The Next Paradigm for Modeling and Simulation?

Integrating Scientific Machine Learning Methods in FMI-based Co-Simulation Tools

PhD Dissertation

Christian Møldrup Legaard

Department of Electrical and Computer Engineering

Aarhus University

October 2023

Abstract

Recent advances in the field of machine learning methods has allowed us to solve increasingly complex problems across a wide range of domains. In particular, the class of models collectively known as neural networks has proven to be especially effective when compared to other data-driven models. A natural question to ask is if the same formalism can be applied to gain insight into the types of systems we study in natural sciences and engineering. Scientific machine learning is a rapidly evolving field that seeks to infuse machine learning methods with knowledge of a system's physics and properties to obtain accurate and parsimonious models directly from data. By encoding knowledge in the models researchers seek to address one of the common critiques of using machine learning models over first-principles modeling, which is that they are unreliable or unpredictable. This has the potential to greatly reduce the barrier towards adopting model-based design for the development of complex systems, since models of the system's components would require much less effort to obtain. The main contribution of this thesis is surveying scientific machine learning techniques and describing how they can be integrated in existing simulation tools based on the Functional Mock-up Interface. Part of this contribution is the development of open-source tools allowing high-level scripting languages and state-of-the-art machine learning libraries to be fully leveraged within this context. The research conducted as part of this PhD project has demonstrated how such techniques can be applied, however there is still a need for more systematic benchmarks to determine their effectiveness on a wide range of systems representative of real-life applications.

Resumé

Fremskridt inden for machine learning har gjort det muligt at løse opgaver som ville være nær umulige at løse med håndskrevne algoritmer. Af datadrevne algoritmer har særligt neurale netværk har vist sig at være særdelses effektive på tværs af en bred vifte af opgaver. Et naturligt spørgsmål er om denne fremgangsmåde er velegnet til at takle typen af opgaver som er karakteristiske indenfor natur- og ingeniørvidenskaben. Scientific machine learning er et hurtigt voksende felt som forsøger at berige machine learning modeller med ekspert-viden om det fysiske systems egenskaber for at kunne aflede præcise og retvisende modeller ud fra målinger. Ved at indkorporere ekspert-viden i modellerne kan man imødekomme et hyppigt kritik punkt ved brugen af machine learning indenfor modellering, hvilket er at de leder til modeller som er upålidlige eller uforudsiglige. Anvendelsen af scientific machine learning har potentialet til at realisere den fulde værdi af modelbaseret design, da det vil blive langt nemmere at modellere de forskellige dele af systemet. Hoved bidraget af denne PhD afhandling er at gennemgå de væsenligste scientific machine learning methoder og beskrive hvordan de kan anvendes i eksisterende simulationsværktøjer som understøtter "Functional Mock-up Inteface"-standarden. Yderligere bidrager PhD projektet med open-source redskaber som giver adgang til høj-niveau scripting sprog, samt de nyeste machine learning redskaber, i denne kontekst. I afhandlingen har vi demonstreret anvendelsen af disse redskaber på en række simple systemer. Der er stadig et behov for at anvende dem på en større skala i et systematisk benchmark.

Acknowledgements

Finishing a doctoral thesis is a milestone along the path paved by countless people in my life. First and foremost I would like to thank my parents, Ulla and Torben — you have given me a curiosity for life and given me the independence I have today. Secondly, I would like to thank my supervisors: *Peter Gorm Larsen*, *Alexandros Iosifides*, *Clá́dio Gomes*, and *Mahdi Abkar*, in no particular order — I appreciate your guidance navigating the many facets of academic and personal life. Third, I would like to thank my long time collaborators and friends *Thomas Schranz* and *Ján Drgoňa* for continuous feedback and providing me the opportunity to visit and stay at the *Pacific Northwest National Laboratory* in Washington as a researcher. Finally, I would like to thank the *Poul Due Jensen Foundation* for funding my research through the project *Digital Twins for Cyber-Physical Systems*, as well as the *DIGIT Aarhus University Centre for Digitalisation, Big Data and Data Analytics* and finally the *MADE Digital* project.

Contents

Part I

Overview

Chapter 1

Introduction

"Data-driven discovery is currently revolutionizing how we model, predict, and control complex systems. The most pressing scientific and engineering problems of the modern era are not amenable to empirical models or derivations based on first-principles. Increasingly, researchers are turning to data-driven approaches for a diverse range of complex systems, such as turbulence, the brain, climate, epidemiology, finance, robotics, and autonomy."

— *Steven L. Brunton & J. Nathan Kutz [20]*

1.1 Motivation

The systems that we are designing today are growing increasingly complex. Most of the systems we design today are comprised of both software and physical components that must interact carefully to ensure the function and safety of system. Additionally, there is an increased trend towards autonomy, eliminating the possibility for a human to intervene if something goes wrong, which puts a greater responsibility on the people engineering such systems. This type of system has been referred to by many names and their design challenges have been discussed extensively: *embedded systems* [58], *Cyber-physical systems* [75, 41, 125], and *systems of systems* [90].

Models are fundamental tools for building an understanding of how a physical system behaves [25]. Not only do these models allow us to predict what the future may look like, but they also allow us to develop an understanding of what causes the observed behavior. In engineering, models are used to improve the system design [43, 114], design optimal control policy [46, 34, 37], simulate faults [97, 88], forecast future behavior [116], or

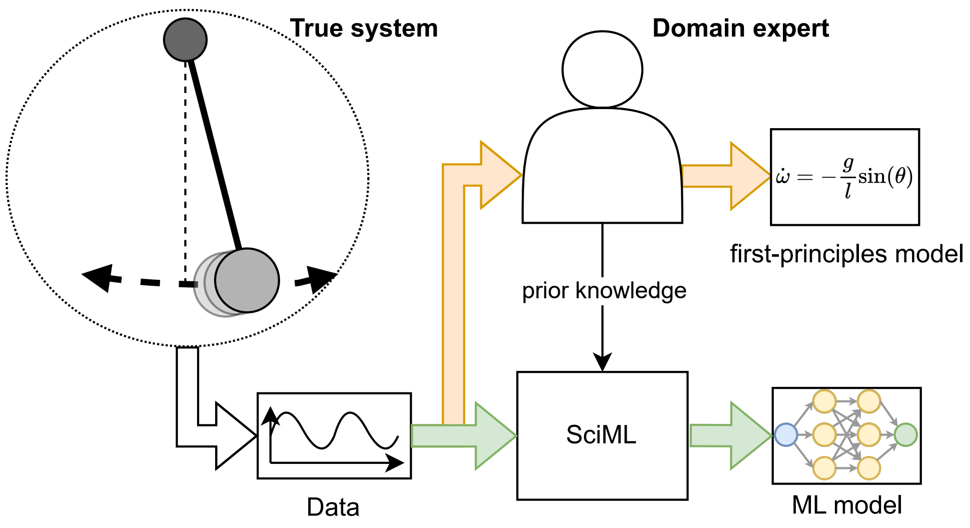


Figure 1.1: Deriving model using first-principles versus obtaining it from data using ML.

assess the desired operational performance [62]. The formalism we use for defining the model depends on which properties of the physical system we are trying to capture. There are two fundamentally different ways to model the true system as shown in fig. 1.1. One is to have a domain expert derive it from first-principles which often involves making a collection of assumptions to simplify the resulting model. The other approach is to use Machine learning (ML) as a means of obtaining a model that hopefully captures the properties of the true system that are important to us. First-principles modeling is sometimes referred to as white-box modeling because we can see and rationalize about the internals of the model [79]. The models obtained using ML are sometimes referred to as black-box models, because the claim is that we can not understand or rationalize about what is going on inside. A strong argument for using an ML based approach is that it reduces the effort/time required to obtain an accurate model of a system. Another argument is that our theoretical understanding of some physical system may be limited or our understanding of the process may be biased. In general the more complex systems we are dealing with the stronger the motivation for using ML becomes because of the effort required to construct an accurate model using first-principles.

In recent years there has been an increased interest in applying ML to solve problems in physical sciences and engineering [24]. The general idea is that a domain expert can encode prior knowledge in one way or the other

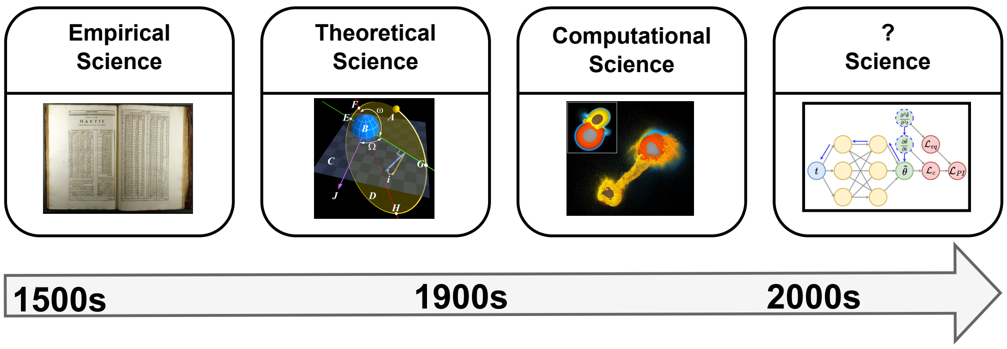


Figure 1.2: Scientific Paradigms. Inspired by [60, fig. 1]

during the ML process to create more accurate and parsimonious models. We use the term Scientific machine learning (SciML) introduced in [9, 102] to refer to the idea leveraging knowledge of the true system to create more accurate ML models. Other terms used to refer to a similar idea are *hybrid modeling* [128, 129], *physics informed ML* [64], and *theory-guided data science* [65], and *data-driven science* [21]. In the context of this thesis we define SciML as the study of methods for learning useful information based on data captured from physical systems or high fidelity simulations. The field of SciML draws from ideas from many disciplines such as applied mathematics, statistics, physics, engineering and computer science. The growing popularity of SciML should also be attributed to the recent explosion in research in the field of *deep learning* (DL) [50]. A byproduct of DL research is the development of software frameworks such as [1, 18, 95] that allows ML algorithms to process large amounts of data, often in parallel on specialized hardware.

SciML can be seen as part of a bigger change that happening to the way that researchers conduct scientific exploration. A collection of top researchers from different domains of science introduced the term *fourth paradigm* [60] in 2009 which they believe represents a new paradigm for scientific exploration. A historical timeline of the preceding paradigms for scientific exploration [13] can be seen in fig. 1.2. The empirical paradigm represents the stage where predictions was made purely based on observations without any formal theory explaining why the world around us behaved like it does. Next, the theoretical paradigm represents the stage where the observations were used to derive mathematical formulas that act as model of reality. Next, the computational paradigm leverages digital computers to simulate the models we derived, allowing us to gain insight into reality that would be difficult if not impossible obtain by experimentation alone. Finally, the proposed fourth paradigm leverages larger quantities of data combined with software tools for deriving knowledge of the system based on the data. Since the ideas was put forth in 2009 many of the predictions have shown to come true [59]. In particular, we have seen an increase in the availability of curated datasets across a wide range of tasks spanning from image classification or protein folding. The increased availability of dataset has resulted in competition between research groups in creating the most accurate models within a given task. At the time of writing, Neural Networks (NNs), the type of model which is at the core of DL, dominates the *state of the art* (SOTA) on many tasks [94]. The exact cause for the explosion in DL popularity is difficult to pinpoint to a single factor [98]. One factor is that the term NN is used to refer to an ever expanding family of mathematical structures, each of which are typically designed to excel at a specific type of task. Additionally, the software libraries for implementing and training NNs are very mature and a lot of tutorial style literature exist online on the process of doing so. We adopt the viewpoint that NNs are not universally superior models, but they are a good candidate for modeling non-linear functions as evident from their success.

A natural question to ask is what is the best way to leverage SciML in the process of engineering complex systems. This question opens up many other such as: Which types of models exist in SciML literature? What are the strengths and limitation of each method? What are best practices for training SciML models for a given type of physical system?

Another practical question is how to best integrate SciML in existing workflows and simulation environments. SciML is not here to replace first-principles but rather to extend the set of tools that are available for engineers and scientists. Being able to integrate SciML models in existing simulation tools and environments makes it possible to adopt the new formalism incrementally and to combine it with models derived by first-principles. One way of achieving this is using *co-simulation* [49], which is a methodology that makes it possible to simulate a system by composing the simulation of multiple models, each modeling part of the systems behavior. For instance, one of these models may be derived using first-principles and another model may be obtained using SciML. A practical question is what the best way to integrate the SciML workflow and the resulting models in co-simulation tools. This opens questions such as: How can we make the process of using SciML in co-simulation as easy as possible? How can we maximize compatibility of models obtained by using SciML, and existing simulation tools?

Before we formalize the research questions of this thesis, it is useful to examine the fields that share common concepts with SciML and co-simulation. Again these are quite diverse, as they originate from different disciplines such as mathematics, physics, computer science and engineering. We cover the topics in section 1.2 and discuss how the various concepts and terms relate to other disciplines. Following this section 1.3 formalize the concrete research questions that this thesis addresses. Finally, section 1.4 describes the structure of the remaining chapters of the thesis and provides links to code snippets that can be accessed online as a supplement to the thesis.

1.2 Related Disciplines

The field of SciML is characterized by an influx of ideas from many disciplines. Some of these originate from mathematics and physics, others from computer science and engineering. Many of these disciplines share the same goals and make use of similar techniques to achieve them. This section provides an outline of some of the disciplines which SciML builds on. Additionally, fig. 1.3 shows a graphical map of the various concepts, model types, algorithms and concepts that are traditionally associated with each field.

1.2.1 Numerical Methods

Numerical methods is a branch of mathematics that is concerned with designing algorithms for computing approximate solutions to continuous problems [23]. Often these methods are iterative allowing us to increase the accuracy of the approximation at the cost of computation time. A closely related discipline is *numerical analysis*, that deals with proving that certain properties hold for these methods. Numerical methods have a wealth of applications such as solving non-linear equations, approximating the values of definite integrals. The field is closely related to *numerical simulation* [132] which employs numerical methods to predict how the state of a *dynamical system* develops through time by applying *numerical integration*. Depending on the nature of the dynamical system and the required accuracy different solvers can be used. For instance, special solvers have been developed to solve stiff problems [53], systems that have discontinuities [47] or to ensure that physical properties of the simulation are preserved, such as it being conservative [85]. We cover numerical methods in more depth in chapter 2 because they are important to understanding the concepts from SciML covered in chapter 3 as those of co-simulation covered in chapter 4.

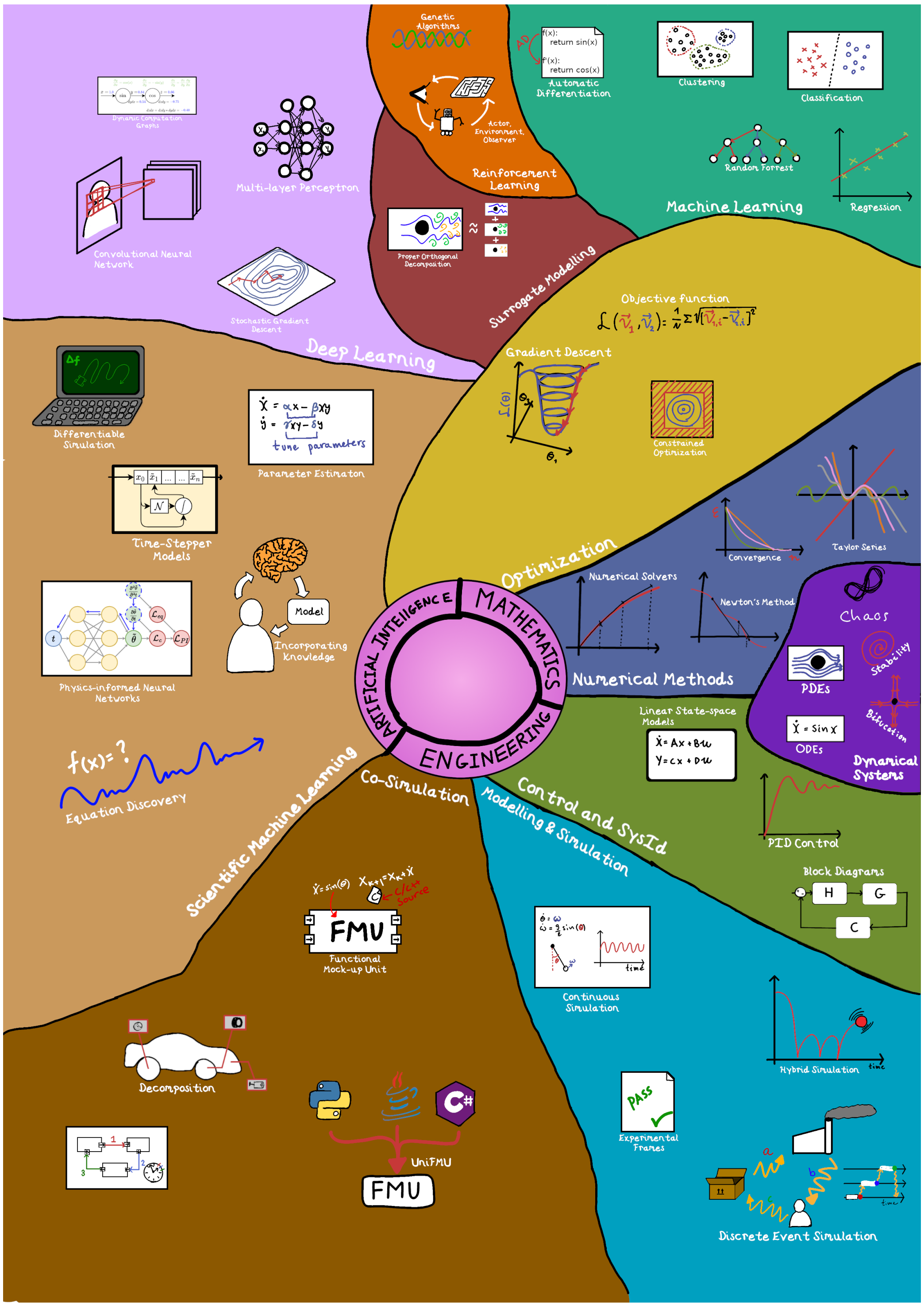


Figure 1.3: Map of topics related to SciML. Inspired by [84].

1.2.2 Dynamical Systems

Dynamical systems theory is a branch of mathematics that studies the properties of *dynamical systems* [118, 5]. A dynamical system is a mathematical construct characterized by a function mapping the systems current state to how the state will evolve within the next infinitesimal increment of time. For instance the systems state may be the position and velocity of an object, and the function is defined through a set of *ordinary differential equations*. Dynamical systems theory is related to numerical methods in the sense that we use numerical integration to obtain a simulation of our model. In literature, a dynamical system are either defined arbitrarily to exhibit certain phenomena or they may be assumed to be a model of some physical system. However, even in the latter case there is still little emphasis on the process for validating that the dynamical system actually behaves like the physical system. This should be seen in contrast to many of the engineering centric disciplines where the end goal is obtaining a model that is a close representation of a physical system. Traditionally, dynamical systems are defined by hand. However, in recent years that has been an increased interest in developing methods for deriving dynamical systems from data captured from physical systems. The authors of [21] to this endeavor as *data-driven dynamical systems*. However, the idea of fitting parameters of ODEs based on data is quite old and there is a wealth of literature using different nomenclature for referring to this. For instance, the authors of [113] refers to it as *numerical data fitting in dynamical systems*.

1.2.3 Modeling and Simulation

Modeling and simulation (M&S) is a discipline that applies a systems based approach for how to create, simulate and validate models of physical systems [133]. Historically, M&S as a field puts a large emphasis on *discrete event simulation* (DEVS) [133] and how models of continuous systems can be integrated in this formalism. The underlying idea is that models described by differential equations can be converted into a form that can be used in DEVS. In other contexts the coupling of models built using different formalism is referred to as *multi-paradigm modeling* [126]. M&S is also used outside the context of DEVS to refer to the general idea of modeling a system and then performing a simulation using the resulting model [68]. For instance, the modeling of continuous time systems is covered by [25] from an M&S perspective. Likewise, the algorithms for simulating such systems is covered by [26], again with an M&S perspective. These aspects of M&S uses differential equations and the numerical integration schemes developed in the field of numerical methods. Another important part of the M&S framework is the emphasis on the need to check the validity of a model by comparing it to the physical system for a range of conditions referred to as an *experimental frame* [133, sec. 17.1][33]. In the author's opinion, the concept of validation as defined in M&S are very similar to that of SysID as presented in [80, sec. 16.5]. There is also a body of literature which does not build on the systems-based modeling framework presented in [133], but rather focus on the process of DEVS and the software tools that support the formalism [106, 10].

1.2.4 Co-simulation

Co-simulation is a methodology that enables the modeling and simulation of a system by decomposing it into several black box models [49]. One of the things that set co-simulation apart from M&S is that models are assumed to be black boxes that can only communicate at certain time intervals, which introduces another layer of complexity in implementing a simulator. Because of the focus on simulating black box models, co-simulation as a field also deals with the practical issue of how to make models interchangeable between different simulation software [124]. We cover these concept in greater detail in chapter 4 since it is an area which we have contributed to as part of this thesis.

1.2.5 Surrogate Modelling

An important quality of a model is that it should allow us to evaluate the outcome of an experiment quickly. This is especially important if we are using it to optimize the design of our system over many trails. Model order reduction (MOR) [101] is a discipline that originates from the dynamical system and control community. As the name indicates the goal is to obtain a lower-order model that approximates a higher-order model that is more computationally expensive to evaluate. More general is the idea of surrogate modeling (SM) [69] which does not suppose any that the system we are trying to reduce is a dynamical system. In recent years, there has been an increased interest in using NNs as means of doing this dimensionality reduction [29, 86, 135].

1.2.6 Optimization

Mathematical optimization is a discipline concerned with developing algorithms for identifying an ideal choice of parameters for a given problem [28].

How good the parameters are is measured by a function referred to as an *optimization criterion* which maps a choice of parameters to a scalar value. An example of such problem is finding the dimensions that makes a mechanical part strong while being cheap to manufacture. Here the parameters would be continuous variables encoding the dimensions and the criterion would be a function that measures the price of producing the part with the given dimensions. Parameters may either be discrete or continuous resulting in different optimization problems. Each type of problem requires a different type of optimization algorithm. Parameters may also be subject to constraints on the value they may assume, which must also be considered by the optimization algorithm.

Optimization algorithms can be divided into two categories based on whether they use the gradient of the optimization criterion or approximate it using numerical differences. We refer to the first class of algorithms as being *gradient-based* and the second as *derivative-free* [105, 72]. In the context of fitting a model's parameters the first requires that we know the internal structure, whereas the second relies on probing the output of the model. There is a close connection between optimization and numerical methods, in the sense that both use iterative algorithms to successively approximate the solution. In fact concept like convergence and stability apply are well-defined concepts in each field. Optimization is a crucial part of ML and other disciplines that make use of parametric models that are fitted to data.

1.2.7 Machine Learning

Machine Learning (ML) encompasses a broad range of methods for learning useful patterns from data allowing predictions to be made based on the uncovered patterns. These methods can broadly be divided into *supervised* and *unsupervised* methods [89, 15]. Supervised learning can be seen as an optimization problem of finding the set of parameters Θ that make the function $\hat{y} = \hat{f}(x, \Theta)$ as close to the true function $y = f(x)$, given knowledge of the desired output for the input variables. Applications of supervised learning algorithms are regression and classifications problems. In the former we map the input to some output variable and in the latter we map the input to a categorical label. Unsupervised learning can be defined as the problem of learning interesting patterns in data without knowledge of the desired output. Applications of unsupervised learning algorithms are clustering and discovering important features in the data [89, sec. 1.3]. Traditionally, many ML methods originate from statistical methods, however over time the term has come to encompass any application where we are fitting the parameters of a model based on some training data [50, 24]. By the general definition disciplines such as Deep learning (DL) [50], System Identification (SysId) [80] and *reinforcement learning* [63] are all examples of ML. Supervised learning is closely connected to the concepts from optimization. What makes a particular ML algorithm is the mathematical structure we employ and the process we use for fitting the parameters of the model.

1.2.8 Deep Learning

DL refers to the idea of using NNs as the means to solve complex problems [50]. We refer to their mathematical structure, that is how their outputs are produced from their inputs as the *architecture* of the network. There are different types of NNs which are more or less suitable for different tasks. For instance convolutional NN are a popular choice for applications such as image classification or segmentation where the input is an image [52]. For modeling dynamical systems two popular approaches such as neural ordinary differential equations (NODEs) and physics-informed neural networks (PINNs) which we cover in section 3.3.3 and section 3.3.4. The topic of modeling dynamical systems using NNs is covered in detail by our survey publication 6.

1.2.9 System Identification and Control

There are other fields concerned with creating mathematical models of physical systems that predates SciML. For instance SysId, is a well established field that explicitly builds on the idea of modeling dynamical systems (DS) using parametric models that are then fitted based on data [80]. Historically, the objective of SysId is to produce models that can be studied to design control algorithms for the system. There is a natural desire to choose a type of model for which there exists simple algorithms for developing a controller. For instance, *linear state-space models* [80] are a popular choice, since concepts from linear algebra can help us analyze the stability of the closed-loop system. Closely related to SysId is the concept of model predictive control (MPC) [46] which is typically employed when constraints are put on the control signal or the state of the closed-loop system. In recent years, there has also been an increased interest in using NNs to model the dynamics of the system and as well as implementing the controller [36, 14, 3, 77].

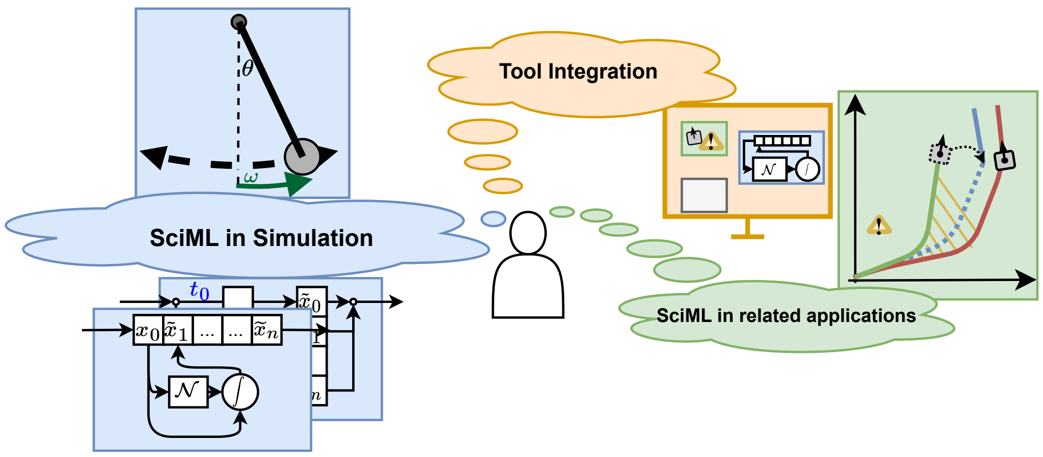


Figure 1.4: Research questions addressed in the work conducted during this PhD project.

1.3 Research Objective

The goal of this PhD project is to:

> Explore how SciML methods can be leveraged in the context of simulating complex physical systems, in order to reduce the effort required to obtain accurate models of the system's components

We have identified four challenges that we believe are central to realize this goal. Based on each challenge we formulated a *research question* that the research conducted during this thesis attempts to answer.
First, the field of SciML draws on inspiration from many disciplines such as applied mathematics, statistics, physics, engineering and computer science, which all have their own distinct nomenclatures. This has led to a situation where practitioners from different fields are referencing the same concepts and methods using different names, making it difficult to determine which methods are effective for specific applications. When compared to traditional ML, SciML addresses some concerns raised by some M&S practitioners when discussing the use of models which are not derived by first-principles. From the author's experience, a commonly voiced concern is the parameters of this type of model does not represent any meaningful quantity of the physical system, which puts the reliability of the model into question. By incorporating prior knowledge of the physical system's physics and by leveraging well-founded theory from numerical simulation, SciML methods can be used to obtain models that have a higher degree of interpretability [107, 35] than traditional ML models, making them more appealing to use in places where

reliability is weighted highly. We define the first challenge as:

SciML in Simulation: How can we make it easier to select and apply the appropriate SciML methods in the context of modeling and simulating a physical system? Additionally, how can knowledge of the physical system be integrated in the modeling process to obtain more accurate and robust models?

Another practical challenge of using SciML methods with existing simulation tools is that they are typically designed to make the simulation of a specific type of physical phenomena as user-friendly as possible, which often comes at the cost of being able to integrate your own code. Generally speaking, SciML research is conducted by implementing models in high level scripting languages using ML libraries. We define the third challenge as:

Tool Integration: How can we make the integration of SciML methods in existing simulation environments as easy as possible?

One of the main uses of SciML methods in the context of simulation is to obtain a model of a system's dynamics based on data captured from the system. However, the methods used in SciML may also be useful for other tasks when we are engineering a system, like tuning a controller or detecting faults.

SciML in related applications: What are other applications of SciML that are relevant in the engineering of systems beyond their use of obtaining models of the system's dynamics?

1.4 Reader's Guide

This dissertation is split into two parts: The chapters of part I provide a summary of the research conducted during the PhD project, as well as a detailed discussion of the extent that the research has addressed the research questions. We recommend that the chapters are read sequentially since the concepts from one chapter are built-upon in the following chapter:

- Chapter 2 describes how differential equation (DE) and numerical integration can be used to model and simulate physical systems
- Chapter 3 describes how SciML can be applied to construct a simulator for a physical system based on data
- Chapter 4 introduces co-simulation and an industry standard for exchanging models

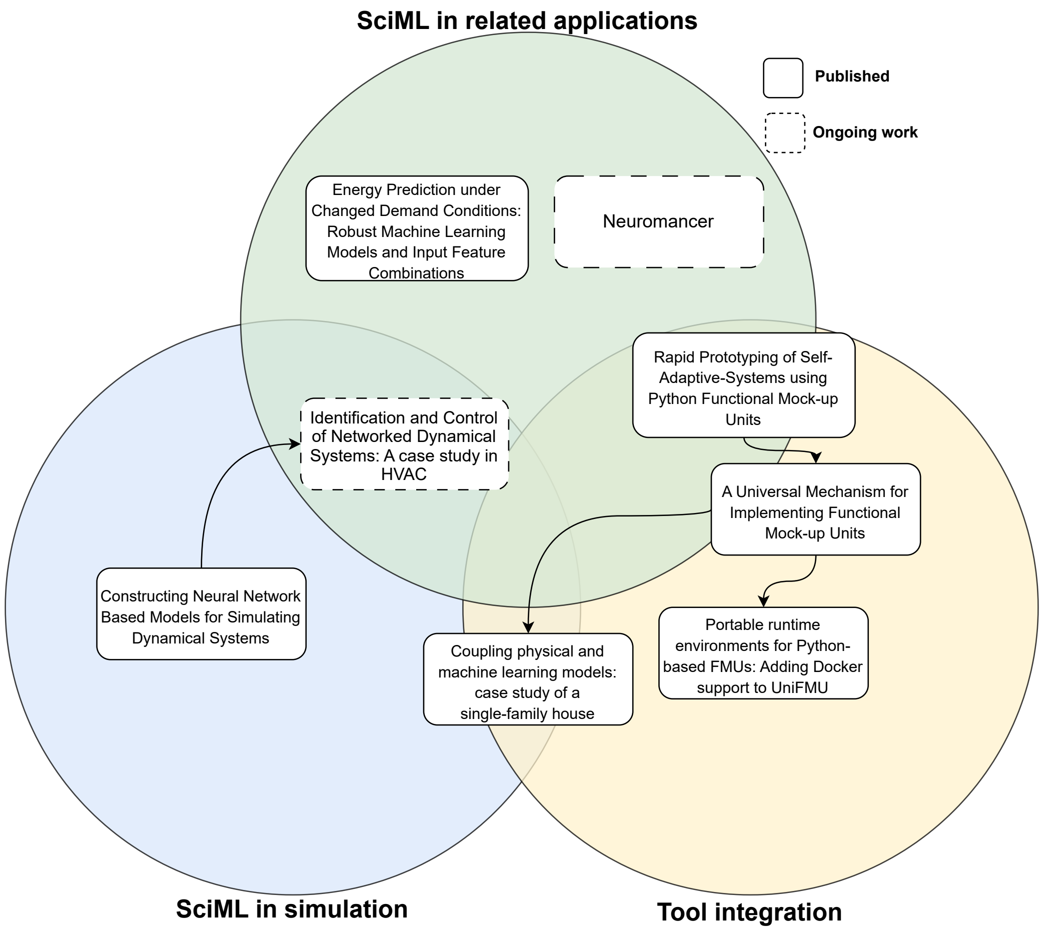

SciML in related applications
Published
Ongoing work
Energy Prediction under Changed Demand Conditions: Robust Machine Learning Models and Input Feature Combinations
Neuromancer
Rapid Prototyping of Self-Adaptive-Systems using Python Functional Mock-up Units
Identification and Control of Networked Dynamical Systems: A case study in HVAC
A Universal Mechanism for Implementing Functional Mock-up Units
Constructing Neural Network Based Models for Simulating Dynamical Systems
Coupling physical and machine learning models: case study of a single-family house
Portable runtime environments for Python-based FMUs: Adding Docker support to UniFMU
SciML in simulation
Tool integration

Figure 1.5: Grouping of publication by topic. The arrows indicate dependencies between the work, pointing from the source to a paper that builds on the idea or uses the tool. Boxes with dotted perimiters indicates work in progress papers. Publications can be found in part II.

- Chapter 5 provides an assessment of the contribution and outlines future work

Part II consists of select publications as shown in fig. 1.5. The order of the publications are sorted in a way that the author believe would be the most useful for a new practitioner in the field.

1.4.1 Contributions

To examine the extent at which the research conducted during the PhD project has addressed the research questions we list the concrete contributions throughout this thesis as shown in contribution 0.

Contribution 0: Example of what a contribution looks like in the thesis

Each contribution can be seen as part of the answer to a specific research question. A concrete example is the taxonomy of SciML methods representing contribution 1 helps address the question of how to pick the right type of SciML method as defined in section 1.3. In general at least one contribution is derived from each of the selected publications shown in fig. 1.5.

1.4.2 Supplemental Code

An important aspect of understanding the SciML is having an understanding of how it can be applied in code in practice. To aid the readers understanding code snippets like the one in listing 1.1 are placed throughout the thesis. The Python programming language [121] is chosen because it is the defacto standard for machine learning research. For brevity, we may simplify the notation and omit statements like importing libraries and calls to plotting libraries.

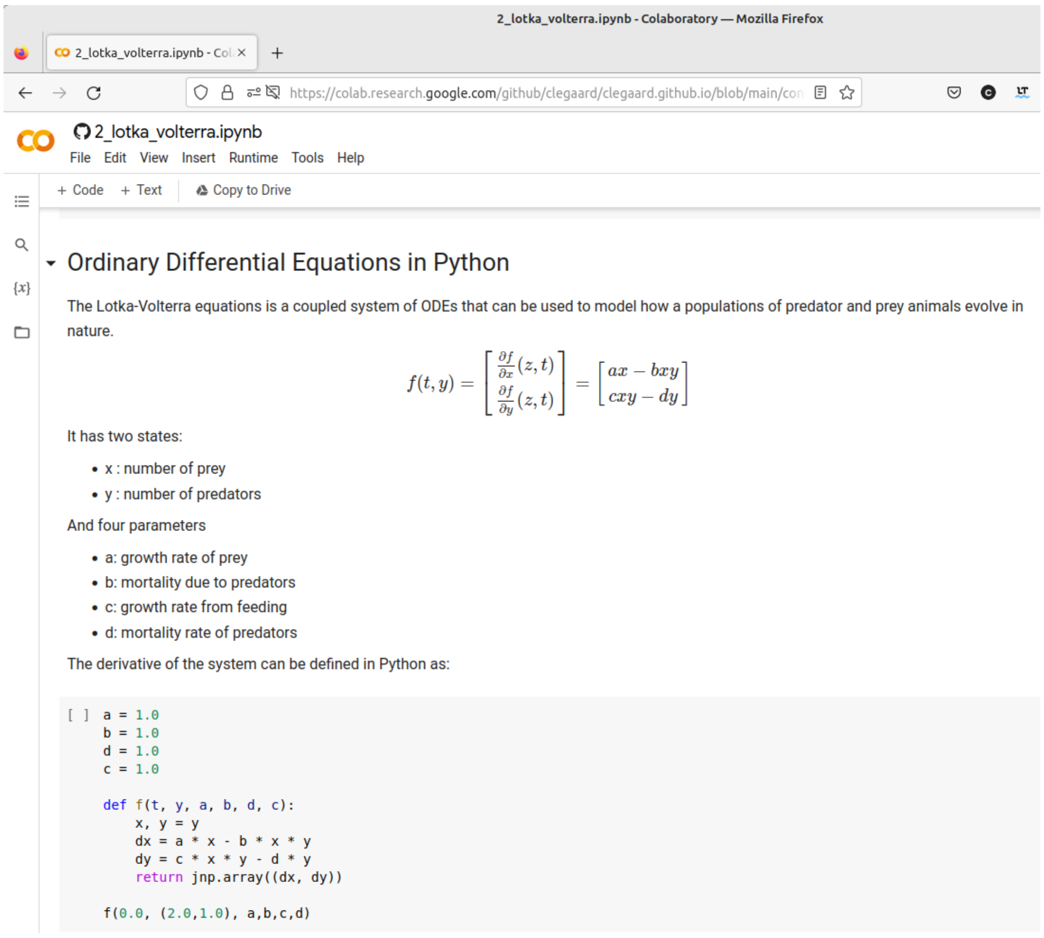
```python
import numpy as np # we omit imports like this
x = np.arange(0.0, 1.0, 0.001)
def f(x):
    return x*2
```

Listing 1.1: Example of Python code snippet.

As a supplement to the thesis document we have published a series of *Jupyter notebooks* [67], which allows the code to be executed through a browser as shown in fig. 1.6. The notebooks can be hosted locally by downloading them from the repository[1] and setting up a Python interpreter locally. Alternatively the notebooks can be accessed through *Google colaboratory* [16], through the following link[2]

[1] https://github.com/clegaard/phd_thesis_supplemental_code

[2] https://githubtocolab.com/clegaard/phd_thesis_supplemental_code/blob/main/notebooks/notebook.ipynb.


2_lotka_volterra.ipynb - Colaboratory — Mozilla Firefox
2_lotka_volterra.ipynb - Col
https://colab.research.google.com/github/clegaard/clegaard.github.io/blob/main/con
2_lotka_volterra.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Copy to Drive

Ordinary Differential Equations in Python

The Lotka-Volterra equations is a coupled system of ODEs that can be used to model how a populations of predator and prey animals evolve in nature.

$$f(t,y) = \begin{bmatrix} \frac{\partial f}{\partial x}(z,t) \\ \frac{\partial f}{\partial y}(z,t) \end{bmatrix} = \begin{bmatrix} ax - bxy \\ cxy - dy \end{bmatrix}$$

It has two states:

- x : number of prey
- y : number of predators

And four parameters

- a: growth rate of prey
- b: mortality due to predators
- c: growth rate from feeding
- d: mortality rate of predators

The derivative of the system can be defined in Python as:

```
[ ] a = 1.0
    b = 1.0
    d = 1.0
    c = 1.0

    def f(t, y, a, b, d, c):
        x, y = y
        dx = a * x - b * x * y
        dy = c * x * y - d * y
        return jnp.array((dx, dy))

    f(0.0, (2.0,1.0), a,b,c,d)
```

Figure 1.6: Notebook running in browser.

Chapter 2

Modeling and Simulation

This chapter introduces the most important concepts from numerical simulation used to simulate continuous systems, which we build upon in chapter 3 and chapter 4. The emphasis in this chapter is the simulation of non-linear Ordinary differential equations (ODEs) since they are comparatively easier to solve than partial differential equations (PDEs) and understanding their solution aids in the understanding of how the latter is solved. To aid in the understanding of how DEs can be used to model a physical system we introduce the ideal pendulum system which is used as a running example throughout the thesis. This chapter will not cover the topic of discrete-event simulation (DES) [133, 10] which is a set of methods used to model systems that evolve in response to events happening in a discrete or periodic fashion.

2.1 What is a simulation?

The concept of *simulation* can be defined as the act of imitating the behavior of a physical system by conducting experiments on a *model* instead of the physical system. For example, one may simulate how a full sized ship handles rough seas by placing a scale model in an environment that mimics the real life condition — but at the scale of the model. In this example the effort required to build the model is miniscule compared to building a full sized ship. This allows us to conduct experiments that would be considered too costly, risky or that are simply infeasible to carry out on the real system. For instance, we may simulate how the real ship would handle a breached hull by boring a hole in the scale model. Based on the outcome of the experiment we may optimize the design of the ship to be more robust under the conditions of the experiment. Not all systems can be simulated effectively by building a scale model. For instance, it would be difficult to build a scale

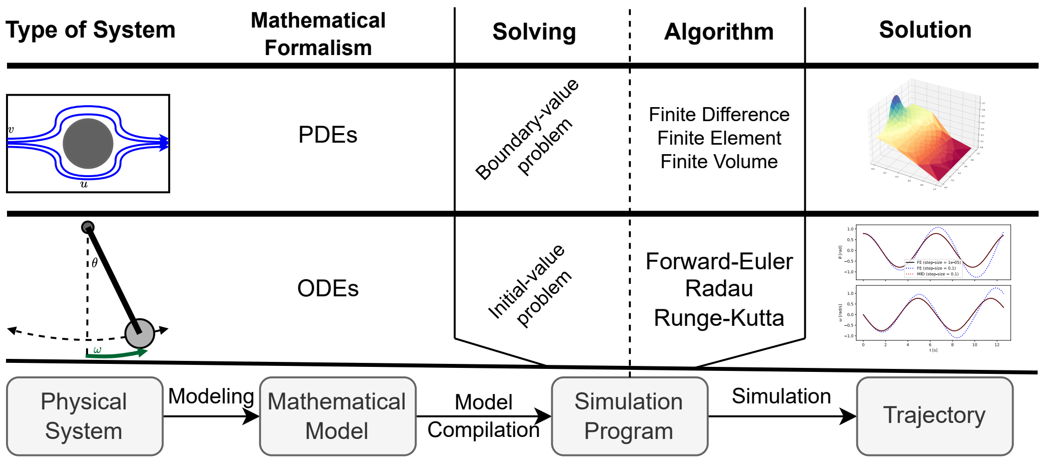


Figure 2.1: Modeling vs simulation. Inspired by [26, p.8]

model of the planets of the solar system which would be massive enough to recreate the underlying gravitational forces. Likewise, it would be impossible to create a scale model that captures the interactions between atoms or molecules since they do not manifest themselves at the macroscopic scale. This motivates the use of mathematics to capture the behavior of the physical world, sometimes referred to as *Mathematical modeling* [48], which has been used for hundreds of years. Traditionally, the computations involved in the simulation of mathematical models had to be carried out on pen and paper, which is a long and laborious process. Since then the invention of the digital computer, increases in computational power, and powerful simulation software has made it feasible to model and simulate increasingly complex systems. In the context of this thesis we focus on *computer simulation*, that is, techniques for defining models and performing simulation using software running on a digital computer.

