased sensations of realism in the vir-
virtual environment that provides force feedback. The haptic display, or force-reflecting interface, the device that allows the user to interact with a virtual environment, consists of a real-time simulation of a virtual environment and a manipulator, which serves as the interface between the human operator and the simulation.

Today, the vast majority of the programming needed to make a haptic system operational is carried out using ad hoc methods in systems-programming languages such as C++. This approach has at least two shortcomings. First, for the resulting system to be reliable or efficient, advanced C++ expertise is generally needed. This is a burdensome requirement to place on the developers of such a system, be they engineers or students. Second, the development time using ad hoc approaches tends to be significant, slowing research and innovation that depends upon having a functioning haptic system by sometimes months or more. This dependence upon expert developers is easier to envision when we look closely at what is involved in programming such systems. Here are some examples:

1. **Device interfacing** involves building a layer of software on top of device drivers to carry out digital-to-analog and analog-to-digital conversions.

2. **Modeling device kinematics** generally requires introducing the various appropriate abstractions for representing the human user, coupled via the haptic device, in the virtual environment.

3. **Simulation infrastructure** involves multi-threading for haptic and graphic rendering loops, and is device-specific. While generic simulation infrastructure often exists, it is often the case that there is no guarantee of the physical fidelity of such generic engines. In essence, simple simulation engines are built from scratch to ensure their conceptual soundness.

4. **Virtual environment modeling** is required to simulate the kinematic and dynamic behavior of the desired virtual system or scene and to track interactions between the human and the objects in the environment.
While some libraries and simulation engines such as CHAI (Computer Haptic Active Interface) 3D [19] are now available, even the most basic offerings require substantial knowledge of C++ for industrial or research-grade use. Tools such as MATLAB and Simulink offer a potentially higher-level interface to modeling dynamics, but they are generally too slow and often are not expressive enough to describe dynamics succinctly. With such tools, development of graphics to accompany the haptic simulation still involves a significant amount of labor.

6.2 The iAcumen Environment: An Environment for Implementing Haptic Tasks

This chapter presents an approach to programming haptic systems that avoids these problems. The centerpiece of the approach is a development environment called interactive Acumen, or iAcumen for short. The key features of this environment include high-level mechanisms for implementing a virtual environment by describing:

1. The physical dynamics of virtual objects. The physical description language, PhyDL, allows us to directly describe dynamic equations governing the system being modeled.

2. Events and their interaction with virtual objects. Modeling of discrete events and control algorithms uses the reactive/interrupt description language, RIDL, in an event-driven programming paradigm.

3. Coupling with external devices, including haptic devices, visual tools, and other application codes. An interface for interaction between the user and the virtual environment allows conceptually clear access to the low-level codes needed to inject these external devices into the real virtual world.

4. Real-time simulation. In iAcumen, high-level parameters are used to specify the speed of the simulation. This specification becomes part of the specification of the virtual environment.
The system is realized by a series of automatic transformations that convert the high-level descriptions into a form that is directly machine executable, and a compilation process is used for converting control algorithms written in RIDL to executable form.

In the rest of this section, I will present the infrastructure of the simulation environment iAcumen, and explain the key components of the iAcumen environment using a simple example.

6.2.1 Inserting Acumen into a Haptic Virtual Environment

A physical system with haptic feedback usually has three parts: continuous behaviors, discrete events, and a haptic device. Building such a system usually requires significant time and effort. To ease this building process, I propose iAcumen, which allows for the realization of such environments in a declarative way.

![Figure 6.1: The iAcumen environment.](image)

The iAcumen environment is an extended version of Acumen with an interactive interface for user input/output. The iAcumen environment has three components (Figure 6.1). PhyDL is used to describe continuous behaviors in the system, and RIDL is used to describe discrete events. A PhyDL program directly captures the physical equations describing the underlying system dynamics. A RIDL program
specifies system responses (control actions) to events. External modules are used for interactive input and output such as haptic feedback and object positions in the system.

6.2.2 A Spring-Mass Example

The diagram in Fig. 6.2 shows a simple example illustrating the basic features of iAcumen. A ball is attached to a spring, and the other end of the spring is attached to the origin. Without loss of generality, we assume that the original position of the spring is at the origin and that the natural length of the spring is zero. The effects of gravity on the ball are omitted. The dynamics of the system are then defined by the following equation:

\[ m \ddot{p} + k \dot{p} = (0, 0); \]  

(6.1)

Suppose the ball is initially placed at position \( p = (20, 5) \). We wish to track the number of times the ball crosses the \( y \)-axis (moving from the positive \( x \)-plane to the negative \( x \)-plane). The iAcumen program to model this system and task is shown in Fig. 6.2. On the left is PhyDL code for the virtual system’s dynamics, and on the right is RIDL code for the controller to count the number of crosses. Interactions between the system and the controller are described as part of PhyDL code in the \texttt{external ridl} section. The \texttt{external GUI} section describes interactive input and output of the system.

6.2.2.1 PhyDL Code

Beginning with the text at the bottom of the PhyDL code, the \texttt{system} section describes system equations. Mass \( m \) and spring constant \( k \) are defined as constants that represent the physical parameters of the simulated system. System dynamics are defined in the same form as equation (6.1). The line \( (x,y)=p \) defines \( x \) and \( y \) as the \( x \)-value and \( y \)-value, respectively, of the ball’s position \( p \).

The \texttt{boundary} section specifies the initial position \( p(0) \) of the ball and its initial
speed $p'(0)$. In this example, we have the ball starting at position $(20,5)$, with an initial velocity of $(0,0)$.

The **external GUI** section specifies system output to the user. The output is the position of the ball $p$ as a function of time, and the user can observe the movement of the ball on the GUI. User input can also be declared in this section, where we use the keyword **reads** instead of **writes** followed by a list of input variable names.

The **external ridl** section describes the communication between the PhyDL code and the RIDL code. The keywords **reads** and **writes** are followed by lists of variables that are read and written by RIDL, respectively. The keyword **observes** **event** is followed by an event name and the rate of the occurrence of the event. We will explain how these declarations are used in a RIDL program in next subsection.

Finally, the **simulation** section defines the starting time and ending time of the simulation, as well as the step size used by iAcumen’s numerical solver.

The new feature of iAcumen in this example is the **external GUI** section. The **external GUI** section facilitates modeling the interaction between the user and the
PhyDL program. An example of calling an external GUI is:

```plaintext
external GUI
  reads pnt1;
  writes pnt2, pnt3;
```

Variables `pnt1`, `pnt2`, and `pnt3` are three 2-D vectors. Variable `pnt1` specified in `reads` is read by PhyDL from the GUI. For example, we can use the mouse to control the position of the point `pnt1` on the GUI and pass the current position to PhyDL. Variables `pnt2` and `pnt3` specified in `writes` are written by PhyDL to the GUI, and the value of the two vectors are specified or computed in PhyDL.

### 6.2.2.2 RIDL Code

The RIDL code of Figure 6.2 defines three variables, namely `cur`, `prev`, and `crosses`. We call these variables *reactive behaviors* or simply *behaviors* because they model system behaviors that respond to *events*. Every reactive behavior has an initial value, which is the value after the keyword `init`, and a set of *event handlers* in a pair of brackets after the keyword `in`. An event handler has the form `A=>B`, where `A` is an event name and `B` is an expression used to update the behavior value when event `A` happens. Event handlers with the keyword `later` will update behavior values after event handlers without the keyword `later`. Thus, for example, when event `clock` happens, behavior `cur` and `crosses` will update their values before behavior `prev`. Temporary names can be used for a behavior if its event handlers will refer to the behavior’s own value. For example, `crosses` is referred to as `a` in its own event handler.

Event `clock` and variable `x` were defined in the *external ridl* section of the PhyDL program. Event `clock` has rate 0.1, which means it happens every 0.1 s. When event `clock` occurs, any reactive behavior that has an event handler for event `clock` will update its value. For example, `cur` gets the current value of `x` from PhyDL, whereas `prev` keeps the previous value of `x` until all other updates are completed. The
delayed update for \texttt{prev} is a result of \texttt{later} annotation. Behavior \texttt{crosses} checks whether the $x$ value is going from a positive value to non-positive. If so, its value is incremented: otherwise it keeps its previous value.

RIDL specifies system behaviors that are reactive to system events. This specification provides a natural way to describe controllers. A controller gets feedback from the system being controlled and sends out control signals only when certain events happen. This is different from continuous-time system dynamics described in PhyDL. That is why we have two language components in iAcumen.