The workflow for simulating a system consists of several stages as illustrated in fig. 2.1. First, a mathematical model is created by the extracting the relevant information from the physical system and the formalization of that information in a mathematical and unambiguous fashion [26, p.8]. The concrete mathematical formalism we apply depends on the nature of the system we are trying to model. DEs would be employed to model physical systems where the state evolves continuously over time according to the laws of physics. There are several types of DEs, most notably ODEs and PDE. ODEs are used to model systems for which the evolution is determined by the system's current state and a single independent variable — typically time. Mechanical systems like the pendulum introduced in section 2.2 are a common example of such system [7]. PDEs can be seen as a generalization of

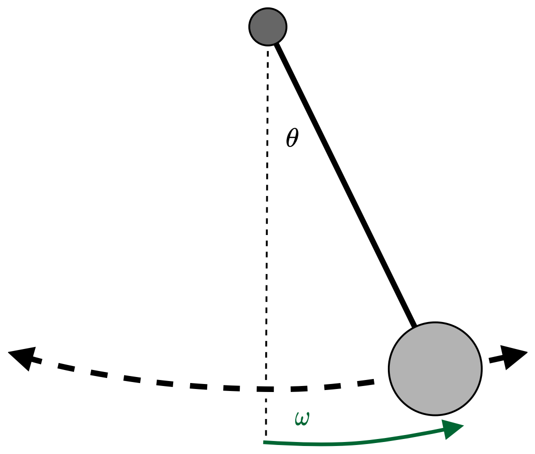


Figure 2.2: Pendulum, characterized by its angle, θ, and its angular velocity, ω. Source publication 6.

ODEs to multiple independent variables — typically time and spatial coordinates. An example of where PDEs are used extensively is in the modeling of fluid dynamics [40]. The next task is to encode this information in the form of a simulation program. The nature of the simulation program and the model are inherently linked. In some cases off the specialized software exist that can be used to import equations and parameters of the model, and in others the simulation program needs to be implemented from scratch for the particular system. For instance, simulating a system modeled by ODEs amounts to solving a initial value problem (IVP) which in practice involves numerical solvers like those described in section 2.3. We refer to the result of the simulation as the *solution* or the *trajectory* because it resembles a set of points moving through state-space in time. The choice of numerical solver determines how well this solution approximates the true solution.

2.2 Example: Pendulum

An *ideal pendulum*, shown in fig. 2.2, refers to a mathematical model of a pendulum that, unlike its physical counterpart, omits the influence of factors such as friction in the pivot or bending of the pendulum arm. The state of this system can be represented by two variables: its angle θ (expressed in radians), and its angular velocity ω. These variables correspond to a mathematical description of the system's state and are referred to as *state variables* or the state of the system. The way that a given point in the state-space evolves over time can be described using DEs. Specifically, for the ideal

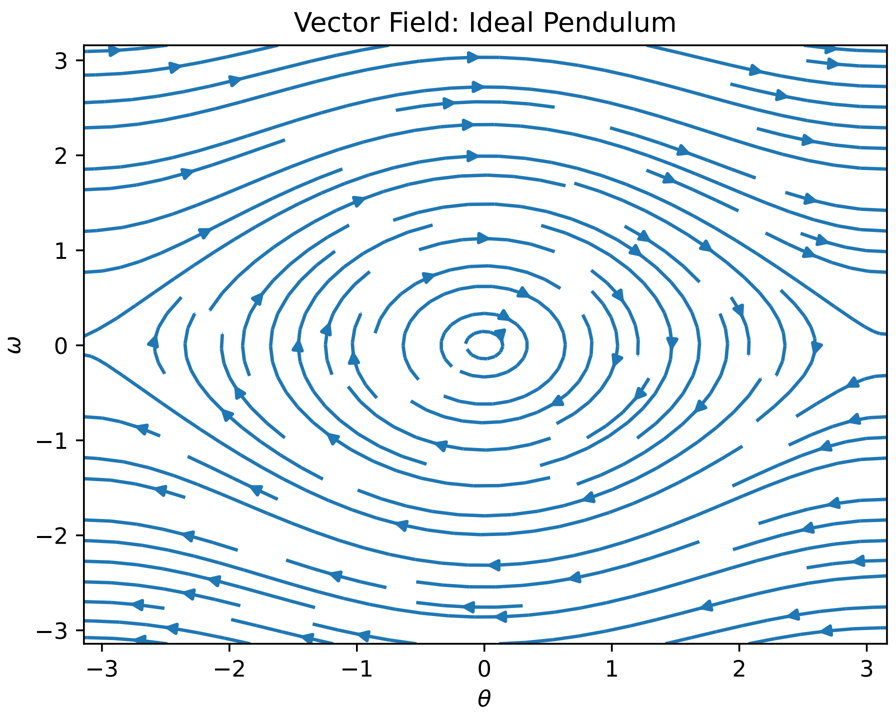


Figure 2.3: Vector field of the ideal pendulum system defined by eq. (2.2)

pendulum, we may use the following ODE:

$$\frac{\partial^2\theta}{\partial t^2} + \frac{g}{l}\sin\theta = 0, \tag{2.1}$$

where g is the gravitational acceleration, and l is the length of the pendulum arm. For simplicity, we assume that $g = l = 1$ in the rest of the thesis. The equation can be rewritten as two first-order differential equations and expressed compactly using vector notation as follows:

$$f(x) = \begin{bmatrix} \frac{\partial\omega}{\partial t} \\ \frac{\partial\theta}{\partial t} \end{bmatrix} = \begin{bmatrix} -\frac{g}{l}\sin\theta \\ \omega \end{bmatrix} \tag{2.2}$$

where x is a vector of the system's state variables. In the context of this document, we refer to $f(x)$ as the *derivative function* or as the derivative of the system. It is worth noting that the fact that $\frac{\partial\theta}{\partial t} \triangleq \omega$ happens to be value of the other state variable is a special case — and not a general characteristic of ODEs.

It is useful to think of eq. (2.2) as a function defining a *vector field* that describes how a given state will evolve over the next infinitesimal increment of time. Evaluating eq. (2.2) in multiple points we can plot the resulting 2-D vectors as arrows as shown in a stream plot as shown in fig. 2.3. Conceptually, starting at a given point and following the direction of the arrows gives us the simulation of the system. A lot about a system's dynamics can be learned by inspecting this vector field. First, we see that the arrows shrink in length going towards the origin which is equivalent to the pendulum bob moving a low speed. At a physical level this makes sense since those state correspond to a small angular velocity and an angle close to the rest position of the pendulum. Secondly, following the arrows close to the middle results in circular trajectories corresponding to the pendulum oscillating. Finally, we can observe that for initial conditions (ICs) where the magnitude of the angular velocity is sufficiently large the trajectories exit the plot either left or right. From a physical perspective this corresponds to the pendulum spinning freely. The reason that the vector field described by eq. (2.1), is due to the fact that $\sin\theta$ is a periodic function and that it is the only term where the angle appears. Conventionally we are used to thinking of angles expressed in radians as wrapping around from $-\pi$ to π and vice-versa. In cases where we are only interested in the angle relative to 0 we can think of trajectories as wrapping around from the right-hand side to the left-hand side. In the case where we are interested in how far the pendulum in the absolute sense we do not need to consider wrapping around the values.

2.3 Numerical Solvers

The goal of performing a simulation of a dynamical system is obtaining a mathematical expression of the solution of a system as a function of time and the state of the system at the start of the simulation. ODEs like eq. (2.2) describe how a system evolves at an infinitesimal increment of time, but do themselves not provide a direct way to compute the solution. The problem of simulating a dynamical system modeled by a set of ODEs defined by $f(\cdot)$ for some IC x_0, is referred to as IVP which can be formalized as:

$$\frac{\partial}{\partial t}x(t) = f(x(t)), \tag{2.3}$$

$$x(t_0) = x_0 \tag{2.4}$$

where $x(\cdot)$ is the *solution*, $x : \mathbb{R} \rightarrow \mathbb{R}^n$ and $n \in \mathbb{N}$ is the dimension of the system's state space. The IVP is satisfied by:

$$x(t) = x(t_0) + \int_{t_0}^{t} f(x(\tau))d\tau \tag{2.5}$$

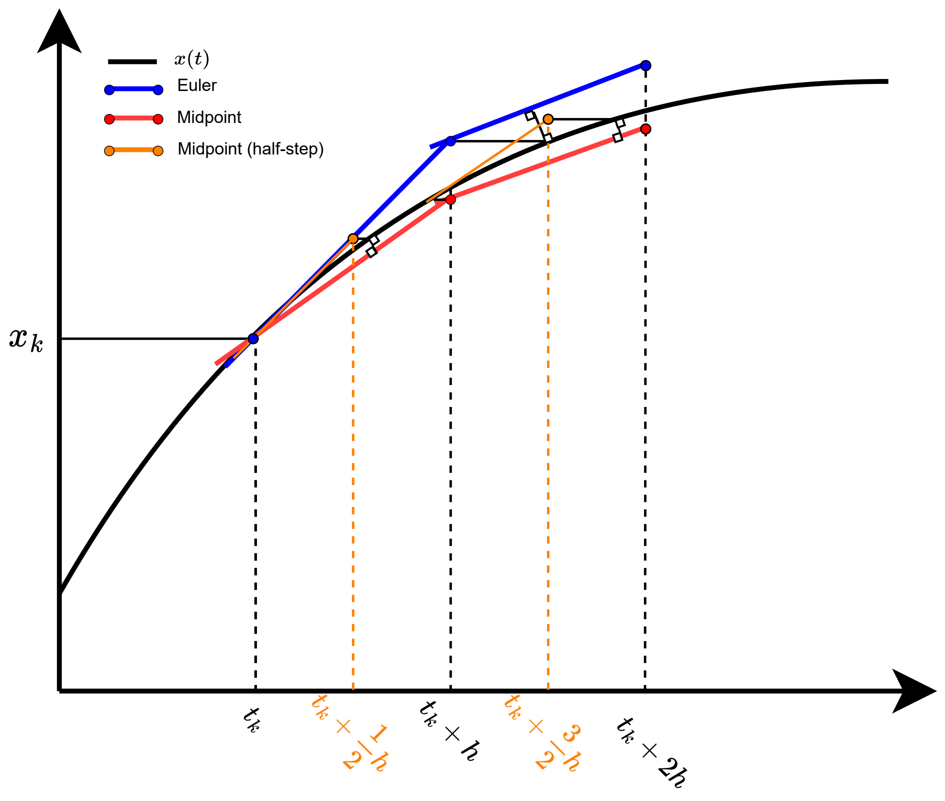


Figure 2.4: Numerical integration using forward-Euler and midpoint methods. The midpoint method has a smaller truncation error, but requires two evaluation of the systems derivative per step.

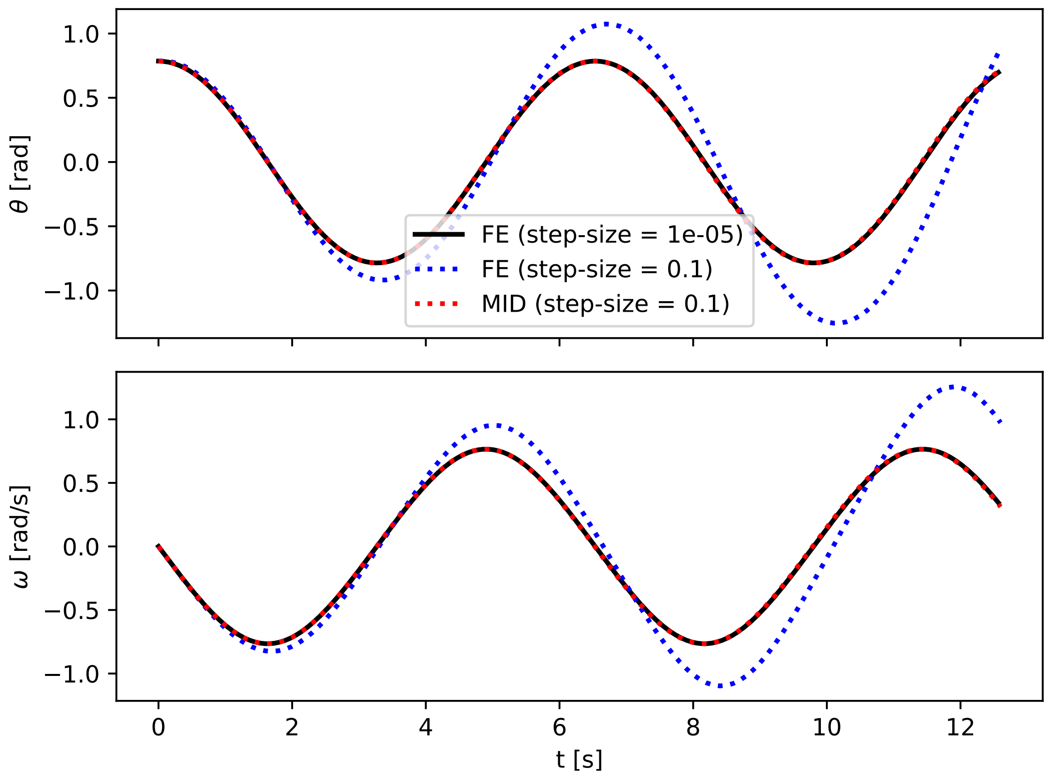


Figure 2.5: Numerical Solutions to eq. (2.3) for the set of ODEs defined by eq. (2.1). The midpoint method is clearly a closer approximation of the true solution, which we approximate accurately using forward Euler method (FE) with a step-size of $1e - 5$.

In the case of non-linear ODE, there is no general way to derive a closed-form solution for the integral in eq. (2.5), and we must resort to approximating it using numerical integration[1]. The basic idea is to approximate the integral by a set of difference equations [26]. If the dynamics of the system f and its derivatives are continuous then the solution x is as well, and can be approximated up to an arbitrary accuracy using a Taylor Series. For a step size of h, letting t be the point about which we base the approximation, and letting $t + h$ be the point which we evaluate the approximation, the Taylor Series is defined as:

$$x(t + h) = x(t) + \frac{\partial x(t)}{\partial t} h + \frac{\partial^2 x(t)}{\partial t^2} \frac{h^2}{2!} + \dots \tag{2.6}$$

Plugging in our model f we get:

$$x(t + h) = x(t) + f(x(t))h + \frac{\partial f(x(t))}{\partial t} \frac{h^2}{2!} + \dots \tag{2.7}$$

If we truncate eq. (2.7) after the linear term we arrive at the simplest integration scheme known as FE:

$$x(t + h) = x(t) + f(x(t))h \tag{2.8}$$

The basic mechanism of the FE and the closely related *midpoint method* [26] is shown in fig. 2.4. One of the limitations of the FE method is that it is only accurate for a small step-sizes. The consequence is that f must be evaluated more times which increases the time it takes to simulate the system. The midpoint method can be seen as a slightly modified version of the FE which uses an additional function evaluation to provide a more accurate estimation for a given h. The result of simulating the pendulum system defined by eq. (2.2) using both methods is shown in fig. 2.4. Despite the fact that the midpoint method requires two function evaluations per step it makes up for this by remaining accurate for much larger values of h, leading to a reduction in the time it takes to run the simulation. Many other solvers exist, many of which are covered in [100, Chap. 17].

Two causes of errors present in numerical methods are: rounding error and truncation error [100, Chap. 1]. The total numerical error can be though of as the sum of those two fig. 2.6, and it can be minimized by picking an appropriate step-size for the numerical method and the dynamics of the system. Rounding error is caused by rounding happening during arithmetic of floating-point numbers, which can not represent every number exactly. The

[1]A large collection of ODEs and their closed form solutions are presented in [99]

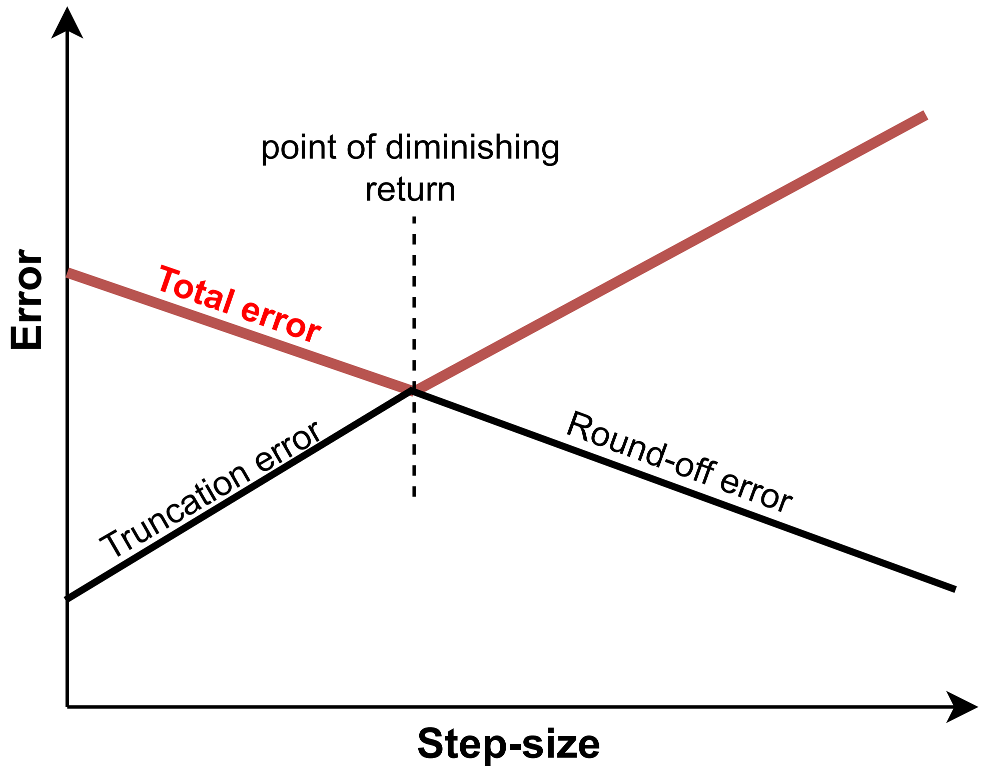


Figure 2.6: Two sources of error in numerical methods. Truncation error is introduced by the approximations used in the numerical method. Round-off error is caused by the fact that floating point numbers can not possibly represent every number in existence. Decreasing the step-size tends to reduce the truncation error at the cost of increasing the round-off error.

magnitude of the roundoff error accumulates with increasing number of calculations. Thus, increasing the step-size actually reduces the roundoff error. Truncation error refers to the error caused by using approximations rather than their continuous quantities in the calculations of numerical methods. For instance, numerical solvers like FE implicitly truncates the Taylor series defined by eq. (2.7) down to its first derivative leading to a large truncation error. Unlike rounding error, truncation error is largely controlled by the implementation of the numerical algorithm. For instance, a higher-order solver like RK45 [100] can be used instead of a first-order solver like FE. Reducing the step-size also reduces the truncation error because numerical approximations like eq. (2.7) are more accurate for small step-sizes.
A third type of error is *accumulation error* [Chap. 2][26]. As the name sug-

gests it is caused by the propagation of error from one step to the next in the solver potentially leading to error accumulating as the simulation progresses. The impact of this error scales with the length of the simulation and based on how sensitive the system is to perturbations. For instance, a chaotic and undamped system would be more sensitive to this type of error than a damped one such as the one shown in fig. 2.7.

2.4 Modeling error

In real-life a physical pendulum is going to be subject to friction in the pivot and other physical phenomena that our model does not account for. Suppose that the physical system was subject to a friction force proportional to the angular velocity and scaled according to a friction coefficient γ. Thus, the new system may be defined as

$$f(x) = \begin{bmatrix} \frac{\partial \omega}{\partial t} \\ \frac{\partial \theta}{\partial t} \end{bmatrix} = \begin{bmatrix} -\frac{g}{l}\sin\theta - \gamma\omega \\ \omega \end{bmatrix}. \tag{2.9}$$

Following the same procedure we may plot the vector field of eq. (2.9) for $\gamma = 1.0$ as shown in fig. 2.7.
Examining the plot we see that arrows are generally attracted to the origin, which corresponds to the pendulum being at rest. This is aligned with our understanding that friction should eventually bring the pendulum to rest. So far we have only considered errors introduced by the algorithms we use to simulate systems. The fact that our model is not a perfect representation of reality is another source of error, which we will refer to as *modeling error*. Modeling error can be caused by incorrect parameterization of our model, such as defining the length of the pendulum in eq. (2.1) to be an incorrect value, or it can be caused by a mismatch between the structure of the model and the physical system. A contrived example would be if we attempted to use the frictionless model defined by eq. (2.2) to a physical system that is subject to friction as defined by eq. (2.9). There are no values for the models parameters that would make the vector field shown in fig. 2.3 and fig. 2.7 equal. In a real life setting we do not have access to a mathematical description of the physical systems, if such a thing even exists. In practice, we can only hope to model the most important aspects of the physical system's behavior. So far we have examined the first-principles approach where a domain-expert derives the dynamics of the system by applying well established physical laws and by making simplifying assumptions about the factors that do and do not affect the system significantly.

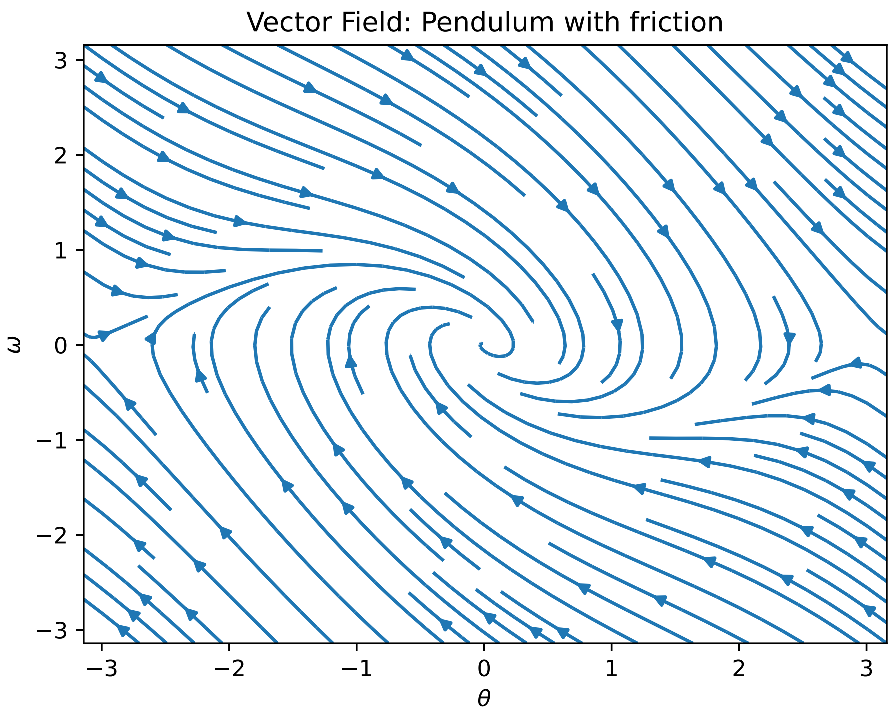


Figure 2.7: Vector field of the pendulum with friction system defined by eq. (2.9). Due to the added friction term the vector field curls towards the origin, which represents the pendulum coming to rest.

Chapter 3

Scientific Machine Learning

SciML [9] is an emerging research field concerned with how ML techniques can be applied to model physical phenomena like those arising in physics and engineering. The core idea is that incorporating domain knowledge of the problem can improve the accuracy and interpretability of the model. Domain knowledge can be expressed in different ways, such as through physical principles, constraints or symmetries. The idea of incorporating prior knowledge in ML models through various means is not unique and the nomenclature is not fully established yet. For instance, the following terms all refer to the same basic idea as SciML: *simulation intelligence* [73], *informed machine learning* [129], *physics-informed machine learning* [64], *Theory-Guided Data Science* [65]. We define SciML by a broad definition that includes all of these ideas:

> A methodology that fuses methods from machine learning and scientific computing with expert knowledge, to solve complex tasks in natural science and engineering, using a data-driven fashion.

The building blocks that make up SciML methods are depicted in fig. 3.1. At the lowest level are concepts from optimization, ML, numerical methods and modeling. The SciML methods described are all derived from this. For instance what characterizes a method like NODEs [27] is the idea of using a NN to in place of a hand-defined ODE, which can then be tuned to match the dynamics of the physical system. Similarly, an application of NODEs is to construct simulators for physical systems from data captured from them. In this chapter we focus on the specific application of constructing a simulator based on data from a physical system. The primary reason for this is that obtaining a simulator that is differentiable is the first step for other SciML applications such as designing a controller for the system. First, section 3.1 explores how gradient-descent can be used to tune the parameters

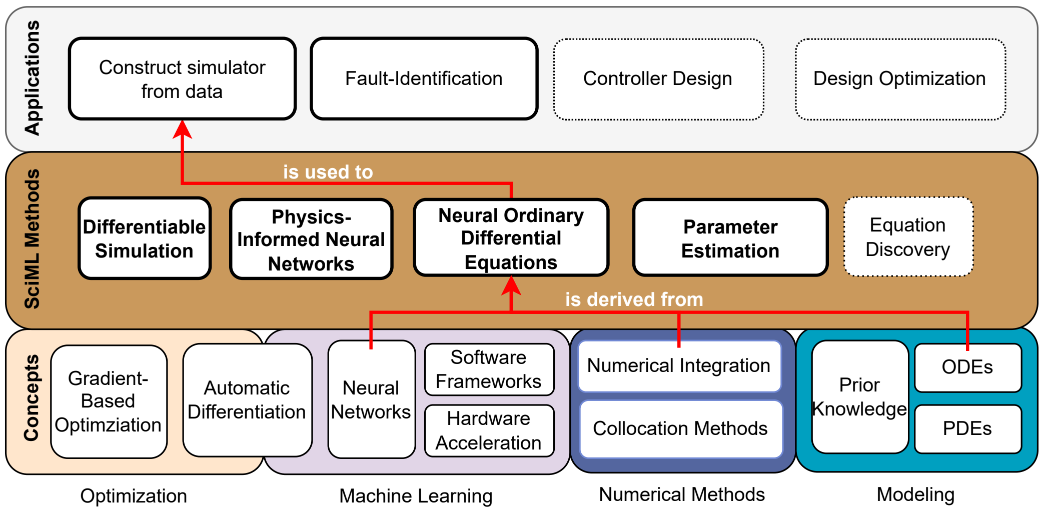


Figure 3.1: SciML methods, their applications and the fields which the methods a built upon. Applications and methods with dotted lines are only covered briefly in this chapter.

of a program. Next, we explore the concept of differentiable simulators and how Automatic differentiation (AD) is used to implement it in section 3.2. Finally, the reader is strongly encouraged to examine the online workbooks that can be accessed at `https://christianlegaard.com/ml/annsim/`.

3.1 Optimization

Modeling a physical system can be framed as an optimization problem where we are looking for the set of parameters θ that provides the closest fit to the physical system. How well the model fits measured by a loss function $\mathcal{L} : \theta \to \mathbb{R}$, which maps a choice of parameters to a scalar value indicating how good the fit is. The best way to understand this is by looking at a concrete problem. Consider the following regression 1-D regression problem:

$$f(x) = ax + b + \epsilon \tag{3.1}$$

where $\epsilon \sim \mathcal{N}(0, 1)$ and $a, b \in \mathbb{R}$. We want to minimize the error between our prediction and our model which we define as:

$$\hat{f}(x) = \hat{a}x + \hat{b} \tag{3.2}$$

A common way to define the error is mean squared error (MSE):

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N-1}\sqrt{(f(x_i) - \hat{f}(x_i))^2} \tag{3.3}$$

where θ is the parameters of the model, in this case (a, b).

A natural question to ask is how to find the values of a, b that minimize eq. (3.3).

3.1.1 Gradient-Descent

We can minimize eq. (3.3) by applying derivative-based optimizer. The gradient of a scalar-valued function ∇f is defined as:

$$\nabla f(\theta) = \begin{bmatrix} \frac{\partial f}{\partial \theta_1}(\theta) \\ \vdots \\ \frac{\partial f}{\partial \theta_n}(\theta) \end{bmatrix} \tag{3.4}$$

For instance the gradient of eq. (3.3) is a new function $\nabla\mathcal{L}(\theta)$, that defines a vector field. In the case of a 2 dimensional parameter space we can visualize this as shown in fig. 3.2. The simplest derivative-based optimizer is gradient descent:

$$\theta_{k+1} = \theta_k - \gamma\nabla_\theta\mathcal{L}(\theta_k) \tag{3.5}$$

where $\theta_k \in \Theta$ is the parameters of the model at the k-th step of the optimizer, Θ being the space of parameters, typically represented as a set of real numbers, $\gamma \in \mathbb{R}$ is the learning rate, Using eq. (3.5) with $\theta_0 = (a_0, b_0) = (0.5, 1.0)$ we arrive at a close fit as shown in fig. 3.3. Examining fig. 3.4 we see that the loss decreases monotonically until around 180 iterations where it stabilizes.

3.1.2 Closed-form Solution

Performing the linear regression using eq. (3.3) as optimization criterion has a closed-form solution. We can augment the observation variable x and the collect the slope and intercept a, b in a vector Θ such the eq. (3.3) can be expressed using dot product:

$$\begin{aligned} \mathcal{L}(\theta) &= \sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\theta_i x_j^i = ||X\theta - Y||^2 \\ &= (X\theta - Y)^T(X\theta - Y) \\ &= Y^TY - Y^TX\theta - \theta^TX^TY + \theta^TX^TX\theta^T \end{aligned} \tag{3.6}$$

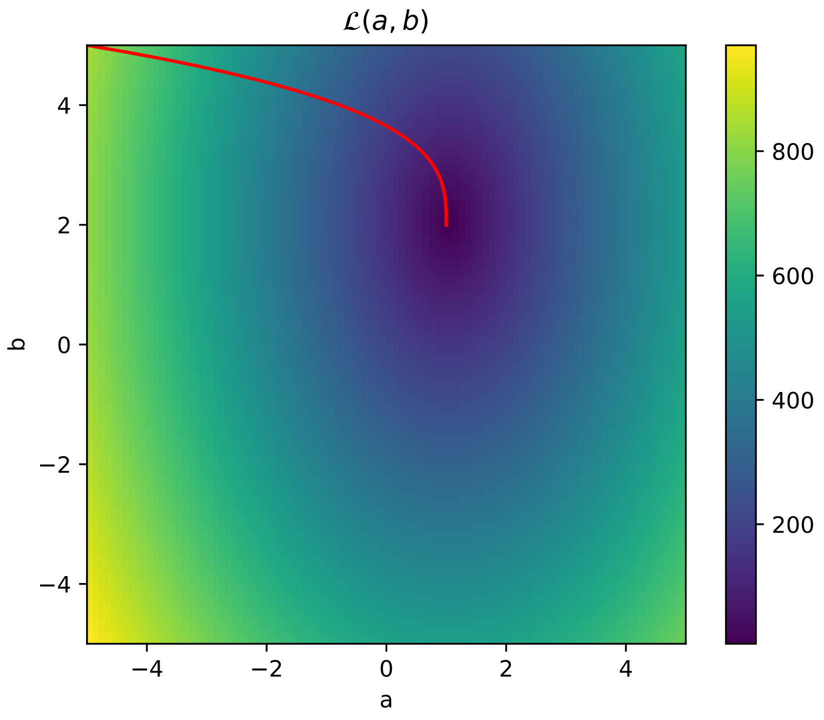


Figure 3.2: Loss landscape obtained by evaluating eq. (3.3) for different combinations of the models parameters a and b. The red line indicates the path of the optimizer starting from the guess at the start of the training to the parameter values at the end of the training.

This is a convex function, which means that the optimum solution is found where the gradient is zero:

$$\nabla_\theta = -2X^TY + X^TX\theta$$

Setting the gradient to zero we get:

$$\begin{aligned} -2X^T\theta + 2X^TX\theta = 0 \\ &\Rightarrow X^TX\theta = X^TY \\ &\Rightarrow \theta = (X^TX)^{-1}X^TY \end{aligned}$$

The cost of this is doing a matrix inversion of X^TX. For the very simple example of linear regression we were able to derive a closed-form solution to optimize eq. (3.3). Had there been any non-linear terms in eq. (3.6) this would not have been possible.

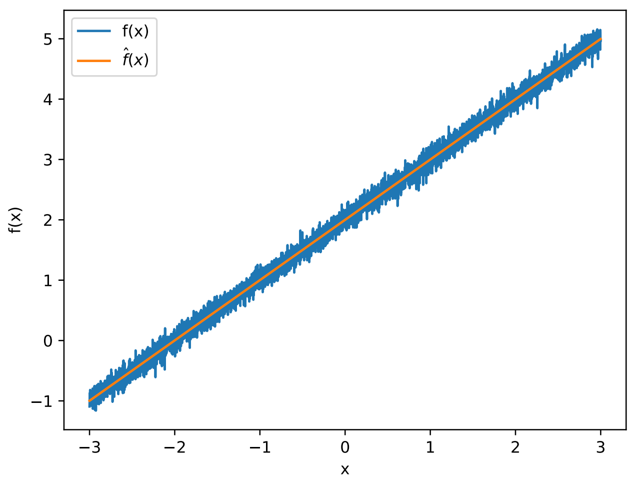


Figure 3.3: Data sampled from eq. (3.1) for $x \in [-\pi, \pi]$, $a = 1, b = 2$

3.1.3 Automatic Differentiation

Using AD allow us to implement a differentiable simulator by writing the code that performs the evaluation of the gradients and numerical integration as discussed in chapter 2. In other words AD makes it easier to optimize the parameters of models that relies on numerical integration. Without the ability to differentiate a simulation, the only alternative to fitting a models parameters are *gradient-free* optimization algorithms like those typically employed in reinforcement learning [63]. The motivation for using gradient-based methods is that they may lead to faster convergence [119]
Much confusion arises from the idea of differentiating computer programs and from the fact that multiple techniques exist for doing so. For ease of understanding we limit ourselves to the task of differentiating the simple univariate function defined by eq. (3.7).

$$f(x) = \cos(\sin x) \tag{3.7}$$

We examine the connection between differentiating a function like eq. (3.7) and the process of training a models parameters a model in section 3.1. For the curious reader, we refer to the survey [12] which provides a detailed description and comparison of the techniques.

Numerical differentiation is a finite difference approximation of a functions gradient obtained by evaluating the function in a set of neighboring sample points in addition to the original point. The simplest scheme is a

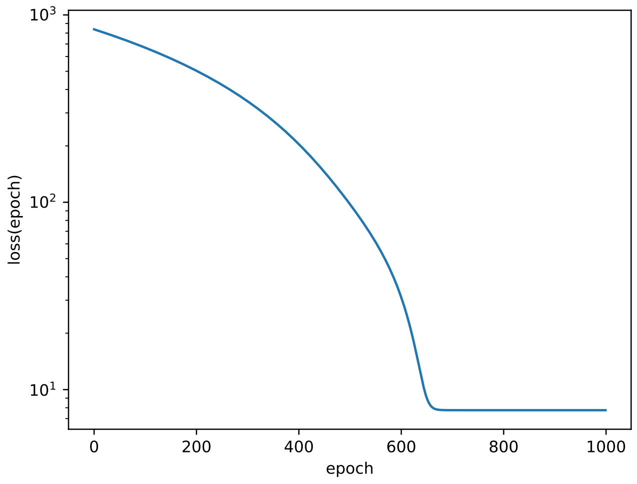


Figure 3.4

forward difference - inspired by the definition of the limit. For a multivariate function $f : \mathbb{R}^N \to \mathbb{R}$, the gradient is approximated by

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}$$

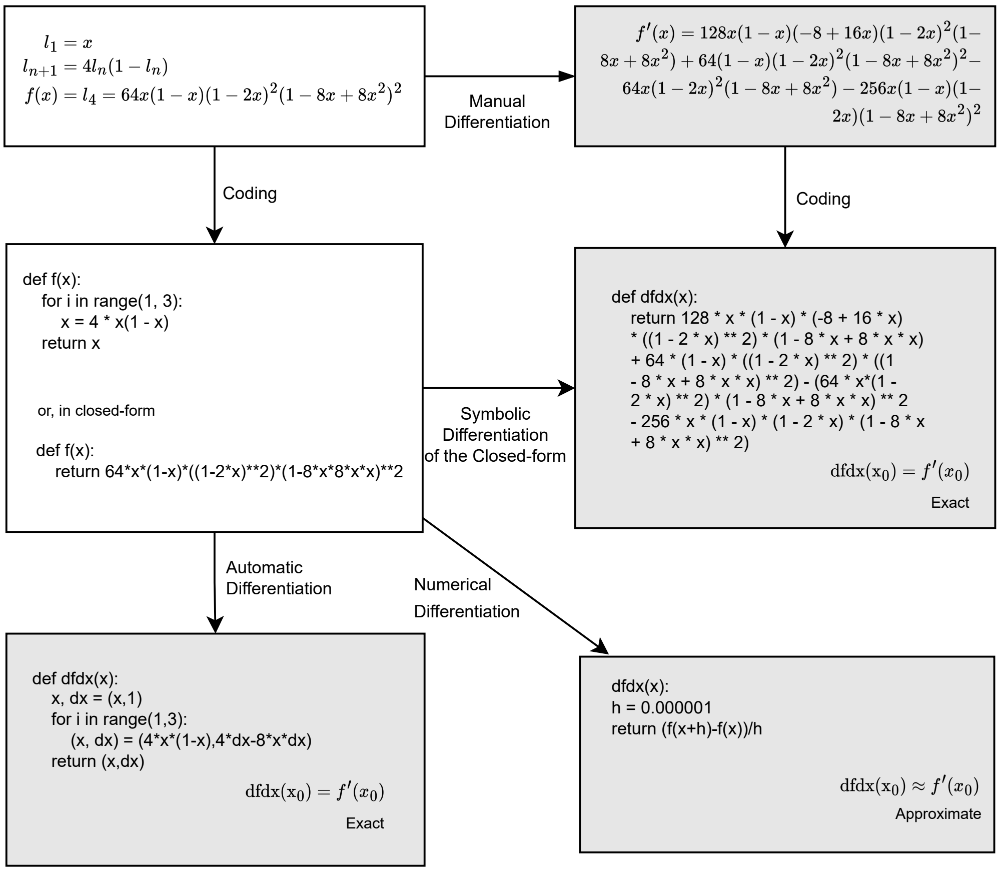
where e_i is the i-th unit vector and $h > 0$ is a sufficiently small step size. The benefit of this approach is that it is simple to implement. For a univariate function like eq. (3.7) it can be implemented as shown in listing 3.1.

```python
h = 0.001
x = 1.0
z0 = cos(sin(x))
z1 = cos(sin(x + h))
dzdx = (z1 - z0) / h
print(dzdx)  # -0.40264575946546977
```

Listing 3.1: Numerical differentiation

An obvious disadvantage of this approach is that it requires an additional function evaluation for each variable that we wish to evaluate the partial derivative for. This is especially problematic if the derivatives are used as a means to train models with a large number of parameters which are typical of NNs. A less obvious disadvantage is that the approximation are ill-conditioned and unstable for most functions [12].

Symbolic differentiation is an approach where symbolic expressions are manipulated to obtain symbolic expressions of the derivatives. The process


$$l_1 = x$$
$$l_{n+1} = 4l_n(1-l_n)$$
$$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

$$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)-256x(1-x)(1-2x)(1-8x+8x^2)^2$$

```
def f(x):
    for i in range(1, 3):
        x = 4 * x(1 - x)
    return x
```

or, in closed-form

```
def f(x):
    return 64*x*(1-x)*((1-2*x)**2)*(1-8*x*8*x*x)**2
```

```
def dfdx(x):
    return 128 * x * (1 - x) * (-8 + 16 * x)
    * ((1 - 2 * x) ** 2) * (1 - 8 * x + 8 * x * x)
    + 64 * (1 - x) * ((1 - 2 * x) ** 2) * ((1
    - 8 * x + 8 * x * x) ** 2) - (64 * x*(1 -
    2 * x) ** 2) * (1 - 8 * x + 8 * x * x) ** 2
    - 256 * x * (1 - x) * (1 - 2 * x) * (1 - 8 * x
    + 8 * x * x) ** 2)
```

$\mathrm{dfdx}(\mathrm{x}_0) = f'(x_0)$

Exact

```
def dfdx(x):
    x, dx = (x,1)
    for i in range(1,3):
        (x, dx) = (4*x*(1-x),4*dx-8*x*dx)
    return (x,dx)
```

$\mathrm{dfdx}(\mathrm{x}_0) = f'(x_0)$

Exact

```
dfdx(x):
h = 0.000001
return (f(x+h)-f(x))/h
```

$\mathrm{dfdx}(\mathrm{x}_0) \approx f'(x_0)$

Approximate

Manual Differentiation
Coding
Coding
Symbolic Differentiation of the Closed-form
Automatic Differentiation
Numerical Differentiation

Figure 3.5: Different approaches for differentiating functions. Reproduction based on [12, fig. 2]

is similar to that of differentiating a function by hand by applying known rules and transformations, the only difference being that we use software to automate the process. In most programming languages symbolic differentiation is implemented as external libraries defining symbolic counterpart of mathematical operators. Using *SymPy* [87] we can differentiate the eq. (3.7) as shown in listing 3.2.

```python
x = symbols("x")
y = sin(x)
z = cos(y)
dzdx = diff(z, x)
print(dzdx)  # -sin(sin(x))*cos(x)
print(dzdx.subs(x, 1.0))  # -0.402862443052853
```

Listing 3.2: Symbolic differentiation in SymPy

A benefit of symbolic differentiation is that they can sometimes valuable insight into the problem such as analytical solutions to problems. The drawbacks of symbolic differentiation as a means to evaluate derivatives is that the expression often explodes in size and that they may take a long time to evaluate.

AD [127, 12] refers to a set of techniques used obtain derivatives of programs by altering its execution. There are multiple ways of tapping into the execution, each with their own set of pros and cons. Dynamic programming languages like Python provide a great deal of flexibility when it comes to implementing AD through external libraries.

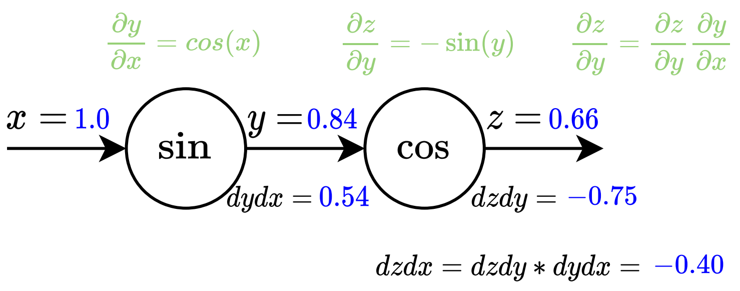

$\frac{\partial y}{\partial x} = cos(x)$
$\frac{\partial z}{\partial y} = -\sin(y)$
$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$
$x = 1.0$
sin
$y = 0.84$
cos
$z = 0.66$
$dydx = 0.54$
$dzdy = -0.75$
$dzdx = dzdy * dydx = -0.40$

Figure 3.6: Dynamic computation graph corresponding to eq. (3.7).

The premise idea of AD is that a program can be differentiated by breaking it down to a sequence of elementary operations, for which the derivative is known. Using the *chain rule* of differentiation the derivative of the composition of two differentiable function functions can be determined. Frameworks such as PyTorch [95], Jax [18] and TensorFlow [1] dynamically builds up a trace of the program during execution. For instance, running the program shown in listing 3.3 results in the computation graph shown in fig. 3.6. They do so by defining a special array type, referred to as a *tensor* in PyTorch, that records the operations applied to it. For instance evaluating the sine of x creates a new node in the graph y which is a child of x. Similarly, evaluating the cosine of y creates yet another node z which is a child of y. Calling the *backward* method on z computes the partial derivative of z and populates the *grad* attribute of all *leaf nodes*. For instance, accessing $x.grad$ gives us the partial derivative of z with respect to x. In the context of ML leaf nodes are either inputs to or parameters of the model and the tensor for which we would be differentiating would be the output of a loss function. There are two modes of AD forward mode and reverse mode, also referred to as *back-propagation*[12]. Both provide us with the ability to compute partial derivatives, but depending on the number of leaf nodes one may be faster to evaluate [12, sec. 4.1].

```python
# imperative
x = torch.ones([], requires_grad=True)
y = torch.sin(x)
z = torch.cos(y)
z.backward()
print(x.grad)  # tensor(-0.4029)

# functional
def h(x):
    return torch.cos(torch.sin(x))

print(grad(h)(x))  # tensor(-0.4029)
```

Listing 3.3: Differentiation in PyTorch using imperative and functional API.

3.2 Differentiable Simulation

Differentiable Programming (DP) is a programming paradigm that leverages AD to obtain derivatives of computer programs. This is extremely useful for data-driven modeling since it provides the derivatives necessary for optimization of a model's parameters, without having derive these by hand. More specific to the topic of modeling and simulation is the concept of *differentiable simulators* or *differentiable physics* [30, 31, 61]. The idea is that a simulator can be implemented in a way that we can train parameters of a model or those of a controller using gradient-based optimization as depicted in fig. 3.7.

In some cases we might have prior knowledge of a system's dynamics in the sense that we already know the structure of the equations, yet we are

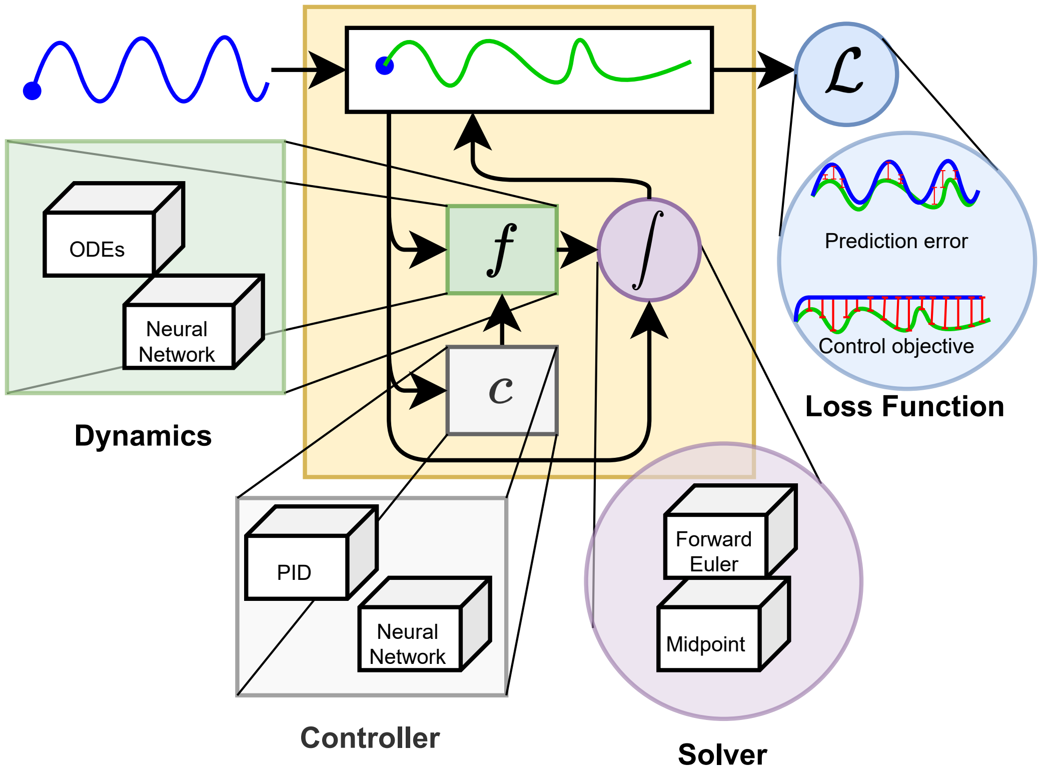


Figure 3.7: Differentiable Simulator. The dynamics of the system is modeled by f, which could be a set of hand-derived ODEs or a NN. A numerical solver simulates the system by evaluating f and integrating it with the current state. The loss function maps the result of the simulation to a scalar value. The choice of loss function and which parameters we differentiate with respect to determines the outcome of the optimization. For instance, differentiating the prediction error with respect to the parameters of f can be thought of as asking: "How can I tweak the parameters to make the simulator and physical system as close as possible?".

missing concrete values for parameters. For instance consider the parameters of the Pendulum system introduced in section 2.2. The dynamics of the system are defined by eq. (2.2) which has two parameters g and l which model the influence of the gravity and the pendulum arms length on the system. Suppose that we knew the structure but only had rough estimates of g and l which we denote $\hat{g}$ and $\hat{l}$. One way to identify them is to treat this as an optimization problem of finding a set of parameters that minimize the difference between the true system and our model. We may define the loss to be the MSE between the predicted trajectory $\hat{x} \in \mathbb{R}^{M \times N}$ where M is the dimension of the state-space and N is the number of steps in the trajectory:

$$\mathcal{L}(\theta) = \frac{1}{MN} \sum_{i=0}^{M} \sum_{j=0}^{N} \sqrt{(x_{ij} - \hat{x}_{ij})^2} \tag{3.8}$$

What makes this scenario special in the context of dynamical systems is that the prediction at a given time x_k depends on the prediction at the previous time x_{k-1} and so forth. An implication of this is that the trajectory of produced by the model is uniquely determined by its parameters Θ and the IC x_0. The exact expression depends on which integration scheme we employ. For instance, using eq. (2.8) we may write up the first three steps of trajectory:

$$\begin{aligned}
&x_1 = x_0 + hf(x_0) \\
&x_2 = x_1 + hf(x_1) = x_0 + hf(x_0) + hf(x_0 + hf(x_0)) \\
&x_3 = x_2 + hf(x_2) = x_0 + hf(x_0) + hf(x_0 + hf(x_0)) \\
&\quad + hf(x_0 + hf(x_0) + hf(x_0 + hf(x_0)))
\end{aligned}$$

During training the number of steps we simulate for is tied to the length of the trajectories from the true system. Suppose, the trajectories from the true system span an interval of 10 seconds and are sampled every second. In this case we would need to simulate the model for a similar time interval and make sure that the time instances where they are sampled are aligned. Based on personal experience, the longer the trajectories used for training are, the more difficult it becomes to train the model. A possible explanation is that the error committed at one step due to an incorrect choice of parameters are propagated to the next step and so forth. In the case where the true system is synthetic, i.e. we defined it ourselves, it is easy to control the length of the trajectories and which initial conditions to simulate for. If the data is captured from a physical system we can split long trajectories into several smaller ones, increasing the number of trajectories but reducing their length.

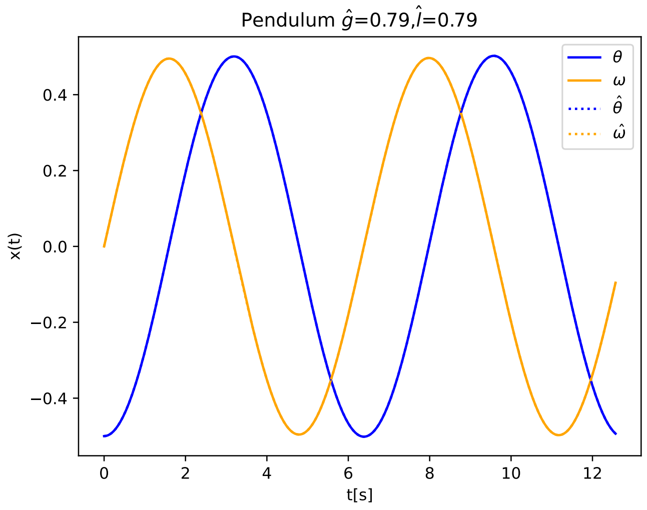


Figure 3.8: Simulation of the true system defined by eq. (2.2) with $g = l = 1.0$ and the model. Only the ratio between the estimated parameters matter because they appear as a fraction as seen in eq. (2.2).

On the extreme end we end up with trajectories that only contain the initial condition and one step ahead in time.

We can apply these ideas to estimate the values of g and l in eq. (2.2) based on trajectories generated by simulating the true system. In this case the model class of the true system and the model class of a model is the same, which means that it is possible to get a perfect fit assuming that our optimizer is able to find the correct parameters. We sample initial conditions in a grid defined by $\theta_0 \in [-\pi, \pi]$ and $\omega_0 \in [-\pi, \pi]$ with a resolution of 0.1 and simulate one step ahead the system using eq. (2.8) and a step size of 0.001 seconds. Minimizing eq. (3.8) for using eq. (3.5) with a learning rate of 0.001 we arrive at the results shown in fig. 3.8 and associated losses shown in fig. 3.9. Examining the plots we see that the estimated values for the systems parameters are $\hat{g} = 0.79$ and $\hat{l} = 0.79$. Despite them being different from the parameters of the true system, the ratio of the fraction in which they appear in eq. (2.2) is the same. This is an example where there is no single combination of parameters that minimizes the problem.

3.3 Neural Network-based models

NNs are a family of parametric models which have proven effective at solving a large variety of tasks. Characteristic of NNs that they implement a non-

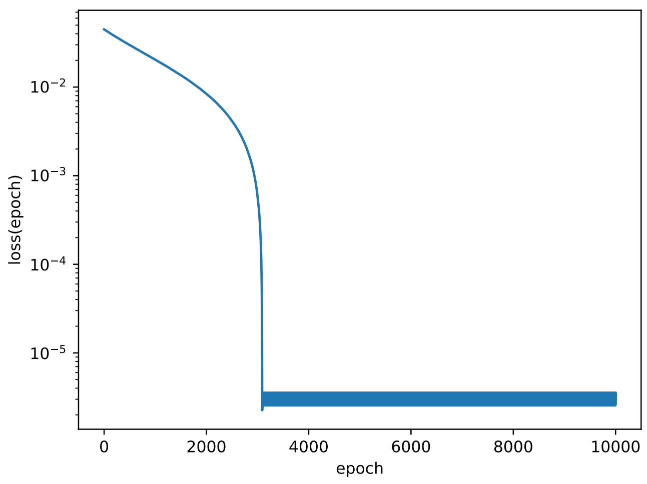


Figure 3.9: Loss resulting from minimizing eq. (3.8) on the pendulum example

linear map from their inputs to their outputs which is trained by tuning the parameters of the network.

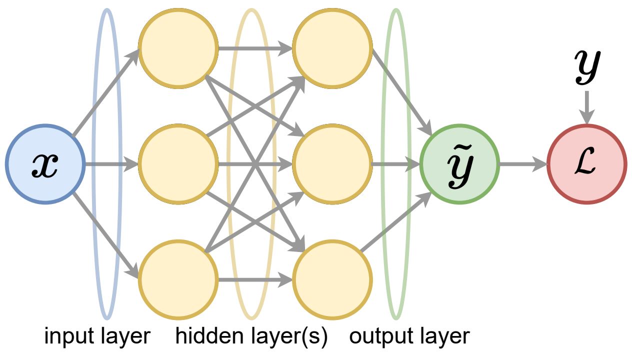


Figure 3.10: Multi-layer perceptron. Source [76]

3.3.1 Multi-Layer Perceptron

One type of network is a Multi-layer perceptron (MLP) [50] or *feedforward* NN, like the one shown in fig. 3.10. A MLP has a single *input layer* one or more *hidden layers* and an *output layer*. Each layer consists of a number of *neurons* which pass on information to neurons in the next layer, hence the feedforward in the name. The number of neurons in the input and output layer is dictated by the application we wish to use the NN in. The number of hidden layers and the number of neurons in each of the hidden layers are so-called *hyperparameters* of the network. The hyperparameters of the network influences the network's capacity to model complex functions. Generally the larger the number of hidden layers and neurons the more complex mappings can be realized. Increasing the number of neurons also results in an increased number of parameters that must be tuned during training.

The parameters of a MLP are a set of weights and biases which we denote w and b respectively. A MLP can be broken down into multiple *perceptrons*, each corresponding to a circle with incoming edges shown in fig. 3.10. The output of a perceptron is a weighted sum of its inputs, which is offset by a bias and then finally passed through a non-linear function referred to as an *activation function*. Formally this can be defined as eq. (3.9).

$$\tilde{y} = \psi(\sum_{i=1}^{n} w_i x_i + b) \tag{3.9}$$

where w is the weights, b are the biases, x is the input and $\psi : \mathbb{R} \rightarrow \mathbb{R}$ is the activation function. The role of the activation function is to introduce non-linearity into the model. The MLP itself is a composition of perceptrons that together form a continuous function from the networks input and parameters to its outputs. MLPs are typically trained using supervised learning by defining a loss function which is itself a continuous function of its inputs. The result of this is that the loss function can be differentiated with respect to its parameters which enables us to use gradient-descent based optimization to tune the parameters like described earlier in the chapter.

The connection between an MLP and the process of modeling a dynamical system is not immediately clear. An MLP is a building block we can use to construct more complex models of dynamical systems. There are different ways to do this, some of which leverage concepts form numerical simulation and others which employ a strategy more reminiscent of collocation methods. Section 3.3.2 introduces a basic taxonomy of the methods. Next, Section 3.3.3, describes how numerical integration can be used with MLPs to simulate dynamical systems. Following this, Section 3.3.4 describe a different approach where an MLP is trained to approximate the solution of a set of ODEs.

3.3.2 Taxonomy

A central challenge to understanding the NN models in SciML literature is that it is a relatively new field and that it draws in ideas and terms from many domains such as SysId, DL, ML and many more. Additionally, the type of models applied are typically tailored to solve a specific problem leveraging prior knowledge of the particular physical phenomenon. Often the core idea of an approach being misunderstood or it being oversold as something it is simply not. A concrete example is the confusion around the term PINN [103], in the authors experience it can mean two very different things, which has also been noted in [66, section 1.1.5]:

- Any type of NN-based model that incorporates prior knowledge of physics
- A specific type of model that we describe in section 3.3.4 that approximates the solution of DEs in way similar to collocation methods.

The fact that this misunderstanding is present in current literature indicate that there are still a lot of confusion around the nomenclature and how the models are implemented in practice. Our survey publication 6 introduces a basic taxonomy describing the two main types of models. It also covers several other topics and variations of the models as shown in fig. 3.11.

Contribution 1: Define a taxonomy for models encountered in SciML literature.

Another challenge of understanding the literature is that there is rarely any emphasis put on how the model is implemented. In addition to the "block diagram"-inspired figures like fig. 3.12 and fig. 3.17, we provide reference implementations for many of the class of models identified in the survey. The code for all the models can be accessed from this GitHub repository[1].

Contribution 2: Provide implementation of the models described by the taxonomy

3.3.3 Neural Ordinary Differential Equations

The idea of combining numerical integration and NNs was recently popularized in [27] which refers to the approach as NODEs. However, the idea of using NNs in a recurrent fashion to model dynamical systems dates back at least 3 decades [112, 115, 44]. In the framework of a differentiable simulator shown in fig. 3.7, NODEs are simply characterized by their use of a

[1]`https://github.com/clegaard/deep_learning_for_dynamical_systems`

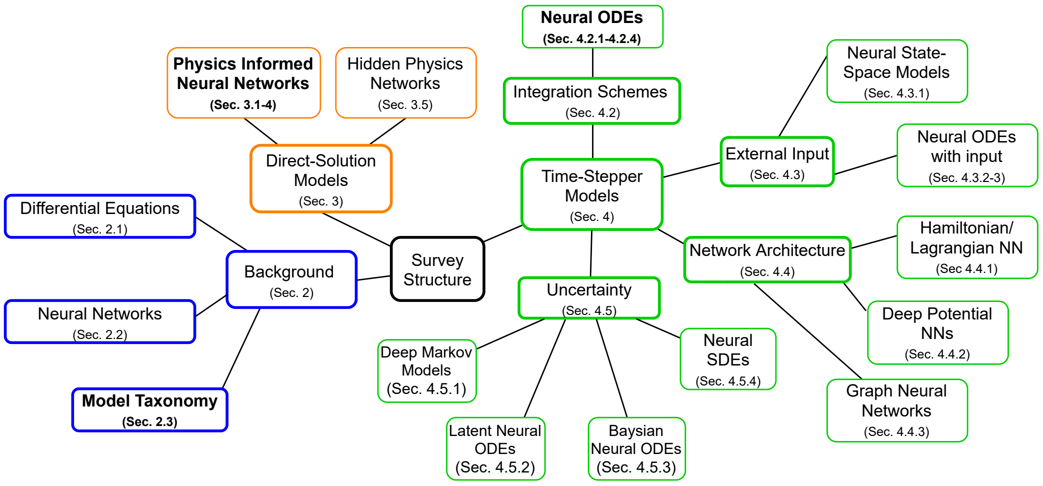


Figure 3.11: A mind map of the topics and model types covered in publication 6. Note that the section numbers match those of the original publication and not those of this thesis. Source publication 6

NN to approximate the physical system's dynamics. The process for tuning the NN's parameters is the same as that of tuning the coefficients of the ODEs described in section 3.2. However, one difference is that the tuned parameters of the NN do not hold any inherent physical meaning — they were simply the choice that minimized the loss function. This is in contrast to the scenario where described in fig. 3.8, where the training resulted in an estimate of g, l.

In publication 6 we applied this type of model to model the ideal pendulum as seen in fig. 3.13. From our experimentation we found that the explicitly encoding the use of numerical integration in the models greatly increased the accuracy of the models. Comparing fig. 3.13 and fig. 3.14, we see that the model using RK45 for numerical integration has a much higher accuracy than the one that uses the NN to directly map from the current state to the next.

Contribution 3: Demonstrate how numerical integration can be used in conjunction with NNs to model and simulate a physical system.

Another advantage of using a numerical integration scheme is that the model can be simulated for a different step-size after training. The procedure for training the models and pseudocode are provided in publication 6.

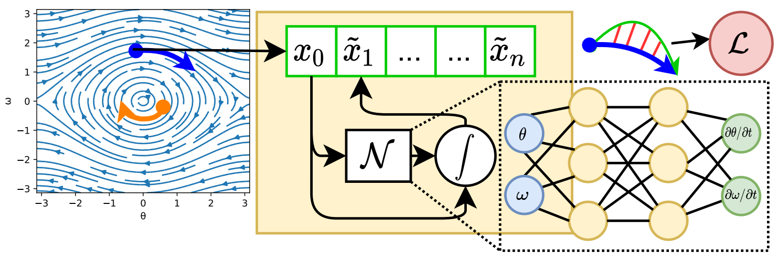


Figure 3.12: NODEs. Starting from a given initial condition x_0, the next state of the system $\tilde{x}_{i+1}$, is obtained by feeding the current state $\tilde{x}_i$ into the derivative network $\mathcal{N}$, producing a derivative that is integrated using an integration scheme $\int$. The loss $\mathcal{L}$ is evaluated by comparing the predicted with the training trajectory. The process can be repeated for multiple trajectories to improve the generalization of the derivative network. Source publication 6.

3.3.4 Physics-Informed Neural Networks

PINNs [103] is another type of model that is popular in SciML literature today. Like the models introduced previously in the chapter they can be used to build simulators for physical systems in a data-driven fashion. Unlike the methods introduced earlier in this chapter PINNs do not use numerical integration. Rather they learn a mapping from the independent variables of DE to the solution at those coordinates. The approach used by PINNs for approximating the solution is similar to *collocation methods* [110] where a function is fitted through a set of collocation points. What makes PINNs special is that they incorporate clever use of AD and terms in the loss function to penalize inconsistency with physics laws. In publication 6 we demonstrated how a PINN can be used to obtain a simulation for the pendulum system defined by eq. (2.2). Additionally, we performed ablation studies to see how various aspect of the method contribute to its effectiveness.

The first experiment was to train the MLP by minimizing the error for a set of collocation points as shown in fig. 3.15. This reveals two issues: First, the predictions are only accurate in the collocation points and not for any other value of t. Second, ω is supposed to represent the derivative of θ — which is clearly not the case for the model's predictions. This outcome is not surprising, since there will be many functions that fit perfectly through the collocation points without necessarily being a solution to the ODE.

The second experiment modified the model slightly such that the MLP only had one output θ. Instead, AD was used to differentiate the angle θ with respect to t giving us the velocity which is per definition ω. As shown in fig. 3.16 this leads to a much higher accuracy for values of t outside the

]

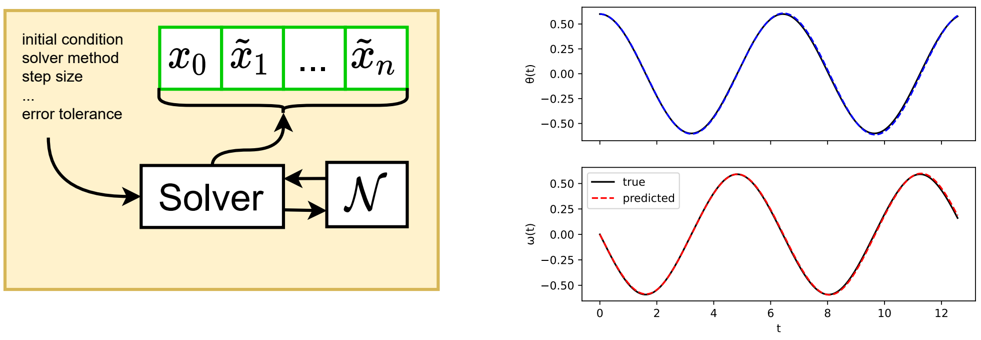


Figure 3.13: NODEs applied to learn the dynamics of the ideal pendulum system defined by eq. (2.2). The RK45 method was used for numerical integration. Source publication 6.

collocation points. By using AD this way we are enforcing that the value and the derivative of the function should go through the collocation points.

The third experiment introduced the "physics-informed" aspect to the model by adding a term to the loss function that penalizes the model for making predictions that do not satisfy eq. (2.1). The new term can be evaluated even without knowing the solution at those points allowing us to evaluate it for any t. It should be emphasized that the collocation points are still needed for training since eq. (2.1) has a trivial solution which $\theta(t) = 0$. By combining the two terms we find a function that: a) fits through the collocation points, b) its derivative fits through the collocation points, c) the solution produced by evaluating the function is consistent with the physical laws governing the system.

The fourth experiment, modifed the model based on the concept of "hidden physics" presented in [104] to infer a hidden state of the system that can not be observed directly. To demonstrate this we considered a pendulum system where the length of the pendulum arm l could change as a function of time as shown in fig. 3.18. The pendulum arm's length, l, was introduced as another output of MLP. Then the equation-based loss term from the PINN is modified to use the estimate of l.

One of the main limitations of PINN as a means to simulate ODEs is that they need to be re-trained every time the system needs to be simulated for a new IC. Likewise, the simulation can not be expected to be accurate beyond the time instances they have been trained for, which is not a problem for methods like NODE. The main appeal of PINNs shines through when solving PDEs which generally requires more complex algorithms than ODEs. One of the challenges of applying PINNs is that optimizing there parameters

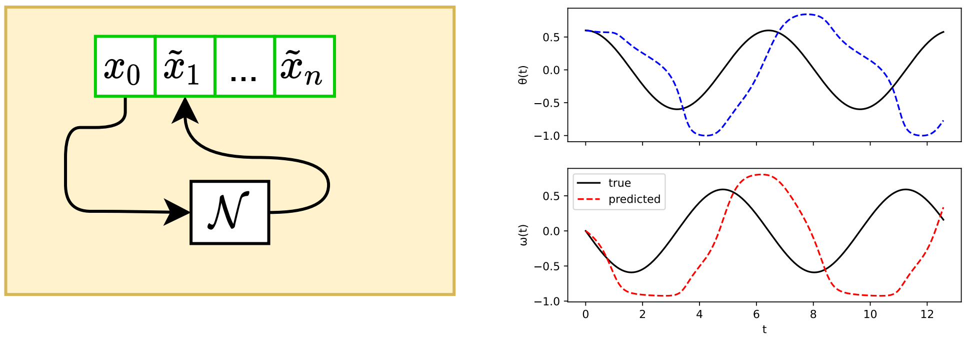


Figure 3.14: NODEs applied to learn the dynamics of the ideal pendulum system defined by eq. (2.2). To demonstrate the importance of explicitly encoding numerical integration the network was trained to map the current state to the next state. Source publication 6.

is difficult [130, 131]. Overcoming these challenges would make PINNs very attractive as a means for modeling physical systems.

Contribution 4: Demonstrate how the significance of using AD and physics based regularization in PINNs

3.4 Model-based Fault Identification

Characteristic for Cyber-Physical Systems (CPSs) is the that they are influenced by the environment they operate in. For instance consider an agricultural robot like the one shown in fig. 3.20. The properties of the soil which it is driving may change if it starts raining or if it drives over a patch where water accumulates. Likewise, parts like the wheel bearings may get worn down and eventually seize. Fault identification [45] refers to a set of techniques for identifying and locating the source of faults in the system.

One approach, referred to as *signal-based* by [45], is to look at the trace of the for abnormalities, like extreme peaks or sudden changes in values. It is also possible to train a model to identify faults that are not easy to formalize into rules by a human. For instance a *recurrent neural network* such as a *gated recurrence unit* (GRU) to emit labels indicating specific faults given a time-series generated from a system exhibiting that fault. The authors of [8] provides a benchmark of various techniques for time-series classification. Applying this method does not require a model of the system but instead relies on labeled data.

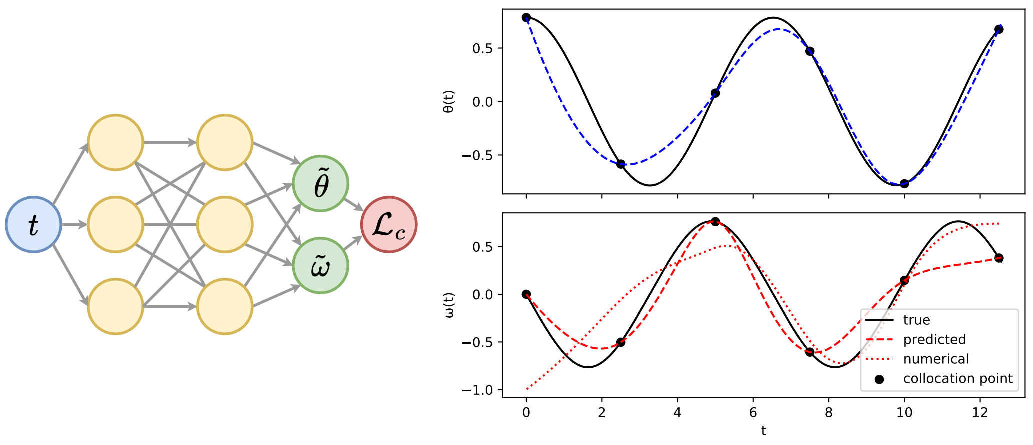


Figure 3.15: Using a MLP with two outputs to approximate the solution at t. The network is trained to minimize the error in the collocation points. This results in a solution that is only accurate in those specific points and not elsewhere. Additionally, the predicted value $\tilde{\omega}$ does not represent the derivative of the predicted variable $\tilde{\theta}$. Source publication 6.

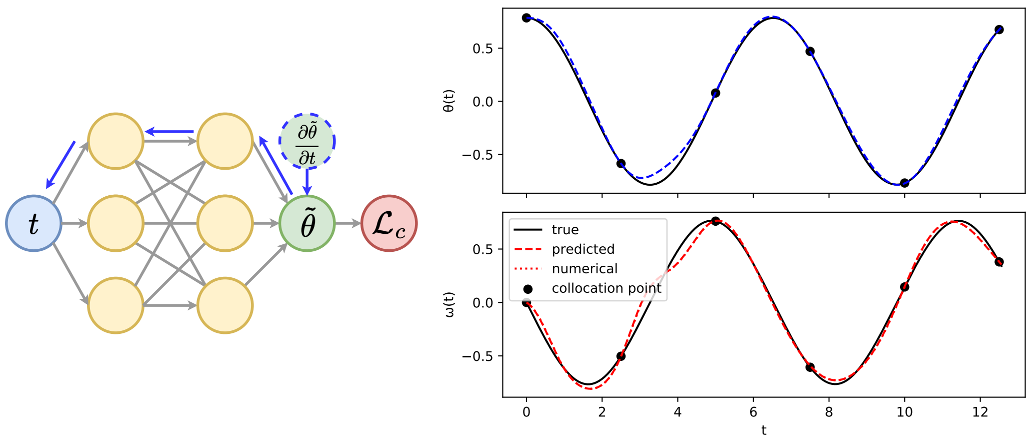


Figure 3.16: Similar approach to fig. 3.15 but the velocity is obtained by differentiating the output θ with respect to t to obtain which per definition gives the angular velocity ω.

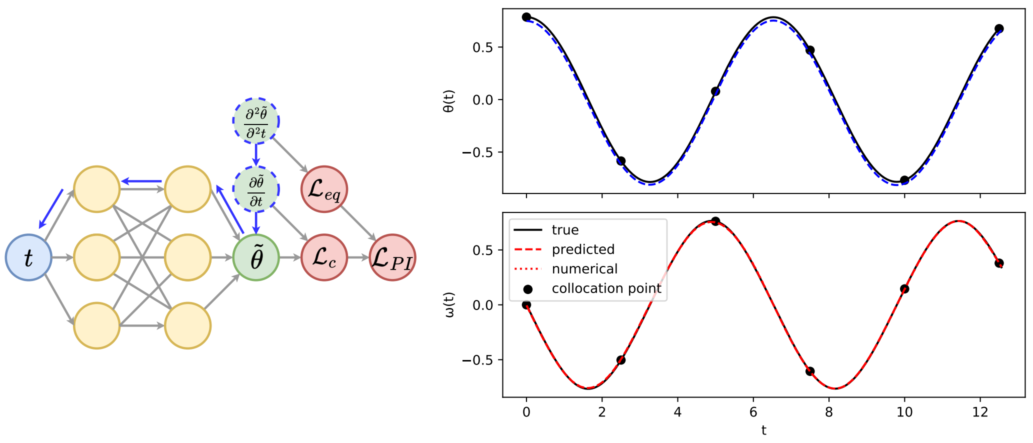


Figure 3.17: Physics-informed neural network. Similar to fig. 3.16 but incorporates eq. (2.1) in the loss function as a means of penalizing physically inconsistent solutions.

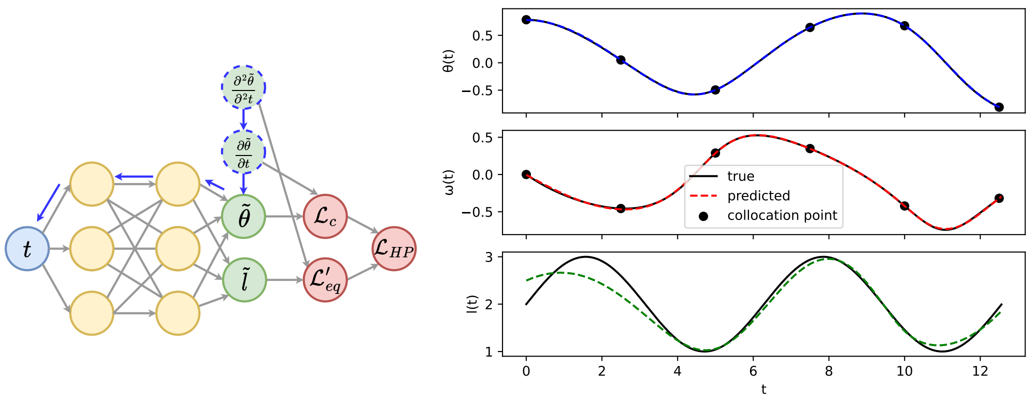


Figure 3.18: Hidden-physics NN. Similar to fig. 3.17 but the model is modified to predict the length of the pendulum arm which varies over time for sake of demonstration.

Another approach, referred to as *model-based* by [45], detects faults by comparing the behavior of the real system with that of a model to identify deviations between the two. The model can be derived by a human using first-principles or a model can be fitted using data to obtain a surrogate of the true system's behavior. Being able to easily estimate parameters on the fly based on observed data makes it possible to not only detect that a fault has occurred, but also to find the set of parameters that would explain the observed behavior. In publication 8 we used this approach to detect when the robot's dynamics deviated and to estimate the parameters that best explain this change which form contribution 5. In the specific scenario we changed the tires' *cornering stiffness* mid-simulation to mimic a scenario where the robot starts slipping. Specifically, we defined a *tracking model* which is a simplified version of the true system which was simulated in parallel with the true system which we referred to as the *reference* as shown in fig. 3.19. Whenever, the two models deviated sufficiently the tracking model would go back to a point in time when the two trajectories were still close and simulate forward in time using a slightly updated estimate for the cornering stiffness. This process was carried out iteratively until the difference between the reference and tracking model was below a certain tolerance. There is a clear resemblance to the *optimal control problem* known from MPC. In MPC we are trying to find a sequence of control actions to achieve the desired trajectory, whereas in our case we are trying to fit a parameter such that we achieve the desired trajectory.

Contribution 5: Demonstrate how model-based parameter estimation can be used to detect anomalies in a self-adaptive system.

At the time, the goal of our work was to demonstrate the feasibility of designing *self-adaptive systems* [82] using co-simulation, which is a topic we cover in chapter 4. As such, we were satisfied with showing that the parameter estimation was able to identify when the parameters of the true system changed with a sufficient degree of accuracy. By defining nominal ranges for each of the estimated parameters it is straight forward to identify faults. However, the real value comes in being able to adapt to changes in one's environment and changes to characteristics of the true system. For instance, if the robot can identify that it is slipping, it can compensate for this by reducing the torque of the wheels or apply a different steering angle. One way to incorporate this behavior in the controller is using a rule-based approach. For instance the controller might reduce its speed by a half if the cornering stiffness falls below a certain threshold. This approach is simple but not optimal in the sense of maximizing the speed at which the robot can move around an area. A more sophisticated approach is to apply MPC

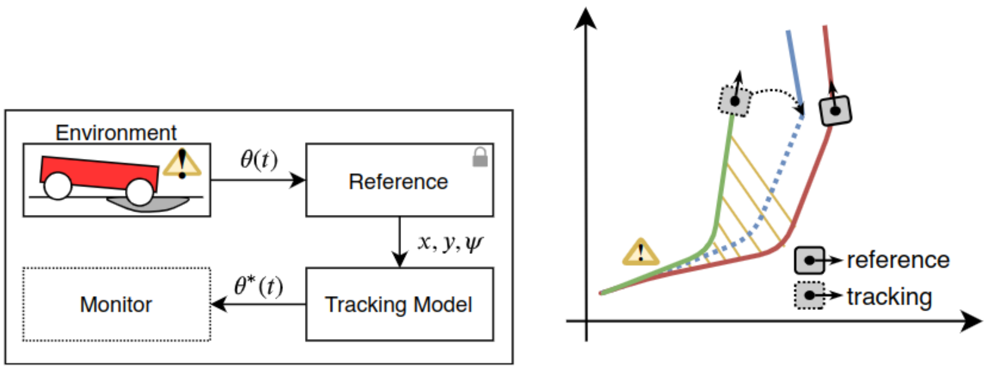


Figure 3.19: Tracking simulator. The term *reference* represents the state of the true system. The *tracking model* represents a model of the true system that we compare with the reference to detect deviations. Source publication 8

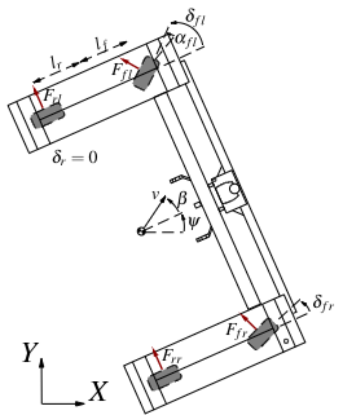
and continuously update the models parameters to the ones estimated by the system. In the MPC community this is referred to as *self-tuning predictive control* [81].

Our implementation of the model did not employ the AD techniques introduced in section 3.1.3. It would be interesting to implement the model in a framework that allows end-to-end differentiation of the model through the numerical solver. This would allow for more robust and efficient estimation of the systems parameters since we avoid issues arising from using finite-differences. It would also be possible to formulate an objective function that measures the performance of the system, which can then be differentiated with respect to design parameters, such as the width of the robot, allowing us to optimize the design through simulation. This idea is sometimes referred to as *differentiable physics* [31, 30] and have gained a lot of traction in the DL control community as an alternative to using *reinforcement learning* [63] which does not assume that our simulation is differentiable.

3.5 Simulation and Control

This section represents ongoing work on applying SciML as a means to design controllers for heating and ventilation systems. We have only completed the first stage of this process which is obtaining a differentiable simulator for the system.
In 2021 the operation of buildings accounted for 30% of global final energy consumption and 27% of total energy sector emissions [22]. Of the total amount of energy, almost 16% are spent on cooling the buildings [117]. Ad-

Front tire slip angle	$\alpha_f = \delta_f - (\dot{y} + l_f \dot{\psi})/\dot{x}$	(1)
Rear tire slip angle	$\alpha_r = (\dot{y} - l_r \dot{\psi})/\dot{x}$	(2)
Lateral tire force at a front wheel	$F_{c_f} = C_{\alpha_f} \alpha_f$	(3)
Lateral tire force at a rear wheel	$F_{c_r} = C_{\alpha_r}(-\alpha_r)$	(4)
Longitudinal acceleration	$\ddot{x} = \dot{\psi}\dot{y} + a$	(5)
Lateral acceleration	$\ddot{y} = -\psi\dot{x} + (1/m)(F_{c_f}\cos(\delta_f) + F_{c_r})$	(6)
Yaw acceleration	$\ddot{\psi} = (1/I_{zz})(l_f F_{c_f} - l_r F_{c_r})$	(7)
Velocity in the global frame	$\dot{X} = \dot{x}\cos(\psi) - \dot{y}\sin(\psi)$	(8)
Velocity in the global frame	$\dot{Y} = \dot{x}\sin(\psi) + \dot{y}\cos(\psi)$	(9)



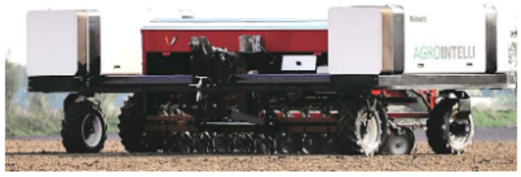


Figure 8: Robotti in the field. Photo: Agrointelli.

Parameter	Magnitude	Description
l_f	10^0	Distance COG to front wheel (m).
m	10^3	Mass of the vehicle.
I_{zz}	10^3	Rotational inertia.
$C\alpha_{f/r}$	10^2	Tire cornering stiffness.

Figure 3.20: Schematic and mathematical model of Robotti robot. Source publication 8.

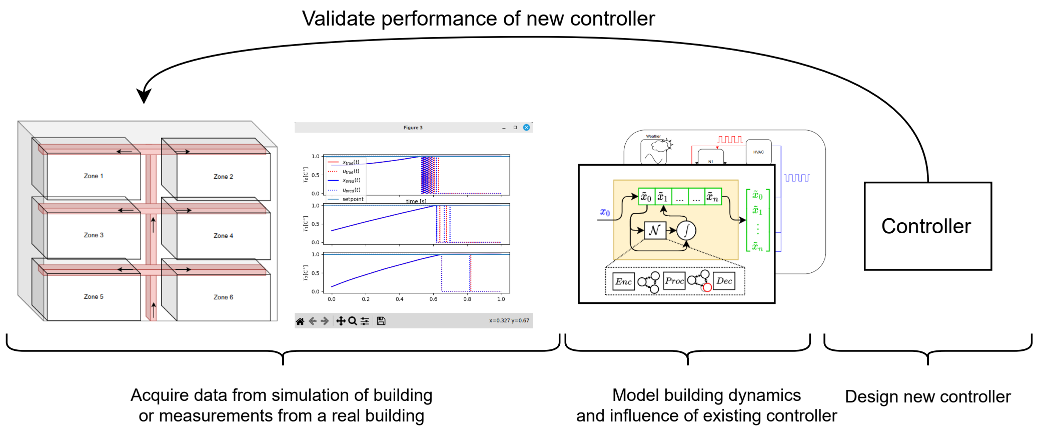
dressing this issue requires that (i) energy is generated in a clean and climate friendly way, (ii) that the produced energy is utilized optimally to achieve the desired objective. We focus our effort on the latter. For a *Heating, ventilation, and air conditioning* (HVAC) system, this amounts to finding a control algorithm that maximizes comfort while minimizing the used to achieve the desired temperature. HVAC provides some unique challenges compared to other types of process control [2]:

- Complexity: The dynamics are non-linear
- Disturbances: Ambient temperature and solar radiation varies throughout the day and with the weather
- Measurement Uncertainty: Temperature sensors may only measure a change in temperature a long time after it has occurred, and it may not be representative of the rest of the room.
- Conflicting Control Loops: Achieving the desired temperature in one zone may be counterproductive to achieving the desired temperature in a neighboring room

Traditional control methods such as ON/OFF or PID[80] controllers are often favored for HVAC systems because of their simplicity [2]. However, they are far from optimal since they can't anticipate that the temperature change is delayed or how external factors may influence the dynamics of the building. In recent years, there has been an increased interest in applying *model predictive control* (MPC) to address some of the issues that arise from simpler controllers [2]. Applied to HVAC the core idea of MPC is that by modeling how the buildings' temperature changes as a function of a control action and external disturbances, it becomes possible to anticipate what control action to apply now such that temperature becomes comfortable - while minimizing the energy to produce the action. The prerequisites for implementing MPC is: (i) a mathematical model must be formalized with captures the dynamics of the system accurately, (ii) a control algorithm must be derived which can infer the optimal control input for the system.

3.5.1 Workflow

To develop optimal control policies, we apply a multi-stage workflow, as seen in fig. 3.21. The first step is to obtain trajectories of the system we want to optimize. We use synthetic data from an RC-model with known coefficients, and known dynamics for both the simple control actions and the



Validate performance of new controller
Zone 1
Zone 2
Zone 3
Zone 4
Zone 5
Zone 6
setpoint
Enc
Proc
Dec
Controller
Acquire data from simulation of building or measurements from a real building
Model building dynamics and influence of existing controller
Design new controller

Figure 3.21: Workflow for implementing optimal control using a data driven approach.1. Dataset describing evolution of the building is captured by sensors in a real life building or obtained through a simulation. 2. A parametric model such as a NODE or RC-network is trained to approximate the dynamics of the building, including the influence of control signals. 3. A controller is designed to replace the existing non-optimal controller, 4. The results are validated by taking the new controller and plugging it into the original system. For simulations this step is straight forward, but for physical buildings it would require that the new controller be installed, and the building be made to operate according to it.

external perturbations, because it allows us to generate an arbitrary number of training trajectories and to conveniently validate the controllers we develop. The second step in the workflow is to develop a parametric model (RC-model/NODE) of the system that closely approximates the synthetic trajectories. Given the parametric model, we design a new controller that optimizes the operation of the system such that it minimizes an objective function. As the final step, we use the controller we designed based on the parametric model in the synthetic simulation model to validate its performance.

To implement MPC-based optimal control, we use data-driven models to learn the dynamics of the system and to obtain a controller capable of performing optimal control of the system. How we choose to model the system and the controller can be characterized by:

- What mathematical structure do we use to model the dynamics
- What training algorithm is used to tune the model to the data

- What type of control algorithm we apply
- What criterion is the control algorithm trying to optimize for

The implication of this is that we can construct models in different ways.. The concrete choice and their implications for the performance is best studied on a building by building basis, as they incorporate insight that is not general to all buildings. The rest of the section introduces the buildings and numerical experiments performed on each of these.