\subsection*{6.2.2.3 External Modules}

Users interact with a virtual environment through physical devices, some of which can be haptic. Physical devices create a closed loop between the user and the virtual environment. Continuous time position and force signals are exchanged between the user and the haptic device, and discrete-time position and force signals are exchanged between the haptic device and the virtual environment.

Our environment provides an interface for interactive input and output. In the PhyDL language, we have an \texttt{external GUI} section, where one specifies variables of discrete signals whose values are passed between iAcumen and the external device. We use a pointing device for user input as positions and a visual display for 2-D animations.

\section*{6.3 Implementing Haptic Tasks in Acumen}

This section presents the complete descriptions of the codes needed to realize two haptic tasks that have been used elsewhere in the literature (Section 6.3). I only focus on the simulation of system dynamics. Haptic feedback can be sent to the user when haptic devices are plugged in.
6.3.1 A Two-ball Task

The first task we consider is a task that has been used extensively in haptics research [15]. It uses a pointing device as an input position to control a pair of balls in a virtual environment. The two balls are connected by a spring and damper in parallel, as shown in Figure 6.3. The white ball is a massless controller, and the black ball is a follower. The effect of gravity on the black ball is neglected, as the system is assumed to be acting in a horizontal plane. The user operates the pointing device to control the white ball. The output is the position of the black ball and the force feedback to the user. We display the positions of both balls on the GUI, and the appropriate force feedback is sent to a haptic device. A snapshot of the GUI with this task running in iAcumen is shown in Fig. 6.4.

The dynamics of this system are defined as follows:

\[
F_k = k * (white\_ball - black\_ball); \\
F_b = b * (white\_ball' - black\_ball'); \\
F = F_k + F_b; \\
F = m * \text{black\_ball''}; .
\]

The PhyDL code for this system is as follows:

```
boundary
black\_ball with black\_ball(0) = (0,0),
```
black_ball’(0) = (0,0);

system

F_k = k * (white_ball - black_ball);
F_b = b * (white_ball’ - black_ball’);
F = F_k + F_b;
F = m2*black_ball’’;
k = 4; b = 1; m = 1;.

In addition to the dynamics above, this task also introduces two boxes. The user’s objective is to control the black ball to cross these two boxes alternately by manipulating only the white ball and leveraging the dynamics of the two-ball system. The white ball is controlled by a pointing device, which acquires real-time input from the user. The positions of the boxes are specified in PhyDL as follows:

box1 = (350, -300); box2 = (-350, 200);

where box1 and box2 define the bottom-left points of the two boxes. The default size of the boxes is 50 for each side.

The counting of the crosses is implemented in RIDL as follows:

cur = init 0 in {clock => black_ball},
prev = init 0 in {clock => black_ball later}.

We first define variables cur and prev, each of which has an initial value of zero. When event clock happens, cur and prev update their values to the current and previous values of black_ball, respectively.

cross1 = init a=0 in {clock =>
    if (cur in box1) && !(prev in box1)
    then x+1 else x}
cross2 = init a=0 in {clock =>
    if (cur in box2) && !(prev in box2)
    then x+1 else x}.

Then variables cross1 and cross2 are defined. Behavior cross1 checks whether the black_ball has just gone into box1 from outside when event clock happens. If so, the value of cross1 is increased by one. Behavior cross2 is defined similarly.
6.3.2 An Inverted Pendulum Task

The second task we consider here has been used in control design with haptic feedback as a performance enhancement [4]. The task uses a pointing device as force input to a cart of mass $M$, on which is placed an inverted pendulum with mass $m$ lumped at the end of a rod of length $l$ (Figure 6.5). The pendulum is subjected to the effect of gravity, and can rotate freely about the pin joint connecting it to the cart. The cart is externally controlled by the pointing device in the horizontal as if there is a spring connected between the cart and the pointing device. Thus, the distance between the pointing device and the cart on the $x$-axis generates a spring force $F$. The purpose of the task is to control the cart to stabilize the position of the pendulum, so that the angle between the rod and vertical remains small. The positions of the cart and the pendulum are visualized on the GUI, and force feedback is sent to a haptic device.

The dynamics of the system are as follows:

$$F = (M + m) \times x'' + m \times l \times (a'' \times \cos(a)$$

$$-a' \times a' \times \sin(a));$$

$$0 = m \times l \times (-g \times \sin(a) + x'' \times \cos(a) + l \times a'');$$

The PhyDL code implementing the equations is as follows:

```
boundary
a with a(0) = 0.1, a'(0) = 0;

system
F = (M+m)x'' + m*l*(a''*cos(a)
```
Figure 6.5: An inverted pendulum.

\[ 0 = m*l*(-g*sin(a)+x''*cos(a)+l*a''); \]
\[ cart = (x, y); M = 10; m = 1; l = 4; \]
\[ ball = cart + rotate((0, l), a);. \]

This is the complete iAcumen specification for the dynamics of this task. The visual and haptic specification are similar to the first example.

6.4 Conclusions and Future Work

We presented a method for building haptic environments at a higher level of abstraction. The approach allows us to realize virtual environments directly from physical equations that describe the dynamics of objects in the virtual environment, augmented with very-high-level specifications of events, event occurrences, and integration with external hardware and software devices.

There are several areas in which technical advances can be made to iAcumen. For example, a static analysis that ensures that the simulation can be carried out within the specified simulation constraints would be invaluable for delivering production-quality applications that are based on iAcumen. We are also very interested in verifying the accuracy of simulations, as well as in providing tools for the automatic
analysis of some control-theoretic properties of the virtual systems described in iAcumen.

Finally, while this work emerged in the context of developing haptic feedback virtual environments, it appears that there are significant potential applications in the broader domain of virtual environments for this approach. As such, we expect that some of the most exciting future work will be in exploring the range of applications for the iAcumen system.
Chapter 7

Examples of Physical Modeling with Acumen

Acumen is a modeling environment that is designed for modeling continuous-time systems with a discrete controller. This controller could be an external event that happens to the system, or simply arbitrary input to which the system reacts. Acumen uses two languages: a Physical Description Language, PhyDL, and a Reactive Interrupt Driven Language, RIDL. PhyDL contains the physics of the system, the equations governing its motion. RIDL models the controller. Some systems can be described entirely through PhyDL.

In this chapter, we present a set of examples developed by users of Acumen from different backgrounds. We use these examples to illustrate the different aspects and usability of Acumen. Most of the examples presented in this chapter were developed by undergraduate students who were not computer scientists.

7.1 Small Examples

We have presented a tutorial in Chapter A. To create a supplemental tutorial for new users of the simulation environment Acumen, in this section, we provide a set of examples in which different systems are modeled step-by-step and the user is given practical advice about different modeling situations, such as what to remember and mistakes to avoid. All the examples in this section were developed by a first year undergraduate student at Rice, Marisa Peralta. These examples can guide the user through the modeling process from the original system diagram to the final simulation results. The systems are basic and ideal, so the general concepts within each model can be applied to more complex problems. A video demo of modeling a system in
7.1.1 A Pendulum

This system is a basic pendulum: a mass suspended without friction from a fixed point in space, with its top at the fixed point and the rest of the mass free to move in two dimensions.

The initial conditions are set for some displacement of angle (\(\theta\)) from the vertical. Acumen then treats the pendulum as though it is released at that angle at the start of the simulation. The initial conditions are set in the boundary module:

\[
\text{boundary} \\
\text{theta with theta(0)=\pi/4, theta(0)=0;}
\]

The pendulum starts at an angle of (\(\pi/4\)) with a velocity of 0.

The angle (\(\theta\)) changes with respect to time, accelerated by the force of gravity. \(\theta\) is defined by its acceleration (\(F\) refers to net external force, not force of gravity):

\[
\text{theta} = -\frac{1}{\text{inertia}} \times (mg\cdot\text{length}\cdot\sin(\theta) + F\cdot\text{length}\cdot\cos(\theta));
\]

This is written in the system module. The variables contained in this equation must also be defined in order for the program to run, so the rest of the system module contains:
\[
\begin{align*}
&\text{m} = 0.5; \\
&\text{g} = 9.8; \\
&\text{l} = 0.5; \\
&\text{inertia} = \text{m}\times\text{length}\times\text{length};.
\end{align*}
\]

The full system module, then, is as follows:

system
\[
\begin{align*}
&\text{m} = 0.5; \\
&\text{g} = 9.8; \\
&\text{length} = 0.5; \\
&\text{theta} = -\frac{1}{\text{inertia}} \times (\text{m}\times\text{g}\times\text{length}\times\sin(\text{theta}) + \text{F}\times\text{length}\times\cos(\text{theta})); \\
&\text{inertia} = \text{m}\times\text{length}\times\text{length};.
\end{align*}
\]

Variable F is written in RIDL, so the external ridl module in PhyDL calls RIDL:

external ridl
\[
\begin{align*}
&writes \text{F}; \\
&\text{observes event clock rate t}=0.0001;.
\end{align*}
\]

The RIDL code for this example is simple:

F=0 \quad (* \text{Note that a semicolon is not used to end lines in RIDL. *}).