Contribution 6: Demonstrate the first stage of a workflow for identifying a buildings thermal dynamics from temperature measurements, enabling gradient-based optimization of the control-policy of the building's HVAC system

3.5.2 Physical System

The first example is a synthetic dataset generated from the building in fig. 3.23. The building has 6 zones, each characterized by a temperature $x_i \in \mathbb{R}$. The building is modeled by an *RC-network*[2] [83, 108] as shown in fig. 3.22. The dynamics of the building are defined by eq. (3.10).

$$\frac{x_i}{\partial t}(t, R, C) = \sum_{j=1}^{M} \frac{x_j(t) - x_i(t)}{C_i R_{ij}} + \frac{u_i(t)}{C_i} \tag{3.10}$$

where M is the number of zones, $C \in \mathbb{R}^M$ is a vector of each zone's specific heat capacity, $R_{ij} \in \mathbb{R}^{M \times M}$ is a matrix defining the thermal resistance between the zones, and $u : \mathbb{R} \to \mathbb{R}^N$ is a vector valued function defining the heat flux of the HVAC system to each zone. An HVAC system is connected to each individual zone and is modeled by an *on-off controller* defined by eq. (3.11). For simplicity, we assume that the set point is constant throughout the entire duration of simulation.

$$u_i(t) = c(x_i(t)) = \begin{cases} 1.0, & \text{if } x_i(t) \leq x_i^*(t) \\ 0, & \text{otherwise} \end{cases} \tag{3.11}$$

The procedure for simulating the true system is depicted in fig. 3.24. First, we sample L initial conditions $X_0 \in \mathbb{L} \times \mathbb{M}$ corresponding to zone temperatures at the start of an experiment from a uniform distribution. Next,

[2] *Network* referring to the zones being connected in a graph-like structure in the model, not the use of NNs.

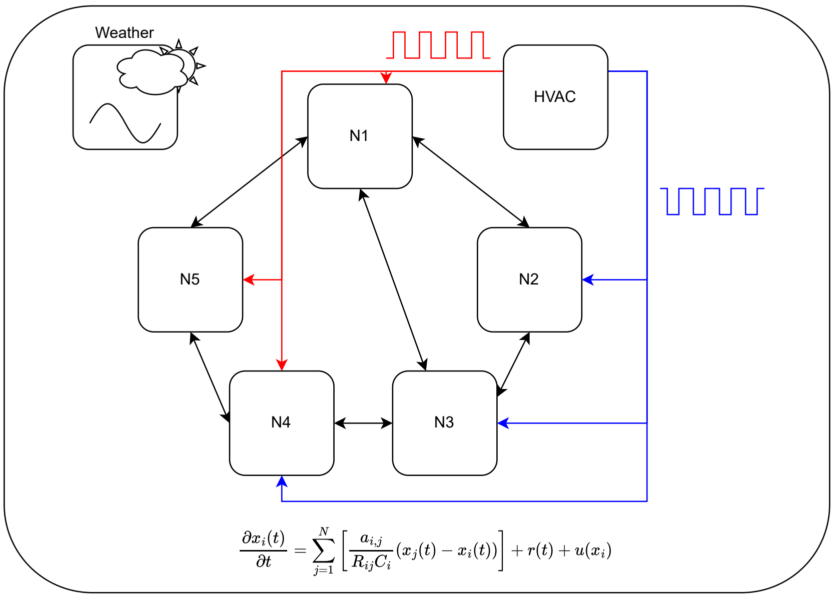

Weather
HVAC
N1
N2
N3
N4
N5
$\frac{\partial x_i(t)}{\partial t} = \sum_{j=1}^{N} \left[\frac{a_{i,j}}{R_{ij}C_i}(x_j(t) - x_i(t)) \right] + r(t) + u(x_i)$

Figure 3.22: RC-model of the building.

we use a numerical solver to simulate the system N steps ahead in time for every each initial condition resulting in a tensor $\bar{X} \in \mathbb{R}^{L \times M \times N}$. This tensor is the training data which we will use for identifying the dynamics of the building and HVAC system.

3.5.3 Identification

In this very contrived case, we attempt to approximate the system described by eq. (3.10) with an RC model. The objective of this is to estimate a set of RC parameters that the model and the true system as close as possible. We do so by minimizing the MSE across all trajectories as defined in eq. (3.12)

$$\mathcal{L}(R, C) = \frac{1}{LMN} \sum_{h=1}^{L} \sum_{i=1}^{M} \sum_{j=1}^{N} \sqrt{(\hat{X}_{hij} - X_{hij})^2} \tag{3.12}$$

Applying gradient descent using eq. (3.12) as loss function we obtain the values for the resistance matrix R and capacitance vector C as shown in fig. 3.26. Examining the value of the loss function shown in fig. 3.25 we see that it decreases quickly and monotonically. This indicates that the loss could probably be reduced by further iterations of the optimization algorithm.

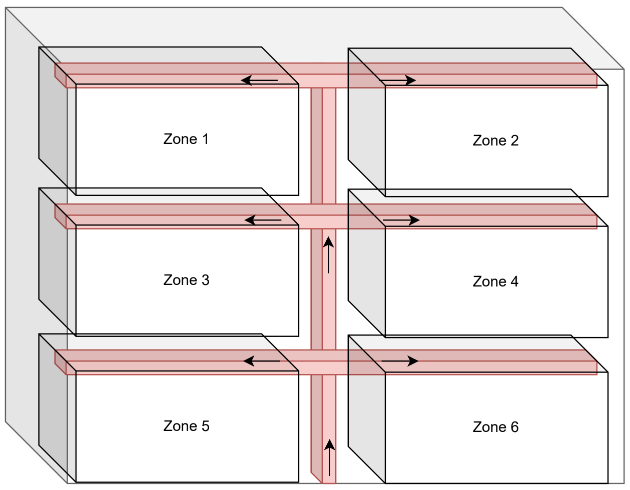


Figure 3.23: High level model of the building.

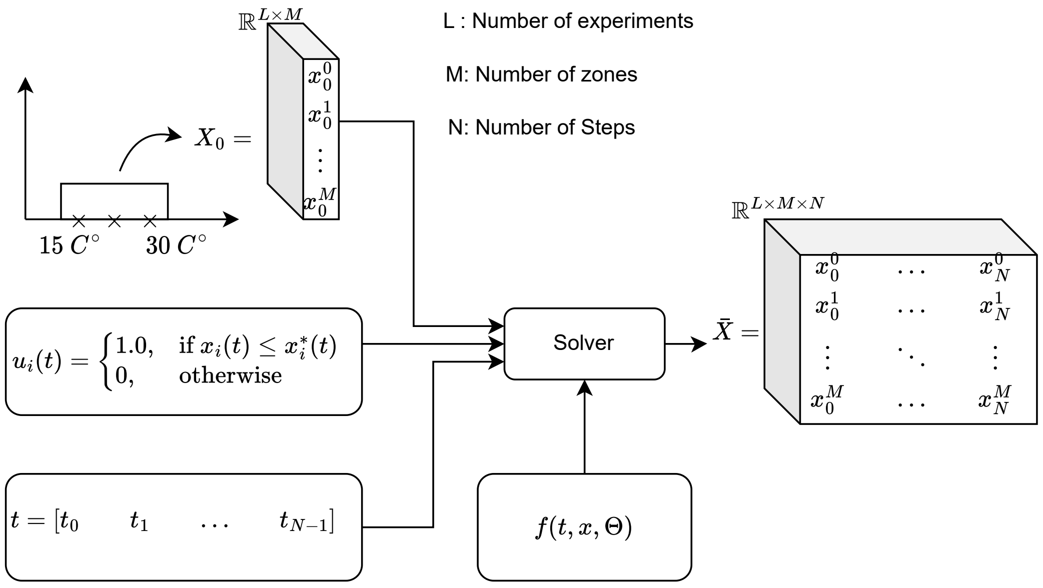

$\mathbb{R}^{L\times M}$
L : Number of experiments
M: Number of zones
N: Number of Steps
$X_0 =$
x_0^0
x_0^1
$\vdots$
x_0^M
15 C°
30 C°
$\mathbb{R}^{L\times M\times N}$
$\bar{X} =$
$x_0^0 \; \dots \; x_N^0$
$x_0^1 \; \dots \; x_N^1$
$x_0^M \; \dots \; x_N^M$
Solver
$u_i(t) = \begin{cases} 1.0, & \text{if } x_i(t) \leq x_i^*(t) \\ 0, & \text{otherwise} \end{cases}$
$t = [t_0 \quad t_1 \quad \dots \quad t_{N-1}]$
$f(t, x, \Theta)$

Figure 3.24: Experimental procedure for synthetic dataset. First L different combinations of zone temperatures are sampled from a random distribution. Next, the system is simulated using a model white-box model for a specific controller.

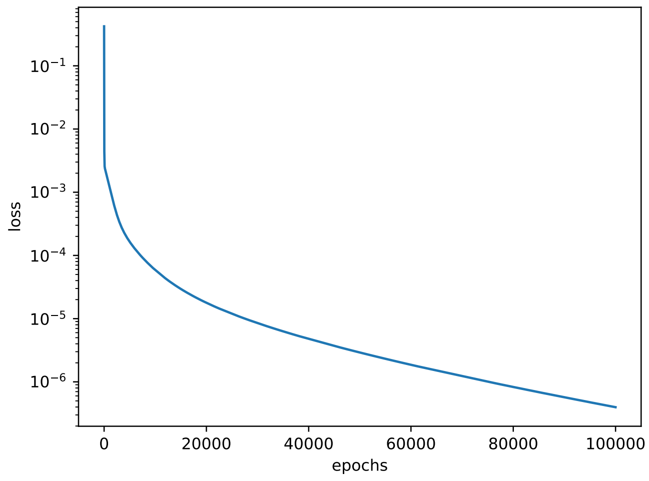


Figure 3.25: Loss eq. (3.8) of RC-model of 6-zone building.

3.5.4 Designing the Controller

The goal is to replace the simple controller defined by eq. (3.11) with one that maximizes comfort while using the minimum amount of energy. This can be formalized as

$$\mathcal{L}_{control}(\theta) = \int_{t_0}^{t_{end}} \alpha c(t) + \beta p(t) dt \tag{3.13}$$

where $c(t)$ measures how comfortable the temperature of the building is, $p(t)$ is the power used by the HVAC and α, β are weighting coefficients determining how important comfort is relative to the power consumption. There are several ways to model how the temperature of a building influences the comfort of humans [38]. For the sake of simplicity, we define this to be the 2-norm between the set point and the actual temperature, averaged across all zones:

$$c(t) = \frac{1}{M} \sum_{i=1}^{M} ||x_i(t) - x_i^*(t)||_2$$

The power consumption of the HVAC system is modelled to be proportional to the control signal:

$$p(t) = \frac{1}{M} \sum_{i=1}^{M} u_i(t) * c * (x_{air} - x_i(t))$$

$$Q = mc\Delta T$$

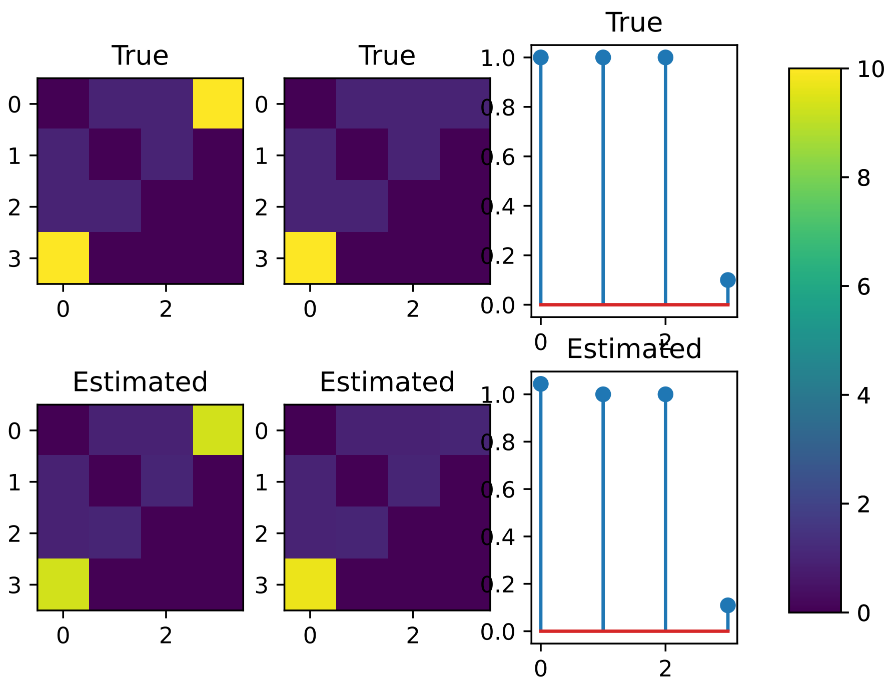


Figure 3.26: True parameters of eq. (3.10) versus those estimated during training. The off diagonal terms are very close to the true values. The diagonal terms are not modified during optimization because they correspond to heat transfer from a node to itself, where the temperature difference will always be zero.

where Q is energy, m is mass, c is heat capacity, ΔT is change in air temperature.

As stated in the disclaimer in the start of the section we have demonstrated the process for tuning an RC-model to the building from the data. The next step in the process is to replace the controller defined by eq. (3.11) with another type of controller that can be tuned to minimize the loss function defined by eq. (3.13). Since the simulator we have constructed is differentiable it is possible to differentiate the loss function with respect to the parameters of the controller, as shown in fig. 3.7. Our intended plan was to compare the effectiveness of different methods such as RC-networks and NODE for modeling the building's dynamics, as well as different types of controllers. However, we have not had time to pursue this work further.

Chapter 4

Co-simulation

Co-simulation refers to a set of techniques that can be used to simulate systems by combining simulations of its components [70, 49]. A primary use-case of co-simulation is to facilitate the development of CPS by allowing different designs to be tested and compared in a virtual setting [51, 41]. One thing that sets co-simulation apart from traditional simulation is the emphasis on developing algorithms that can simulate models [49, sec 2.4] without knowledge of the models internals. The underlying assumption is that models of the systems components may be defined by different vendors using a formalism that is appropriate for modeling the given component, as illustrated in fig. 4.1.

The rest of this chapter is structured as follows. First, Section 4.1 introduces a system consisting of a robotic arm and controller that is used throughout the chapter. Next, section 4.2 describes the ad-hoc way of writing co-simulation program. Following this, Section 4.3 introduces the FMI standard for coupling models to perform a co-simulation of the full system. Next, Section 4.4 introduces a tool which was developed during the PhD project, which allows high-level languages to be leveraged in FMI-based co-simulation. Finally, Section 4.5 demonstrate how we can integrate ML component in our existing simulation software using the tools introduced in the previous section.

4.1 Example: Controlled Robotic Arm

We use the *controlled robotic arm* system introduced in publication 10 as a running example in this chapter.

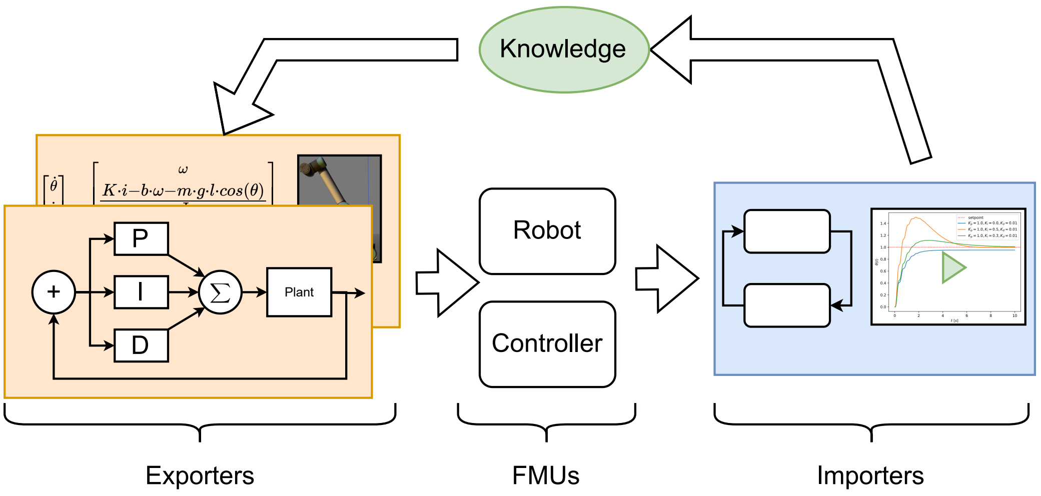


Figure 4.1: Co-simulation workflow. Exporters refer to tools that can package and export models referred to as Functional Mock-up Units (FMUs) in the context of Functional Mock-up Interface (FMI). Importers refer to tools that can load and interact with FMUs typically in order to perform a simulation.

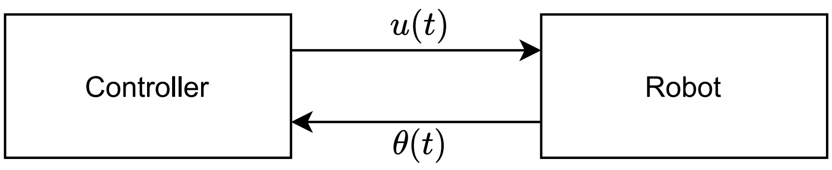


Figure 4.2: Connection between controller and robot model. Source publication 10

4.1.1 Controlled Robot

The states of the system are its angle θ, the angular velocity ω and the current running through the coils of the electrical motor i. The dynamics of the Robot are described by the set of ODEs:

$$f(x) = \begin{bmatrix} \dot{\theta} \\ \dot{\omega} \\ \dot{i} \end{bmatrix} = \begin{bmatrix} \omega \\ \frac{K \cdot i - b \cdot \omega - m \cdot g \cdot l \cdot cos(\theta)}{J} \\ \frac{u(t) \cdot V_{abs} - R \cdot i - K \cdot \omega}{L} \end{bmatrix} \tag{4.1}$$

Where: the derivative of the angle $\dot{\theta}$ is, per definition, equal to the velocity of the arm ω. The derivative of the angular velocity $\dot{\omega}$ is determined by the torque coefficient $K = 7.45\ s^{-2}A^{-1}$, the current i, the motor-shaft friction $b = 5.0\ kg \cdot m^2 \cdot s^{-1}$, the gravity acting on the arm, denoted by $m \cdot g \cdot l \cdot \cos(\theta)$, with $m = 5.0\ kg$, $g = 9.81 ms^{-2}$, $l = 1.0m$ and the moment of inertia $J = \frac{1}{2}ml^2 = 0.5$. The change in current is determined by the input from the controller u, the voltage across the coils $V_{abs} = 12.0\ V$, the resistance $R = 0.15\ \Omega$ and the motor's inductance $L = 0.036\ H$. Comparing eq. (4.1) with eq. (2.9) we see that the main difference is the introduction of third state variable modeling the current running through the motor.

Contrary to the visualization shown in fig. 4.3 the model only considers a single joint that rotates around a single axis. In reality, the majority of industrial robots have more rotational axis and joints. At a conceptual level modeling this amounts to expanding the state space with angles and velocities in the additional spatial dimensions, and storing a copy for each joint. However, the derivation of the dynamics quickly becomes very involved for more complex robots. In robotics literature there is a distinction between the concepts of dynamics and kinematics. A model of a robot's dynamics is a mapping from the forces applied at each joint to a change of position. A model of a robot's kinematics is a mapping from the joint's angles to the position of the joints. There is also a distinction between the *forward* and *inverse* problem. Forward dynamics is the process of calculating the changes in joint position based on torques applied, whereas inverse dynamic is the process of calculating the torques that must be applied to change the angles of robot in a given way [92]. For reference [71] describes one way to derive the forward dynamics of a 6-joint robot. For more general description and historical perspective on how to derive the dynamics of robot we refer the reader to [39].

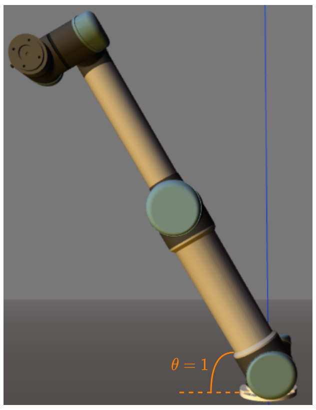


Figure 4.3: Screenshot from visualization of robotic arm. For simplicity, the relative angles of all joints except the first one is set to 0.0. Source publication 10

4.1.2 Controller

A proportional-integral-derivative (PID) controller [6] is used to generate the control signals sent to the Robot. The continuous formalization of the controller is given by:

$$u(t) = K_p e(t) + K_d \dot{e}(t) + K_i \int_0^t e(\tau) d\tau \tag{4.2}$$

where $e(t)$ is a measure of the error of the variable being controlled and K_p, K_i and K_d are coefficients used to tune how the proportional and derivative terms are weighted. In case of the robot, the controller is trying to minimize the error between the desired angle $\theta^*(t)$ and the true angle $\theta(t)$. Thus, the error is defined as $e(t) = \theta^*(t) - \theta(t)$. Tuning the coefficients allow us to influence the rise time, settling time and overshoot of the system [42]. Increasing K_p generally leads to a faster rise time but at the cost of more overshoot. The role of the K_i term is to weigh the integral of the error. Intuitively the integral term is useful to forcing the system out of a state the error is so low that the control action can not force the system closer to the setpoint. For instance, as the Robot approaches its setpoint forces from gravity and the control action may cancel out leading the arm to get stuck, since the error never changes. In this case the integral will still be growing which will eventually lead the control action to overpower the forces generated by gravity acting on the arm.

In practice, most controllers are implemented digitally, which means that derivatives and integrals must be replaced by discrete approximations. There are several ways to do this, the simplest being to replace derivatives by first-order differences

$$\dot{e}(t_k) \approx \dot{e}_k = \frac{e_k - e_{k-1}}{T},$$

and integrals by sums

$$\int_0^{t_k} e(t_k) \approx E_k = \sum_{n=1}^{N} e_{k_n} \cdot T$$

where $e_k = e(t_k)$, T is the sampling time and $N = t_k/T$ is the number of samples between time 0 and t_k. After replacing the continuous definitions in Equation 4.2 we obtain an equation that can be implemented on a discrete controller

$$u_k = K_p e_k + K_i E_k + K_d \dot{e}_k \tag{4.3}$$

Another consideration to take into account is that in real life the controller must first sample the state of the system using from a sensor, compute the output of the control algorithm and then write that value to the physical output of the controller. This is sometimes referred to as the *sample time* of the controller. In cases where the sampling time is large enough that the real system may change its state significantly this can lead to instability because the output of the controller is delayed [42]. For analysis done in the frequency domain, there exist way to emulate a discrete time delay, such as the *pade* approximation [42]. When simulating in the time-domain using numerical simulation the delay can be modelled by holding the value of the controller output until the simulation has advanced by an amount of time corresponding to the sampling time of the controller.

4.2 Monolithic vs Co-simulation

There are two fundamentally different ways to simulate a coupled system like the controlled robotic arm. The first approach is to write a program defining a loop that:

1. Computes the output of the controller using eq. (4.3)
2. Computes derivatives of the robotic arm for the given control input using eq. (4.1)
3. Integrates the derivatives with the current state
4. Repeat 1-4 until the simulation has reached its termination

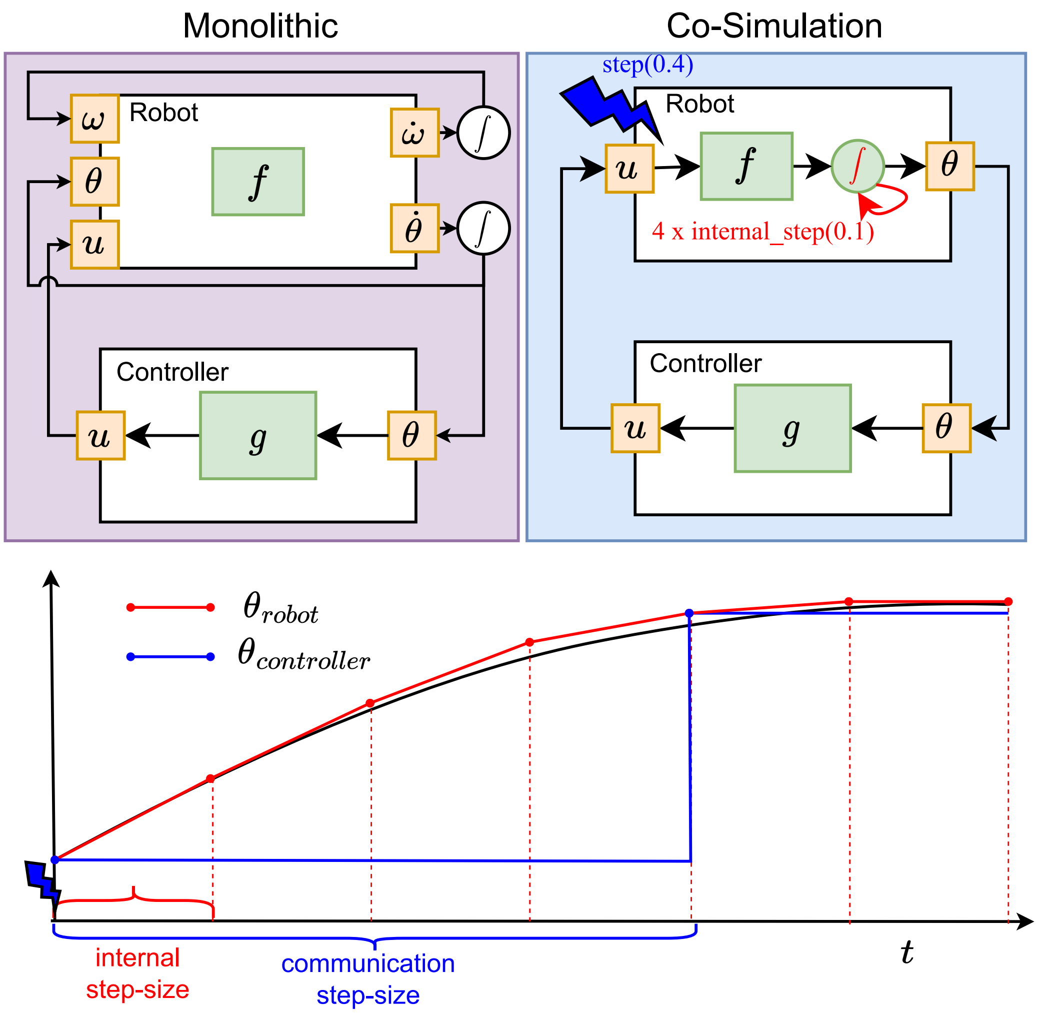


Figure 4.4: Difference between a monolithic simulation and a co-simulation.

We refer to this approach as building a *monolithic simulator*. This approach is suitable for simple systems where we have full access to the internals of each sub-system. The drawback of this approach is that the complexity of the simulation problems scales poorly with the number of components. A different approach is to write a program that allow for the simulation of the system while treating the models as black-boxes, which we refer to as a *co-simulation algorithm*. The differences the monolithic and co-simulation approach can be seen in fig. 4.4.

An implication of treating models as black-boxes is that the numerical integration must happen inside the models, because only it has access to its derivatives. The role of the co-simulation algorithm is orchestrating a

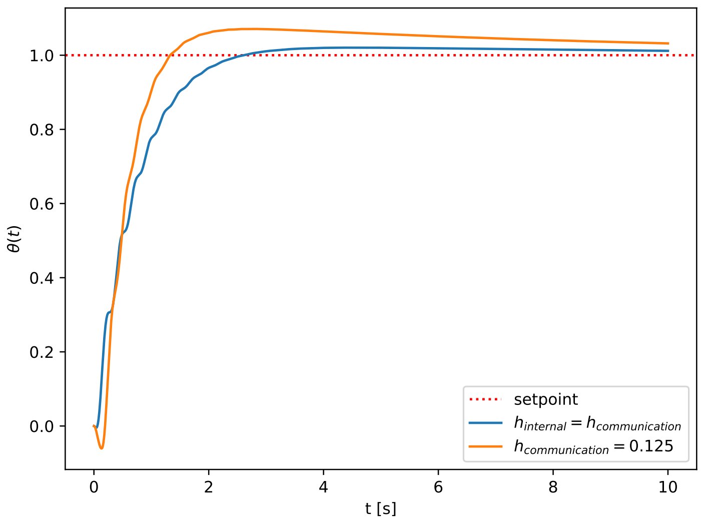


Figure 4.5: Co-simulation of the controlled robotic arm system with varying communication step-sizes. The blue line depicts the case where the internal and communication step-size are equal, which results in synchronization at each internal step. The orange line depicts a case where the models are synchronized every 1/8 of a second. The internal-step size is for both simulations is 1/1024.

sequence of calls to the models to advance their simulation time, which we refer to as *stepping*. A key concept is that when models are asked to step for some duration, they may choose to take multiple smaller steps internally to reduce the numerical error. We refer to the length of the steps taken internally as the *internal step-size* of a model. Another important concept is that the outputs of the models are not synchronized between the internal steps but rather at *communication points* that are typically spaced further apart. We refer to the spacing of the communication points as the *communication step size*.

Picking the right communication step-size is important for the accuracy of the simulation as shown in fig. 4.5. At one end of the extreme the communication step-size can be chosen to be the smallest of the internal steps-sizes, leading to frequent synchronizations. At the other extreme the communication step-size can be very large compared to the internal step-sizes of the models. For the controlled robotic arm this may result in the system becoming unstable, because the output of the controller is held constant between each communication point from the perspective of the robotic arm.

4.3 Functional Mock-up Interface

Given the wide range of tools for modeling and simulating systems, standardized interfaces have been developed to enable these different tools to communicate their simulation results. One such interface is the FMI [17, 4]. It specifies a file format for packaging a model's assets into an archive that can be loaded by tools supporting the standard, the package is referred to as a FMU. The other part of the standard is a set of functions that can be called by the simulation tool to interact with the FMU. We will refer to tools that loads and interacts with FMUs as *importers* and tools that produce FMUs as *exporters*. The most recent major version of the standard, FMI 3.0, defines 3 different interfaces that a FMU must implement one or more of. The use-cases and the functionality that the FMU must implement can be summarized as:

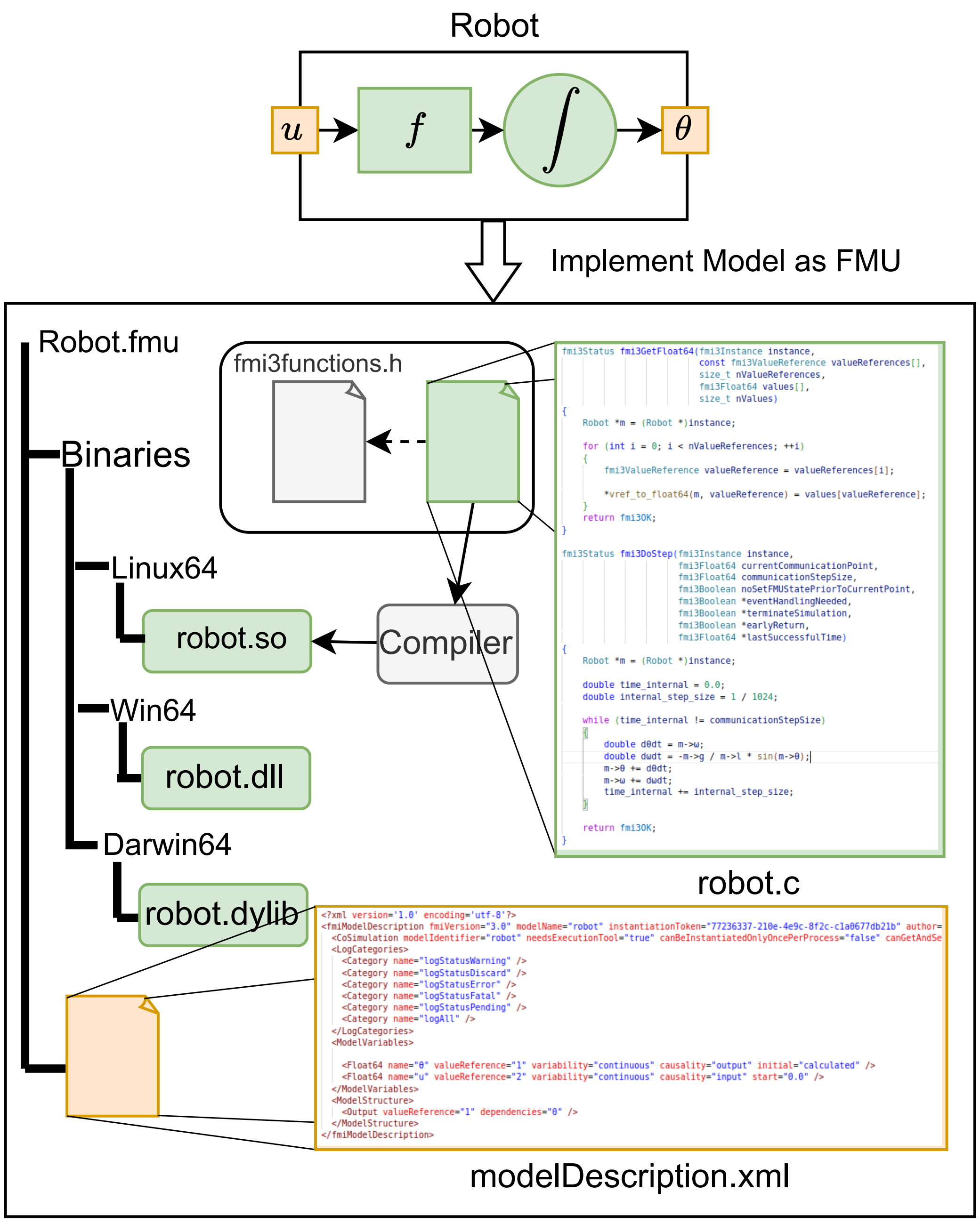
- Model-Exchange (ME): The FMU exposes its derivative which the importer integrates to advance the simulation
- Co-Simulation (CS): The FMU expose a method to advance the state of the FMU ahead in time, which typically means that the FMU contains its own solver
- Scheduled Execution (SE): Newly introduced and designed to deal with periodic and discrete events

The focus of this section is on the second type, CS FMUs.

4.3.1 Implementing a FMU

The process for implementing a FMU is depicted in fig. 4.6 From a practical perspective a FMU is a collection of files that define its interface and behavior. For ease of distribution the files are compressed in a zip archive and have the extension `.fmu` rather than the conventional `.zip`.

The behavior of the model is defined by implementing a set of functions declared by a C-header file distributed as part of the standard. For instance, to implement a FMI 3 compliant FMU one would have to implement the functions from the header `fmi3Functions.h`. The functions that must be implemented depends on which of the 3 interfaces we wish to support. In our case we wish to support the co-simulation part of the standard. After the interface has been implemented the C files are compiled to a shared library and stored in the `binaries` directory. Shared libraries are referred to by different names in the context of different operating systems and the library



```c
fmi3Status fmi3GetFloat64(fmi3Instance instance,
                          const fmi3ValueReference valueReferences[],
                          size_t nValueReferences,
                          fmi3Float64 values[],
                          size_t nValues)
{
    Robot *m = (Robot *)instance;

    for (int i = 0; i < nValueReferences; ++i)
    {
        fmi3ValueReference valueReference = valueReferences[i];

        *vref_to_float64(m, valueReference) = values[valueReference];
    }
    return fmi3OK;
}

fmi3Status fmi3DoStep(fmi3Instance instance,
                      fmi3Float64 currentCommunicationPoint,
                      fmi3Float64 communicationStepSize,
                      fmi3Boolean noSetFMUStatePriorToCurrentPoint,
                      fmi3Boolean *eventHandlingNeeded,
                      fmi3Boolean *terminateSimulation,
                      fmi3Boolean *earlyReturn,
                      fmi3Float64 *lastSuccessfulTime)
{
    Robot *m = (Robot *)instance;

    double time_internal = 0.0;
    double internal_step_size = 1 / 1024;

    while (time_internal != communicationStepSize)
    {
        double dθdt = m->ω;
        double dωdt = -m->g / m->l * sin(m->θ);
        m->θ += dθdt;
        m->ω += dωdt;
        time_internal += internal_step_size;
    }

    return fmi3OK;
}
```

```xml
<?xml version='1.0' encoding='utf-8'?>
<fmiModelDescription fmiVersion="3.0" modelName="robot" instantiationToken="77236337-210e-4e9c-8f2c-c1a0677db21b" author=
  <CoSimulation modelIdentifier="robot" needsExecutionTool="true" canBeInstantiatedOnlyOncePerProcess="false" canGetAndSe
  <LogCategories>
    <Category name="logStatusWarning" />
    <Category name="logStatusDiscard" />
    <Category name="logStatusError" />
    <Category name="logStatusFatal" />
    <Category name="logStatusPending" />
    <Category name="logAll" />
  </LogCategories>
  <ModelVariables>

    <Float64 name="θ" valueReference="1" variability="continuous" causality="output" initial="calculated" />
    <Float64 name="u" valueReference="2" variability="continuous" causality="input" start="0.0" />
  </ModelVariables>
  <ModelStructure>
    <Output valueReference="1" dependencies="0" />
  </ModelStructure>
</fmiModelDescription>
```

Figure 4.6: Conventional process for implementing eq. (4.1) as a FMU. The derivative and numerical integration scheme of the model is defined by the implementation of the `fmi3DoStep` function declared in the `fmi3functions.h` header file. The inputs, outputs, and parameters of the model are declared in the `modelDescription.xml` file which is placed in the root of the FMU.

have different file-extension. For instance, Linux refers to them as shared objects and use the extension `.so`. When an importer loads a FMU it links the shared libraries allowing it to call methods of the FMU.

The interface of the FMU is declared in a special configuration file modelDescription.xml which we will refer to as the model description. The file declares the inputs, outputs, and parameters of the model as well as other meta information. When an importer loads a FMU it reads the model description and parses its contents and presents this information to the user of the tool. Using this information the user can then define the coupling between the different models.

4.4 Universal Functional Mock-up Units

Research in SciML is generally conducted in high-level scripting languages such as Python which makes it possible to rapidly prototype ideas using modern libraries for scientific computing and ML. Being able to use the same languages and reuse existing code inside FMI-based co-simulation has the potential to greatly reduce the barrier to integrating these techniques. Based on this need a new tool Universal Functional Mock-up Units (UniFMU) was developed in publication 7. Specifically, the tool provides the following utility to the user:

- Support for Python, C#, Java and MATLAB FMUs out of the box.
- An easy-to-use extension mechanism to provide support for any language.
- CLI to generate template FMUs using a single command.
- Pre-built binaries for Windows, Linux and macOS, eliminating the need for cross-compilation and complex tool-chain setup.
- Flexible configuration of the execution environment, such as running inside a Docker container or activating a virtual environment.

Contribution 7: Create a tool for coupling languages like Python, Java and C# code into a FMI based co-simulation.

The process for implementing a FMU using the tool can be seen in fig. 4.7. The most important concept is that the tool eliminates the need to compile the binaries of the FMU by providing pre-compiled binaries for Linux, macOS and Windows. The behavior of the model is instead implemented by a script or a program located in the `resources` folder, which we refer to as the *implementation* of the model. A configuration file `launch.toml` defines the procedure for initializing the model. For instance, for a model implemented in Python it would invoke the interpreter with the name of the script implementing the model. The configuration file is made to be modified by the user, which gives a great flexibility towards bootstrapping models that may require a complex setup to initialize.

The tool itself is a command-line interface (CLI) which can be used to generate templates of FMUs for a selection of languages. The generated FMUs are fully functional out of the box and serve as a starting point for implementing the behavior of the specific system. For instance the template for a Python FMU contains a file `model.py` which mirrors the methods of the FMI-API in an object-oriented fashion as depicted in listing 4.1.

```python
class Model:
    def fmi2DoStep(self, current_time, step_size, no_step_prior):
        return Fmi2Status.ok

    def fmi2EnterInitializationMode(self):
        return Fmi2Status.ok
```

Listing 4.1: Implementation of FMI-API in Python template

To implement a model of the robotic arm, the equations defining the derivatives would have to be defined in the script as well as the procedure for applying numerical integration. An important point is that the script itself can be executed outside context of a co-simulation because it is valid Python. From practical experience this makes development much easier because the model can be implemented and tested using the same workflow as you would otherwise.

The tool uses remote procedure call (RPC) as a means to communicate from the generic binary to the implementation of the model as shown in fig. 4.8. When the FMU is instantiated a communication channel is established between the binary and the model. Whenever a call to the C-API is made it is converted into a message and sent to the model. In the case of a Python template, a script `backend.py` is responsible for receiving the commands and invoking the corresponding methods on the model implementation. More details about this can be found in publication 7. A discussion on the limitations and future work on the tool can be found in section 5.2.1.