Though the event clock is not explicitly stated in the RIDL code, it is syntactically necessary to define the rate of “clock” in the external ridl module. This sets F at the desired value (here, 0) at each step of the “clock” rate. For this example, clock rate is set at the same step size as the step size for the simulation. (The simulation module will show this.)

F being zero indicates that there is no additional or external force acting on the pendulum. Because the force of gravity is m*g, which is present in the acceleration of theta, the pendulum’s natural motion due to gravity is defined as
\(-1/\text{inertia} m g \text{length} \sin(\text{theta})\), with the term containing \(F\) being equal to zero.

Because the term containing \(F\) is equal to zero in this example, it would be possible to eliminate it, and model the pendulum without using RIDL at all. However, because the next example uses a nonzero \(F\), seeing this example in the same form, with \(F=0\), makes the next example easier to follow.

The simulation module defines how long the simulation lasts and what the calculation step size is:

```plaintext
simulation
  starting_time = 0; ending_time = 20, step_size = 0.0001;
```

The simulation will model 20 of the system’s behavior, calculating results 10,000 times per second.

**User tip:** A faster simulation can be obtained by setting the step size to a larger value, such as 0.01, but this can cause error to build up in the simulation. (In this example, if the step size is too large, the pendulum swings with a greater angle of displacement on each oscillation, appearing to gain energy when it should not.) In many systems, if the results appear to be physically incorrect in this way, the problem can often be solved by using a smaller step size.

The final module to be written is the `external matlab` module, which defines the desired results. In this system, \(\theta\) is the desired variable for which to show results. MATLAB will automatically plot these results with respect to time. It is also helpful to plot \(F\) so that at a glance, the plot will show if this system has an external force or not.

```plaintext
external matlab
  reads theta, F;
```

MATLAB will therefore produce two plots, one for each variable. The table of values corresponding to these plots is shown in Acumen’s simulation window, under
the tab “Intermediate Trace.” (The “Final States” tab shows what values the system has attained by the end of the simulation.)

### 7.1.2 A Pendulum with Interference

This system is the same pendulum as in the previous example with an additional, external force. A steady breeze blows on the pendulum in one direction, aligned with the pendulum’s plane of motion. This breeze exerts a force of 8\( N \) on the pendulum.

![Diagram of a pendulum with breeze interference](image)

**Figure 7.2**: A simple pendulum with breeze interference.

This system shall therefore be modeled in the same way as the previous example, expect that the RIDL code will now read:

\[ F = 8. \]

This gives a nonzero value to the term in the acceleration of theta, which changes the pendulum’s motion in the x-direction. (The y-direction is aligned with the pendulum, and the x-direction is the direction of motion, both changing to follow the pendulum as it moves.)

As can be seen in the results, this changes the pendulum’s motion so that it oscillates around some nonzero point, instead of oscillating around zero (which is the position of the pendulum at rest). Essentially, the arc pathway that the tip of the pendulum travels has now been shifted upwards to one side due to the breeze force.
It can be seen that the pendulum now oscillates around -0.2 instead of 0. The diagram demonstrates this motion: the pendulum’s amplitude is unchanged, simply shifted over to the negative x-direction.

7.1.3 A Bouncing Ball

This system consists of an elastic ball beginning from rest at some height above a flat surface, and being released to bounce in 1 dimension.

The initial condition of the system has the ball beginning at a height of 5, with
the surface located at 0. The position of the ball (along the vertical dimension) is given by \( p_{\text{ball}} \).

The total force that acts on the ball is composed entirely of the force of gravity while the ball is in the air. Whenever the ball is in contact with the surface, however, the surface exerts an upward contact force on the ball that is much greater than the force of gravity, thereby pushing the ball up into the air for another bounce. Therefore, when the ball is in contact with the surface, the total force is the force of gravity added to the contact force (pointing in opposite directions).

The contact force between surface and ball can be modeled as a spring force with a high coefficient. The spring displacement \( x \) is defined as the difference in position between the ball and the surface. The spring force only applies when this difference is negative, meaning the position of the ball is below the surface. Because the ball is not actually going into the surface, this refers to the ball deforming on contact with the surface. (The amount of deformation and, therefore, the amount of force the surface applies to the ball is determined by the ball’s kinetic energy at the moment of contact, which itself is determined by the ball’s height at the moment it began traveling downward.)

The force on the ball, then, can be described with the following if-then-else statement:

\[
F = \text{if } (p_{\text{ball}} - p_{\text{surface}}) < 0 \text{ then } k(x)(p_{\text{surface}} - p_{\text{ball}}) + m \cdot g \text{ else } m \cdot g;.
\]

This line of code belongs in the system module. \( F = kx \) is used as the spring force equation, with \( k \) being unique to the particular ball and surface being modeled. As explained above, \( x \) is the difference between the positions of the ball and surface. If the ball is above the surface (if the difference is positive), then the force is only \( m \cdot g \),
the gravitational force. Otherwise, both the spring force and the gravitational force act on the ball.

**User tip**: The $k$-value needs to be sufficiently high so that the ball does not travel too far below the surface. Here, $k=300$ gives physically reasonable results. (The spring force is high enough to push the ball back in the air before the ball gets too far into the surface.)

The constants in the force equations are also defined in the system module, along with the definition of the ball’s acceleration:

$$m = 2;$$
$$k = 300;$$
$$g = -9.8;$$
$$p_{ball} = F / m;$$
$$p_{surface} = 0;.$$ 

**User tip**: the velocity of the ball needs to be defined as a variable, not a derivative, in order for MATLAB to recognize it and plot it in the results:

$$P_{ball} = v_{ball};.$$ 

This completes the code for the system module. The variables to be plotted are specified in the external `matlab` module as follows:

```matlab
external matlab
    reads p_ball, v_ball, p_surface, F;.
```

Finally, the simulation module defines how long the simulation will run and what step size will be used:

```matlab
simulation
    starting_time = 0; ending_time = 30; step_size = 0.0001;.
```
7.1.4 A Bouncing Ball on An Oscillating Surface

This system can be likened to bouncing a tennis ball up and down on a racket. The surface upon which the ball bounces oscillates consistently at a certain rate. Depending on the rate of oscillation and the starting height for the ball, the ball will
hit the surface while the surface is in a different position from bounce to bounce.

It is modeled the same way as the previous example, with the following difference: the position of the surface is no longer 0, but rather oscillates up and down between -1 and 1. This new surface position, which must be written at each step, can be modeled using RIDL. The definition of $p_{\text{surface}}$ is removed from the system module, and the external ridl module is added in PhyDL to produce the following PhyDL code:

```
simulation
    starting_time=0; ending_time=30; step_size=0.0001;
external ridl
    writes $p_{\text{surface}}'$;
    observes event clock rate $t=3$;
external matlab
    reads $p_{\text{ball}}$, $v_{\text{ball}}$, $p_{\text{surface}}$, $F$;
boundary
    $p_{\text{ball}}$ with $p_{\text{ball}}(0) = 5$, $p_{\text{ball}}'(0) = 0$;
system
    $m = 2$;
    $k = 300$;
    $g = -9.8$;
    $p_{\text{ball}}'' = F / m$;
    $p_{\text{ball}} = v_{\text{ball}}$;
```

Here, PhyDL calls RIDL to write the position of the surface. The position will oscillate between $-1$ and $1$ every 3, so the event “clock” is given the rate of $t = 3$. In RIDL, this event is written to be the reversal of the surfaces velocity. The RIDL program is as follows:

```
$p_{\text{surface}}' = \text{init } x=1 \text{ in } \{ \text{clock } \Rightarrow x^{*}-1 \}$.
```

The results of this system are given in Fig. 7.6.
Figure 7.6: Ball bouncing on oscillating surface.

Here, the sharp spikes in force again occur at the points of contact, and since the velocity of the surface is not taken into account in calculating the resulting force on the ball, the contact force is approximately the same for each bounce, rather than increasing for bounces during which the surface is moving up, which would be physically true.

The velocity of the ball follows the same pattern as in the previous example, decreasing slowly (gravity only) and increasing sharply (quick change of direction at moment of contact).

The surface’s position oscillates from 0 to 3, changing instantaneously every three seconds. The ball’s position is the most interesting plot here, showing alternately high bounces and low bounces. The difference in bounce height attained depends upon the position of the surface at the moment when the ball hits. (Again, the velocity of the surface is not taken into account, for the sake of simplicity.)
7.2 Alfa Romeo Suspension Model

In this example, we would like to model the suspension of a 1989 Alfa Romeo Milano, specifically the Rice Society of Automotive Engineers custom modified Alfa\textsuperscript{1}. These custom modifications include suspension upgrades. We will model both the original suspension setup (factory) and current setup (custom) in order to ascertain the handling differences between the two. Once a four-wheel model is created for each suspension setup, we can test each of them on two different types of road conditions, street and racetrack, in order to establish the performance differences.

Also, some of the suspension components are adjustable, and it would be interesting to optimize these settings for a given road condition.

7.2.1 Background

The suspension design of this vehicle is very unique. The factory design uses 22.8\text{mm} torsion bars (torsion springs) in the front, a de Dion tube in the rear, and shock absorbers (dampers) in both the front and the back. The de Dion tube is a type of rear semi-independent live axle suspension, using coil springs in this implementation. The advantages of the de Dion tube are a reduction in unsprung weight by mounting the transmission and differential directly to the chassis rather than the suspension, and also its handling characteristics. The de Dion tube, which connects the two wheels directly to each other, keeps the rear wheels parallel to each other and perpendicular to the ground on rebound, which means that there are no camber changes. This is advantageous because it creates the maximum contact patch with the ground. There are shock absorbers (dampers) in the front and the rear suspension.

We have made a few modifications to this setup. First, we have stiffer torsion bars, 27\text{mm}. We have also replaced the stock shock absorbers with KONI Sport shock absorbers, which offer an adjustable damping rate. The Force vs. Displacement graph

\textsuperscript{1}This example was developed by Kevin Hirshberg from Department of ME at Rice University
from their Web site is in Fig. 7.7.

![Force-velocity diagram of a KONI damper.](image)

Figure 7.7: Force-velocity diagram of a KONI damper.

We have also lightened the car quite a bit, which will need to be taken into account in the models.

**The Model:**

The first step of creating this model will be to generate a one-wheel prototype carrying a quarter of the car’s weight for each of the four suspension types involved — stock front, stock rear, and custom front, custom rear. The stock front model will be approximated by a spring with the equivalent spring constant of the smaller-diameter torsion bar, and the damping ratio of the original shock absorbers. The stock rear will consist of the spring rate of the original coil springs, and the damping ratio of the original shocks. The custom front will consist of the equivalent spring rate of the larger diameter (27 mm) torsion bars and an average damping ratio value for the new shocks. Some sort of average representative value will be used because the dampers are adjustable. The custom rear will be modeled with the new spring rate and the same value for the custom shocks. Once all of these models are created, they should be tested on a sample path to ensure that they behave as predicted.
The next step will be to connect two wheels together and make a prototype half-car. With a front and a rear from each car connected, it will be possible to see how this model reacts to the given inputs.

The next step is to create a four-wheel model for each car. It is important to create a four-wheel model instead of assuming symmetry about the longitudinal axis due to some of the unique constraints in this direction. For example, the de Dion tube keeps the rear wheels parallel, so this needs be modeled. Also, the lateral stiffness of the car, as controlled by the sway bars (anti-roll bars), should be modeled.

Once the four-wheel cars have been modeled, they should be tested on various road conditions to test their handling qualities. The main aspects that this study is interested in are the differences between the responses of the two cars, and the optimization of the variable settings of the modified car.

**Alternative Model:**

Although this model progression would logically build complexity mechanically, it might not be the best approach in terms of programming complexity progression. In order to build up a working model of the system in simple programming steps, another model progression is in order. The following proposed progression also generates several working models that can be used independently in order to validate the modeling software.

- The first model in this system would consist of a four-wheel, rigid-body suspension car. This system would lack springs, dampers, and material elasticity. It would be a perfectly inelastic body.

- The next model would add vertical springs to this model, neglecting damping and any torsional flexibility.

- Next, dampers would be added to increase the accuracy of the model.

- The next step would be to add spring and damping ratios to the tires, allowing
them to flex as the vehicle traverses the courses.

- The final step would be to add all torsional effects — anti-roll bars, de Dion constraint equations, etc.

Practical Model Progression:

Since each modeling method has its own strengths and weaknesses, a compromise model will likely be implemented. By starting with quarter car models of each suspension type and then generating four-wheel models, both mechanical and programming implementation will be simplified.

Road Conditions:

The “street” conditions should be modeled by a bumpy road, most likely a sinusoid with varying amplitude. The factory suspension should be much smoother (better) on the street than the custom suspension.

The “racetrack” conditions could be modeled by a series of turns linked together, which could be used to ascertain the body roll of the car. Since reducing body roll is advantageous in racing, this will be used as an indicator of handling ability. Turns of varying radii that lead into each other will simulate the actual racetracks that we race on. If greater complexity is required, a decreasing radius turn would be useful in determining handling ability.

Equations:

\[
\sum F_{Wheel} = F_{K2} + F_{B2} - F_{K1} - F_{B1} - m_1g = m_1x''
\]

\[
K_2(x_2 - x_1) + B_2(x_2' - x_1') - K_1(x_1 - x_p) - B_1(x_1' - x_p') = m_1g + m_1x''
\]

\[
\sum F_{CarBody} = -F_{K2} - F_{B2} - m_2g = m_2x''
\]

\[
-K_2(x_2 - x_1) - B_2(x_2' - x_1') = m_2g + m_2x''.
\]
7.2.2 One-Wheel Model and Four-Wheel Model

7.2.2.1 Example — One-Wheel Model

The following Acumen file is a one-degree-of-freedom suspension test program. It models a simple suspension system traversing a sinusoidal speed bump. See code and graphs (Figure 7.9, 7.10) below.

(*1DofSpringDamper.phydl
Kevin Hirshberg
10/26/2007
Description: This model is a one-degree-of-freedom
Spring/Damper quarter-car suspension. The spring rate, k, is 500 N/m, and the damping ratio, b, is 200 N*s/m. *)

simulation
starting_time=0;
ending_time=30;
step_size=.01;

external matlab
reads x1, rh;

boundary
x1 with x1(0)=0, x1'(0)=0; (*Vehicle height IC's *)

system
m=250; (* 1/4 car mass is 250kg *)
k=500; (* spring rate is 500 N/m *)
b=200; (* damping ratio is 200 N*s/m *)

rh=if(t<pi) then .05*sin(t) else 0; (* RoadHeight=.05*sin(t) for a half period*)

x1''=1/m*(-k*(x1-rh)-b*(x1'-rh'));.

7.2.2.2 Example — Four-Wheel Model

The following is a four-wheel suspension model, with the associated graphs. Note that both of the front wheels and both of the rear wheels traverse the speed bump at the same time, a car-length apart (Figure 7.11).

(*1DofSpringDamper-4Wheel.phydl
Kevin Hirshberg
11/07/2007
Description: This model is a one-degree-of-freedom Spring/Damper full-car suspension model. There are four wheels, all of which travel over the same bump in the road. The rear wheels go over
Figure 7.9: Acumen 1DOF test graph, underdamped.

the bump a carlength after the front wheels. *)

simulation
starting_time=0;
ending_time=30;
step_size=.01;

external matlab
reads x1, x2, x3, x4, rh, rh2;

boundary
x1 with x1(0)=0, x1'(0)=0; (*Vehicle height Initial Conditions's *)
x2 with x2(0)=0, x2'(0)=0;
x3 with x3(0)=0, x3'(0)=0;
x4 with x4(0)=0, x4'(0)=0;
Figure 7.10: Acumen 1DOF test graph, overdamped.

system
m=250; (* 1/4 car mass is 250kg *)
k1=500; (* front spring rate is 500 N/m *)
b1=200; (* front damping ratio is 200 N*s/m *)
k2=400; (* rear spring rate is 400 N/m *)
b2=200; (* rear damping ratio is 200 N*s/m *)
l=2.5; (* wheelbase is 2.5m *)

rh=if(t<pi) then .05*sin(t) else 0; (* RoadHeight=.05*sin(t) for a half period*)

rh2=if(t<l) then 0 else
  if(t<pi+l) then .05*sin(t-l) else 0; (* RoadHeight=.05*sin(t) for a half period*)
x₁''=1/m*(-k₁*(x₁-rh)-b₁*(x₁'')'); (*front left*)
x₂''=1/m*(-k₁*(x₂-rh)-b₁*(x₂'')'); (*front right*)
x₃''=1/m*(-k₂*(x₃-rh₂)-b₂*(x₃'')'); (*rear left*)
x₄''=1/m*(-k₂*(x₄-rh₂)-b₂*(x₄'')'); (*rear right*).