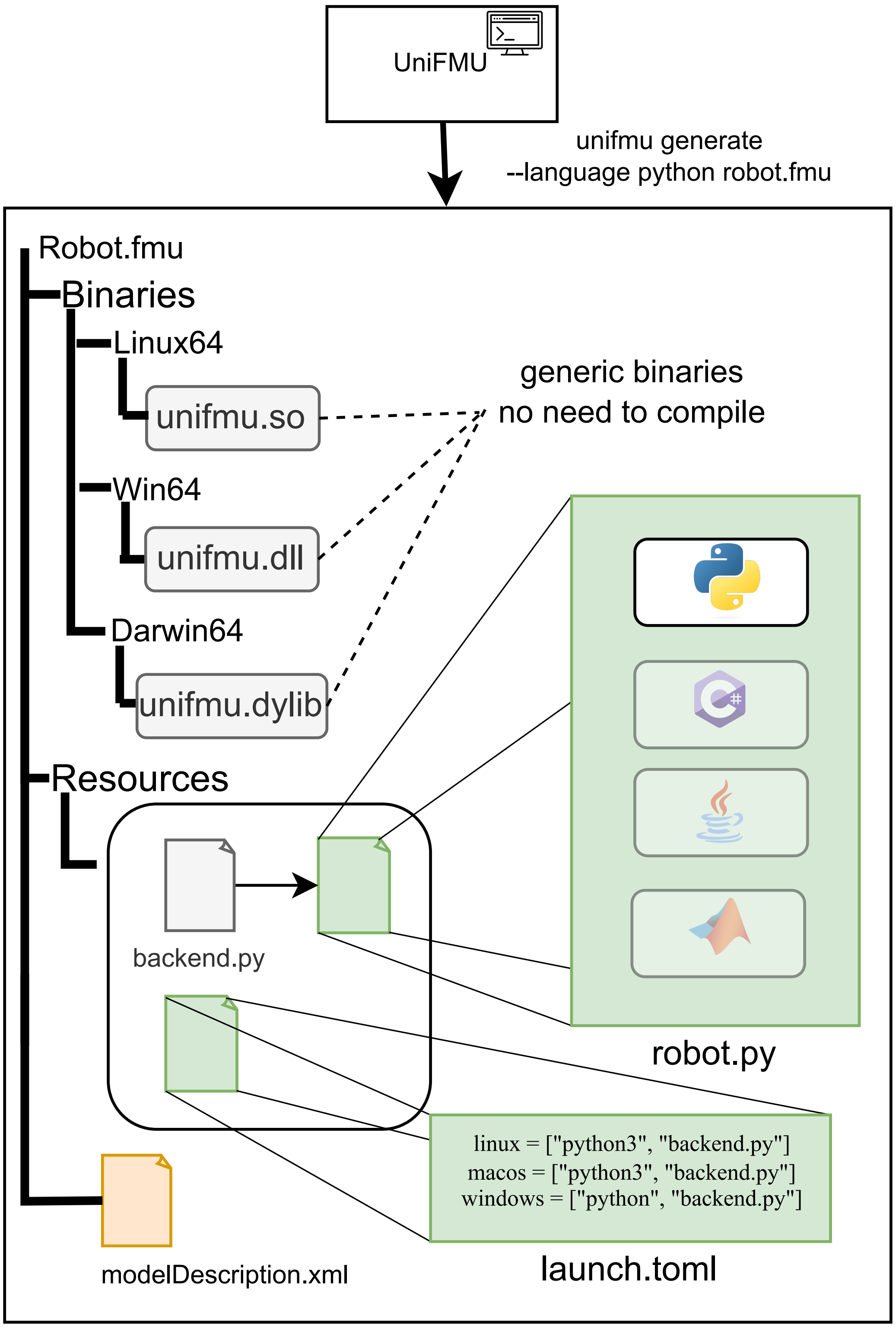


Figure 4.7: Implementing a FMU using UniFMU

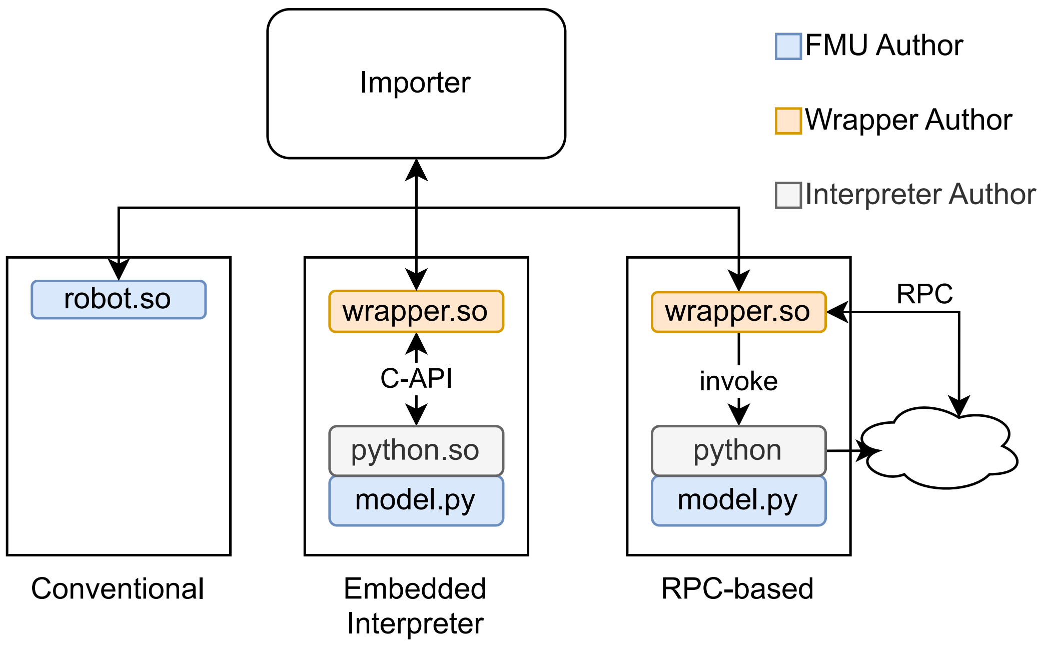


Figure 4.8: Different approaches for implementing FMUs.

4.5 Machine Learning and Co-Simulation

When modeling a large complex CPS there may be parts that are well studied and amenable to traditional first-principles based modeling techniques and there may be other parts where it is difficult to derive a model. In situations like these we may create FMUs of the former parts using existing tools and use ML to create models that we then pack as FMUs and integrate into our co-simulation environment. In publication 11 we investigated this approach in the context of a single family house as shown in fig. 4.9.

Contribution 8: Demonstrate how a ML model can be integrated in a tool supporting FMI-based co-simulation

The house was modeled in Dymola [19] using the build in block design workflow for all components but the *Hot Water Tank* shown in fig. 4.10. Based on our work in publication 12 where we compared the performance of various combinations of features and ML models, and found *random forest regression* [96] to be the most robust. The template FMU was generated using UniFMU and the method was implemented as shown in listing 4.2

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
...
```

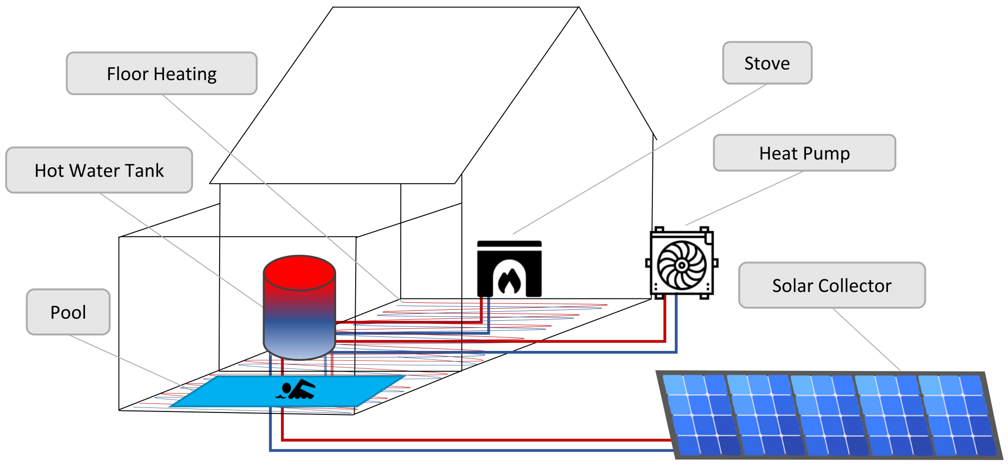


Figure 4.9: Single family house. All components except the *Hot Water Tank* are modelled in Dymola and the Hot Water Tank as a FMU created by UniFMU. Source contribution 8