7.3 Stability of a Pendulum

The purpose of this example is to explore stability theory in the context of a simple mechanical pendulum. This is an early step towards one of the goals of Acumen, which is to provide stability assurance for arbitrary mechanical systems. The pendulum is a good first example to look at it because it is very simple while still retaining some more complicated properties such as a nonlinear term in the equation of motion.

Figure 7.11: Four-wheel suspension model.
Hopefully, by looking at a simple example like the pendulum, we will gain insight into evaluating more complicated systems\(^2\).

To achieve this goal, we first want to find an appropriate notion of stability for a pendulum using a specific controller. Next, we want to write down the proof for the stability of this system. Finally, we want to formalize the proof in the Coq type theory.

### 7.3.1 Pendulum Systems and Their Stability

A pendulum is a mass suspended from a pivot so it can swing freely. When the mass is below the pivot point, it is called *normal pendulum* or *simple pendulum*, as shown on the left-hand side of Fig. 7.12. When the mass is above its pivot, it is called *inverted pendulum*, as shown on the right-hand side of Fig. 7.12.

![Diagram of a pendulum showing stable and unstable configurations.](image)

**Figure 7.12**: Uncontrolled pendulum: stable vs. unstable configuration.

---

\(^2\)Examples in this section are developed by Joshua Langsfeld, Jun Inoue, Marcie O’Malley, and Walid Taha.
The normal pendulum is stable when hanging downwards, and an inverted pendulum is inherently unstable, if it is uncontrolled. The characteristic equations of these two systems are as follows:

- **Characteristically Stable System:**
  \[
  I \theta'' = F_{\text{restoring}} \cdot L \\
  ML^2 \theta'' = -F_g \cdot L \cdot \sin(\theta) \\
  ML^2 \theta'' + MgL \cdot \sin(\theta) = 0,
  \]
  with linearization assumption \( ML^2 \theta'' + MgL = 0 \).

- **Characteristically Unstable System:**
  \[
  I \theta'' = F_{\text{counter,restoring}} \cdot L \\
  ML^2 \theta'' = F_g \cdot L \cdot \sin(\theta) \\
  ML^2 \theta'' - MgL \cdot \sin(\theta) = 0,
  \]
  with linearization assumption \( ML^2 \theta'' - MgL = 0 \).

For normal pendulum, a controller can be applied to accelerate the stabilization of oscillation or change the equilibrium position. Inverted pendulums must be actively balanced in order to remain upright, either by applying a torque at the pivot point or by moving the pivot point horizontally as part of a feedback system.

A basic PD controller \( F_{\text{external}} = -K_d \cdot \theta' - K_p \cdot \theta \) can be used to control a pendulum, which gives:

\[
ML^2 \theta'' + MgL \cdot \sin(\theta) = F_{\text{external}} \\
ML^2 \theta'' + K_d \cdot \theta' + MgL \cdot \sin(\theta) + K_p \cdot \theta = 0.
\]

Linear Variety: \( ML^2 \theta'' + K_d \cdot \theta' + (MgL + K_p) \cdot \theta = 0 \).

### 7.3.2 Pendulum Behavior

We look at the behavior of a pendulum when it is left uncontrolled. For the normal stable pendulum, using the following settings, the equilibrium of the system is at
\( \theta = 0 \). The trace of \( \theta \) is shown in Fig. 7.13.

\[
K_p = 0, K_d = 1, M = 1, L = 1, g = 9.8, \theta(0) = \pi/3, \theta' = 0.
\]

Figure 7.13 : Stable pendulum.

For the unstable inverted pendulum, using the following settings, the equilibrium of the system is at \( \theta = \pi \). The trace of \( \theta \) is shown in Fig. 7.14.

\[
K_p = 0, K_d = 1, M = 1, L = 1, g = 9.8, \theta(0) = 0.001, \theta' = 0.
\]

### 7.3.3 Stability

The notion of stability is a quality used to describe the trajectories of dynamical systems under small perturbations of initial conditions. In other words, a system is
said to be stable if the trajectories do not change too much under small perturbations. Various criteria have been developed to prove stability or instability of an orbit. Under favorable circumstances (e.g., linear systems), the question of stability may be reduced to a well-studied problem involving eigenvalues of matrices. A more general method involves Lyapunov functions.

For a linear system, root locus (pole placement) is used as the stability criteria. Let the characteristic equations be

\[ M L^2 s^2 + K_d s + M g L + K_p = 0. \]

Poles anywhere on the complex plane can be determined with values of \( K_p \) and \( K_d \). A stable system is one with the poles in the open left-half plane.

For nonlinear systems, Lyapunov functions are used as stability criteria. A Lyapunov function is a scalar function \( V(x) \) defined on the state space of the system, which is positive definite, continuous, and has a continuous partial derivative. The time derivative of the Lyapunov function along the trajectories of the system is negative definite, indicating that the system is stable.

Figure 7.14: Unstable pendulum.
punov function shows that the “norm” of the system returns to zero at the point where stability is demonstrated. To show this, you need a candidate function that is positive definite (always positive except at zero) and that has a derivative that is negative definite. This is equivalent to showing that solution trajectories always return to the equilibrium point and the system is stable.

7.3.4 Stability Analysis of the Pendulum Example

The energy of a non-linear pendulum system can be computed by

\[ E = M \cdot g \cdot L - M \cdot g \cdot L \cdot \cos(\theta) + \frac{1}{2} \cdot M \cdot L^2 \cdot (\theta')^2. \]

The energy of a linear pendulum system is computed by

\[ E = M \cdot g \cdot L \cdot \text{abs}(\theta) + \frac{1}{2} \cdot M \cdot L^2 \cdot (\theta')^2. \]

As can be seen in Fig.7.15 of the pendulum’s position, the system is clearly stable and tends towards the equilibrium at \( \theta = 0 \). In the context of Lyapunov stability, this can also be observed by looking at the graph of the energy in the system, which is simply the sum of the potential and kinetic energies. Assuming that the pendulum has no potential energy at the equilibrium point, then the total energy decays to zero over time. Therefore, Lyapunov stability can be proved if one can find a positive definite function that will always be greater than the energy of the system but that will decay to zero.

One function candidate can be suggested by observing the other ways that the energy of the system can be calculated. The blue line in the above graph shows the exact energy content of the system calculated using the nonlinear equation above for the potential and kinetic energies as the system itself. The black line is the energy from the linear system calculated the same way as the nonlinear system. They are very close because the only difference is the approximation of \( \sin(\theta) \) as \( \theta \) in the linear system. It is worth noting that the linear system energy is always greater than or
equal to the nonlinear system energy. Finally, the red line represents a completely linear system that is the equivalent of a mass-spring-damper. Here, the energy is much higher than before because it involves a completely different approximation (essentially replacing $1 - \cos(\theta)$ with $\theta$, which is not a very accurate approximation). However, it stills has the property of being greater than or equal to the actual energy. This happens because the approximation is always greater, with the exception being at $\theta = 0$, when they are equal. This is helpful because it places a bound on the energy of the system, which is well defined by a simple linear differential equation. It is then possible to define a function that can serve as a Lyapunov function candidate.

Looking at the graph of the linear energy content, it would seem that the peaks of each period define an exponentially decaying curve. This is, in fact what they do
define, if a time domain solution is found for the energy equation. It would seem to follow, then, that the Lyapunov candidate could be this function, of the form $Ae^{-bt}$, where $A$ and $b$ are positive constants. This function satisfies the Lyapunov restrictions of being positive-definite, and having a derivative that is negative-definite. Therefore, since it can be shown to be greater than the energy of the pendulum system for all $t$, and it collapses to zero, the nonlinear pendulum can be shown to be Lyapunov stable.

The PhyDL files for the models are as follows:

- **Nonlinear**

  simulation
  ```
  starting_time = 0; ending_time = 10; step_size = 0.001;
  external matlab
  reads theta, E;
  boundary
  theta with theta(0)= pi/3, theta’(0)= 0;
  system
  m = 1; g = 9.8; L = 1; K_d = 1;
  I = m * L * L;
  theta'' = -1/I * (K_d*theta’ + m*g*L*sin(theta));
  E = m*g*L - m*g*L*cos(theta) + .5*I*(theta’)^2;
  ```

- **Linear**

  simulation
  ```
  starting_time = 0; ending_time = 10; step_size = 0.001;
  external matlab
  reads theta, E;
  boundary
  theta with theta(0)= pi/3, theta’(0)= 0;
  system
  ```
\[ m = 1; \quad g = 9.8; \quad L = 1; \quad K_d = 1; \]
\[ I = m \cdot L \cdot L; \]
\[ \theta'' = -\frac{1}{I} \left( K_d \cdot \theta' + m \cdot g \cdot L \cdot \sin(\theta) \right); \]
\[ E = m \cdot g \cdot L \cdot \text{abs}(\theta) + 0.5 \cdot I \cdot (\theta')^2; \]

### 7.4 Summary

In this chapter, we have introduced a variety of examples developed in Acumen that are contributed by different people. These examples can help beginning users of Acumen to learn how to model physical system behaviors, and how to convert a system model to Acumen code. The analysis of those simulation results shows the connection between Acumen simulation results and physical system properties. Acumen can be used to study controller designs and their stability issues. It is easier to program in Acumen than in MATLAB/Simulink, while the simulation results are very close. All examples show that the Acumen environment is very easy to use. People with little programming background can learn to use it very quickly.

These examples also involve many interesting issues from different aspects. For example, the Alfa Romeo suspension model can be enriched by adding more details to the model and making the model more realistic. Also, further exploration in Acumen design may provide better facilities for system controller design.

There is much that can still be done to obtain a complete analysis of system stability issues. The stability analysis in Section 7.3 is only for the simple case of a stable pendulum, and so it would be helpful to expand it to include other scenarios. Also, the exact details of the Lyapunov function should be worked out to prove the stability conclusively. Finally, it will be helpful to incorporate the effects of controllers. The analyzed system has only a damper, and any thorough analysis should take into account more complicated controllers.
Chapter 8

Conclusions

This thesis presented a method for modeling and simulating physical systems. The method allows the user to rapidly prototype cyber-physical systems from high-level descriptions. Physical systems and virtual environments can be simulated directly from physical equations that describe the dynamics of the physical objects, augmented with very-high-level specifications of events, event occurrences, and integration with external hardware and software devices.

The centerpiece of the method is the carefully designed, high-level language PhyDL, which describes physical systems in a manner close to mathematical equations familiar to engineers. The key feature of PhyDL is that it provides a natural notation for mathematical modeling of a continuous environment. By incorporating the discrete-event modeling language RIDL in Acumen, users are able to model both the continuous physical environment and discrete computational elements of a cyber-physical system.

Our experience shows that Acumen is easy to understand and use by non-programmers in various examples. Acumen also demonstrated experimentally in practice acceptable numerical accuracy on a large industrial case study.

There are places where further considerations can be taken into the language design of Acumen.

1. **Formalization of translations and analysis of their properties.** Acumen makes simulation easier by program transformations, which allows the user to model a physical system from high-level descriptions. We have shown the translation steps taken in Acumen by explaining the key ideas and using
some examples. We also want to formalize these translations and analyze their properties.

2. **Extension to partial differential equations.** Currently, we allow only differentiation of a variable with respect to time. We want to explore a broader range of applications for Acumen and see whether it is necessary to allow derivatives on expressions. More generally, we also want to add partial derivatives into PhyDL syntax and semantics.

3. **Dealing with surfaces and contacts.** Collision detection and simulation of rigid bodies with contact are important topics in physical system simulation. Using our acausal modeling, it requires powerful equation-solving techniques to define contact/collision conditions and check for them. As such, we expect that some of the most exciting future work will be in exploring the range of applications with contact simulation and how to make the simulation of such systems easier.

4. **More aggressive integration with other tools, such as MATLAB and other computational engines, as well as high-level modeling formalisms.** So far, the design of Acumen has been focused on the structure of the simulation environment and exploring modeling issues and approaches. In its current implementation, Acumen is a self-contained environment in which computations are all handled within the tool. To handle some of the computation steps more efficiently (for example, the topological sorting in program translation) we can integrate Acumen with some other tools that can take care of one or more steps in the computation. One possibility is to use MATLAB for linear system modeling and analysis. Another is to use Mathematica or similar tools for equation solving.

5. **Enriching Acumen Examples.** Acumen can be used for modeling physical systems and studying controller designs. Chapter 7 presented several case
studies of complete modeling and analysis of different systems. We are very interested in expanding those examples by making the physical system models more realistic and including more scenarios in the analysis.

6. **Verifying the correctness of the results of simulation.** For the accuracy evaluation, we currently only compare the simulation result of the same physical model between Acumen and Simulink. It would be useful to have an exact reference point with which to compare all of these numerical results. Thus, for future work, we want to look at solving general ODEs and running the simulation using exact real arithmetic computation.
Bibliography


Appendix A

Acumen User Guide

Acumen is a simulation environment for continual-time physical systems containing event-driven controllers. This chapter is aimed at mechanical engineers interested in modeling and simulating such systems. The chapter familiarizes the user with the layout and functionality of the graphical user interface, and explains how to write physical descriptions in the PhyDL component of Acumen, as well as how to write reactive controllers using the RIDL component. The chapter also defines the syntax for the primitive operations provided by Acumen, and it explains how to control a simulation.

A.1 Acumen = PhyDL + RIDL

When the Acumen environment is started, the language for modeling the physical part is located in the left-hand column of the screen, which is called PhyDL (Physical Description Language). The language for modeling the controller part is located on the upper right-hand area of the screen, which is called RIDL (Reactive Interrupt Driven Language) [31, 14]. Following is a snapshot of the Acumen user interface.

A.1.1 Comments

Comments in Acumen are enclosed in (* *). The text within these starred parentheses will be ignored by Acumen and will serve solely as a comment for the user:

(* This is just a comment of the program. *)
**Important Note on Syntax:** Throughout this guide, when example syntax is given for a code, the semi-colons found at the end of some lines of code are necessary for the code to run properly.

### A.2 PhyDL

A PhyDL program is organized into several sections, namely:

- System
- Boundary
- Simulation
- Interaction with Control Description (External Section for RIDL code)
- Interaction with MATLAB (External Section for MATLAB)
- Interaction with GUI (External Section for an interactive GUI).
To model a given system in Acumen, we need to give PhyDL certain information for each section. Each section starts with a heading. This is followed by specific information about either the system or how to simulate or visualize it. For example, boundary conditions should be listed in the Boundary Section. In the rest of this section, we explain each of these sections in more detail.

A.2.1 System Section (Inputting System Equations and Constants)

A dynamical system can be described by a set of differential equations together with definitions of the variables in these equations. For example, a simple pendulum can be described by a differential equation with some boundary conditions:

\[
\text{theta}'' = -\frac{1}{I} (m*l*\sin(\text{theta}) + F*l*\cos(\text{theta}) ; \\
\text{theta}(0) = 0.1 ; \\
\text{theta}'(0) = 0 ; .
\]

The first equation above describes the dynamics of this system: that is, the second derivative of theta. The next two equations give the initial conditions of the system; in this case, the initial position and initial velocity of the pendulum (theta and the first derivative of theta.) The following is a picture of a simple pendulum.

Figure A.2 : A simple pendulum.
Differential equations and constants for a system are listed under the System Section in Acumen. First, let us explain how to code differential equations:

PhyDL allows you to write differential equations directly. The prescribed syntax of the system section is:

```plaintext
system
  <exp> = <exp>;.
```

An example of a correct System Section code is:

```plaintext
system
  theta'' = -1/I* (m*l*sin(theta) + F*l*cos(theta));.
```

The above equation appears in the form of assignment, which is, a variable appears on the left-hand side of the equation and its value is to be computed using the expression on the right-hand side. PhyDL also allows to specify constraints instead of assignments. The above code is thus equivalent to

```plaintext
system
  I*theta'' + m*l*sin(theta) + F*l*cos(theta) = 0;.
```

Now, we will cover how to code variables. We can solve the above equation for \( \theta \) if and only if all other variables are defined or constrained in the program. For example, the differential equation we have been working with has variables that represent mass \( (m) \), length \( (l) \), and a force \( F \). So in order to describe the system these variables must be defined. In Acumen, variables represent physical quantities that should be defined in the System Section \( (m \text{ and } l) \). Variables that represent quantities from an external computational element can be declared in an external section. We will explain this in Section A.2.3. The following code for the System Section is an example of how to code both variables and the differential equation of the system:

```plaintext
system
  m = 0.5; (* mass *)
  l = 0.3; (* length *)
  theta'' = -1/I* (m*l*sin(theta) + F*l*cos(theta));.
```
\[ I = m \cdot l \cdot l; \quad (\text{moment of inertia}) \]
\[ I \cdot \theta'' + m \cdot l \cdot \sin(\theta) + F \cdot l \cdot \cos(\theta) = 0; \]
\[ (\text{differential equation of system}). \]