```
def do_step(current_time,step_size,no_step_prior):
    self.temp_next=self.forrest(self.temp_prevs)
    return Fmi2Status.ok
```

Listing 4.2: Implementation of `fmi2DoStep` by storage model.

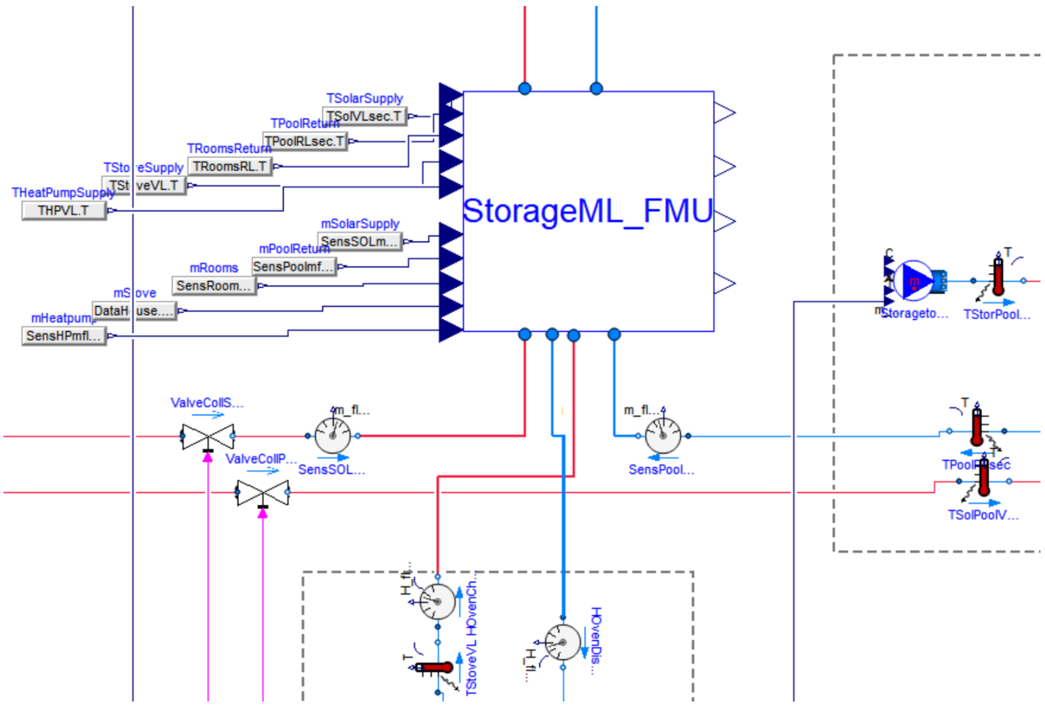


Figure 4.10: Schematic of the house in Dymola, the box in the middle is a FMU export by UniFMU

Chapter 5

Conclusion

The two previous chapters presented the contributions made during this PhD project in the area of SciML and its use in FMI-based co-simulation. This chapter provides a discussion of the extent of which the contributions have addressed the research questions outlined defined in section 1.3. Based on this discussion we outline future work that the author believe is important to advancing the use of SciML for simulating physical systems. This can be seen as a critical review of what has not been accomplished in this PhD thesis to fully answer the research questions. Recall the relationship between the publications included in part II and the research questions was shown in fig. 1.5 in the introduction. The main contribution presented in each paper and how they are related is shown in fig. 5.1. The following three sections addresses the extent that each research question has been addressed by the contributions and outlines future research directions. Finally, section 5.4 provides a brief concluding remark to round up the thesis.

5.1 SciML in Simulation

The first research question was defined in section 1.3 as follows:

SciML in Simulation: How can we make it easier to select and apply the appropriate SciML methods in the context of modeling and simulating a physical system? Additionally, how can knowledge of the physical system be integrated in the modeling process to obtain more accurate and robust models?

In an ideal world we would be able to identify the most effective type of model to apply to a given problem based on a simple set of rules. When deriving a model based on data, as we do in ML, we answer this question by

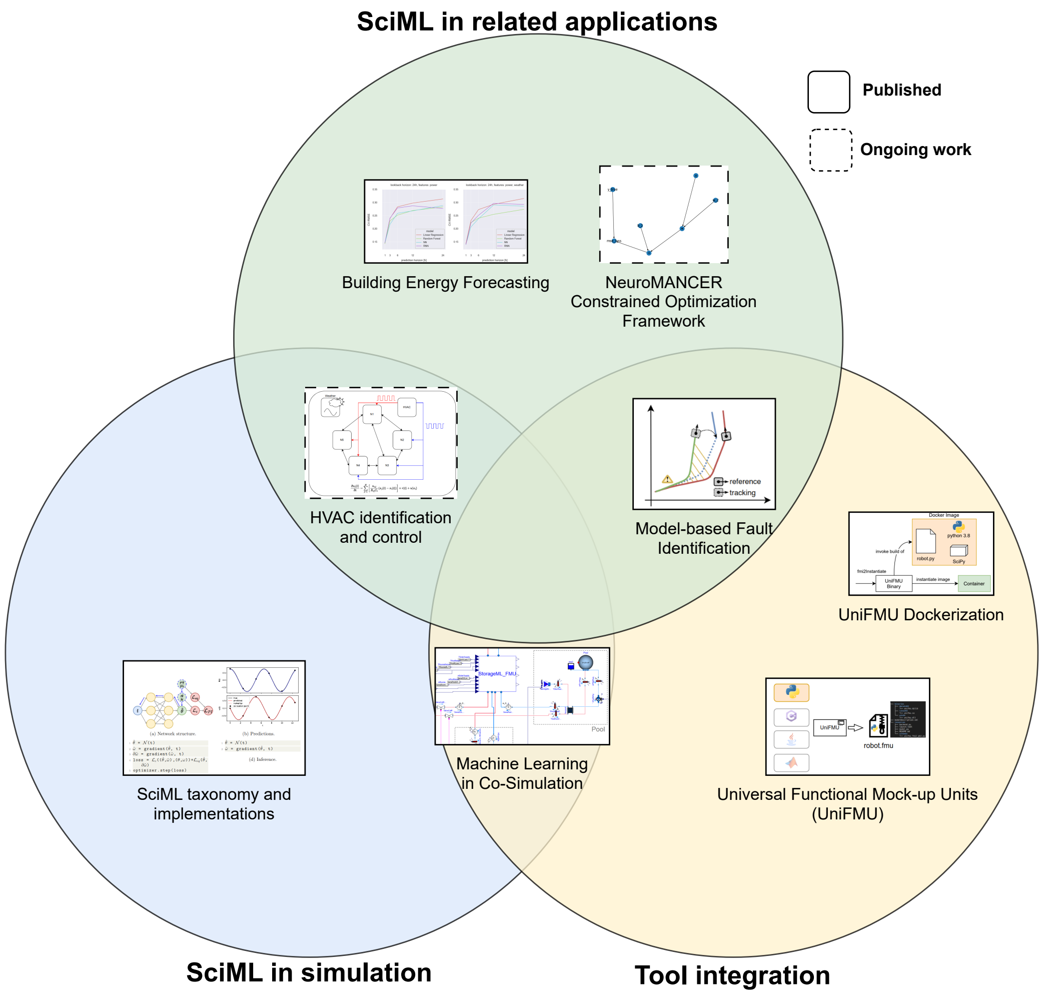

SciML in related applications
Published
Ongoing work
Building Energy Forecasting
NeuroMANCER Constrained Optimization Framework
HVAC identification and control
Model-based Fault Identification
UniFMU Dockerization
SciML taxonomy and implementations
Machine Learning in Co-Simulation
Universal Functional Mock-up Units (UniFMU)
SciML in simulation
Tool integration

Figure 5.1: Map of publications and the contributions made by them. The colored shapes next to each contribution indicate which area the contribution is in

benchmarking different types of models and combinations of hyperparameters to see which one performs the best. By running benchmarks across several large datasets we start to see patterns in the characteristics of the models that performs well across all datasets, and we will discover *tricks* that lead to improved performance. For instance, it has been observed that convolutional NNs [74] perform well in the context image processing tasks. Intuitively this makes sense, convolution is a mathematical operation which has been used extensively in image processing before the recent growth in NNs popularity. Another example is the use of *shortcuts* between the layers

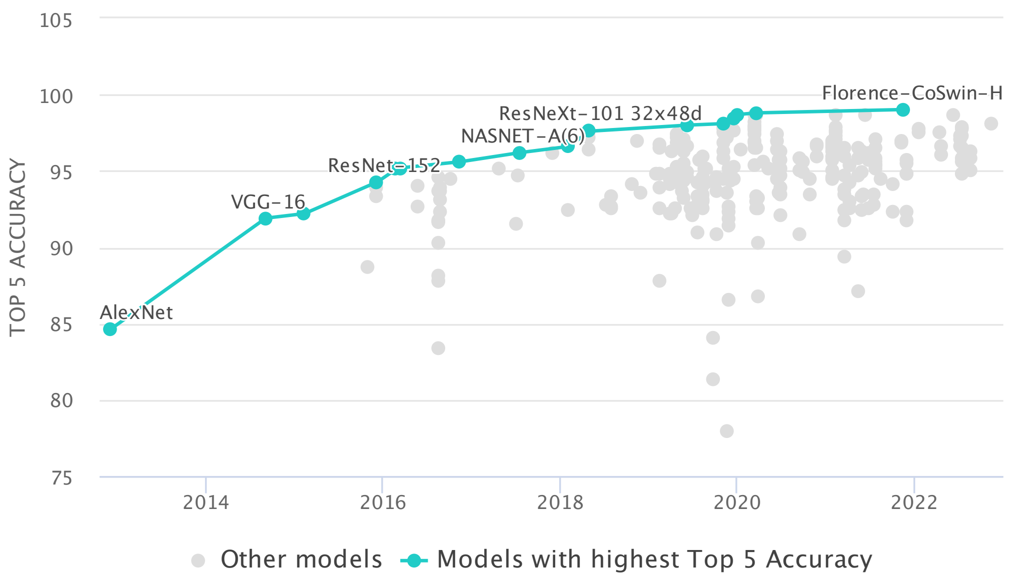


Figure 5.2: Evolution of model accuracy on ImageNet dataset. Source `https://paperswithcode.com/sota/image-classification-on-imagenet`

of ResNets which makes them easier to train [57, 78]. Ideas like these have been incorporated along with others to create increasingly effective methods for tasks where large datasets exist. For instance, the 99% top-5 accuracy on the ImageNet [32] dataset went from 85% to 99% from 2014 to 2022, as shown in fig. 5.2, well above human performance [109]. Benchmarks like these gives provides a good basis for selecting which type of model to apply to a related problem. For instance, when faced with an image classification task, a model that achieves SOTA performance is probably a good starting point for further experimentation. Additionally, the results also establish an upper bound for the accuracy our model could achieve given a comparable amount of data for training and effort spent on fine-tuning. Unfortunately, we do not have the same amount of empirical evidence at applying ML in the context of scientific applications which makes it difficult to give concrete recommendations on which type of model to apply and what kind of performance to expect.

One of the characteristics separating SciML from traditional ML is that they leverage whatever prior knowledge the domain expert may have of the physical system's behavior or properties. As such their implementation must be custom tailored to the specific problem based on its physics — which contrasts most ML methods that are more general purpose. This makes benchmarking the performance of SciML methods more time-consuming since the models would have to be adapted to reflect the physics of a new problem. Knowledge about the physical properties can take many forms and may be incorporated in the modeling process in different ways. One example is the clever use of AD in PINNs used to obtain the derivative of one state from another. The effect of this is quite significant as seen by comparing fig. 3.15 and fig. 3.16. Likewise, we saw that explicitly integrating the state with the derivative was much more effective than relying on the NN to learn this implicitly in the case of the time-stepper models described in section 3.3.3.

The names used to reference different methods varies in SciML literature due to the influx of ideas from neighboring fields that have their own distinct nomenclature. This can lead to confusion for newcomers to the field. A concrete example is the use of the term PINN which is used by some to reference a specific type of model we discussed in section 3.3.4, but also the general idea of incorporating physics in a NN. The primary contribution to addressing this issue is the taxonomy presented in contribution 1, that describes the most commonly encountered SciML methods in terms of how they produce a prediction, how they are trained, and their inherent strengths and weaknesses. In addition to the taxonomy contribution 2 provides source code implementations for the models described by the taxonomy all applied to the same running example. It is the authors hope that this code has helped readers to try out the different types of models on the particular systems they are studying. At the time of writing 43 people have starred the repository containing the source code, which indicates that people have found the reference implementations useful, although the exact impact is difficult to quantify. Finally, contribution 6 demonstrates how different choices of models influences the accuracy of the model in the context of modeling a building's thermal dynamics. Choosing an RC-model as the basis of modeling the system implicitly encodes the concept of thermal resistance and capacitance. Based on our knowledge of the physics we ensured that the resistance matrix was symmetric which is equivalent to saying that the thermal resistance between two rooms is the same in both directions.

Our contributions to answering the research question is demonstrating that these techniques can be effectively using data generated synthetically by simulating a selection of dynamical systems for varying initial conditions. Using synthetic data allow us to study the influence that the training data has on the final model's accuracy. For instance, we can examine how the density of initial conditions and the length of the resulting trajectories has on generalization of the model to unseen data. We have not systematically studied the influence of how real-life factors such as noise and partial observability of states may influence the learning process. These factors are still possible to study in a synthetic setting by defining a noise model and how the value of latent variables are reflected in the observable variables.

5.1.1 Systematic Benchmarking

It is important for the SciML community to develop best practices and to compile a benchmark dataset consisting of data captured from systems that are representative of what would be encountered in real life. This will allow for a systematic evaluation of the effectiveness of various SciML methods and allow us to identify promising classes of models. The first step is to define the concrete task we are trying to solve, how to measure the performance, and finally how to characterize the data from the physical systems which we will be learning from. For ODEs, data could be time-series characterized by the value of each state sampled at a number of points in time. For PDEs data could be a series of snapshots in time, each characterized by the system's state sampled at a number of spatial coordinates. Ideally, prior knowledge of the system should be encoded in a way that allows a model to make use of it, without the need to custom tailor the model's implementation to a given physical system. Using prior knowledge in the context benchmarking models is interesting because the performance of a model is not determined by its architecture alone but also by how effectively the knowledge can be incorporated.

Some benchmarks have been proposed in the SysId community such as these[1], but the number of datasets is rather limited, and they are distributed in a manner that requires considerable effort to load the data for each dataset. There has also been some effort in the SciML community such as this benchmark for PDEs [120], which also provides an in depth discussion on the challenges of creating benchmarks for SciML. More qualitative insight can be gained by looking at `paperswithcode.com` which list 223 datasets with 442 benchmarks published related to image classification and 21 with 0 benchmarks related to physics. Clearly, SciML is still very niche compared to more well established application like image processing. However, it is the author's belief that finding effective ways to benchmark SciML is well worth the effort required despite the challenges.

5.1.2 Experimental Design and Training

Applying SciML methods to real-life applications require that we capture data from the physical system through measurements. Because capturing data might be time-consuming and expensive we are interested in capturing

[1]`https://www.nonlinearbenchmark.org/`

data that tells us important things about its dynamics. In the SysID community the process of figuring out how to illicit an interesting response from the system is referred to as *experimental design* [80, chap.13]. This process is just as important when applying SciML methods to real-life systems than it is in SysID. This process should be informed based on factors such as our understanding of the physical system, the type of model we are trying to fit, and the optimization algorithm we employ. For example, fitting the coefficients of an ODE to data as described in section 3.2 likely requires fewer data points than fitting an NODE to the same problem. Generally speaking, a model with many parameters like a NN is able to realize a function that goes through the data points in the training data, but without it actually capturing the underlying dynamics, as shown in fig. 3.15. Incorporating knowledge in the modeling process can mitigate this issue by constraining the set of parameters considered to be good solutions by the optimizer. A concrete example is how PINNs incorporate physical equations in the loss function as described in section 3.3.4 which effectively filters out solutions that fit through the training data, yet are physically inconsistent.

A closely related topic is how to formulate the optimization problem. When training time-stepper models we are interested in finding a model for which the simulation matches results in trajectories that closely resemble those sampled from the true system. Error in the derivatives produced by the model are propagated through the steps of the solver leading to error being accumulated over time. From personal experience, the more integration steps are taken during training, the more difficult optimization of the time-stepper's parameters become. Others have noted that the step-size used during training has an effect on the dynamics realized by the model after training [93]. These factors highlight the need to establish guidelines for how to effectively train time-stepper models.

5.1.3 Knowledge Incorporation

An interesting direction would be to see if AD could be used in time-stepper models to enforce that a given state may represent the derivative of a different state, as was the case for the PINNs. As seen in section 3.3.4 when comparing fig. 3.15 to fig. 3.16 this has a huge impact on what the model learns. Another promising approach is to combine modeling using first-principles with the black-box models used in SciML. This idea has already been explored in [102], however it would be interesting to apply this hybrid approach to real life systems for which accurate first-principles models already exist to see if a hybrid model can outperform them.

5.2 Tool Integration

The second research question was defined in section 1.3 as follows:

Tool Integration: How can we make the integration of SciML methods in existing simulation environments as easy as possible?

A practical challenges to integrating SciML methods in existing modeling and simulation tools is that they are rarely designed with the execution of a users code in mind. The typical approach is to provide a user-friendly interface for modeling, simulating and plotting results. Contrary to this, research in SciML methods typically use high-level scripting languages where the researcher has full control of the simulation algorithm as described in section 4.2. Expecting the developers of existing tools to provide integrated support for SciML methods is unreasonable, and it is difficult to imagine that any tool would be able to keep up with the rapid advances in ML. Likewise, expecting the user to stop using simulation tools and re-implement the tool's functionality in a script along with the SciML methods is also not an ideal scenario. The research outlined in this thesis focuses on simulation using the FMI standard, which defines a standard interface for exchanging and simulating models originating from different simulation tools. The primary contribution to addressing this is the development of the tool UniFMU representing contribution 7 which makes it possible to use high level scripting languages in FMI-based co-simulations. Building on this, contribution 8 represents a practical example of how this can be used to integrate an ML model in Dymola — a popular modeling and simulation tool. Finally, contribution 5 represents a novel utility of this approach allowing model-based parameter estimation within an FMI-based co-simulation.

The tool UniFMU developed during this PhD project use models written in selection of popular high-level programming languages in any tool support FMI-based co-simulation. For instance, the user might choose to use Python and the PyTorch library as a basis for implementing and training a SciML model. This greatly reduces the barrier to creating FMUs, since the user can use programming languages and software libraries that are appropriate for a given modeling task. Additionally, it removes a lot of the complicating factors that makes implementing an FMU correctly difficult such as memory management, cross-compiling for multiple platforms and the packaging of assets. There are other tools out there that target specific programming language like *FMI.jl* [123] for Julia and *PythonFMU* [56] for Python. An advantage of the approach taken by UniFMU is that it can be extended by the user to integrate new language and simulation environments, without having to modify the tool itself or having to deal with many of the technical complexities of the FMI standard. The tool is used frequently in our own research group and by researchers from at least one other university. The tool is also made publicly available on GitHub where it has been starred 22 times.

5.2.1 Ease of use

A challenge of working with FMUs is ensuring that their model description is consistent with the model's implementation. The approach used by PythonFMU is to provide an object-oriented API that the user must use to declare the models variables inside the script itself. By running the script the model description is regenerated such that it is consistent the implementation. The drawback is that it requires that the same concept be implemented and maintained for all programming languages supported by the tool. The semantics of FMI variables are complex/obscure enough that around half of the FMUs found in a cross-check repository have one or more errors in their model descriptions [11, Sec. 4]. Likewise, the logic that would need to be implemented for each programming supported programming language would non-trivial. An alternative solution would be to develop a standalone application for declaring interface of an FMU through a GUI or from a browser. One benefit of a standalone approach is that it would be useful to anyone implementing FMUs from scratch or those who develop tools for generating them.

5.2.2 Extend Support for ML features

The new FMI 3 standard introduces features like array types and new ways of accessing derivatives, which would be very useful for anyone wanting to integrate ML-models like NNs in their systems. Packages like *Flux.jl* [123] makes it possible to integrating FMUs in ML applications by enabling gradient information to pass through the FMU — provided it implements the appropriate FMI functions. A potential use case of this would be tuning the parameters of a controller using gradient-based methods in cases where the dynamics of the physical system are implemented by an FMU to protect intellectual property.

5.2.3 Performance Benchmarking

UniFMU relies on RPC for communication between the binary and the model which is executing on another process. When a function is invoked on the FMI interface, a message describing the which function was called and which arguments were passed is created and sent to the model. When the model receives this message it unpacks it, does some computation, and then returns a message describing the result of the function. The overhead scales inversely with the communication step size used in the co-simulation, because we need to take more steps to simulate the system for a given time, than had we taken a fewer but larger steps. The exact size of this overhead depends on the machines operating system, hardware, and other programs that might be taking up resources at the same time. A benchmark is provided in [55] measures this overhead to be around 150 μs for a different but comparable RPC implementation to ours. For models based on NNs this is likely not a significant issue since the forward pass is likely to take significantly longer. Extending the UniFMU with the capabilities to benchmark FMUs would be useful for tracking the performance of those generated by the tool, and it would also be a useful for researchers developing similar tools.

5.3 SciML in related applications

The third and final research question was defined in section 1.3 as follows:

SciML in related applications: What are other applications of SciML that are relevant in the engineering of systems beyond their use of obtaining models of the system's dynamics?

The gradient based techniques used to train SciML models can also be useful for other applications during the engineering of systems. Some of these has been touched apart in this thesis. Contribution 5 demonstrates how gradient-based optimization can be used to monitor for changes in a system's parameters, enabling the detection and localization of faults. Contribution 6 demonstrates a workflow for tuning a controller for an HVAC system in a building. First, a model of the building's thermal dynamics is obtained by fitting the coefficients of an RC-model to the data. The model can now be used as a surrogate for buildings thermal dynamics as a means of tuning the controller since the simulation is differentiable. The parameters of the controller can then be tuned end-to-end because both the model and the controllers are differentiable. Common for both applications is that they rely on being able to differentiate the solution produced by the simulator with respect to parameters of the system or the controller. Using AD makes it much less technical to write differentiable simulators, because the gradient of a program can be obtained from the program automatically. These contributions only scratch the surface of the problems SciML can be applied to. The field of SciML and its applications is rapidly expanding as the techniques are being applied to different branches of natural sciences. Recent works like [73, 134, 91, 122] attempts to provide a broad overview of the many applications and provide future perspectives. In the author's opinion the challenge of SciML is not finding new applications, but rather moving from the proof of concept stage to applying them to complex real-life problems. This effort would benefit greatly form standardized benchmarks representing the most common application areas.

5.3.1 Parameter Estimation

The parameter estimation described in publication 8 was done using the *downhill simplex method* [100] which only uses function evaluations and not derivatives. This is not optimal since each evaluation of requires that the numerical solver takes multiple steps. Re-implementing the simulation algorithm using AD would eliminate this problem and allow the optimization algorithm to scale better to a larger number of parameters.

5.3.2 Control

The work presented in section 3.5 on the control of Heating, Ventilation, and Air Conditioning (HVAC) systems is still under progress. There are several ways that SciML methods can be used to design controllers for such as system. One way would be to treat the control inputs to the system as parameters of an optimization problem that we could solve using gradient descent. Solving this at each step would lead to a classical MPC problem. An extension of this idea is to use a NN as a surrogate of the system's dynamics to speed up the optimization step as described in [111]. Another approach would be to use a conventional PID controller, but treat the coefficients as parameters of an optimization problem. Finally, the same idea could be applied using a NN, again tuning the parameters using gradient-based optimization.

5.3.3 Design Optimization

Gradient-based optimization can also be used to optimize the design of an engineered system. The loss function can be defined such that it measures the performance and the price of the system for a given choice of parameters. For instance, we could consider the task of choosing the optimal wheel size for the agricultural robot seen in fig. 3.20. THE procedure for optimizing the wheel size would be picking some initial guess for what a good wheel size would be, then simulate the system for a period of time given, compute the loss, then differentiate the loss with respect to the parameter describing the wheel size. It would be interesting to examine the effectiveness of this approach compared to ad-hoc design optimization.

5.4 Thank you

All that remains is to thank the reader for their attention. Hopefully it managed to convey the most important concepts from the field of SciML and have convinced you of the potential it has when applied to the modeling and simulation of physical systems.

The tools necessary to apply SciML are already here today. What remains is to apply it at a larger scale to develop robust methods and gain insight into the requirements and limitations of the techniques — just as researchers have done for image and natural language processing.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-scale Machine Learning on Heterogeneous Systems.* 2015. URL: https://www.tensorflow.org/.

[2] Abdul Afram and Farrokh Janabi-Sharifi. "Theory and Applications of HVAC Control Systems – A Review of Model Predictive Control (MPC)". In: *Building and Environment* 72 (Feb. 1, 2014), pp. 343–355. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2013.11.016. URL: https://www.sciencedirect.com/science/article/pii/S0360132313003363 (visited on 03/13/2023).

[3] R. K. Al Seyab and Y. Cao. "Nonlinear System Identification for Predictive Control Using Continuous Time Recurrent Neural Networks and Automatic Differentiation". In: *Journal of Process Control* 18.6 (July 1, 2008), pp. 568–581. ISSN: 0959-1524. DOI: 10.1016/j.jprocont.2007.10.012. URL: http://www.sciencedirect.com/science/article/pii/S095915240700159X (visited on 09/08/2020).

[4] Andreas Junghanns et al. "The Functional Mock-up Interface 3.0 - New Features Enabling New Applications". In: 14th Modelica Conference 2021. Sept. 27, 2021, pp. 17–26. DOI: 10.3384/ecp2118117. URL: https://ecp.ep.liu.se/index.php/modelica/article/view/178 (visited on 05/23/2023).

[5] D. K. Arrowsmith and C. M. Place. *Dynamical Systems: Differential Equations, Maps and Chaotic Behaviour.* Dordrecht: Springer Netherlands, 1992. ISBN: 978-94-011-2388-4.

[6] Karl Johan Aström and Richard M Murray. *Feedback Systems: An Introduction for Scientists and Engineers.* 2010. ISBN: 978-1-4008-2873-9. URL: https://doi.org/10.1515/9781400828739 (visited on 05/06/2021).

[7] Jan Awrejcewicz. *Ordinary Differential Equations and Mechanical Systems.* Cham: Springer International Publishing, 2014. ISBN: 978-3-319-07658-4 978-3-319-07659-1. DOI: 10.1007/978-3-319-07659-1. URL:

https://link.springer.com/10.1007/978-3-319-07659-1 (visited on 05/22/2023).

[8] Anthony Bagnall et al. "The Great Time Series Classification Bake off: A Review and Experimental Evaluation of Recent Algorithmic Advances". In: *Data Mining and Knowledge Discovery* 31.3 (May 1, 2017), pp. 606–660. ISSN: 1573-756X. DOI: 10.1007/s10618-016-0483-9. URL: https://doi.org/10.1007/s10618-016-0483-9 (visited on 05/12/2020).

[9] Nathan Baker et al. "Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence". In: (Feb. 2019). DOI: 10.2172/1478744. URL: https://www.osti.gov/biblio/1478744.

[10] Jerry Banks et al. *Discrete-Event System Simulation*. 5th edition. Upper Saddle River: Pearson, June 26, 2009. 648 pp. ISBN: 978-0-13-606212-7.

[11] Nick Battle et al. "Towards a Static Check of FMUs in VDM-SL". In: *Formal Methods. FM 2019 International Workshops*. Ed. by Emil Sekerinski et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 272–288. ISBN: 978-3-030-54997-8. DOI: 10.1007/978-3-030-54997-8_18.

[12] Atilim Gunes Baydin et al. "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43. ISSN: 1533-7928. URL: http://jmlr.org/papers/v18/17-468.html (visited on 03/17/2023).

[13] Gordon Bell, Tony Hey, and Alex Szalay. "Beyond the Data Deluge". In: *Science (New York, N.Y.)* 323.5919 (2009), pp. 1297–1298. DOI: 10.1126/science.1170411. eprint: https://www.science.org/doi/pdf/10.1126/science.1170411. URL: https://www.science.org/doi/abs/10.1126/science.1170411.

[14] Katharina Bieker et al. "Deep Model Predictive Control with Online Learning for Complex Physical Systems". May 24, 2019. arXiv: 1905.10094 [cs, math, stat]. URL: http://arxiv.org/abs/1905.10094 (visited on 06/17/2020).

[15] Christopher Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York: Springer-Verlag, 2006. ISBN: 978-0-387-31073-2. URL: https://www.springer.com/gp/book/9780387310732 (visited on 11/04/2020).

[16] Ekaba Bisong and Ekaba Bisong. “Google Colaboratory”. In: *Building machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners* (2019), pp. 59–64.

[17] T. Blockwitz et al. “Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models”. In: *Proceedings*. 9th International Modelica Conference. München, 2012. URL: https://elib.dlr.de/78486/ (visited on 03/08/2023).

[18] James Bradbury et al. *JAX: Composable Transformations of Python+NumPy Programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[19] Dag Brück et al. “Dymola for Multi-Engineering Modeling and Simulation”. In: *Proceedings of Modelica*. Vol. 2002. Citeseer. 2002.

[20] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge: Cambridge University Press, 2019. ISBN: 978-1-108-42209-3. DOI: 10.1017/9781108380690. URL: https://www.cambridge.org/core/books/datadriven-science-and-engineering/77D52B171B60A496EAFE4DB662AD (visited on 10/18/2021).

[21] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. 2nd ed. Cambridge: Cambridge University Press, 2022. DOI: 10.1017/9781009089517.

[22] *Buildings – Analysis*. IEA. URL: https://www.iea.org/reports/buildings (visited on 03/13/2023).

[23] Richard L. Burden, J. Douglas Faires, and Annette M. Burden. *Numerical Analysis*. Tenth edition. Boston, MA: Cengage Learning, 2016. 896 pp. ISBN: 978-1-305-25366-7.

[24] Giuseppe Carleo et al. “Machine Learning and the Physical Sciences”. In: *Reviews of Modern Physics* 91.4 (Dec. 6, 2019), p. 045002. ISSN: 0034-6861, 1539-0756. DOI: 10.1103/RevModPhys.91.045002. arXiv: 1903.10563. URL: http://arxiv.org/abs/1903.10563 (visited on 06/15/2020).

[25] François Edouard Cellier. *Continuous System Modeling*. Springer Science & Business Media, 1991.

[26] François Edouard Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer Science & Business Media, 2006. ISBN: 978-0-387-26102-7.

[27] Ricky T. Q. Chen et al. “Neural Ordinary Differential Equations”. Dec. 13, 2019. arXiv: `1806.07366 [cs, stat]`. URL: `http://arxiv.org/abs/1806.07366` (visited on 05/12/2020).

[28] Edwin Kah Pin Chong and Stanislaw H. Żak. *An Introduction to Optimization.* Fourth edition. Wiley Series in Discrete Mathematics and Optimization. Hoboken, New Jersey: Wiley, 2013. 622 pp. ISBN: 978-1-118-27901-4.

[29] Erick De la Rosa, Wen Yu, and Xiaoou Li. “Nonlinear System Modeling with Deep Neural Networks and Autoencoders Algorithm”. In: *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC).* 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC). Budapest, Hungary: IEEE, Oct. 2016, pp. 002157–002162. ISBN: 978-1-5090-1897-0. DOI: `10.1109/SMC.2016.7844558`. URL: `http://ieeexplore.ieee.org/document/7844558/` (visited on 06/08/2020).

[30] Filipe de Avila Belbute-Peres et al. “End-to-End Differentiable Physics for Learning and Control”. In: *Advances in neural information processing systems* 31 (2018).

[31] Jonas Degrave et al. “A Differentiable Physics Engine for Deep Learning in Robotics”. In: *Frontiers in Neurorobotics* 13 (2019). ISSN: 1662-5218. DOI: `10.3389/fnbot.2019.00006`. URL: `https://www.frontiersin.org/articles/10.3389/fnbot.2019.00006/full` (visited on 07/09/2020).

[32] Jia Deng et al. “ImageNet: A Large-Scale Hierarchical Image Database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition.* 2009, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848`.

[33] Joachim Denil et al. “The Experiment Model and Validity Frame in M&S”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation.* TMS/DEVS ’17. San Diego, CA, USA: Society for Computer Simulation International, Apr. 23, 2017, pp. 1–12.

[34] Moritz Diehl et al. “Real-Time Optimization and Nonlinear Model Predictive Control of Processes Governed by Differential-Algebraic Equations”. In: *Journal of Process Control* 12.4 (2002), pp. 577–585. ISSN: 0959-1524. DOI: `10.1016/S0959-1524(01)00023-3`. URL: `https://www.sciencedirect.com/science/article/pii/S0959152401000233`.

[35] Finale Doshi-Velez and Been Kim. *Towards A Rigorous Science of Interpretable Machine Learning.* Mar. 2, 2017. DOI: 10.48550/arXiv.1702.08608. arXiv: 1702.08608 [cs, stat]. URL: http://arxiv.org/abs/1702.08608 (visited on 07/06/2023). preprint.

[36] Jan Drgona, Aaron Tuor, and Draguna Vrabie. "Constrained Physics-Informed Deep Learning for Stable System Identification and Control of Unknown Linear Systems". Aug. 17, 2020. arXiv: 2004.11184 [cs, eess]. URL: http://arxiv.org/abs/2004.11184 (visited on 08/19/2020).

[37] Ján Drgoňa et al. "All You Need to Know about Model Predictive Control for Buildings". In: *Annual Reviews in Control* 50 (2020), pp. 190–232. ISSN: 1367-5788. DOI: 10.1016/j.arcontrol.2020.09.001. URL: https://www.sciencedirect.com/science/article/pii/S1367578820300584.

[38] Poul O Fanger et al. "Thermal Comfort. Analysis and Applications in Environmental Engineering." In: *Thermal comfort. Analysis and applications in environmental engineering.* (1970).

[39] R. Featherstone and D. Orin. "Robot Dynamics: Equations and Algorithms". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065).* Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065). Vol. 1. Apr. 2000, 826–834 vol.1. DOI: 10.1109/ROBOT.2000.844153.

[40] Joel H. Ferziger, Milovan Perić, and Robert L. Street. *Computational Methods for Fluid Dynamics.* Cham: Springer International Publishing, 2020. ISBN: 978-3-319-99691-2 978-3-319-99693-6. DOI: 10.1007/978-3-319-99693-6. URL: http://link.springer.com/10.1007/978-3-319-99693-6 (visited on 02/10/2022).

[41] John Fitzgerald et al. "Cyber-Physical Systems Design: Formal Foundations, Methods and Integrated Tool Chains". In: *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering.* 2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering. May 2015, pp. 40–46. DOI: 10.1109/FormaliSE.2015.14.

[42] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems.* Eighth edition. Ny, NY: Pearson, 2019. 902 pp. ISBN: 978-0-13-468571-7.

[43] Jonathan Friedman and Jason Ghidella. “Using Model-Based Design for Automotive Systems Engineering - Requirements Analysis of the Power Window Example”. In: *Transactions Journal of Passenger Cars: Electronic and Electrical Systems*. SAE 2006 World Congress & Exhibition. Vol. 115. Automotive Systems Engineering. Detroit, USA: SAE Technical Paper, Apr. 3, 2006, p. 8. ISBN: 0148-7191. DOI: 10.4271/2006-01-1217.

[44] Ken-ichi Funahashi and Yuichi Nakamura. “Approximation of Dynamical Systems by Continuous Time Recurrent Neural Networks”. In: *Neural Networks* 6.6 (Jan. 1, 1993), pp. 801–806. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(05)80125-X. URL: http://www.sciencedirect.com/science/article/pii/S089360800580125X (visited on 07/19/2020).

[45] Zhiwei Gao, Carlo Cecati, and Steven X. Ding. “A Survey of Fault Diagnosis and Fault-Tolerant Techniques—Part I: Fault Diagnosis With Model-Based and Signal-Based Approaches”. In: *IEEE Transactions on Industrial Electronics* 62.6 (June 2015), pp. 3757–3767. ISSN: 1557-9948. DOI: 10.1109/TIE.2015.2417501.

[46] Carlos E. García, David M. Prett, and Manfred Morari. “Model Predictive Control: Theory and Practice—A Survey”. In: *Automatica* 25.3 (1989), pp. 335–348. ISSN: 0005-1098. DOI: 10.1016/0005-1098(89)90002-2. URL: https://www.sciencedirect.com/science/article/pii/0005109889900022.

[47] C W Gear and O Osterby. “Solving Ordinary Differential Equations with Discontinuities”. In: *ACM Trans. Math. Softw.* 10.1 (Jan. 1984), pp. 23–44. ISSN: 0098-3500. DOI: 10.1145/356068.356071. URL: http://doi.acm.org/10.1145/356068.356071.

[48] Neil A. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1999. 268 pp. ISBN: 978-0-521-57095-4. Google Books: zYAcGbp17nYC.

[49] Cláudio Gomes et al. “Co-Simulation: A Survey”. In: *ACM Computing Surveys* 51 (May 23, 2018). DOI: 10.1145/3179993.

[50] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[51] Matthias Gries. “Methods for Evaluating and Covering the Design Space during Early Design Development”. In: *Integration* 38.2 (Dec. 1, 2004), pp. 131–183. ISSN: 0167-9260. DOI: 10.1016/j.vlsi.2004.

06.001. URL: https://www.sciencedirect.com/science/article/pii/S016792600400032X (visited on 05/26/2023).

[52] Jiuxiang Gu et al. "Recent Advances in Convolutional Neural Networks". In: *Pattern recognition* 77 (2018), pp. 354–377.

[53] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 14. Springer-Verlag Berlin Heidelberg, 1996. ISBN: 3-540-60452-9.

[54] Charles R. Harris et al. "Array Programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[55] L. I. Hatledal et al. "A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface". In: *IEEE Access* 7 (2019), pp. 109328–109339. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2933275.

[56] Lars Ivar Hatledal, Houxiang Zhang, and Frederic Collonval. "Enabling Python Driven Co-Simulation Models with PythonFMU." In: *ECMS*. 2020, pp. 235–239.

[57] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016, pp. 770–778. URL: http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html (visited on 05/18/2020).

[58] Thomas A. Henzinger and Joseph Sifakis. "The Embedded Systems Design Challenge". In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 1–15. ISBN: 978-3-540-37216-5. DOI: 10.1007/11813040_1.

[59] Tony Hey and Anne Trefethen. "The Fourth Paradigm 10 Years On". In: *Informatik Spektrum* 42.6 (Jan. 1, 2020), pp. 441–447. ISSN: 1432-122X. DOI: 10.1007/s00287-019-01215-9. URL: https://doi.org/10.1007/s00287-019-01215-9 (visited on 06/28/2023).

[60] Tony Hey et al. *The Fourth Paradigm: Data-intensive Scientific Discovery*. Microsoft Research, Oct. 2009. ISBN: 978-0-9825442-0-4. URL: https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery/.

[61] Yuanming Hu et al. *DiffTaichi: Differentiable Programming for Physical Simulation.* Feb. 14, 2020. DOI: 10.48550/arXiv.1910.00935. arXiv: 1910.00935 [physics, stat]. URL: http://arxiv.org/abs/1910.00935 (visited on 06/12/2023). preprint.

[62] Zhihao Jiang et al. "Closed-Loop Verification of Medical Devices with Model Abstraction and Refinement". In: *International Journal on Software Tools for Technology Transfer* 16.2 (Apr. 2014), pp. 191–213. ISSN: 1433-2787. DOI: 10.1007/s10009-013-0289-7. URL: https://doi.org/10.1007/s10009-013-0289-7.

[63] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement Learning: A Survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[64] George Em Karniadakis et al. "Physics-Informed Machine Learning". In: *Nature Reviews Physics* (May 24, 2021), pp. 1–19. ISSN: 2522-5820. DOI: 10.1038/s42254-021-00314-5. URL: https://www.nature.com/articles/s42254-021-00314-5 (visited on 05/31/2021).

[65] Anuj Karpatne et al. "Theory-Guided Data Science: A New Paradigm for Scientific Discovery from Data". In: *IEEE Transactions on Knowledge and Data Engineering* 29.10 (Oct. 1, 2017), pp. 2318–2331. ISSN: 1041-4347. DOI: 10.1109/TKDE.2017.2720168. URL: http://ieeexplore.ieee.org/document/7959606/ (visited on 06/24/2020).

[66] Patrick Kidger. "On Neural Differential Equations". Feb. 4, 2022. arXiv: 2202.02435 [cs, math, stat]. URL: http://arxiv.org/abs/2202.02435 (visited on 02/16/2022).

[67] Thomas Kluyver et al. *Jupyter Notebooks-a Publishing Format for Reproducible Computational Workflows.* Vol. 2016. 2016.

[68] Ernesto Kofman and Sergio Junco. "Quantized-State Systems: A DEVS Approach for Continuous System Simulation". In: *Transactions of The Society for Modeling and Simulation International* 18.3 (2001), pp. 123–132. ISSN: 0740-6797.

[69] Slawomir Koziel and Anna Pietrenko-Dabrowska. "Basics of Data-Driven Surrogate Modeling". In: *Performance-Driven Surrogate Modeling of High-Frequency Structures.* Cham: Springer International Publishing, 2020, pp. 23–58. ISBN: 978-3-030-38925-3 978-3-030-38926-0. DOI: 10.1007/978-3-030-38926-0_2. URL: http://link.springer.com/10.1007/978-3-030-38926-0_2 (visited on 12/10/2020).

[70] R. Kübler and W. Schiehlen. “Two Methods of Simulator Coupling”. In: *Mathematical and Computer Modelling of Dynamical Systems* 6.2 (June 1, 2000), pp. 93–113. ISSN: 1387-3954. DOI: 10.1076/1387-3954(200006)6:2;1-M;FT093. URL: https://www.tandfonline.com/doi/abs/10.1076/1387-3954%28200006%296%3A2%3B1-M%3BFT093 (visited on 03/24/2021).

[71] Vladimir Kvrgic and Jelena Vidakovic. “Efficient Method for Robot Forward Dynamics Computation”. In: *Mechanism and Machine Theory* 145 (Mar. 1, 2020), p. 103680. ISSN: 0094-114X. DOI: 10.1016/j.mechmachtheory.2019.103680. URL: https://www.sciencedirect.com/science/article/pii/S0094114X19322864 (visited on 05/23/2023).

[72] Jeffrey Larson, Matt Menickelly, and Stefan M. Wild. “Derivative-Free Optimization Methods”. In: *Acta Numerica* 28 (May 1, 2019), pp. 287–404. ISSN: 0962-4929, 1474-0508. DOI: 10.1017/S0962492919000060. arXiv: 1904.11585 [math]. URL: http://arxiv.org/abs/1904.11585 (visited on 06/26/2023).

[73] Alexander Lavin et al. *Simulation Intelligence: Towards a New Generation of Scientific Methods*. Nov. 27, 2022. DOI: 10.48550/arXiv.2112.03235. arXiv: 2112.03235 [cs]. URL: http://arxiv.org/abs/2112.03235 (visited on 07/19/2023). preprint.

[74] Y. Lecun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[75] Edward A. Lee. “Cyber Physical Systems: Design Challenges”. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). May 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

[76] Christian Legaard et al. “Constructing Neural Network Based Models for Simulating Dynamical Systems”. In: *ACM Computing Surveys* 55.11 (Nov. 30, 2023), pp. 1–34. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3567591. URL: https://dl.acm.org/doi/10.1145/3567591 (visited on 02/27/2023).

[77] Ian Lenz, Ross Knepper, and Ashutosh Saxena. “DeepMPC: Learning Deep Latent Features for Model Predictive Control”. In: *Proceedings of Robotics: Science and Systems*. Rome, Italy, July 2015. DOI: 10.15607/RSS.2015.XI.012.

[78] Sihan Li et al. *Demystifying ResNet.* May 20, 2017. DOI: 10.48550/arXiv.1611.01186. arXiv: 1611.01186 [cs, stat]. URL: http://arxiv.org/abs/1611.01186 (visited on 07/18/2023). preprint.

[79] Lennart Ljung. “Perspectives on System Identification”. In: *IFAC Proceedings Volumes.* 17th IFAC World Congress 41.2 (Jan. 1, 2008), pp. 7172–7184. ISSN: 1474-6670. DOI: 10.3182/20080706-5-KR-1001.01215. URL: http://www.sciencedirect.com/science/article/pii/S1474667016400984 (visited on 06/21/2020).

[80] Lennart Ljung. *System Identification (2nd Ed.): Theory for the User.* USA: Prentice Hall PTR, 1999. 609 pp. ISBN: 978-0-13-656695-3.

[81] Matthias Lorenzen, Mark Cannon, and Frank Allgöwer. “Robust MPC with Recursive Model Update”. In: *Automatica* 103 (May 1, 2019), pp. 461–471. ISSN: 0005-1098. DOI: 10.1016/j.automatica.2019.02.023. URL: https://www.sciencedirect.com/science/article/pii/S0005109819300731 (visited on 06/12/2023).

[82] Frank D Macías-Escrivá et al. “Self-Adaptive Systems: A Survey of Current Approaches, Research Challenges and Applications”. In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279.

[83] H. Madsen and J. Holst. “Estimation of Continuous-Time Models for the Heat Dynamics of a Building”. In: *Energy and Buildings* 22.1 (Mar. 1, 1995), pp. 67–79. ISSN: 0378-7788. DOI: 10.1016/0378-7788(94)00904-X. URL: https://www.sciencedirect.com/science/article/pii/037877889400904X (visited on 03/14/2023).

[84] *Map of Control.* Engineering Media. URL: https://engineeringmedia.com/map-of-control (visited on 06/21/2023).

[85] J. E. Marsden and M. West. “Discrete Mechanics and Variational Integrators”. In: *Acta Numerica* 10 (May 2001), pp. 357–514. ISSN: 0962-4929, 1474-0508. DOI: 10.1017/S096249290100006X. URL: https://www.cambridge.org/core/product/identifier/S096249290100006X/type/journal_article (visited on 02/20/2020).

[86] Romit Maulik, Bethany Lusch, and Prasanna Balaprakash. “Reduced-Order Modeling of Advection-Dominated Systems with Recurrent Neural Networks and Convolutional Autoencoders”. In: *Physics of Fluids* 33.3 (Mar. 1, 2021), p. 037106. ISSN: 1070-6631. DOI: 10.1063/5.0039986. URL: https://aip.scitation.org/doi/full/10.1063/5.0039986 (visited on 12/09/2021).

[87] Aaron Meurer et al. "SymPy: Symbolic Computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[88] Mehrdad Moradi et al. "Optimizing Fault Injection in FMI Co-simulation". In: *Proceedings of the 2019 Summer Simulation Conference*. Berlin, Germany: Society for Computer Simulation International, 2019, p. 12. DOI: 10.5555/3374138.3374170.

[89] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0-262-01802-0.

[90] Claus Ballegaard Nielsen et al. "Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions". In: *ACM Computing Surveys* 48.2 (Sept. 24, 2015), 18:1–18:41. ISSN: 0360-0300. DOI: 10.1145/2794381. URL: https://doi.org/10.1145/2794381 (visited on 06/21/2020).

[91] Frank Noé et al. "Machine Learning for Molecular Simulation". In: *Annual Review of Physical Chemistry* 71.1 (2020), pp. 361–390. DOI: 10.1146/annurev-physchem-042018-052331. pmid: 32092281. URL: https://doi.org/10.1146/annurev-physchem-042018-052331 (visited on 09/02/2020).

[92] Shimon Y. Nof, ed. *Handbook of Industrial Robotics*. 2nd ed. New York: John Wiley, 1999. 1348 pp. ISBN: 978-0-471-17783-8.

[93] Katharina Ott et al. "When Are Neural ODE Solutions Proper ODEs?" July 30, 2020. arXiv: 2007.15386 [cs, stat]. URL: http://arxiv.org/abs/2007.15386 (visited on 09/08/2020).

[94] *Papers with Code - Browse the State-of-the-Art in Machine Learning*. URL: https://paperswithcode.com/sota (visited on 07/01/2023).

[95] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html (visited on 02/27/2023).

[96] Fabian Pedregosa et al. "Scikit-Learn: Machine Learning in Python". In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[97] Ludovic Pintard et al. “Fault Injection in the Automotive Standard ISO 26262: An Initial Approach”. In: *Dependable Computing.* Ed. by Marco Vieira et al. Vol. 7869. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 126–133. ISBN: 978-3-642-38788-3 978-3-642-38789-0. DOI: 10.1007/978-3-642-38789-0_11. URL: http://link.springer.com/10.1007/978-3-642-38789-0_11 (visited on 03/05/2019).

[98] Alessio Plebe and Giorgio Grasso. “The Unbearable Shallow Understanding of Deep Learning”. In: *Minds and Machines* 29.4 (Dec. 1, 2019), pp. 515–553. ISSN: 1572-8641. DOI: 10.1007/s11023-019-09512-8. URL: https://doi.org/10.1007/s11023-019-09512-8 (visited on 12/07/2020).

[99] Andrei D Polyanin and Valentin F Zaitsev. *Handbook of Ordinary Differential Equations: Exact Solutions, Methods, and Problems.* CRC Press, 2017.

[100] William H. Press. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, Sept. 6, 2007. 1195 pp. ISBN: 978-0-521-88068-8. Google Books: 1aAOdzK3FegC.

[101] Alfio Quarteroni and Gianluigi Rozza, eds. *Reduced Order Methods for Modeling and Computational Reduction.* Cham: Springer International Publishing, 2014. ISBN: 978-3-319-02089-1 978-3-319-02090-7. DOI: 10.1007/978-3-319-02090-7. URL: http://link.springer.com/10.1007/978-3-319-02090-7 (visited on 03/22/2023).

[102] Christopher Rackauckas et al. “Universal Differential Equations for Scientific Machine Learning”. Aug. 6, 2020. arXiv: 2001.04385 [cs, math, q-bio, stat]. URL: http://arxiv.org/abs/2001.04385 (visited on 09/07/2020).

[103] M. Raissi, P. Perdikaris, and G. E. Karniadakis. “Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations”. In: *Journal of Computational Physics* 378 (Feb. 1, 2019), pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.10.045. URL: http://www.sciencedirect.com/science/article/pii/S0021999118307125 (visited on 08/19/2020).

[104] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. “Hidden Fluid Mechanics: Learning Velocity and Pressure Fields from Flow Visualizations”. In: *Science (New York, N.Y.)* 367.6481 (Feb. 28, 2020), pp. 1026–1030. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.

aaw4741. pmid: 32001523. URL: https://science.sciencemag.org/content/367/6481/1026 (visited on 05/18/2020).

[105] Luis Miguel Rios and Nikolaos V. Sahinidis. “Derivative-Free Optimization: A Review of Algorithms and Comparison of Software Implementations”. In: *Journal of Global Optimization* 56.3 (July 1, 2013), pp. 1247–1293. ISSN: 1573-2916. DOI: 10.1007/s10898-012-9951-y. URL: https://doi.org/10.1007/s10898-012-9951-y (visited on 06/26/2023).

[106] Stewart Robinson. *Simulation: The Practice of Model Development and Use*. Second edition. Houndmills, Basingstoke, Hampshire, UK ; New York, NY: Palgrave Macmillan, 2014. 367 pp. ISBN: 978-1-137-32802-1.

[107] Ribana Roscher et al. “Explainable Machine Learning for Scientific Insights and Discoveries”. In: *IEEE access : practical innovations, open solutions* 8 (2020), pp. 42200–42216. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2976199. arXiv: 1905.08883. URL: http://arxiv.org/abs/1905.08883 (visited on 09/01/2020).

[108] Simon Rouchier, Mickaël Rabouille, and Pierre Oberlé. “Calibration of Simplified Building Energy Models for Parameter Estimation and Forecasting: Stochastic versus Deterministic Modelling”. In: *Building and Environment* 134 (Apr. 15, 2018), pp. 181–190. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2018.02.043. URL: https://www.sciencedirect.com/science/article/pii/S036013231830115X (visited on 03/14/2023).

[109] Olga Russakovsky et al. *ImageNet Large Scale Visual Recognition Challenge*. Jan. 29, 2015. DOI: 10.48550/arXiv.1409.0575. arXiv: 1409.0575 [cs]. URL: http://arxiv.org/abs/1409.0575 (visited on 07/18/2023). preprint.

[110] RD Russell and Lawerence F Shampine. “A Collocation Method for Boundary Value Problems”. In: *Numerische Mathematik* 19 (1972), pp. 1–28.

[111] Tim Salzmann et al. “Real-Time Neural MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms”. In: *IEEE Robotics and Automation Letters* 8.4 (Apr. 2023), pp. 2397–2404. ISSN: 2377-3766. DOI: 10.1109/LRA.2023.3246839.

[112] Masa-aki Sato and Yoshihiko Murakami. "Learning Nonlinear Dynamics by Recurrent Neural". In: *Some Problems on the Theory of Dynamical Systems in Applied Sciences-Proceedings of the Symposium.* Vol. 10. 1991, p. 49.

[113] Klaus Schittkowski. *Numerical Data Fitting in Dynamical Systems: A Practical Introduction with Applications and Software.* Vol. 77. Springer Science & Business Media, 2002.

[114] Dieter Schramm, Wildan Lalo, and Michael Unterreiner. "Application of Simulators and Simulation Tools for the Functional Design of Mechatronic Systems". In: *Solid State Phenomena* 166–167 (Sept. 2010), pp. 1–14. ISSN: 1662-9779. DOI: `10.4028/www.scientific.net/SSP.166-167.1`. URL: `http://www.scientific.net/SSP.166-167.1`.

[115] D.R. Seidl and R.D. Lorenz. "A Structure by Which a Recurrent Neural Network Can Approximate a Nonlinear Dynamic System". In: *IJCNN-91-Seattle International Joint Conference on Neural Networks.* IJCNN-91-Seattle International Joint Conference on Neural Networks. Vol. ii. July 1991, 709–714 vol.2. DOI: `10.1109/IJCNN.1991.155422`.

[116] B. Sohlberg and E.W. Jacobsen. "GREY BOX MODELLING – BRANCHES AND EXPERIENCES". In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 11415–11420. ISSN: 14746670. DOI: `10.3182/20080706-5-KR-1001.01934`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S1474667016408025` (visited on 05/12/2020).

[117] *Space Cooling – Analysis.* IEA. URL: `https://www.iea.org/reports/space-cooling` (visited on 03/13/2023).

[118] Shlomo Sternberg. *Dynamical Systems.* Mineola, N.Y: Dover Publications, 2010. 265 pp. ISBN: 978-0-486-47705-3.

[119] Hyung Ju Suh et al. "Do Differentiable Simulators Give Better Policy Gradients?" In: *Proceedings of the 39th International Conference on Machine Learning.* International Conference on Machine Learning. PMLR, June 28, 2022, pp. 20668–20696. URL: `https://proceedings.mlr.press/v162/suh22b.html` (visited on 06/18/2023).

[120] Makoto Takamoto et al. *PDEBENCH: An Extensive Benchmark for Scientific Machine Learning.* Mar. 13, 2023. DOI: `10.48550/arXiv.2210.07182`. arXiv: `2210.07182 [physics]`. URL: `http://arxiv.org/abs/2210.07182` (visited on 07/19/2023). preprint.

[121] *This Is Python Version 3.12.0 Alpha 6*. Python, Mar. 9, 2023. URL: https://github.com/python/cpython (visited on 03/09/2023).

[122] Nils Thuerey et al. "Physics-Based Deep Learning". Sept. 11, 2021. arXiv: 2109.05237 [physics]. URL: http://arxiv.org/abs/2109.05237 (visited on 09/30/2021).

[123] Tobias Thummerer, Lars Mikelsons, and Josef Kircher. "NeuralFMU: Towards Structural Integration of FMUs into Neural Networks". In: 14th Modelica Conference 2021. Sept. 27, 2021, pp. 297–306. DOI: 10.3384/ecp21181297. URL: https://ecp.ep.liu.se/index.php/modelica/article/view/207 (visited on 05/26/2023).

[124] *Tools — Functional Mock-up Interface*. URL: https://fmi-standard.org/tools/ (visited on 06/26/2023).

[125] Martin Törngren and Ulf Sellgren. "Complexity Challenges in Development of Cyber-Physical Systems". In: *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 478–503. ISBN: 978-3-319-95246-8. DOI: 10.1007/978-3-319-95246-8_27. URL: https://doi.org/10.1007/978-3-319-95246-8_27 (visited on 03/12/2023).

[126] Hans Vangheluwe, Juan De Lara, and Pieter J. Mosterman. "An Introduction to Multi-Paradigm Modelling and Simulation". In: *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal*. Vol. 21. 1. 2002.

[127] Arun Verma. "An Introduction to Automatic Differentiation". In: *Current Science* 78.7 (2000), pp. 804–807. ISSN: 0011-3891. JSTOR: 24103956. URL: https://www.jstor.org/stable/24103956 (visited on 02/26/2023).

[128] Laura von Rueden et al. "Combining Machine Learning and Simulation to a Hybrid Modelling Approach: Current and Future Directions". In: *Advances in Intelligent Data Analysis XVIII*. Ed. by Michael R. Berthold, Ad Feelders, and Georg Krempl. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 548–560. ISBN: 978-3-030-44584-3. DOI: 10.1007/978-3-030-44584-3_43.

[129] Laura von Rueden et al. “Informed Machine Learning – A Taxonomy and Survey of Integrating Prior Knowledge into Learning Systems”. In: *IEEE Transactions on Knowledge and Data Engineering* 35.1 (Jan. 2023), pp. 614–633. ISSN: 1558-2191. DOI: 10.1109/TKDE.2021.3079836.

[130] Sifan Wang, Yujun Teng, and Paris Perdikaris. “Understanding and Mitigating Gradient Pathologies in Physics-Informed Neural Networks”. Jan. 13, 2020. arXiv: 2001.04536 [cs, math, stat]. URL: http://arxiv.org/abs/2001.04536 (visited on 03/17/2021).

[131] Sifan Wang, Xinling Yu, and Paris Perdikaris. “When and Why PINNs Fail to Train: A Neural Tangent Kernel Perspective”. July 28, 2020. arXiv: 2007.14527 [cs, math, stat]. URL: http://arxiv.org/abs/2007.14527 (visited on 03/17/2021).

[132] G. Wanner and E. Hairer. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer S. Vol. 1. Springer-Verlag, 1991.

[133] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. 3rd ed. San Diego (Calif.): Academic Press, 2019. ISBN: 978-0-12-813370-5.

[134] Xuan Zhang et al. *Artificial Intelligence for Science in Quantum, Atomistic, and Continuum Systems*. July 17, 2023. DOI: 10.48550/arXiv.2307.08423. arXiv: 2307.08423 [physics]. URL: http://arxiv.org/abs/2307.08423 (visited on 07/21/2023). preprint.

[135] Yinhao Zhu et al. “Physics-Constrained Deep Learning for High-Dimensional Surrogate Modeling and Uncertainty Quantification without Labeled Data”. In: *Journal of Computational Physics* 394 (Oct. 1, 2019), pp. 56–81. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.05.024. URL: http://www.sciencedirect.com/science/article/pii/S0021999119303559 (visited on 05/18/2020).

Part II

Publications

Chapter 6

Constructing Neural Network Based Models for Simulating Dynamical Systems

The paper presented in this chapter has been published in the peer-reviewed journal *ACM Computing Surveys*.

Constructing Neural Network Based Models for Simulating Dynamical Systems

CHRISTIAN LEGAARD, Aarhus University
THOMAS SCHRANZ and GERALD SCHWEIGER, TU Graz
JÁN DRGOŇA, Pacific Northwest National Laboratory
BASAK FALAY, AEE–Institute for Sustainable Technologies
CLÁUDIO GOMES, ALEXANDROS IOSIFIDIS, MAHDI ABKAR, and
PETER LARSEN, Aarhus University

Dynamical systems see widespread use in natural sciences like physics, biology, and chemistry, as well as engineering disciplines such as circuit analysis, computational fluid dynamics, and control. For simple systems, the differential equations governing the dynamics can be derived by applying fundamental physical laws. However, for more complex systems, this approach becomes exceedingly difficult. Data-driven modeling is an alternative paradigm that seeks to learn an approximation of the dynamics of a system using observations of the true system. In recent years, there has been an increased interest in applying data-driven modeling techniques to solve a wide range of problems in physics and engineering. This article provides a survey of the different ways to construct models of dynamical systems using neural networks. In addition to the basic overview, we review the related literature and outline the most significant challenges from numerical simulations that this modeling paradigm must overcome. Based on the reviewed literature and identified challenges, we provide a discussion on promising research areas.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Continuous simulation**; **Continuous models**; *Supervised learning by regression*; • **Applied computing** → *Physics*; *Engineering;*

Additional Key Words and Phrases: Neural ODEs, physics-informed neural networks, physics-based regularization

We acknowledge the Poul Due Jensen Foundation for funding the project Digital Twins for Cyber-Physical Systems (DiT4CPS) and Legaard would also like to acknowledge partial support from the MADE Digital project.
This research was supported by the Data Model Convergence (DMC) initiative via the Laboratory Directed Research and Development (LDRD) investments at Pacific Northwest National Laboratory (PNNL). PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DoE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL0-1830.
The Digital Twins for Cyber-Physical Systems (DiT4CPS) project was funded by the Poul Due Jensen Foundation. C. Legaard received partial support from the MADE Digital project. This research was supported by the Data Model Convergence (DMC) initiative via the Laboratory Directed Research and Development (LDRD) investments at Pacific Northwest National Laboratory (PNNL). PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DoE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL0-1830.
Authors' addresses: C. Legaard, C. Gomes, A. Iosifidis, M. Abkar, and P. Larsen, Aarhus University, Aarhus, Denmark; emails: {cml, claudio.gomes, ai}@ece.au.dk, abkar@mpe.au.dk, pgl@ece.au.dk; T. Schranz and G. Schweiger, TU Graz, Graz, Austria; emails: {thomas.schranz, gerald.schweiger}@tugraz.at; J. Drgoňa, Pacific Northwest National Laboratory, Richland, WA; email: jan.drgona@pnnl.gov; B. Falay, AEE–Institute for Sustainable Technologies, Gleisdorf, Austria; email: b.falay@aee.at.

0360-0300/2023/02-ART236 $15.00
https://doi.org/10.1145/3567591

ACM Reference format:
Christian Legaard, Thomas Schranz, Gerald Schweiger, Ján Drgoňa, Basak Falay, Cláudio Gomes, Alexandros Iosifidis, Mahdi Abkar, and Peter Larsen. 2023. Constructing Neural Network Based Models for Simulating Dynamical Systems. *ACM Comput. Surv.* 55, 11, Article 236 (February 2023), 34 pages.
https://doi.org/10.1145/3567591

1 INTRODUCTION

Mathematical models are fundamental tools for building an understanding of the physical phenomena observed in nature [13]. Not only do these models allow us to predict what the future may look like, but they also allow us to develop an understanding of what causes the observed behavior. In engineering, models are used to improve the system design [33, 118], design optimal control policy [23, 25, 35], simulate faults [84, 94], forecast future behavior [122], or assess the desired operational performance [51].

The focus of this survey is on the type of models that allow us to predict how a physical system evolves over time for a given set of conditions. Dynamical systems theory provides an essential set of tools for formalizing and studying the dynamics of this type of model. However, when studying complex physical phenomena, it becomes increasingly difficult to derive models by hand that strike an acceptable balance between accuracy and speed. This has led to the development of fields that are concerned with creating models directly from data such as *system identification* [76, 87], **machine learning (ML)** [9, 85], and, more recently, **deep learning (DL)** [40].

In recent years, the interest in DL has increased rapidly, as is evident from the volume of research being published on the topic [95]. The exact causes behind the success of **neural networks (NNs)** are hard to pinpoint. Some claim that practical factors like the availability of large quantities of data, user-friendly software frameworks [1, 93], and specialized hardware [82] are the main cause for its success, whereas others claim that the success of NNs can be attributed to their structure being well suited to solving a wide variety of problems [95].

The goal of this survey is to provide a practical guide on how to construct models of dynamical systems using NNs as primary building blocks. We do this by walking the reader through the most important classes of models found in the literature, for many of which we provide an example implementation. We put special emphasis on the process for training the models, since it differs significantly from traditional applications of DL that do not consider evolution over time. More specifically, we describe how to split the trajectories used during training, and we introduce optimization criteria suitable for simulation. After training, it is necessary to validate that the model is a good representation of the true system. Like other data-driven models, we determine the validity empirically by using a separate set of trajectories for validation. We introduce some of the most important properties and how they can be verified.

It should be emphasized that the type of model we wish to construct should allow us to obtain a simulation of the system. Rather than providing a formal definition of simulation, we refer to Figure 1, which shows several topics related to simulation that are not covered in this article.

The source code and instructions for running the experiments can be accessed at GitHub.[1]

1.1 Related Surveys

We provide an overview of existing surveys related to our work. Then we compare our work with these surveys and describe the structure of the remainder of the article.

[1]https://github.com/clegaard/deep_learning_for_dynamical_systems.

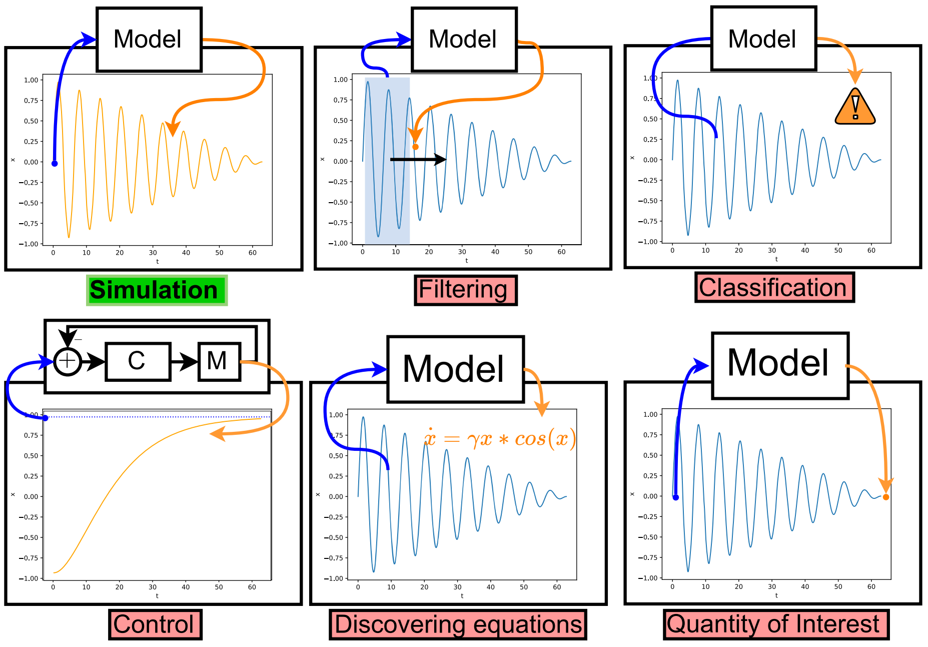


Fig. 1. *Simulation* and related application areas where ML techniques are commonly applied. The focus of the survey is exclusively on techniques that can generate a simulation based on an initial condition, as shown in the top left. Although interesting on their own, topics other than simulation are not covered by the survey. *Filtering* refers to applications where a sliding window over past observations is used to predict the next sample or some other quantity of interest. *Classification* refers to applications where a model takes a sequence of observations and produces a categorical label, for instance, indicating that the system is in an abnormal state. *Control* refers to applications where a NN-based controller is used to drive the system to a desired state. *Discovering Equations* refers to techniques based on ML that aim to discover the underlying equations of the system. *Quantity of Interest* refers to applications where an NN is used to provide a mapping from an initial condition to some quantity of interest, such as the steady-state of the system.

Application Domain. The broader topic of using ML in scientific fields has received widespread attention within several application domains [11, 12, 19, 108]. These review papers commonly focus on providing an overview of the prospective use cases of ML within their domains but put limited emphasis on how to apply the techniques in practice.

Surrogate Modeling. The field of surrogate modeling—that is, the theory and techniques used to produce faster models—is intimately related to the field of simulation with NNs. So it is important that we highlight some surveys in this field. The work of Koziel and Pietrenko-Dabrowska [61] presents a thorough introduction to data-driven surrogate modeling, which encompasses the use of NNs. Viana et al. [127] summarize advanced and yet simple statistical tools commonly used in the design automation community: (i) screening and variable reduction in both the input and the output spaces, (ii) simultaneous use of multiple surrogates, (iii) sequential sampling and optimization, and (iv) conservative estimators. Since optimization is an important use case of surrogate modeling, Forrester and Keane [31] reviewed advances in surrogate modeling in this field. Finally, with a focus on applications to water resources and building simulation, we highlight the work of Razavi et al. [105] and Westermann and Evins [135].

Prior Knowledge. One of the major trends to address some challenges arising in NN-based simulation is to encode prior knowledge such as physical constraints into the network itself or during the training process, ensuring the trained network is physically consistent. The work of Kelly et al. [54] coins this *theory-guided data science* and provides several examples of how knowledge

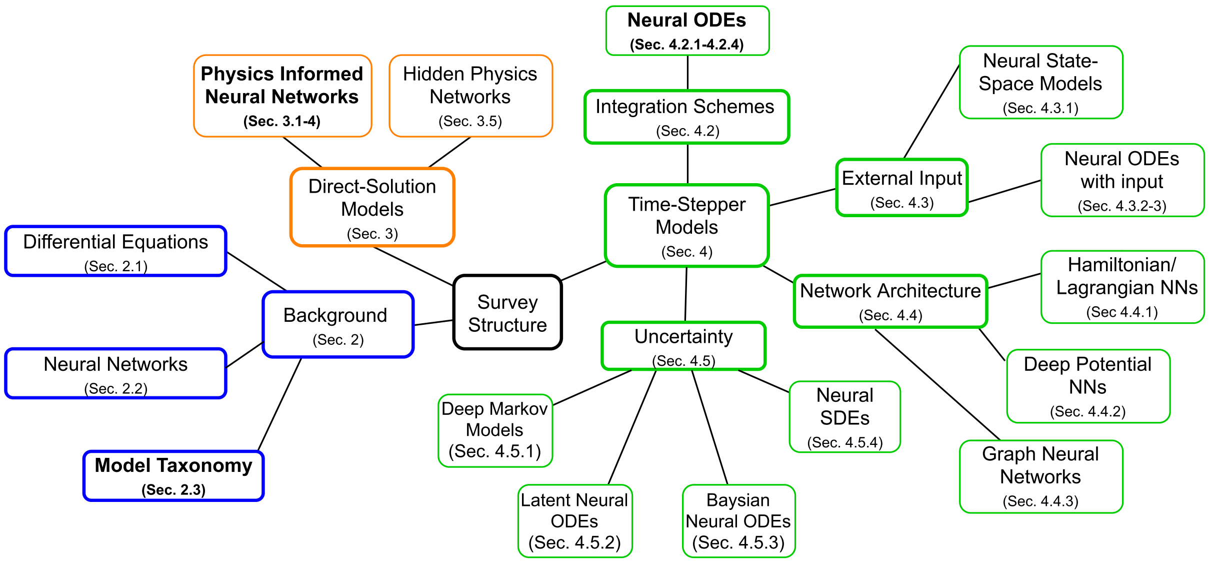


Fig. 2. A mind map of the topics and model types covered in the survey.

may be incorporated in practice. Closely related to this is the work of Rai and Sahu [100] and von Rueden et al. [128, 129], which propose a detailed taxonomy describing the various paths through which knowledge can be incorporated into a NN model.

Comparison with This Survey. Our work complements the preceding surveys by providing an in-depth review focused specifically on NNs rather than ML as a whole. The concrete example helps the reader's understanding and highlights the similarities and inherent deficiencies of each approach.

We also outline the inherent challenges of simulation and establish a relationship between numerical simulation challenges and DL-based simulation challenges. The benefit of our approach is that the reader gets the intuition behind some approaches used to incorporate knowledge into the NNs. For instance, we relate energy-conserving numerical solvers to Hamiltonian NNs, whose goal is to encode energy conservation, and we discuss concepts such as numerical stability and solver convergence, which are crucial in long-term prediction using NNs.

1.2 Survey Structure

The remainder of the article is structured according to the mind map shown in Figure 2. First, Section 2 introduces the central concepts of dynamical systems, numerical solvers, and NNs. In addition, the section proposes a taxonomy describing the fundamental differences of how models can be constructed using NNs. The following two sections are dedicated to describing the two classes of models identified in the taxonomy: *direct-solution models* and *time-stepper models* in Sections 3 and 4, respectively. For each of the two categories, we describe the following:

- The structure of the model and the mechanism used to produce simulations of a system
- How the parameters are tuned to match the behavior of the true system
- Key challenges and extensions of the model designed to address them

Following this, Section 5 discusses the advantages and limitations of the two distinct model types and outlines future research directions. Finally, Section 6 provides a brief summary of the contributions of the article and the outlined research directions.

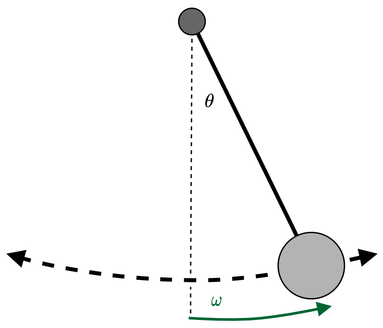


Fig. 3. The ideal pendulum system used as a case study throughout the article. The pendulum is characterized by an angle, θ, and an angular velocity, ω.

2 BACKGROUND

Models are an integral tool in natural sciences and engineering that allow us to deepen our understanding of nature or improve the design of engineered systems. One way to categorize models is by the *modeling* technique used to derive the model: *first principles* models are derived using fundamental physical laws, and *data-driven* models are created based on experimental data.

First, in Section 2.1, a running example is introduced, where we describe how differential equations can be used to model a simple mechanical system and how a solver is used to obtain a simulation. Then, Section 2.2 introduces the different ways NN-based models of the system can be constructed and trained. Finally, Section 2.3 introduces a taxonomy of the different ways NNs can be used to construct models of dynamical systems.

2.1 Differential Equations

An *ideal pendulum*, shown in Figure 3, refers to a mathematical model of a pendulum that, unlike its physical counterpart, neglects the influence of factors such as friction in the pivot or bending of the pendulum arm. The state of this system can be represented by two variables: its angle θ (expressed in radians) and its angular velocity ω. These variables correspond to a mathematical description of the system's state and are referred to as *state variables*. The way that a given point in the state-space evolves over time can be described using *differential equations*. Specifically, for the ideal pendulum, we may use the following **ordinary differential equation (ODE)**:

$$\frac{\partial^2 \theta}{\partial t^2} + \frac{g}{l} \sin \theta = 0, \tag{1}$$

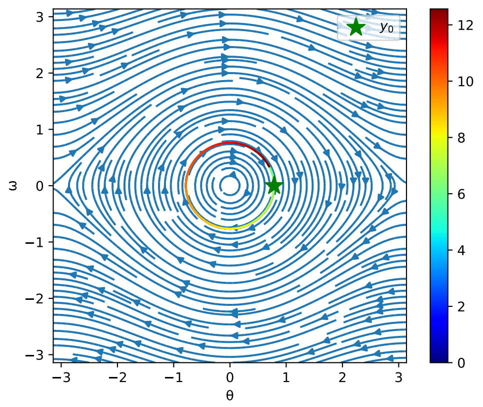
where g is the gravitational acceleration and l is the length of the pendulum arm. The ideal pendulum (Equation (1)) falls into the category of *autonomous* and *time-invariant* systems, since the system is not influenced by external stimulus and the dynamics do not change over time. Although this simplifies the notation and the way in which models can be constructed, it is not the general case. We discuss the implication of these issues in Section 4.3.1.

The equation can be rewritten as two first-order differential equations and expressed compactly using vector notation as follows:

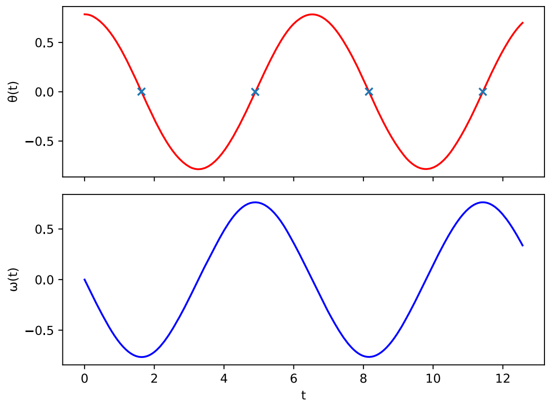
$$f(x) = \begin{bmatrix} \frac{\partial \omega}{\partial t} \\ \frac{\partial \theta}{\partial t} \end{bmatrix} = \begin{bmatrix} -\frac{g}{l} \sin \theta \\ \omega \end{bmatrix}. \tag{2}$$

where x is a vector of the system's state variables. In the context of this article, we refer to $f(x)$ as the *derivative function* or as the derivative of the system.

Although the differential equations describe how each state variable will evolve over the next time instance, they do not provide any way of determining the solution $x(t)$ on their own.



(a) Phase portrait of the ideal pendulum with a single trajectory drawn onto the phase space. The color denotes time.



(b) Solution of equation (3) for the initial condition marked with a star in figure 4(a).

Fig. 4. Diagram of the pendulum system and an example of the trajectory generated when solving the equation using a numerical solver.

Obtaining the solution of an ODE $f(x)$ given some *initial conditions* x_0 is referred to as an **initial value problem (IVP)** and can be formalized as follows:

$$\frac{\partial}{\partial t}x(t) = f(x(t)), \tag{3}$$

$$x(t_0) = x_0 \tag{4}$$

where $x(\cdot)$ is called the *solution*, $x : \mathbb{R} \to \mathbb{R}^n$, and $n \in \mathbb{N}$ is the dimension of the system's state space.

The result of solving the IVP corresponding to the pendulum can be seen in Figure 4(b), which shows how the two state variables θ and ω evolve from their initial state. An alternative view of this can be seen in the *phase portrait* in Figure 4(a).

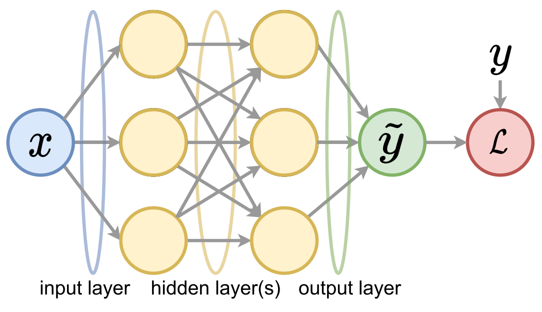
In many cases, it is impossible to find an exact analytical solution to the IVP, and instead numerical methods are used to approximate the solution. Numerical solvers are algorithms that approximate a continuous IVP, as the one in Equation (2), into a discrete-time dynamical system. These systems are often modeled with difference equations:

$$x_{i+1} = F(x_i), \tag{5}$$

where x_i represents the state vector at the i-th time point, x_{i+1} represents the next state vector, and $F : \mathbb{R}^n \to \mathbb{R}^n$ models the system behavior. Just as with ODEs, the initial state can be represented by a constraint on x_0, and the solution to Equation (5) with an initial value defined by such constraint is a function x_i defined for all $i \geq 0$. In Equation (5), time is implicitly defined as a discrete set.

We start by introducing the simplest and most intuitive numerical solver because it highlights the main challenges well. There are many numerical solvers, each presenting unique trade-offs. The reader is referred to the work of Cellier and Kofman [14] for an introduction to this topic, to the work of Hairer and Wanner [44] and Wanner and Hairer [133] for more detailed expositions on the numerical solution of ODEs and differential-algebraic system of equations (DAEs), to the work of LeVeque [69] for the numerical solution to **partial differential equations (PDEs)**, to the work of Marsden and West [78] for an overview of more advanced numerical schemes, and to the work of Kofman and Junco [60] for an introduction to quantized state solvers.

Given an IVP (Equation (3)) and a simulation step size $h > 0$, the **forward Euler (FE)** method computes a sequence in time of points $\tilde{x}_i$, where $\tilde{x}_i$ is the approximation of the solution to the



(a)

```
ỹ = 𝒩(x)
```

(b) Inference.