A.2.2 Boundary Section (Inputting Boundary Conditions)

To describe a system in PhyDL, we must list boundary conditions in the Boundary Section. Keep in mind that if the highest order of derivative of a variable is \( n \), then the boundary conditions listed must include the value of this variable for each derivative from the \((n - 1)\)th derivative to the 0th derivative. The syntax for listing boundary conditions is as follows:

\[
\text{boundary} \quad <\text{var}> \text{ with } <\text{var}>(0) = <\text{number}>, <\text{var}>'(0)=<\text{number}>;.
\]

For example:

\[
\text{boundary} \quad \theta \text{ with } \theta(0) = 0.1, \theta'(0)=0;.
\]

A.2.3 External Section for RIDL Code (Interaction with Control Description)

A RIDL program described the control information of the system. A PhyDL program can interact with a RIDL program in the external ridl section. However, the External Section is not simply for RIDL functions. (We shall see soon how PhyDL can interact with an interactive GUI or MATLAB as well in the following section. The control input is taken from an external program written in RIDL. The syntax for calling an external RIDL controller is:

\[
\text{external ridl "filename.ridl"}
\]

reads \( <\text{var}>, <\text{var}>, \ldots ; \)

writes \( <\text{var}>, <\text{var}>, \ldots ; \)

observes \( \text{event } <\text{var}>, \text{ rate } t = <\text{number}>, \)
event <var> when t = <number>;;

Each var above is a string that represents a variable name in the specified RIDL program. During simulation, PhyDL needs to get control information from RIDL, while RIDL needs to “observe” some system state from PhyDL. The sections reads and writes provide a mechanism to communicate between the PhyDL program (the physical system) and a RIDL program (that system’s controller). Variables specified in reads are those whose value is “read” by a RIDL program and are generally used to calculate the new controller input. Variables specified in writes are those whose value is calculated by the RIDL program.

In the observes section, event, rate, and when are keywords that cannot be changed. Since RIDL models event-driven, discrete-time controllers, we need to specify when the controller will “read” a variable’s value at an instantaneous state of the system, determine what event handlers will be triggered depending on current event instance, compute (“write”) a new control value based on the system’s instantaneous state that was “read,” and communicate this control information back to the PhyDL program. If observes event ... rate t= n is coded, the reading of the instantaneous state of the system and computation (writing) of the control values for that particular state happens at a rate of once every n seconds. If observes event ... when t= n is coded, the observation and computation happens just once when t is equal to the specified time n. An example of syntax for the external ridl section calling a RIDL program is:

```
external ridl "simple_pendulum_control.ridl"
    reads theta;
    writes F;
    observes event clock rate t = 0.1;;
```

A.2.4 External Section for MATLAB (Plot Data Using MATLAB)

The External Section in PhyDL links the PhyDL program to an external program. One commonly used external program is MATLAB. In order to plot simulation results
of a function to produce a program visualization, you use the following syntax:

external matlab

reads  <variable>, <variable>, ... ;.

Here, we have "reads," which specifies that MATLAB will read the system data for the listed variables and plot it. These variables are the dependent variables of a system, and MATLAB plots them with respect to time. If a variable is defined as a constant in system equations, Acumen will choose not to display it and throw an error message. The plotting in MATLAB only takes place at the completion of a simulation.

A.2.5 External Section for GUI (Interaction with an Interactive GUI)

Another commonly used external program in PhyDL is an interactive GUI. Instead of plotting simulation results at the end of the simulation, one can also choose to see the objects moving on a 2-D canvas during the simulation. Based on the visual display, interactive input can also be sent to the simulated system via the GUI. The syntax to specify interactive input/output is as follows:

external GUI

reads  <variable>;
writes  <variable>, <variable>, ... ;.

Here, each <variable> must be a 2-D vector. The GUI can “read” the mouse position as the position of the variable listed after the keyword reads, display it on the GUI, and also display (writes) the positions of variables listed after the keyword writes, whose values are computed in the system Section. Since every dynamic variable is a function of time, the user can see the objects moving on the GUI if the value of the variable to be displayed changes over time.
A.2.6 PhyDL Module System

A dynamical system may have several components that subject to the same set of dynamic equations. To exploit the reusability of the definition of such components, we can use the PhyDL module system. We define the component as a module in PhyDL once and use it multiple times by referring to the the name of the module definition.

A.2.6.1 Syntax of Module Definition

A component in a physical system is an object that can be defined independently from the rest of the system, and it can be deemed as a subsystem. Thus, the definition of the dynamics of a component is also independent and is similar to the System Section code:

\[ \texttt{<exp>} = \texttt{<exp>};. \]

On the other hand, a component interacts with the rest of the system. The interaction can be specified by sharing variables, which will be used as the interfaces and declared as the “ports” of the component. A complete definition of a module in PhyDL as a component of a physical system is as follows:

\[
\text{module } \texttt{<variable>} = \{
\texttt{ports: <variable>, ..., <variable>};
\texttt{<exp>} = \texttt{<exp>};
\texttt{...}
\texttt{<exp>} = \texttt{<exp>};\}.
\]

The keyword \texttt{module} is followed by the name of the model to be defined. The curly brackets enclose the definition of the module. The keyword \texttt{ports} is followed by a list of interfacing variables of the module. Definition for the behavior of the module has the same form as System Section equations in PhyDL.

A module definition can be used by referring to the name of the module, followed by a list of variables instantiating the interfacing variables:
A PhyDL program can have multiple module definitions. Modules can also be defined hierarchically. A completely defined module can be used in the definition of another module.

### A.2.6.2 An Example Using PhyDL Module System

Suppose we have a two spring-and-ball system (Figure A.3). System dynamics can be described as:

\[
\begin{align*}
  m \cdot p_1'' + k \cdot p_1 &= (0, 0); \\
  m \cdot p_2'' + k \cdot p_2 &= (0, 0);.
\end{align*}
\]

Let one ball initially be located at \((20, 5)\), and the other ball initially be located at \((20, -5)\). We can code this system in PhyDL in two ways as follows:

![Diagram of balls on springs](image)

**Figure A.3**: Example: balls on springs.
boundary
p1 with p1(0) = (20, 5),
p1'(0) = (0,0);
p2 with p2(0) = (20,-5),
p2'(0) = (0,0);

system
m = 1; k = 10;
m*p'' + k*p = (0,0);
(x1, y1) = p1;
m*p2'' + k*p2 = (0,0);
(x2, y2) = p2;

module spring =
{ports: m, k, p, x;
 m*p'' + k*p = (0,0);
 (x, y) = p; };

boundary
p1 with p1(0) = (20, 5), p1'(0) = (0,0);
p2 with p2(0) = (20,-5), p2'(0) = (0,0);

system
m = 1; k = 10;
spring(m, k, p1, x1);
spring(m, k, p2, x2);

The code on the left models two identical springs by repeating the dynamic equations twice. The code on the right defines a module called spring. The module has variables m and k as inputs, and has variables p and x as outputs. The dynamics of this module spring is defined as equation \( m*p'' + k*p = (0,0) \). Variables x and y are defined as the first and second components of vector p. Now, to define the whole system of the two spring-and-balls we refer to the module definition by its name twice as

\[
\text{spring}(m, k, p1, x1);
\text{spring}(m, k, p2, x2);
\]

A.2.7 Simulation Section

The first section listed in PhyDL, the Simulation Section, gives PhyDL information about the simulation. With \(<\text{number}>\) representing any number, the syntax for the
commands that are used to specify the conditions of starting time, ending time and step size at which the system should be evaluated are:

```
simulation
    starting_time = <number>;
    ending_time = <number>;
    step_size = <number>;
```

Here, the unit of time is second. The value of the starting time must be less than or equal to the value of the ending time. The step size value is used for numerical computation; it is the mesh of partition to discretize the continuous time. An example of correct simulation code is:

```
simulation
    starting_time = 0;
    ending_time = 30;
    step_size = 0.001;
```

### A.3 RIDL

RIDL [31, 14] programs are coded separately from PhyDL programs and are meant to act as interchangeable controllers for a given PhyDL program. As stated before, while simulation takes place, the PhyDL program needs to acquire control information from a given RIDL program, while this RIDL programs must "observe" the system state from the PhyDL program. In order to understand how to code a RIDL program, it is best to study examples. Let us first take a look at the following simple RIDL program:

```
theta = 0.1;
F = init 0 in
    { clock => -4.9 * theta }.
```

We suppose here that we have a global event named `clock`. The frequency of this event is not determined in the RIDL program, but is specified in the PhyDL code.
that calls the RIDL program. The first line in this example program states \texttt{theta} is simply a variable with value \(0.1\). The second and third lines in this program define a variable \(F\). In this case, \(F\) is a “reactive behavior,” which means that the value of \(F\) will be updated when some event happens (in this case, the event is called \texttt{clock}). The syntax \texttt{init 0} means that the initial value of \(F\) is 0, that is, at time \(t=0\) or just after the system is reset \(F = 0\). The information within the curly brackets is the event handler, \texttt{clock \Rightarrow -4.9 * theta} states that when the event called \texttt{clock} happens, RIDL should compute \(F\), using the formula \(F=-4.9 \times \text{theta}\).

\textbf{Syntax note}: vector values are allowed in RIDL as well as in the communication between RIDL and PhyDL.

Now let’s examine a more complex RIDL example:

\begin{verbatim}
theta = init y=0.1 in
  { clock => -1 * y }.
\end{verbatim}

We now have re-defined \texttt{theta} to be a “reactive behavior” as well. The first line of code states that at time \(t = 0\), \texttt{theta} = 0.1. The variable \texttt{y} is a temporary variable name for \texttt{theta}. Temporary variables are used when the new value of a variable is dependent on the former value. The second line specifies that when the event \texttt{clock} occurs, RIDL should compute \texttt{theta} using the formula \(\texttt{theta}=-1*y\). We see here that the temporary variable \texttt{y} was needed so that a new \texttt{theta} value could be assigned based on the old one. So, this example code states that at time \(t = 0\), \texttt{theta} = 0.1. After the event \texttt{clock} occurs once, \texttt{theta} = -0.1. When the event \texttt{clock} occurs a second time, \texttt{theta} = 0.1, etc.

\section*{A.4 A Pendulum Example}

In Acumen, PhyDL and RIDL work together through the External RIDL Section in PhyDL. (Please see “External Section for RIDL Code” above.) Thus, it is possible to use a variable in RIDL whose value is only defined in PhyDL or vice versa. For
example, the following RIDL program uses $\theta$ in order to compute $F$ but only defines $\theta$ in PhyDL.

(* RIDL program *)

$$F = \text{init } 0 \text{ in}$$
$$\{ \text{clock} \Rightarrow -4.9 \times \theta \}$$

(* PhyDL program *) simulation

$$\text{starting\_time} = 0; \text{ending\_time} = 30; \text{step\_size} = 0.001;$$

external ridl "pendulum\_control.ridl"

reads $\theta$;
writes $F$;
observes event clock rate $t = 0.1$;

external matlab

reads $\theta$, $F$;

boundary

$\theta$ with $\theta(0) = 0.1$, $\theta'(0) = 0$;

system

$m = 0.5$;
$g = 9.8$;
$l = 0.3$;
$I = m \times l \times l$;

$\theta'' = - \frac{1}{I} \left( m g l \sin(\theta) + F l \cos(\theta) \right)$.

Figure A.4 shows the complete code.

### A.5 Primitive Values and Functions in PhyDL

In PhyDL, we support primitive values and functions as shown in Table A.1.
A.6 Running an Acumen Program

In the Simulation Controls section located in the middle of the right-hand portion of the screen, there are three buttons that control the running of a simulation. The first one is “Start,” the second one is “Stop,” and the third one is “Reset.” The “Start” button triggers the simulate, and resets system state before the simulation if necessary. The “Stop” button can abort a simulation if the simulation is run in multi-thread mode. One can choose to run simulation in multi-thread mode in the menu of “Settings” on the top-left menu bar. The “Reset” button resets the system settings and state to the initial conditions without running the simulation. It can be used to check whether the PhyDL program and RIDL program are valid programs.

After the simulation is done, Acumen will generate information on the final state of the system that can be found within the Simulation Controls box under the tab “Final State” as well as run this information through MATLAB (assuming the External
Table A.1: Primitive values and functions in PhyDL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>pi</td>
<td></td>
<td>pi, sin(pi), cos(pi/4)</td>
</tr>
<tr>
<td>array</td>
<td>(, )</td>
<td>(&lt;exp&gt;, &lt;exp&gt;, ...)</td>
<td>(1,2,3), (x,y), ((1,2),(3,4))</td>
</tr>
<tr>
<td>add</td>
<td>+</td>
<td>&lt;exp&gt; + &lt;exp&gt;</td>
<td>1+2, 3.5+1, x+y, (1,2)+(3,4)</td>
</tr>
<tr>
<td>subtract</td>
<td>-</td>
<td>&lt;exp&gt; - &lt;exp&gt;</td>
<td>1-2, 3.5-1, x-y, (1,2)-(3,4)</td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>&lt;exp&gt; * &lt;exp&gt;</td>
<td>1<em>2, 3.5</em>1, x<em>y, 2</em>(3,4)</td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td>&lt;exp&gt; / &lt;exp&gt;</td>
<td>1/2, 3.5/1, x/y, (1,2)/3</td>
</tr>
<tr>
<td>inner</td>
<td>inner()</td>
<td>inner(&lt;array&gt;, &lt;array&gt;)</td>
<td>inner((1,2), (3,4)), inner(x,y)</td>
</tr>
<tr>
<td>cross</td>
<td>cross()</td>
<td>cross(&lt;array&gt;, &lt;array&gt;)</td>
<td>cross((1,2), (3,4)), cross(x,y)</td>
</tr>
<tr>
<td>sin</td>
<td>sin()</td>
<td>sin(&lt;exp&gt;)</td>
<td>sin(1.1), sin(x)</td>
</tr>
<tr>
<td>cos</td>
<td>cos()</td>
<td>cos(&lt;exp&gt;)</td>
<td>cos(1.1), cos(x)</td>
</tr>
<tr>
<td>exponential</td>
<td>power()</td>
<td>power(&lt;exp&gt;, &lt;exp&gt;)</td>
<td>power(1,2), power(3.4,1),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>power(x,y)</td>
</tr>
<tr>
<td>absolute value</td>
<td>abs()</td>
<td>abs(&lt;exp&gt;)</td>
<td>abs(1.1), abs(1,2), abs(x)</td>
</tr>
<tr>
<td>array indexing</td>
<td>index()</td>
<td>index(&lt;array&gt;, &lt;exp&gt;)</td>
<td>index((1.1,2.2),1), index((3,4), 2),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>index(x,y)</td>
</tr>
<tr>
<td>unit vector</td>
<td>direction()</td>
<td>direction(&lt;array&gt;)</td>
<td>direction(1,2), direction(x,y)</td>
</tr>
<tr>
<td>vector rotation</td>
<td>rotate()</td>
<td>rotate(&lt;array&gt;, &lt;exp&gt;)</td>
<td>rotate((1.1,2.2),pi), rotate(x,y),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>rotate((3,4), -1.5)</td>
</tr>
<tr>
<td>vector project</td>
<td>proj()</td>
<td>proj(&lt;array&gt;, &lt;array&gt;)</td>
<td>proj((1.1,2.2),(1,0)),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>proj((3,4),(1,1)), proj(x,y)</td>
</tr>
</tbody>
</table>

Module for MATLAB is included in the PhyDL code) which will return the desired plot. The "Intermediate Trace" in the Simulation Controls box displays information about the system at intervals from the initial state to the final state.

As far as plotting the results of a simulation, whenever "Reset" is clicked Acumen creates a new canvas for the next plot. Then, when "Start" is clicked, Acumen updates the current canvas with the newest simulation results. In the current Acumen package,
“trace_matlab” is the file that keeps the trace after simulation. If the External Module for MATLAB is used in a code, MATLAB is triggered when we click “Start.” The file “trigger_matlab” is a communication channel that sends comments to MATLAB, such as “plot now” or “clear now,” when an action happens in Acumen. So the file “trigger_matlab” is used to monitor Acumen actions and notify MATLAB. The file “trace_matlab” is actual data to be plotted by MATLAB.

If the external GUI section is used instead of the external MATLAB section in the program, the program will run in an interactive model. An external GUI will pop up after clicking the “Start” button. Mouse position can be read by the program and 2-D vectors can be displayed dynamically on the GUI.