```
ỹ = 𝒩(x)
loss = ℒ(ỹ, y)
optimizer.step(loss)
```

(c) Training. This step is typically repeated many times for different inputs and desired output values.

Fig. 5. An FC NN is used to perform regression from an input x to y, where $\tilde{y}$ represents the approximation provided by the NN. Each layer of the network is characterized by a set of weights that are tuned during training to produce the desired output for a given input. During training, the loss function $\mathcal{L}$ is used to measure the divergence between the output produced by the network, $\tilde{y}$, and the desired output, y.

IVP at time hj: $\tilde{x}_i \approx x_i = x(hi)$. It starts from the given initial value $\tilde{x}_0 = x(0)$ and then computes iteratively:

$$\tilde{x}_{i+1} = \tilde{x}_i + hf(t_i, \tilde{x}_i), \tag{6}$$

where $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is the ODE right-hand side in Equation (2) and $t_i = hi$.

A graphical representation of the solutions IVP starting from different initial conditions can be seen in Figure 4(a). For a specific point, the solver evaluates the derivative (depicted as curved arrows in the plot) and takes a small step in this direction. Applying this process iteratively results in the full trajectory, which for the pendulum corresponds to the circle in the phase space. The circle in the phase space implies that the solution is repeating itself—that is, corresponds to an oscillation in time as seen in Figure 4(b).

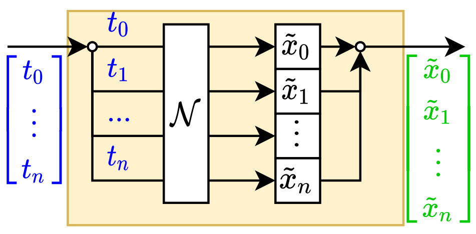
The ideal pendulum is an example of a well-studied dynamical system for which the dynamics can be described using simple ODEs that can be solved using standard solvers. Unfortunately, the simplicity of the idealized model comes at the cost of neglecting several factors that are present in a real pendulum. For example, the arm of the real pendulum may bend and energy may be lost in the pivot due to friction. The idealized model can be extended to account for these factors by incorporating models of friction and bending. However, this is time consuming, leads to a model that is harder to interpret, and does not guarantee that all factors are accounted for.

2.2 Neural Networks

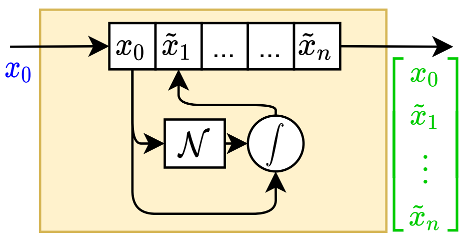
Today, the term *neural network* has come to encompass a whole family of models, which collectively have proven to be effective building blocks for solving a wide range of problems. In this article, we focus on a single class of networks, the **fully connected (FC)** NNs, due to their simplicity and the fact that they will be used to construct the models introduced in later sections. We refer the reader to the work of Goodfellow et al. [40] for a general introduction to the field of DL.

Like other data-driven models, NNs are generic structures that have no behavior specific to the problem they are being applied to before training. For this reason, it is essential to consider not only how the network produces its outputs but also how the network's parameters are tuned to solve the problem. For instance, we may consider using an FC NN to perform regression from a scalar input, x, to a scalar output, y, as shown in Figure 5(a).

We will refer to the process of producing predictions as *inference* and the process of tuning the network's weights to produce the desired results as *training*. There can be quite drastic differences



(a) Direct-solution model. An NN is used to parameterize a mapping from a time instance to the solution corresponding to that time instance.



(b) Time-stepper model. The network, $\mathcal{N}$, provides the derivative of the system at various points in state-space, which is then integrated by a numerical solver, here depicted as $\int$.

Fig. 6. Overview of two distinct model types. Direct-solution models are trained to produce a simulation without performing numerical integration explicitly. Conversely, time-stepper models use the same techniques known from numerical simulation to produce a simulation of the system.

in the complexity of the two phases, the training phase typically being the most complex and computationally intensive. During training, a loss function defines a mapping from the predicted quantity to a scalar that is a measure of how close the prediction is to the true trajectory. Differentiating the loss function with respect to the parameters of the NN allows us to update them in a way such that the loss is minimized.

Batching and Notation. During the training of an NN, we often wish to perform the forward pass individually for multiple inputs grouped in a batch. By convention, many DL frameworks treat any leading dimensions as being batches of samples. We adopt this convention as well to simplify notation. Thus, $\mathcal{N}(x) \in \mathbb{R}^{*\times n}$ when $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^n$ and $x \in \mathbb{R}^{*\times n}$ ('$*$' indicating any number of leading dimensions).

2.3 Model Taxonomy

A challenge of studying any fast-evolving research field such as DL is that the terminology used to describe important concepts and ideas may not always have converged. This is especially true in the intersection between DL, numerical simulation, and physics, due to the influx of ideas and terminology from the different fields. In the literature, there is also a tendency to focus on the success of a particular technique in a specific application, with little emphasis on explaining the inner workings and limitations of the technique. A consequence of this is that important contributions to the field become lost due to the papers being hard to digest.

In an attempt to alleviate this, we propose a simple taxonomy describing how models can be constructed consisting of two categories: *direct-solution models* and *time-stepper models*, as shown in Figure 6. Direct-solution models, described in Section 3, do not employ integration but rather produce an estimate of the state at a particular time by feeding in the time as an input to the network. Time-stepper models, found in Section 4, can be characterized by using a similar approach to numerical solvers, where the current state is used to calculate the state at some time into the future. The difference between the time-stepper and continuous models has significant implications for how the model deals with varying initial conditions and inputs. Per design, the time-stepper models handle different initial conditions and inputs, whereas direct-solution models have to be retrained. In other words, the time-stepper models learn the dynamics while the direct-solution models learn a solution to an IVP for a given initial state and set of inputs.

Table 1. Comparison of Direct-Solution Models

Name	In_{NN}	Out_{NN}	Out_{AD}	Uses Equations
Vanilla Direct-Solution	t	θ, ω		
Automatic Differentiation Direct-Solution	t	θ	ω	
Physics-Informed Neural Network	t	θ	$\omega, \partial\omega$	✓
Hidden Physics Neural Network	t	θ, l	$\omega, \partial\omega$	✓

3 DIRECT-SOLUTION MODELS

One approach to obtaining the trace of a system is to construct a model that maps a set of time instances $t \in \mathbb{R}^m$ to the solution $\tilde{x} \in \mathbb{R}^{m \times n}$. We refer to this type of model as a *direct-solution* model.

To construct the model, an NN is trained to provide an exact solution for a set of *collocation* points that are sampled from the true system. Another way to view this is that the NN acts as a trainable interpolation engine, which allows the solution to be evaluated at arbitrary points in time, not only those of the collocation points. An important limitation of this approach is that a trained model is fixed for a specific set of initial conditions. To evaluate the solution for different initial conditions, a new model would have to be trained on new data.

In the literature, this type of model is often applied to learn the dynamics of systems governed by PDEs and less frequently for systems governed by ODEs. Several factors are likely to influence this pattern of use. First, PDEs are generally harder and more computationally expensive to solve than ODEs, which provides a stronger motivation for applying NNs as a means to obtain a solution. Second, many practical uses of ODEs require that they can easily be evaluated for different initial conditions, which is not the case for direct-solution models.

Although the motivation for applying direct-solution networks may be strongest for PDEs, they can also be applied to model ODEs. The main difference is that a network to model an ODE takes time as the only input, whereas the network used to model a PDE would take both time and spatial coordinates.

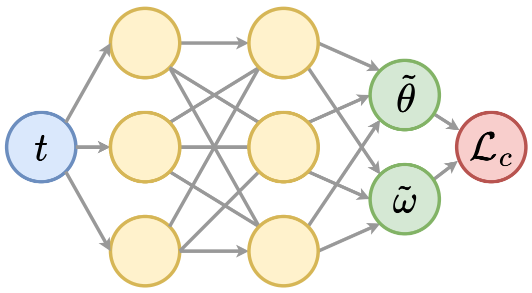
A key challenge in training direct-solution NNs is the amount of data required to reach an acceptable level of accuracy and generalization. A vanilla approach that does not leverage prior knowledge, like the one described in Section 3.2, is likely to fit the collocation points very well but fails to reproduce the underlying trend. A recent trend popularized by **physics-informed neural networks (PINNs)** [101] is to apply **automatic differentiation (AD)** and to use equations encoding prior knowledge to improve the generalization of the model.

The remaining part of this section describes how the different types of direct-solution models, shown in Table 1, can be applied to simulate the ideal pendulum system for a specific initial condition. First, the architecture of the NNs used for the experiments is introduced in Section 3.1. Next, the simplest approach is introduced in Section 3.2, before progressively building up to a model type that incorporates features from all prior models in Section 3.5.

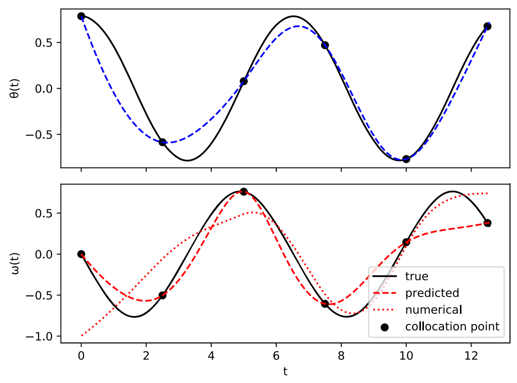
3.1 Methodology

The examples of direct-solution models shown in this section use an FC NN with three hidden layers consisting of 32 neurons each. The output of each hidden layer is followed by a softplus activation function.

Each model is trained on a trajectory corresponding to the simulation for a single initial condition, which is sampled to obtain a set of collocation points as shown in Figure 7(b). The goal is to obtain a model that can predict the solution at any point in time, not only those coinciding with the collocation points.



(a) Network structure.



(b) Predictions.

```
θ̃,ω̃ = 𝒩(t)
loss = ℒc((θ̃, ω̃), (θ, ω))
optimizer.step(loss)
```

(c) Training.

```
θ̃,ω̃ = 𝒩(t)
```

(d) Inference.

Fig. 7. Vanilla direct-solution model. The network $\mathcal{N} : \mathbb{R} \to \mathbb{R}^2$ maps time instances t to the solution $\tilde{\theta}, \tilde{\omega}$. Black dots indicate the collocation points (i.e., the points in which the loss function is minimized). The network fits all collocation points well but fails to generalize in the interval between points. In addition, $\tilde{\omega}$ is very different from the approximation obtained using numerical differentiation of $\tilde{\theta}$.

3.2 Vanilla Direct-Solution

Direct-solution models produce an estimate of the system's state at a given time, t_i, by introducing it as an input to an NN.

To model the pendulumm we use a feed-forward network with a single input t and two outputs $\tilde{\theta}$ and $\tilde{\omega}$, as depicted in Figure 7(a). To obtain the solution for multiple time instances, the network can simply be evaluated multiple times. There are no dependencies between the estimates of multiple states, allowing us to evaluate all of these in parallel.

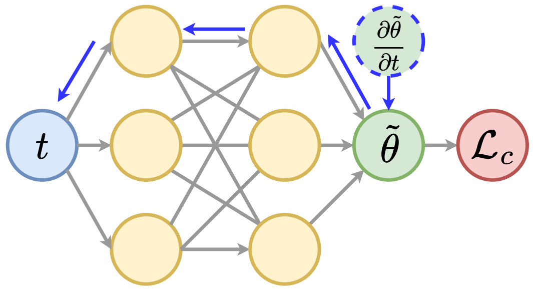
The network is trained by minimizing the difference between the predicted and the true trajectory in the collocation points shown in Figure 7(b) using a distance metric such as MSE defined by Equation (7):

$$\mathcal{L}_c(\tilde{x}, x) = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (\tilde{x}_{ij} - x_{ij})^2, \tag{7}$$

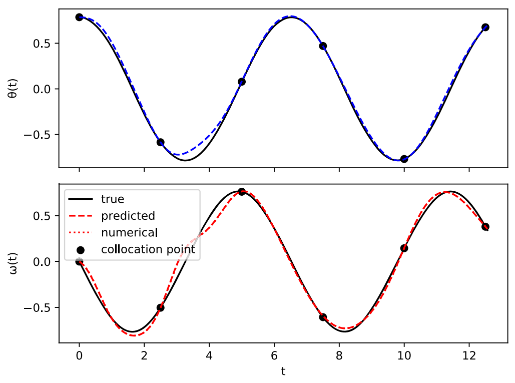
where m is the length of the trajectory, n is the dimension of the system's state-space, and x_{ij} denotes the value of the j-th state at the i-th point of time of the trajectory.

It is important to emphasize that the models learn a sequence of system states characterized by a specific set of initial conditions—that is, the initial conditions are encoded into the trainable parameters of the network during training and cannot be modified during inference.

Direct-solution models are sensitive to the quality of training data. NNs are used to find mappings between sparse sets of input data and the output. Even a simple example in the data-sampling strategy can influence their generalization performance. Consider the trajectory in Figure 7(b); the model has only learned the correct solution in the collocation points and fails to generalize anywhere else.



(a) Network structure.



(b) Predictions

```
θ̃ = 𝒩(t)
ω̃ = gradient(θ̃, t)
loss = ℒc((θ̃, ω̃), (θ, ω))
optimizer.step(loss)
```

(c) Training.

```
θ̃ = 𝒩(t)
ω̃ = gradient(θ̃, t)
```

(d) Inference.

Fig. 8. AD in the direct-solution model. The network $\mathcal{N} : \mathbb{R} \rightarrow \mathbb{R}$ maps the time instances t to the pendulum's angle $\tilde{\theta}$. The angular velocity $\tilde{\omega}$ is obtained by differentiating $\tilde{\theta}$ with respect to time using AD. This approach ensures that an output, representing the derivative of another output, acts like a true derivative. As a result, the network generalizes significantly better across both state variables.

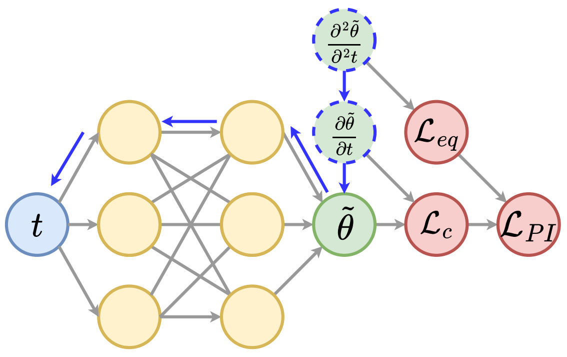
It is worth noting that there are many ways that this can go wrong—that is, given a sufficiently sparse sampling, it is not just one specific choice of training points that makes it impossible for the network to learn the true mapping. The obvious way to mitigate the issue is to obtain more data by sampling at a higher rate. However, there are cases where data acquisition is expensive, impractical, or it is simply impossible to change the sampling frequency.

Consider a system where one state variable is the derivative of the other, a setting that is quite common in systems that can be described by differential equations. A vanilla direct-solution model cannot guarantee that the relationship between the predicted state variables respects this property. Figure 7(b) provides a graphical representation of the issue. Although the model predicts both system state variables correctly in the collocation points, it can clearly be seen that the estimate for ω is neither the derivative of $\tilde{\theta}$ nor does it come close to the true trajectory.

3.3 AD Direct-Solution

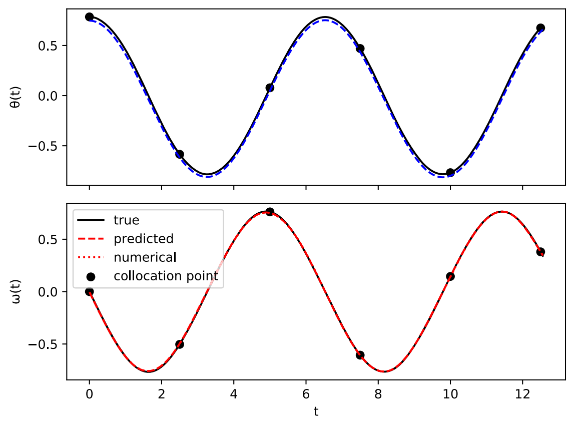
One way to leverage known relations is to calculate derivatives of state variables using automatic differentiation instead of having the network predict them as explicit outputs. In the case of the pendulum, this means using the network to predict $\tilde{\theta}$ and then obtaining $\tilde{\omega}$ by calculating the first-order derivative of $\tilde{\theta}$ with respect to time, as described in Figure 8(c) and (d). Figure 8(b) shows how much closer the predicted trajectories are to the true ones when using this approach.

A drawback of obtaining ω using AD is an increased computation cost and memory consumption depending on which mode of AD is used. Using reverse mode AD (backpropagation) as depicted in Figure 8(a) requires another pass of the computation graph, as indicated by the arrow going from output θ to input t. For training, this is not problematic since the computations carried out during backpropagation are necessary to update the weights of the network as well. However, using backpropagation during inference is not ideal because it introduces unnecessary memory and computation cost. An alternative is to use forward AD where the derivatives are computed



$\frac{\partial^2\tilde{\theta}}{\partial^2 t}$
$\frac{\partial\tilde{\theta}}{\partial t}$
$\mathcal{L}_{eq}$
t
$\tilde{\theta}$
$\mathcal{L}_c$
$\mathcal{L}_{PI}$

(a) Network structure.



(b) Predictions.

```
θ̃ = 𝒩(t)
ω̃ = gradient(θ̃, t)
∂ω̃ = gradient(ω̃, t)
loss = ℒc((θ̃,ω̃),(θ,ω))+ℒeq(θ̃,∂ω̃)
optimizer.step(loss)
```

(c) Training.

```
θ̃ = 𝒩(t)
ω̃ = gradient(θ̃, t)
```

(d) Inference.

Fig. 9. Physics-informed neural network. The network $\mathcal{N} : \mathbb{R} \to \mathbb{R}$ maps the time instances t to the pendulum's angle $\tilde{\theta}$. The angular velocity $\tilde{\omega}$ and its derivative are obtained using AD. The network is trained by minimizing Equation (8).

during the forward pass, thus dispensing of the separate backward pass. Unfortunately, not all DL frameworks provide support for forward mode AD (see Table 5 in the work of Baydin et al. [5]). A likely explanation is that the typical task of evaluating the derivative of the loss with respect to the network's weights is more efficient using reverse-mode AD (backpropagation).

3.4 Physics-Informed Neural Networks

In modeling scenarios where the equations describing the dynamics of the system are known, we can use them to train the model as another way of addressing the data-sampling issue. In what is known as *physics-informed neural networks* [101], knowledge about the physical laws governing the system is used to impose structure on the NN model. This can be accomplished by extending the loss function with an *equation loss* term that ensures the solution obeys the dynamics described by the governing equations. Although this technique was originally proposed for solving PDEs, it can also be applied to solve ODEs. For instance, to model the ideal pendulum using a PINN, we could integrate the expression of $\frac{\partial \tilde{\omega}}{\partial t}$ from Equation (1) to formulate the loss as

$$\mathcal{L}_{PI}\left(\tilde{\theta}, \frac{\partial\tilde{\theta}}{\partial t}, \frac{\partial\tilde{\omega}}{\partial t}\right) = \mathcal{L}_c\left(\tilde{\theta}, \frac{\partial\tilde{\theta}}{\partial t}\right) + \mathcal{L}_{eq}\left(\tilde{\theta}, \frac{\partial\tilde{\omega}}{\partial t}\right) \tag{8}$$

$$\mathcal{L}_{eq}\left(\tilde{\theta}, \frac{\partial\tilde{\omega}}{\partial t}\right) = \frac{1}{m}\sum_{i=0}^{m-1}\left(\frac{\partial\tilde{\omega}_i}{\partial t} - \frac{g}{l}\sin\tilde{\theta}_i\right)^2.$$

Again, we can use automatic differentiation to obtain $\frac{\partial\tilde{\omega}}{\partial t}$ by differentiating $\tilde{\theta}$ twice, depicted in the computation graph shown in Figure 9(a). As shown in Figure 9(c), this requires only a few lines of code when using AD.

A motivation for incorporating the equation loss term is to constrain the search space of the optimizer to parameters that yield physically consistent solutions. It should be noted that both the loss term penalizing the prediction error and the equation error are necessary to constrain the predictions of the network. On its own, the equation error guarantees that the predicted state satisfies the ODE, but not necessarily that it is the solution at a particular time. Introducing the prediction error ensures that the predictions are not only valid but also the correct solutions for the particular points used to calculate the prediction error. In addition, it should be noted that the collocation and equation loss terms may be evaluated for a different set of times. For instance, the equation-based loss term may be evaluated for an arbitrary number of time instances, since the term does rely on accessing the true solution for particular time instances.

In addition to proposing the introduction of the equation loss, PINNs also apply the idea of using backpropagation to calculate the derivatives of the state variables rather than adding them as outputs to the network, as depicted in Figure 9(a). Being able to obtain the *n*-th order derivatives is very useful for PINNs, as they often appear in differential equations on which the equation loss is based. For the ideal pendulum, this technique can be used to obtain $\frac{\partial^2 \theta}{\partial t^2}(t)$ from a single output of the network θ, which can then be plugged into Equation (2) to check that the prediction is consistent. A benefit of using backpropagation compared to adding state variables as outputs of the network is that this structurally ensures that the derivatives are in fact partial derivatives of the state variables.

Training PINNs using gradient descent requires careful tuning of the learning rate. Specifically, it has been observed that the boundary conditions and the physics regularization terms may converge at different rates. In some cases, this manifests itself as a large misfit specifically at the boundary points. Wang et al. [131, 132] propose a strategy for weighing the different terms of the loss function to ensure consistent minimization across all terms.

3.5 Hidden Physics Networks

Hidden physics neural networks (HNNs) [103] can be seen as an extension of PINNs that use governing equations to extract features of the data that are not present in the original training data. We refer to the unobserved variable of interest as a *hidden variable*. This technique is useful in cases where the hidden variable is difficult to measure compared to the known variables or simply impossible to measure since no sensor exists that can reliably measure it.

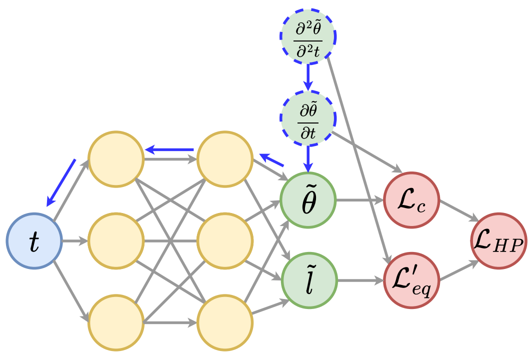
For the sake of demonstration, we may suppose that the length of the pendulum arm is unknown and that it varies with time, as shown in Figure 10(b). For the training, this is problematic since l is required to calculate the equation loss. A solution to this is to add an output $\tilde{l}$ to the network that serves as an approximation of the true length l, as depicted in Figure 10(d). We modify Equation (8) to define a new loss function that takes the estimate of $\tilde{l}$ into account

$$\mathcal{L}_{HP}\left(\tilde{\theta}, \frac{\partial \tilde{\theta}}{\partial t}, \frac{\partial \tilde{\omega}}{\partial t}, \tilde{l}\right) = \mathcal{L}_c\left(\tilde{\theta}, \frac{\partial \tilde{\theta}}{\partial t}\right) + \mathcal{L}'_{eq}\left(\tilde{\theta}, \frac{\partial \tilde{\omega}}{\partial t}, \tilde{l}\right) \tag{9}$$

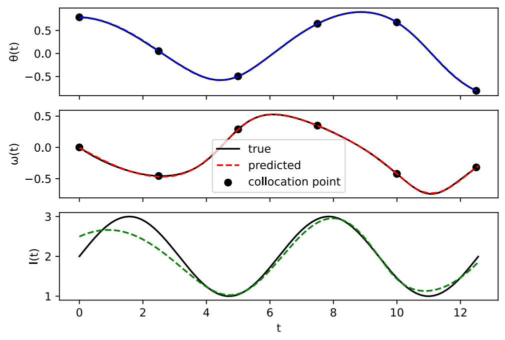
$$\mathcal{L}'_{eq}\left(\tilde{\theta}, \frac{\partial \tilde{\omega}}{\partial t}, \tilde{l}\right) = \frac{1}{m}\sum_{i=0}^{m-1}\left(\frac{\partial \tilde{\omega}_i}{\partial t} - \frac{g}{\tilde{l}_i}\sin\tilde{\theta}_i\right)^2.$$

It should be emphasized that $\tilde{l}$ is not part of the collocation loss term, since the true value l is not known. It is only as a result of the equation loss that the network is constrained to produce estimates of l satisfies the system's dynamics.

Raissi et al. [103] use this technique to extract pressure and velocity fields based on measured dye concentrations. In this particular case, the dye concentration can be measured by a camera, since


$\frac{\partial^2\tilde{\theta}}{\partial^2 t}$
$\frac{\partial\tilde{\theta}}{\partial t}$
t
$\tilde{\theta}$
$\tilde{l}$
$\mathcal{L}_c$
$\mathcal{L}'_{eq}$
$\mathcal{L}_{HP}$

(a) Network structure.



(b) Predictions.

```
θ̃, l̃ = 𝒩(t)
ω̃ = gradient(θ̃, t)
∂ω̃ = gradient(ω̃, t)
loss = ℒc((θ̃,ω̃),(θ,ω))+ℒ'eq(θ̃, ∂ω̃, l̃)
optimizer.step(loss)
```

(c) Training.

```
θ̃, l̃ = 𝒩(t)
ω̃ = gradient(θ̃, t)
```

(d) Inference.

Fig. 10. Hidden physics network. This network $\mathcal{N} : \mathbb{R} \to \mathbb{R}^2$ is an extension of the PINN and maps the time instances t to the pendulum's angle $\tilde{\theta}$ and the length of the pendulum $\tilde{l}$ that is set to vary in time for the sake of demonstration. Note that $\tilde{l}$ is not part of $\mathcal{L}_c$ since there is no training data for it; instead, it is part of the equation loss $\mathcal{L}'_{eq}$.

the opacity of the fluid is proportional to the dye concentration. They show that this technique also works well even in cases where the dye concentration is sampled at only a few points in time and in space. Like PINNs, HNNs are easily applied to PDEs, but at the cost of the initial conditions being encoded in the network during training.

The difference between PINNs and HNNs is very subtle; both utilize similar network architectures and use loss functions that penalize any incorrect prediction violations of governing equations. A distinguishing factor is that, in HNNs, the hidden variable is inferred based on physical laws that relate the hidden variable to the observed variables. Since the hidden variables are not part of the training data, they can only be enforced through equations.

4 TIME-STEPPER MODELS

Consider the approach used to model an ideal pendulum, described in Section 2. First, a set of differential equations, Equation (2), was used to model the derivative function of the system. Next, using the function, a numerical solver was used to obtain a simulation of the system for a particular initial condition. The challenge of this approach is that identifying the derivative function analytically is difficult for complex systems.

An alternative approach is to train an NN to approximate the derivative function of the system, allowing the network to be used in place of the hand-derived function, as depicted in Figure 11. We refer to this type of model as a *time-stepper model* since it produces a simulation by taking multiple steps in time, like a numerical solver. An advantage of this is that it allows well-studied numerical solvers to be integrated into a model with relative ease.

The main differences between two given models can be attributed to (i) how the derivatives are produced by the network and (ii) what sort of integration scheme is applied. For instance, the difference between the *direct* (Section 4.2.1) and *Euler* time-stepper models (Section 4.2.2) is that the former does not employ any integration scheme, whereas the latter is similar to the FE

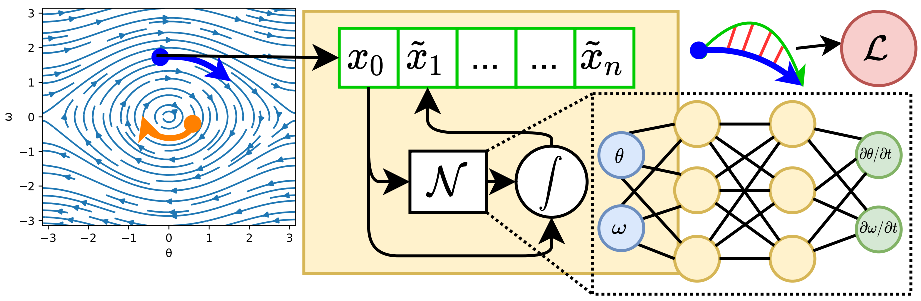


Fig. 11. Time-stepper model. Starting from a given initial condition x_0, the next state of the system, $\tilde{x}_{i+1}$, is obtained by feeding the current state $\tilde{x}_i$ into the derivative network $\mathcal{N}$, producing a derivative that is integrated using an integration scheme $\int$. The loss $\mathcal{L}$ is evaluated by comparing the predicted with the training trajectory. The process can be repeated for multiple trajectories to improve the generalization of the derivative network.

(recall Equation (6)), leading to a significant difference in predictive ability. Other networks, such as the *Lagrangian* time-stepper, Section 4.4.1, distinguish themselves by the way the NN produces the derivatives. Specifically, this approach does not obtain $\partial\theta$ and $\partial\omega$ as outputs from a network but instead uses AD in an approach similar to Section 3.3. Similar to how an ODE can be solved with different numerical solvers, the Lagrangian time-stepper could be modified to use a different integration scheme than FE.

Given the independent relationship between the choice of NN and the numerical solver used, the models introduced in the sections should not be viewed as an exhaustive list of combinations. Rather, the aim is to describe and compare the models commonly encountered in the literature.

4.1 Methodology

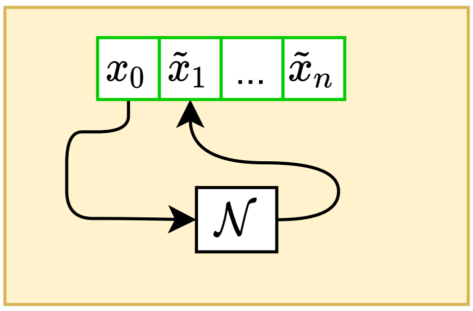
A time-stepper must be able to produce accurate simulations for different initial conditions. It would be possible to train a time-stepper using a single trajectory; however, this is unlikely to generalize well to different initial conditions. Another approach is to use multiple, potentially shorter trajectories as training data. We can extend Equation (7) to take the mean error over l trajectories:

$$\mathcal{L}_{ts}(\tilde{X}, X) = \frac{1}{l}\sum_{i=0}^{l-1}\mathcal{L}_c(\tilde{X}_i, X_i), \tag{10}$$

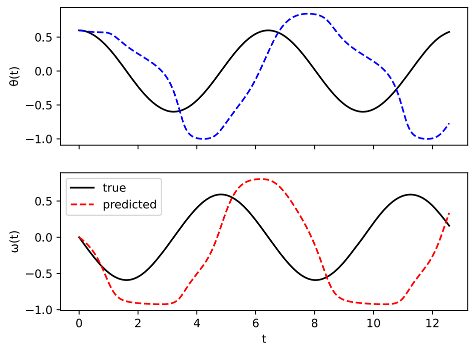
where $X \in R^{l\times m\times n}$ are the training trajectories and $\tilde{X} \in R^{l\times m\times n}$ are the predicted trajectories.

Each time-stepper model is trained on 100 trajectories, each consisting of two samples: the initial state and the state one step into the future. The initial states are sampled in the interval $\theta : (-1, 1)$ and $\omega : (-1, 1)$ using Latin hyper-cube sampling (see Figure 11). Each model uses an FC network consisting of eight hidden layers with 32 neurons each. Each layer of the network applies a softplus activation function. The number of inputs and outputs is determined by the number of states characterizing the system, which is 2 for the ideal pendulum. Exceptions to this are networks such as the Lagrangian network described in Section 4.4.1, for which the derivatives are obtained using AD rather than as outputs of a network.

To validate the performance of each model, 100 new initial conditions are sampled in a grid. For each initial condition in the validation set, the system is simulated for 4π seconds using the original ODE and compared with the corresponding prediction made by the trained model. For simplicity, we show only the trajectory corresponding to a single initial condition, like the one in Figure 12(b).



(a) Network structure.



(b) Predictions.

```
$\tilde{X}$[:,0,:] = $X_{t_0}$
for i in 0...m-1
  $\tilde{X}$[:,i+1,:] = $\mathcal{N}$($\tilde{X}$[:,i,:])
loss = $\mathcal{L}_{ts}$($X$,$\tilde{X}$)
optimizer.step(loss)
```

(c) Training.

```
$\tilde{x}$[0,:] = $x_{t_0}$
for i in 0...m-1
  $\tilde{x}$[i+1,:] = $\mathcal{N}$($\tilde{x}$[i,:])
```

(d) Inference.

Fig. 12. Direct time-stepper. The output of the network $\mathcal{N} : \mathbb{R}^2 \to \mathbb{R}^2$ is used as the prediction for the next step without any form of numerical integration. An issue of this type of model is that it fails to generalize beyond the exact points in state space that it has been trained for. Over several steps, the error compounds, which leads to an inaccurate simulation.

4.2 Integration Schemes

An important characteristic of a time-stepper model is how the derivatives are evaluated and integrated to obtain a simulation of the system. Again, it should be emphasized that the choice of the numerical solver is independent of the architecture of the NN used to approximate the derivative function. In other words, for a given choice of NN architecture, the performance of the trained model may depend on the choice of solver.

The choice of numerical solver not only determines how the model produces a simulation of the system but also influences how the model must be trained. Specifically, when minimizing any criterion that is a function of the integrated state, the choice of solver determines how the state is produced.

In the following section, we demonstrate how various numerical solvers can be used and evaluate their impacts on the performance of the models.

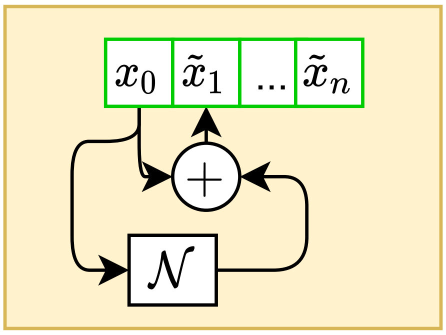
4.2.1 Direct Time-Stepper. The simplest approach to obtaining the next state is to use the prediction produced by the network directly, as summarized in Figure 12(a):

$$\tilde{x}_{i+1} = \mathcal{N}(\tilde{x}_i),$$

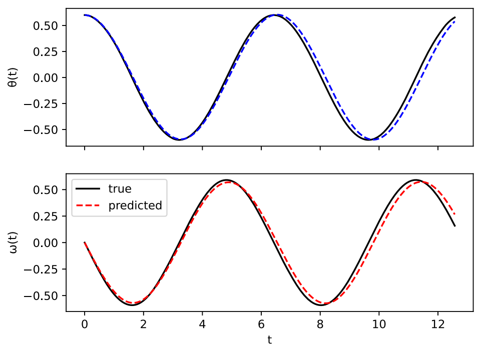
where N represents a generic NN with arbitrary architecture and $\tilde{x}_0 = x_0$.

The network is trained to produce an estimate of the next state, $\tilde{x}_{i+1}$, from the current state, x_i. During training, this operation can be vectorized such that every state at every timestamp, omitting the last, is mapped one step into the future using a single invocation of the network, as shown in Figure 12(c). The reason for leaving out the last sample in when invoking the NN is that this would produce a prediction, x_{N+1}, for which there does not exist a sample in the training set.

At inference time, only the initial state x_0 is known. The full trace of the system is obtained by repeatedly introducing the current state into the network, as depicted in Figure 12(d). Note that the inference phase cannot be parallelized in time, since predictions for time $i + 1$ depend on



(a) Network structure.



(b) Predictions.

```
X̃[:,0,:] = $X_0$
for i in 0...m-1
  $\Delta X$ = $\mathcal{N}$(X̃[:,i,:])
  X̃[:,i+1,:] = X̃[:,i,:] + $\Delta X$
loss = $\mathcal{L}_{ts}$($X$,$\tilde{X}$)
optimizer.step(loss)
```

(c) Training.

```
x̃[0,:] = $x_0$
for i in 0...m-1
  $\Delta x$ = $\mathcal{N}$(x̃[i,:])
  x̃[i+1,:] = x̃[i,:] + $\Delta x$
```

(d) Inference.

Fig. 13. Residual time-stepper. The output of the network is added to the current state to form a prediction of the next state. Compared to the direct time-stepper, this method produces simulations that are much closer to the true system.

predictions for time k. However, it is possible to simulate the system for multiple initial states in parallel, as they are independent of each other.

The simulation for a single initial condition can be seen in Figure 12(b). Although the simulation is accurate for the first few steps, it quickly diverges from the true dynamics.

4.2.2 *Residual Time-Stepper.*

A network can be trained to predict a derivative-like quantity that can then be added to the current state to yield the next as shown in Figure 13(a):

$$\tilde{x}_{i+1} = \tilde{x}_i + \mathcal{N}(\tilde{x}_i).$$

DL practitioners may recognize this as a residual block that forms the basis for *residual networks* (ResNets) [45], which are used with great success in applications spanning from image classification to natural language processing. Readers familiar with numerical simulation will likely notice that the previous equation closely resembles the accumulated term in the FE integrator (recall Equation (6)) but without the term that accounts for the step size. If the data is sampled at equidistant timesteps, the network scales the derivative to adapt the step size.

The central motivation for using a residual network is that it may be easier to train a network to predict how the system will change rather than a direct mapping between the current and next state.

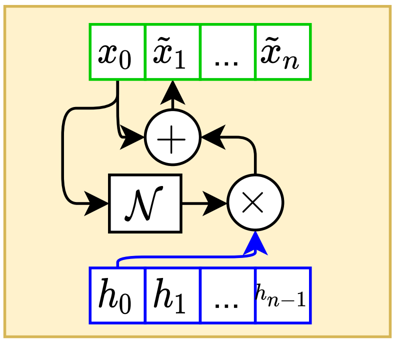
4.2.3 *Euler Time-Stepper.*

Alternatively, the step size can be encoded in the model by scaling the contribution of the derivative by the step size h_i as shown in Figure 14(a):

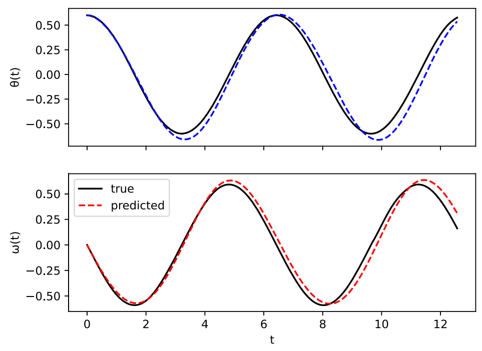
$$\tilde{x}_{i+1} = \tilde{x}_i + h_i * \mathcal{N}(\tilde{x}_i). \tag{11}$$

This resemblance has been noted several times [97] and has resulted in work that interprets residual networks as ODEs allowing classical stability analysis to be used [15, 110, 111].

The FE integrator shown in Equation (11) is simple to implement. However, it accumulates a higher error than more advanced methods, such as the Midpoint, for a given step size. This issue



(a) Network structure.



(b) Predictions.

```
X̃[:,0,:] = X₀
for i in 0...m-1
  ΔX = 𝒩(X̃[:,i,:])
  X̃[:,i+1,:] = X̃[:,i,:] + h[i]*ΔX
loss = ℒ_ts(X,X̃)
optimizer.step(loss)
```

(c) Training.

```
x̃[0,:] = x₀
for i in 0...m-1
  Δx = 𝒩(x̃[i,:])
  x̃[i+1,:] = x̃[i,:] + h[i]*Δx
```

(d) Inference.

Fig. 14. Euler time-stepper. The output of the network is multiplied by the step size and is added to the current state to form a prediction for the next state. In this case, accounting for the step size leads to minimal improvements, if any, compared to the residual time-stepper. This is likely due to the fact that the step size used during training is the same as the one used to plot the trajectory in Figure 14(b).

has motivated the integration of more sophisticated numerical solvers in time-stepper models. For example, **linear multistep (LMS)** methods are used in the work of Raissi et al. [102]. LMS uses several past states and their derivatives to predict the next state, resulting in a smaller error compared to FE. Like FE, LMS only requires a single function evaluation per step, making it a very efficient method. But if the system is not continuous, this method needs to be reinitialized after a discontinuity occurs [36].

4.2.4 Neural Ordinary Differential Equations. **Neural ordinary differential equations (NODEs)** [18] is a method used to construct models by combining a numerical solver with an NN that approximates the derivative of the system. Unlike the previously introduced models, the term *NODEs* is not used to refer to models using a specific integration scheme but rather to the idea of treating an ML problem as a dynamical system that can be solved using a numerical solver. Part of their contribution was the implementation of differentiable solvers accessible via a simple API, allowing users to implement NODEs in only a few lines of code, as shown in Figure 16.

Some confusion may arise from the fact that NODEs are frequently used for image classification throughout the literature, which may seem completely unrelated to numerical simulations. The underlying idea is that an image can be represented as a point in state-space that moves on a trajectory defined by an ODE, as shown in Figure 15. The goal of this is to find an ODE that results in images of the same class converging to a cluster that is easily separable from that of unrelated classes. For single inference (e.g., in image classification), intermediate predictions have no inherent meaning—that is, they typically do not correspond to any measurable quantity of the system; we are only interested in the final estimate $\hat{x}_m$. Due to the lack of training samples corresponding to intermediate steps, it is impossible to minimize the single-step error.

Chen et al. [18] motivate the use of an adaptive step-size solver by its ability to adjust the step size to match the desired balance between numerical error and performance. An alternative way

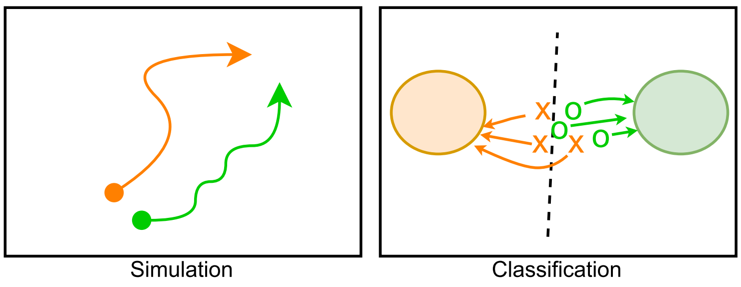
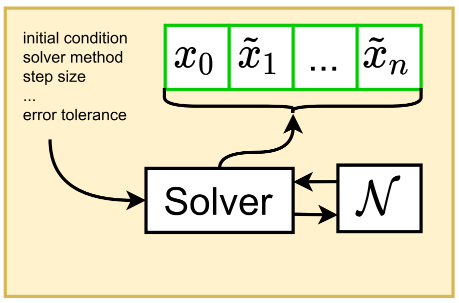
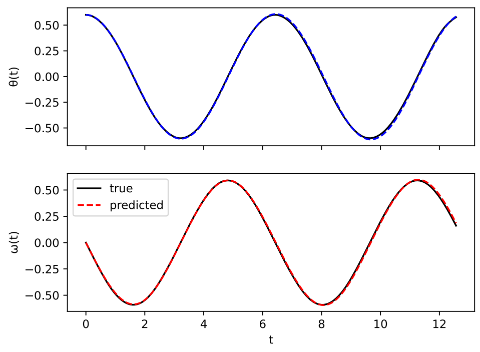


Fig. 15. Different applications of NODEs. NODEs can be used to simulate a dynamical system with the goal of obtaining a trajectory corresponding to an initial condition. In this case, the goal is to train the network to produce a derivative that provides a good estimate of the true state at every step of the trajectory. Another use is for classification by treating each input sample as a point in state-space, which evolves according to the derivative produced by the network. In this case, the goal is to train the network to learn dynamics that leads to samples belonging to each class ending in distinct clusters that are easily separable.



(a) Network Structure.



(b) Predictions.

```
$\tilde{X}$ = odeint($\mathcal{N}$,$X_0$,$t_{start}$,$t_{end}$,"rk4")
loss = $\mathcal{L}_{ts}$($X$,$\tilde{X}$)
optimizer.step(loss)
```

(c) Training.

```
$\tilde{x}$ = odeint($\mathcal{N}$,$x_0$,$t_{start}$,$t_{end}$,"rk4")
```

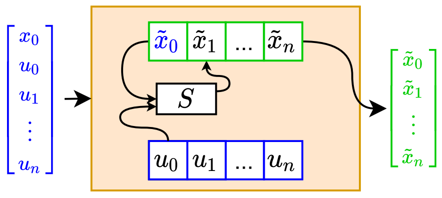
(d) Inference.

Fig. 16. Neural ordinary differential equations. NODEs generally refer to models that are constructed to use a numerical solver to integrate the derivatives through time. Unlike the previously introduced integration schemes that mapped to concrete architectures, NODEs refer to the idea of using well-established numerical solvers inside a model. Part of NODEs' popularity is due to the fact that it mimics the programming APIs of traditional numerical solvers, which makes it easy to switch between different types of solvers.

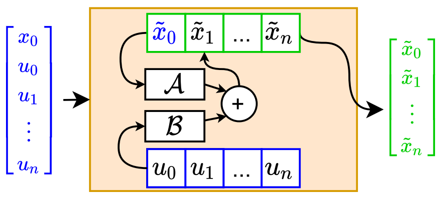
to view NODEs is as a *continuous-depth model* where the number of layers is a result of the step size chosen by the solver.

From this perspective, the stability of NODEs is closely related to the stability of integration schemes of classical ODEs. To address the convergence issues during training, some authors propose NODEs with stability guarantees by exploiting Lyapunov stability theory [79] and spectral projections [99]. Another standing issue of NODEs is their large computational overhead during training compared to classical NNs. Finlay et al. [28] demonstrated that stability regularization may improve convergence and reduce the training times of NODEs. Poli et al. [96] propose graph NODEs resulting in training speedups, as well as improved performance due to incorporation of prior knowledge.

To improve the performance, others have introduced various inductive biases such as Hamiltonian NODE architecture [142] or penalizing higher-order derivatives of the NODEs in the



(a) State and input stacked and fed into the same network.



(b) State and input fed into separate networks.

Fig. 17. Incorporation of inputs in the time-stepping model.

loss function [55]. To account for the noise and uncertainties, some authors proposed stochastic NODEs [42, 48, 70, 74] as generalizations of deterministic NODEs.

A fundamental issue of interpreting trained NODEs as proper ODEs is that they may have trajectory crossings, and their performance can be sensitive to the step size used during inference [92]. Contrary to this, the solutions of ODEs with unique solutions would never have intersecting trajectories, as this would imply that for a given state (the point of intersection), the system could evolve in two different ways. Some authors have noted that there seems to be a critical step size for which the trained network starts behaving like a proper ODE [92]. In other words, if trained with a particular step size, the network will perform equally well or better if used with a smaller step size during inference. Another approach is to use regularization to constrain the parameters of the network to ensure that solutions are unique. For ResNets, this can be achieved by ensuring that the Lipschitz constant of the network is less than 1 for any point in the state-space, which guarantees a unique solution [7].

To deal with external inputs in NODEs, Dupont et al. [27] and Norcliffe et al. [88] propose lifting the state-space via additional augmented variables. A more general way of explicitly modeling the input dynamics via additional NNs is proposed by Massaroli et al. [80].

4.3 External Input

So far, we have only considered how to apply time-stepper models to systems where the derivative function is determined exclusively by the system's state. In practice, many systems encountered are influenced by an external stimulus that is independent of the dynamics, such as external forces acting on the system or actuation signals of a controller. To avoid confusion, we refer to these external influences as *external input* to distinguish them from the general concept of an NN's inputs.

The structure of a time-stepper model lends itself well to introducing external inputs at every evaluation of the derivative function. As a result, it is possible to integrate external inputs in time-stepper models in many ways.

4.3.1 Neural State-Space Models. Inputs can be added to the time-stepper models in a couple of ways. One way is to concatenate the inputs with the states, as illustrated in Figure 17(a):

$$\tilde{x}_{i+1} = \mathcal{N}([x_i, u_i]), \tag{12}$$

where $\tilde{x}_i$ and u_i represent states and inputs at time t_i, respectively. The evolution of the future state x_{i+1} is fully determined by the derivative network $\mathcal{N}$. A possible rationale for lumping system states and inputs are parameter-varying systems, where the inputs influence the system differently depending on the current state. This approach does not impose any structure on how the state and input information are aggregated in the network, since the layers of the network make no distinction between the two.

Alternatively, two separate networks $\mathcal{A}$ and $\mathcal{B}$ can be used to model contributions of the autonomous and forced parts of the dynamics, respectively, as seen in Figure 17(b). This information can then be aggregated by taking the sum of the two terms:

$$\tilde{x}_{i+1} = \mathcal{A}(\tilde{x}_i) + \mathcal{B}(u_i). \tag{13}$$

This approach is suitable for systems where the influence of the inputs is known to be independent of the state of the system since it structurally enforces models that are independent.

In system identification and control theory, both variants (12) and (13) are referred to as **state-space models (SSMs)** [56, 66, 116, 117]. More recently, researchers [43, 64, 104, 123] proposed to model non-linear SSMs by using NNs, which we refer to as neural SSMs.

Some works proposed to combine neural approximations with classical approaches with linear state transition dynamics $\mathcal{A}$, resulting in Hammerstein [91] and Hammerstein-Wiener [47] architectures, or using linear operators representing transfer function as layers in deep NNs [30]. However, others leverage encoder-decoder neural architectures to handle partially observable dynamics [37, 81]. Some authors [26, 120, 121] applied principles of gray-box modeling by imposing physics-informed constraints on a learned neural SSM. Ogunmolu et al. [90] analyzed the effect of different neural architectures on the system identification performance of non-linear systems and concluded that compared to classical non-linear regressive models, deep NNs scale better and are easier to train.

4.3.2 NODEs with External Input. The challenge of introducing external input to NODEs is that the numerical solver may try to evaluate the derivative function at time instances that align with the sampled values of the external input. For instance, an adaptive step-size solver may choose its own internal step size based on how rapidly the derivative function changes in the neighborhood of the current state. The issue can be solved using interpolation to obtain values of external inputs for time instances that do not coincide with the sampling.

External input can also be used to represent static parameter values that remain constant through a simulation. In the context of the ideal pendulum system, we could imagine that the length of the pendulum could be made a parameter of the model, allowing the model to simulate the system under different conditions. Lee and Parish [67] call this approach *parameterized* NODEs and use this mechanism to train models that can solve PDEs for different parameter values.

Another approach is **neural controlled differential equations (NCDEs)** [57]. The term *controlled* should not be confused with the field of *control theory* but rather the mathematical concept of controlled differential equations from the field of rough analysis. The core idea of NCDEs is to treat the progression of time and the external inputs as a signal that *drives* the evolution of the system's state over time. The way that a specific system responds to this signal is approximated using an NN. A benefit of this approach is that it generalizes how a system's autonomous and forced dynamics are modeled. Specifically, it allows NCDEs to be applied to systems where NODEs would be applied, as well as systems where the output is purely driven by the external input to the system.

4.4 Network Architecture

Part of the success of NNs can be attributed to the ease of integrating specialized architectures into a model. In this section, we introduce a few examples of how to integrate domain-specific NNs into a time-stepper model.

First, Section 4.4.1 describes how energy-conserving dynamics can be enforced by encoding the problem using Hamiltonian or Lagrangian mechanics. Next, Section 4.4.2 demonstrates another way of enforcing energy conservation, which is often encountered in **molecular dynamics (MD)**.

Finally, Section 4.4.3 describes how graphs can be integrated with a time-stepper to solve problems that can naturally be represented as graphs.

4.4.1 Hamiltonian and Lagrangian Networks. Recall that the movement in some physical systems happens as a result of energy transfers within the system, as opposed to systems where energy is transferred to/from the system. The former is called an *energy conservative system*. For instance, if the pendulum introduced in Figure 3 had no friction and no external forces acting on it, it would oscillate forever, with its kinetic and potential energy oscillating without a change in its total energy. In physics, a special class of closely related functions, called *Hamiltonian and Lagrangian functions*, has been developed for describing the total energy of a system. Both Hamiltonian $\mathcal{H}$ and Lagrangian $\mathcal{L}$ are defined as a sum of total kinetic T and potential energy V of the system. We start with the Hamiltonian defined as

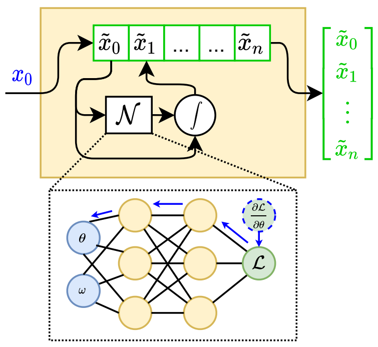


Fig. 18. Lagrangian time-stepper. The Lagrangian, $\mathcal{L}$ (not to be confused with the loss function), is differentiated using AD to obtain the derivative of the state.

$$\mathcal{H}(x) = T(x) - V(x), \tag{14}$$

where $x = [q, p]$ represents the concatenated state vector of generalized coordinates q and generalized momenta p. By taking the gradients of the energy function (14), we can derive a corresponding differential $\dot{x} = f(x)$ equation as

$$\dot{x} = S\nabla\mathcal{H}(x), \tag{15}$$

where S is a symplectic matrix. Please note that the difference between $\mathcal{H}$ and $\mathcal{L}$ is their corresponding coordinate system: for the Lagrangian, instead of $x = [q, p]$, we consider $x = [q, \dot{q}]$, where $\dot{p} = M(q)\dot{q}$, with $M(q)$ being a generalized mass matrix.

Despite their mathematical elegance, deriving analytical Hamiltonian and Lagrangian functions for complex dynamical systems is a grueling task. In recent years, the research community turned its attention to deriving these types of scalar-valued energy functions by means of data-driven methods [41, 77, 142]. Specifically, the goal is to train an NN to approximate the Hamiltonian/Lagrangian of the system, as shown in Figure 18. A key aspect of this approach is that the derivatives of the states are not outputs of the network but are instead obtained by differentiating the output of the network $\mathcal{L}$, with respect to the state variables $[\theta, \omega]$ and plugging the results into Equation (14). The main advantage of Hamiltonian [41, 124] NNs and the closely related Lagrangian [21, 77] NNs is that they naturally incorporate the preservation of energy into the network structure itself. Research into the simulation of energy-preserving systems has yielded a special class of solvers, called *symplectic solvers*. Jin et al. [52] propose a new specialized network architecture, referred to as *symplectic networks*, to ensure that the dynamics of the model are energy conserving. Similarly, Finzi et al. [29] propose extensions for including explicit constraints via Lagrange multipliers for improved training efficiency and accuracy.

4.4.2 Deep Potential Energy Networks. A similar concept to that of Hamiltonian and Lagrangian NNs involves learning neural surrogates for potential energy functions $V(x)$ of a dynamical system, where the primary difference with Hamiltonians and Lagrangians is that the kinetic terms are

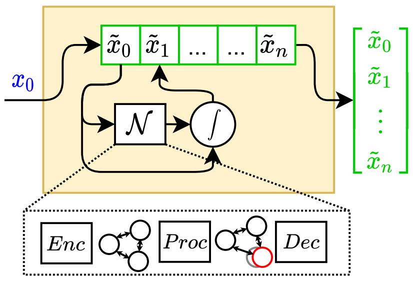


Fig. 19. A simplified view of a graph time-stepper. During each step of the simulation, the current state is encoded as a graph (Enc) that is then used to compute the change in state variable between the current and next timestep (Proc). Finally, the change in state is decoded to the original state space to update the state of the system (Dec).

encoded explicitly in the time stepper by considering classical Newtonian laws of motion:

$$\tilde{x}_{i+1} = \tilde{x}_i + \tilde{v}_i, \tag{16a}$$

$$\tilde{v}_{i+1} = \tilde{v}_i - \frac{\nabla\mathcal{V}(x)}{m}, \tag{16b}$$

where x_i, and v_i are positional and velocity vectors of the system. The gradients of the potential function are equal to the interaction forces $F = -\nabla\mathcal{V}(x)$, whereas m is a vector of "masses."

This approach is extensively used, mainly in the domain of MD simulations [6, 50, 125, 126, 130, 139]. In modern data-driven MD, the learned neural potentials $V(x)$ replace expensive quantum chemistry calculations based, for example, on density functional theory (DFT). The advantage of this approach for large-scale systems, compared to directly learning high-dimensional maps of the time-steppers, is that the learning of the scalar-valued potential function $V(x) : \mathbf{R}^n \to \mathbf{R}$ represents a much simpler regression problem. Furthermore, this approach allows prior information to be encoded in the architecture of the deep potential functions $V(x)$, such as considering only local interactions between atoms [119], and encoding spatial symmetries [34, 140]. As a result, these methods are allowing researchers in MD to achieve unprecedented scalability, allowing simulation of up to 100M atoms on supercomputers [49]. In contrast, training a single naive time-stepper for such a model would require learning a 300M-dimensional mapping.

4.4.3 Graph Time-Steppers. Many complex real-world systems from social networks and molecules to power grid systems can be represented as graph structures describing the interactions between individual subsystems. Recent research in **graph neural networks (GNNs)** embraces this idea by embedding or learning the underlying graph structure from data. There exists a large body of work on GNNs, but covering this is outside the scope of this survey. We refer the interested reader to overview papers [3, 10, 115, 137, 141, 143]. For the purposes of this section, we focus solely on GNN-based time-stepper models applied to model dynamical systems [58, 71].

The core idea of using GNNs inside time-steppers is to use a GNN-based pipeline to estimate the derivatives of the system, as shown in Figure 19. Generally, the pipeline can be split into three steps; first, the current state of the system is encoded as a graph; next, the graph is processed to produce an update of the system's state; and finally, the update is decoded and used to update the state of the system.

One of the early works includes interaction networks [4] or *neural physics engine* (NPE) [16] demonstrating the ability to learn the dynamics in various physical domains in smaller scale dimensions, such as *n*-body problems, rigid-body collision, and non-rigid dynamics. Since then, the use of GNNs rapidly expanded, finding its use in NODE time-steppers [112] including control inputs [72, 114], dynamic graphs [109], or considering feature encoders enabling learning dynamics directly from the visual signals [134]. Modern GNNs are trained using message-passing algorithms introduced in the context of quantum chemistry application [39]. In GNNs, each node has associated latent variables representing values of physical quantities such as positions, charges, or velocities, then in the message-passing step, the aggregated values of the latent states are passed through the edges to update the values of the neighboring nodes. This abstraction efficiently encodes local structure-preserving interactions that commonly occur in the natural world. Although early implementations of GNN-based time-steppers suffered from larger computational complexity, more recent works [113] have demonstrated their scalability to ever larger dynamical systems with thousands of state variables over long prediction horizons. Due to their expressiveness and generic nature, GNNs could in principle be applied in all the time-stepper variants summarized in this article, some of which would represent novel architectures up to date.

4.5 Uncertainty

So far, we have considered only the cases of modeling systems where noise-free trajectories were available for training. In reality, it is likely that the data captured from the system does not represent the true state of the system, x, but rather a noisy version of the original signal perturbed by measurement noise. Another source of uncertainty is that the dynamics of the system itself may exhibit some degree of randomness. One cause of this would be unidentified external forces acting on the system. For instance, the dynamics of a physical pendulum may be influenced by vibrations from its environment. The following sections introduce several models that explicitly incorporate uncertainty in their predictions.

4.5.1 Deep Markov Models. A **deep Markov model (DMM)** [2, 32, 65, 73, 86] is a probabilistic model that combines the formalism of Markov chains with the use of an NN for approximating unknown probability density functions. A Markov chain is a latent variable model, which assumes that the values we observe from the system are determined by an underlying latent variable, which cannot be observed directly. This idea is very similar to an SSM, the difference being that a Markov chain assumes that the mapping from the latent to the observed variable is probabilistic and that evolution of the latent variable is not fully deterministic.

The relationship between the observed and latent variables of a DMM can be specified as follows:

$$z_{i+1} \sim \mathcal{Z}(N_t(z_i)), \qquad \text{(Transition)} \qquad (17a)$$

$$x_i \sim \mathcal{X}(N_e(z_i)), \qquad \text{(Emission)} \qquad (17b)$$

where z_i represents the latent state vector and x_i is the output vector. Here, $\mathcal{Z}$ and $\mathcal{X}$ represent probability distributions, commonly Gaussian distributions, modeled by maps $N_T(\mathbf{z}_i)$ or $N_e(\mathbf{z}_i)$, respectively.

A natural question to ask is how the observed and latent variables are represented, given that they are probability density functions and not numerical values—a solution to pick distributions that can be represented in terms of a few characteristic parameters. For instance, a Gaussian can be represented by its mean and covariance. The process of performing inference using a DMM is shown in Figure 20.

An obstacle to training DMMs using supervised learning is that the training data only contains targets for the observed variables x, not the latent variables z. A popular approach for training DMMs is using **variational inference (VI)**. It should be noted that VI is a general method for fitting the parameters of statistical models to data. In this special case, we happen to be applying it in a case where there is a dependence between samples in time. For a concrete example of a

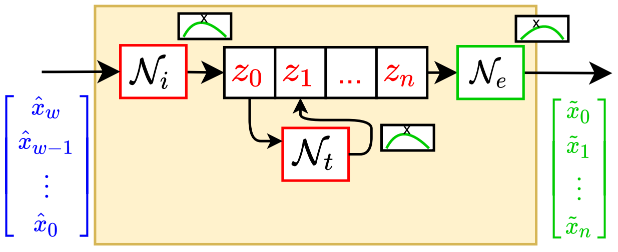


Fig. 20. Deep Markov model with an inference network. The value of z_0 is estimated by an inference network N_i based on several samples of the observed variable. The transmission function, approximated by the network N_t, maps the current value of z to a distribution over z one step ahead in time. The emission function, approximated by N_e, maps each predicted latent variable to a distribution of the corresponding x value in the original observed space. Note that the output of each network is the parameters of a distribution, which is then sampled to obtain a value that can be fed into the next stage of the model.

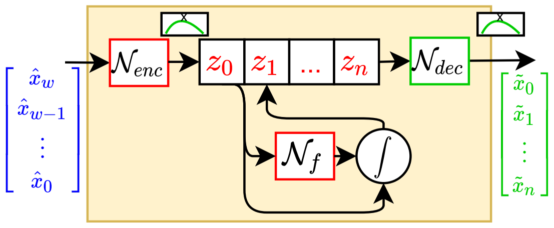


Fig. 21. Latent NODEs. An encoder network is used to obtain a latent representation of the system's initial state, z_0, by aggregating information from several observations of the systems $[\hat{x}_w, \hat{x}_{w-1}, \ldots, \hat{x}_0]$. The system is simulated for multiple steps to obtain $[z_0, z_1, \ldots, z_n]$. Finally, the latent variables are mapped back to the original state space by a decoder network.

training algorithm based on VI that is suitable for training DMM, we refer to the work of Krishnan et al. [65].

Although probability distributions in classical DMMs are assumed to be Gaussian, recent extensions proposed the use of more expressive but also more computationally expensive deep normalizing flows [38, 106]. Another variant of DMM includes additional graph structure for possible encoding of useful inductive biases [98]. DMMs are typically trained using the stochastic counterpart of the backpropagation algorithm [107], which is part of popular open source libraries such as PyTorch-based Pyro [8] or TensorFlow Probability [24]. Applications in dynamical systems modeling span from climate forecasting [17], MD [136], or generic time series modeling with uncertainty quantification [83].

4.5.2 Latent NODEs. Latent NODEs [18] is an extension of NODEs that introduces an encoder and decoder NN to the model as shown in Figure 21. The core of the idea is that information from multiple observations can be aggregated by the encoder network N_{enc} to obtain a latent state z_0, which characterizes the specific trajectory. A convenient choice of encoder network for time series is an RNN because it can handle a variable number of observations. The system can then be simulated using the same approach as NODEs to produce a solution in the latent space. Finally, a decoder network maps each point of the latent solution to the observable space to obtain the final solution.

Separating the measurement, $\mathbf{x}_i$, from the latent system dynamics, $\mathbf{z}_i$, allows us to exploit the modeling flexibility of wider NNs capable of generating more complex latent trajectories. However, by doing so, it creates an inference problem of estimating unknown initial conditions of the hidden states for both deterministic [68, 121] and stochastic time-steppers [20, 62, 63, 68].

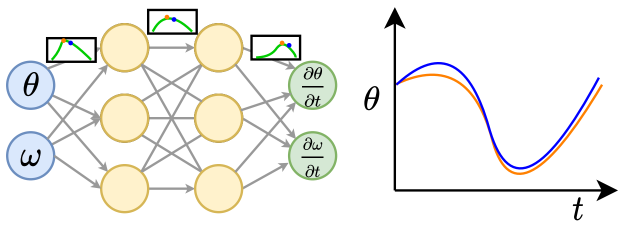

θ
ω
$\frac{\partial\theta}{\partial t}$
$\frac{\partial\omega}{\partial t}$
θ
t

Fig. 22. Bayesian neural ordinary differential equations. The parameters of the network are characterized by a probability distribution. The parameter distributions are sampled multiple times and used to simulate the system, producing multiple trajectories as shown on the right. To get a single prediction, the predictions can be averaged.

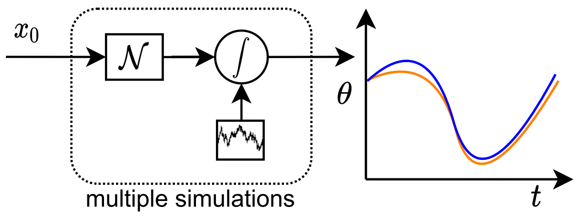


Fig. 23. Neural SDEs. The network $\mathcal{N}$ is used to approximate the deterministic drift term of the SDE and the diffusion term is a Wiener process. Multiple trajectories are produced by solving the SDE multiple times, corresponding to different realizations of the Wiener process.

A difference between latent NODEs and DMMs is that the former treats the state variable as a continuous-time variable and the latter treats it as discrete-time. In addition, latent NODEs assume that the dynamics are deterministic.

4.5.3 Bayesian Neural Ordinary Differential Equations. **Bayesian neural ordinary differential equations (BNODEs)** [22] combine the concept of a NODE with the stochastic nature of **Bayesian neural networks (BNNs)** [53]. In the context of a BNN, the term *Bayesian* refers to the fact that the parameters of the network are characterized by a probability density function rather than an exact value. For instance, the weights of the networks may be assumed to be approximately distributed according to a multivariate Gaussian.

A possible motivation for applying this formalism is that the uncertainty of the model's predictions can be quantified, which would otherwise not be possible. To obtain an estimate of the uncertainty, the model can be simulated several times using different realizations of the model's parameters, resulting in several trajectories as shown in Figure 22. The ensemble of trajectories can then be used to infer confidence bounds and to obtain the mean value of the trajectories.

A drawback of using BNNs and extensions like BNODEs is that they use specialized training algorithms that generally do not scale well to large network architectures. An alternative approach is to introduce sources of stochasticity during the training and inference, such as by using dropout. A categorization of ways to introduce stochasticity that do not require specialized training algorithms is provided in Section 8 of the work of Jospin et al. [53].

4.5.4 Neural Stochastic Differential Equations. Neural **stochastic differential equations (SDEs)** [75] can be viewed as a generalization of an ODE that includes one or more stochastic terms in addition to the deterministic dynamics, as shown in Figure 23. Like the DTMC, an SDE often includes a deterministic drift term and a stochastic diffusion term, such as *Wiener process*:

$$dX = f(x(t))dt + g(x(t))dW_t. \tag{18}$$

Table 2. Comparison of Direct-Solution and Time-Stepper Models

Name	Advantages	Limitations
Direct-solution	+ Easy to apply to PDEs + No discretization of time and spatial coordinates + No accumulation of error during simulation + Parallel evaluation of simulation	- Fixed initial condition - Fixed temporal and spatial domain - Difficult to incorporate inputs
Time-stepper	+ Initial condition not fixed + Easy to incorporate inputs + Leverage knowledge from numerical simulation	- Not trivial to apply to PDEs - Accumulation of error during simulation - No parallel evaluation of simulation

Conventionally, SDEs are expressed in *differential form* unlike the derivative form of an ODE. The reason for this is that many stochastic processes are continuous but cannot be differentiated. The meaning of Equation (18) is per definition the integral equation:

$$x(t) = x_0 + \int_0^t f(x(s))ds + \int_0^t g(x(s))dW_s. \tag{19}$$

As is the case for ODEs, most SDEs must be solved numerically, since only very few SDEs have analytical solutions. Solving SDEs requires the use of algorithms that are different from those used to solve deterministic ODEs. Covering the solvers is outside the scope of this work; instead, we refer to Chapter 9 of the work of Kloeden and Platen [59] for an in-depth coverage. However, in the context of neural SDEs, we can simply think of the solver as a means to simulate systems with stochastic dynamics.

There are several choices for how to incorporate the use of NNs for modeling SDEs. For instance, if the stochastic diffusion term is known, an NN can be trained to approximate the deterministic drift term in Equation (18) as in the case of the work of Liu et al. [74] and Oganesyan et al. [89]. Another approach is to use NNs to parameterize both the drift and diffusion terms [46]. In addition, there are approaches such as those of Xu et al. [138] that incorporate the idea from both neural SDEs and BNNs, by modeling both evolution of the state variables and network parameters as SDEs.

Although neural SDEs provide a strong theoretical framework for modeling uncertainty, they are complex compared to their deterministic counterparts. One way to address this is to examine if simpler and computationally efficient mechanisms like injecting noise or using dropout can achieve some of the same effects as adopting a fully SDE-based framework.

5 DISCUSSION

An important question is how to pick the right type of model for a given application. The two fundamentally different approaches for simulating a system are (i) having an NN approximate the solution of the problem, as described in Section 3, or (ii) having an NN approximate the dynamics of the system, as described in Section 4. Each approach has inherent advantages and limitations, which can be derived by looking at what the NN is used for within the respective type of model. A comparison between the two types of models can be seen in Table 2.

In this survey, we described several variants of direct-solution and time-stepper models. The way that these are presented in the literature often gives the impression that they are fundamentally different. However, applying them to the ideal pendulum system makes it clear that many models are closely related, set apart only by a small extension of the original idea. In the case of the direct-solution models, we observed that the differences between the vanilla direct-solution and the PINN is the application of physics-based regularization and use of AD for obtaining the velocity. In the case of time-stepper models, the main differences boil down to the architecture of the NN and the numerical integration scheme being applied. The ability to pick an NN architecture for a specific application makes it possible to model a wide range of physical phenomena. In addition, the ideas of one model can easily be transferred to another, allowing for the creation of novel architectures. This inherent variability makes it difficult to define concrete guidelines for picking a type of model for a certain application. Instead, we urge the reader to consider what capabilities are needed for the application and how knowledge of the physics incorporated. The topics described by Figure 2 may serve as a starting point for this.

Evaluating the performance of different models on a benchmark dataset consisting of data from various dynamical systems would be quite useful. This dataset should be representative of the systems that are encountered in disciplines such as physics, chemistry, and engineering. This would allow us to identify general trends and heuristics, which would serve as a starting point for new practitioners and future applications. Drawing inspiration from other applications of DL, such as image classification, we see that large image databases have contributed greatly toward developing better NN architectures. A standardized benchmark dataset is an essential step toward gaining more insight into which types of models work well. Not only would it allow for a fair comparison between the NN-based models, but it would also allow us to answer the question of how well these models work compared to traditional models originating from various fields.

Another valuable contribution, would be to define a procedure for evaluating a model's ability to approximate a dynamical system. We are interested in verifying that the model can produce accurate simulations for the initial conditions we would encounter when using the model. Given the diverse nature of these dynamical systems, some may be more difficult for an NN to approximate than others. For instance, a small approximation error in a chaotic system may result in the accumulation of a large error over time. An interesting research topic is determining metrics that allow a fair comparison across multiple dynamical systems.

Another valuable contribution would be to develop concrete guidelines on how to train models of dynamical systems. Finding a rule of thumb for how much training data is necessary to reach a certain degree of accuracy would make it easier to determine if a data-driven approach is feasible for a given application. In addition to determining how much data we need, it would be useful to develop best practices on how to split the data into training and validation sets. For instance, in the context of training time-stepper models, we may examine which length of trajectories result in a good ratio between accuracy and training time. Likewise, it would be useful to determine how to formulate the loss function such that the process of optimizing the model's parameters is fast and robust.

6 SUMMARY

In recent years, there has been an increased interest in applying NNs to solve a diverse set of problems encountered in various branches of engineering and natural sciences. This has resulted in a wealth of papers, each proposing how a particular physical phenomenon can be simulated using NNs. As a consequence, the terminology and notation used in each paper vary greatly, making it difficult to digest for all but experts in the respective field. These papers, often constrained in space, put great emphasis on describing the application and the physics involved, often at a cost of omitting details like how the NN was trained and limitations of proposed methods.

This survey provides an easy-to-follow overview of the techniques for simulating dynamical systems based on NNs. Specifically, we categorized the models encountered in the literature into two distinct types: direct-solution- and time-stepper models. For each type of model, we provided a concrete guide on how to construct, train, and use the model for simulation. Starting from the simplest possible model, we incrementally introduced more advanced variants and established the differences and similarities between the models. In addition, we supply source code for many of the models described in the article, which can be used as a reference for detailed implementation of each model.

An open research question is determining how well these methods work across a broad set of problems that are representative of real-world applications. It is our hope that this survey will support this goal by presenting the most important concepts in a way that is accessible to practitioners coming from DL as well as various branches of physics and engineering.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467 (2016).

[2] Maren Awiszus and Bodo Rosenhahn. 2018. Markov chain neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2180–2187.

[3] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, et al. 2018. Relational inductive biases, deep learning, and graph networks. *CoRR* abs/1806.01261 (2018).

[4] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. 2016. Interaction networks for learning about objects, relations and physics. *CoRR* abs/1612.00222 (2016).

[5] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: A survey. arXiv:1502.05767 [cs, stat] (Feb. 2018).

[6] Jörg Behler. 2015. Constructing high-dimensional neural network potentials: A tutorial review. *International Journal of Quantum Chemistry* 115, 16 (2015), 1032–1050. https://doi.org/10.1002/qua.24890

[7] Jens Behrmann, Will Grathwohl, Ricky T. Q. Chen, David Duvenaud, and Joern-Henrik Jacobsen. 2019. Invertible residual networks. In *Proceedings of the 36th International Conference on Machine Learning*. Proceedings of Machine Learning Research, Vol. 97, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 573–582.

[8] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research* 20, 1 (2019), 973–978.

[9] Christopher Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag, New York, NY.

[10] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. 2021. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *CoRR* abs/2104.13478 (2021).

[11] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. 2020. Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics* 52, 1 (2020), 477–508. https://doi.org/10.1146/annurev-fluid-010719-060214

[12] Keith T. Butler, Daniel W. Davies, Hugh Cartwright, Olexandr Isayev, and Aron Walsh. 2018. Machine learning for molecular and materials science. *Nature* 559, 7715 (July 2018), 547–555. https://doi.org/10.1038/s41586-018-0337-2

[13] François Edouard Cellier. 1991. *Continuous System Modeling*. Springer Science & Business Media.

[14] François Edouard Cellier and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer Science & Business Media.

[15] Bo Chang, Lili Meng, Eldad Haber, Frederick Tung, and David Begert. 2018. Multi-level residual networks from dynamical systems view. arXiv:1710.10348 [cs, stat] (Feb. 2018).

[16] Michael B. Chang, Tomer Ullman, Antonio Torralba, and Joshua B. Tenenbaum. 2016. A compositional object-based approach to learning physical dynamics. *CoRR* abs/1612.00341 (2016).

[17] Zhengping Che, Sanjay Purushotham, Guangyu Li, Bo Jiang, and Yan Liu. 2018. Hierarchical deep generative models for multi-rate multivariate time series. In *Proceedings of the 35th International Conference on Machine Learning*. Proceedings of Machine Learning Research, Vol. 80, Jennifer Dy and Andreas Krause (Eds.). PMLR, 784–793.

[18] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2019. Neural ordinary differential equations. arXiv:1806.07366 [cs, stat] (Dec. 2019).

[19] Travers Ching, Daniel S. Himmelstein, Brett K. Beaulieu-Jones, Alexandr A. Kalinin, Brian T. Do, Gregory P. Way, Enrico Ferrero, et al. 2018. Opportunities and obstacles for deep learning in biology and medicine. *Journal of the Royal Society Interface* 15, 141 (April 2018), 20170387. https://doi.org/10.1098/rsif.2017.0387

[20] Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C. Courville, and Yoshua Bengio. 2015. A recurrent latent variable model for sequential data. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates.

[21] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. 2020. Lagrangian neural networks. arXiv:2003.04630 [physics, stat] (July 2020).

[22] Raj Dandekar, Karen Chung, Vaibhav Dixit, Mohamed Tarek, Aslan Garcia-Valadez, Krishna Vishal Vemula, and Chris Rackauckas. 2021. Bayesian neural ordinary differential equations. arXiv:2012.07244 [cs] (March 2021).

[23] Moritz Diehl, H. Georg Bock, Johannes P. Schlöder, Rolf Findeisen, Zoltan Nagy, and Frank Allgöwer. 2002. Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations. *Journal of Process Control* 12, 4 (2002), 577–585. https://doi.org/10.1016/S0959-1524(01)00023-3
[24] Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matthew D. Hoffman, and Rif A. Saurous. 2017. TensorFlow distributions. *CoRR* abs/1711.10604 (2017).
[25] Ján Drgoňa, Javier Arroyo, Iago Cupeiro Figueroa, David Blum, Krzysztof Arendt, Donghun Kim, Enric Perarnau Ollé, et al. 2020. All you need to know about model predictive control for buildings. *Annual Reviews in Control* 50 (2020), 190–232. https://doi.org/10.1016/j.arcontrol.2020.09.001
[26] Jan Drgona, Aaron R. Tuor, Vikas Chandan, and Draguna L. Vrabie. 2020. Physics-constrained deep learning of multi-zone building thermal dynamics. arXiv:2011.05987 [cs.LG] (2020).
[27] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. 2019. Augmented neural ODEs. arXiv:1904.01681 [stat.ML] (2019).
[28] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M. Oberman. 2020. How to train your neural ODE: The world of Jacobian and kinetic regularization. arXiv:2002.02798 [stat.ML] (2020).
[29] Marc Finzi, Ke Alexander Wang, and Andrew Gordon Wilson. 2020. Simplifying Hamiltonian and Lagrangian neural networks via explicit constraints. *CoRR* abs/2010.13581 (2020).
[30] Marco Forgione and Dario Piga. 2020. dynoNet: A neural network architecture for learning dynamical systems. arXiv:2006.02250 [cs.LG] (2020).
[31] Alexander I. J. Forrester and Andy J. Keane. 2009. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences* 45, 1-3 (Jan. 2009), 50–79. https://doi.org/10.1016/j.paerosci.2008.11.001
[32] Marco Fraccaro, Søren Kaae Sønderby, Ulrich Paquet, and Ole Winther. 2016. Sequential neural models with stochastic layers. arXiv preprint arXiv:1605.07571 (2016).
[33] Jonathan Friedman and Jason Ghidella. 2006. Using model-based design for automotive systems engineering—Requirements analysis of the power window example. *Journal of Passenger Cars: Electronic and Electrical Systems* 115 (2006), 516–521. https://doi.org/10.4271/2006-01-1217
[34] Xiang Gao, Farhad Ramezanghorbani, Olexandr Isayev, Justin S. Smith, and Adrian E. Roitberg. 2020. TorchANI: A free and open source PyTorch-based deep learning implementation of the ANI neural network potentials. *Journal of Chemical Information and Modeling* 60, 7 (2020), 3408–3415. https://doi.org/10.1021/acs.jcim.0c00451
[35] Carlos E. García, David M. Prett, and Manfred Morari. 1989. Model predictive control: Theory and practice—A survey. *Automatica* 25, 3 (1989), 335–348. https://doi.org/10.1016/0005-1098(89)90002-2
[36] C. W. Gear and O. Osterby. 1984. Solving ordinary differential equations with discontinuities. *ACM Transactions on Mathematical Software* 10, 1 (Jan. 1984), 23–44. https://doi.org/10.1145/356068.356071
[37] Daniel Gedon, Niklas Wahlström, Thomas B. Schön, and Lennart Ljung. 2020. Deep state space models for nonlinear system identification. arXiv:2003.14162 [eess.SY] (2020).
[38] Anubhab Ghosh, Antoine Honoré, Dong Liu, Gustav Eje Henter, and Saikat Chatterjee. 2021. Robust classification using hidden Markov models and mixtures of normalizing flows. *CoRR* abs/2102.07284 (2021).
[39] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural message passing for quantum chemistry. *CoRR* abs/1704.01212 (2017).
[40] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*, Vol. 1. MIT Press, Cambridge, MA.
[41] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. 2019. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, 15379–15389.
[42] Batuhan Güler, Alexis Laignelet, and Panos Parpas. 2019. Towards robust and stable deep learning algorithms for forward backward stochastic differential equations. arXiv:1910.11623 [stat.ML] (2019).
[43] Danijar Hafner, Timothy P. Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. 2018. Learning latent dynamics for planning from pixels. *CoRR* abs/1811.04551 (2018).
[44] Ernst Hairer and Gerhard Wanner. 1996. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Number 14. Springer-Verlag, Berlin, Germany.
[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. arXiv:1512.03385 [cs] (Dec. 2015).
[46] Pashupati Hegde, Markus Heinonen, Harri Lähdesmäki, and Samuel Kaski. 2018. Deep learning with differential Gaussian process flows. arXiv:1810.04066 [cs, stat] (Oct. 2018).
[47] Jeen-Shing Wang and Yi-Chung Chen. 2008. A Hammerstein-W recurrent neural network with universal approximation capability. In *Proceedings of the 2008 IEEE International Conference on Systems, Man, and Cybernetics*. 1832–1837. https://doi.org/10.1109/ICSMC.2008.4811555
[48] Junteng Jia and Austin R. Benson. 2019. Neural jump stochastic differential equations. *CoRR* abs/1905.10403 (2019).

[49] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, Weinan E, and Linfeng Zhang. 2020. Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning. arXiv:2005.00223 [physics.comp-ph] (2020).

[50] Bin Jiang, Jun Li, and Hua Guo. 2016. Potential energy surfaces from high fidelity fitting of ab initio points: The permutation invariant polynomial–neural network approach. *International Reviews in Physical Chemistry* 35, 3 (2016), 479–506. https://doi.org/10.1080/0144235X.2016.1200347

[51] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. 2014. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer* 16, 2 (April 2014), 191–213. https://doi.org/10.1007/s10009-013-0289-7

[52] Pengzhan Jin, Aiqing Zhu, George Em Karniadakis, and Yifa Tang. 2020. Symplectic networks: Intrinsic structure-preserving networks for identifying Hamiltonian systems. *CoRR* abs/2001.03750 (2020).

[53] Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. 2020. Hands-on Bayesian neural networks—A tutorial for deep learning users. arXiv:2007.06823 [cs, stat] (July 2020).

[54] Anuj Karpatne, Gowtham Atluri, James H. Faghmous, Michael Steinbach, Arindam Banerjee, Auroop Ganguly, Shashi Shekhar, Nagiza Samatova, and Vipin Kumar. 2017. Theory-guided data science: A new paradigm for scientific discovery from data. *IEEE Transactions on Knowledge and Data Engineering* 29, 10 (Oct. 2017), 2318–2331. https://doi.org/10.1109/TKDE.2017.2720168

[55] Jacob Kelly, Jesse Bettencourt, Matthew James Johnson, and David Duvenaud. 2020. Learning differential equations that are easy to solve. arXiv:2007.04504 [cs.LG] (2020).

[56] Gaëtan Kerschen, Keith Worden, Alexander F. Vakakis, and Jean-Claude Golinval. 2006. Past, present and future of nonlinear system identification in structural dynamics. *Mechanical Systems and Signal Processing* 20, 3 (2006), 505–592. https://doi.org/10.1016/j.ymssp.2005.04.008

[57] Patrick Kidger, Ricky T. Q. Chen, and Terry Lyons. 2020. "Hey, that's not an ODE": Faster ODE adjoints with 12 lines of code. arXiv:2009.09457 [cs, math] (Sept. 2020).

[58] Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. 2018. Neural relational inference for interacting systems. arXiv:1802.04687 [stat.ML] (2018).

[59] Peter E. Kloeden and Eckhard Platen. 1992. *Numerical Solution of Stochastic Differential Equations*. Springer.

[60] Ernesto Kofman and Sergio Junco. 2001. Quantized-state systems: A DEVS approach for continuous system simulation. *Transactions of the Society for Modeling and Simulation International* 18, 3 (2001), 123–132.

[61] Slawomir Koziel and Anna Pietrenko-Dabrowska. 2020. *Basics of Data-Driven Surrogate Modeling*. Springer International Publishing, Cham, Switzerland, 23–58. https://doi.org/10.1007/978-3-030-38926-0_2

[62] R. Krishnan, U. Shalit, and D. Sontag. 2017. Structured inference networks for nonlinear state space models. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*. 2101–2109.

[63] Rahul G. Krishnan, Uri Shalit, and David Sontag. 2015. Deep Kalman filters. arXiv:1511.05121 [stat.ML] (2015).

[64] Rahul G. Krishnan, Uri Shalit, and David Sontag. 2016. Structured inference networks for nonlinear state space models. arXiv:1609.09869 [stat.ML] (2016).

[65] Rahul G. Krishnan, Uri Shalit, and David Sontag. 2016. Structured inference networks for nonlinear state space models. arXiv:1609.09869 [cs, stat] (Dec. 2016).

[66] Andreas Kroll and Horst Schulte. 2014. Benchmark problems for nonlinear system identification and control using soft computing methods: Need and overview. *Applied Soft Computing* 25 (2014), 496–513. https://doi.org/10.1016/j.asoc.2014.08.034

[67] Kookjin Lee and Eric J. Parish. 2021. Parameterized neural ordinary differential equations: Applications to computational physics problems. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 477, 2253 (Sept. 2021), 20210162. https://doi.org/10.1098/rspa.2021.0162

[68] I. Lenz, Ross A. Knepper, and A. Saxena. 2015. DeepMPC: Learning deep latent features for model predictive control. In *Proceedings of the Conference on Robotics: Science and Systems*.

[69] Randall J. LeVeque. 2007. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, Vol. 98. SIAM.

[70] Xuechen Li, Ting-Kam Leonard Wong, Ricky T. Q. Chen, and David Duvenaud. 2020. Scalable gradients for stochastic differential equations. arXiv:2001.01328 [cs.LG] (2020).

[71] Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B. Tenenbaum, and Antonio Torralba. 2018. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *CoRR* abs/1810.01566 (2018).

[72] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B. Tenenbaum, Antonio Torralba, and Russ Tedrake. 2018. Propagation networks for model-based control under partial observation. *CoRR* abs/1809.11169 (2018).

[73] Dong Liu, Antoine Honoré, Saikat Chatterjee, and Lars K. Rasmussen. 2019. Powering hidden Markov model by neural network based generative models. arXiv preprint arXiv:1910.05744 (2019).

[74] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. 2019. Neural SDE: Stabilizing neural ODE networks with stochastic noise. arXiv:1906.02355 [cs.LG] (2019).

[75] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. 2019. Neural SDE: Stabilizing neural ODE networks with stochastic noise. arXiv:1906.02355 [cs, stat] (June 2019).

[76] Lennart Ljung. 2006. Some aspects of non linear system identification. *IFAC Proceedings Volumes* 39, 1 (2006), 110–121. https://doi.org/10.3182/20060329-3-AU-2901.00009

[77] Michael Lutter, Christian Ritter, and Jan Peters. 2019. Deep Lagrangian networks: Using physics as model prior for deep learning. arXiv:1907.04490 [cs, eess, stat] (July 2019).

[78] J. E. Marsden and M. West. 2001. Discrete mechanics and variational integrators. *Acta Numerica* 10 (May 2001), 357–514. https://doi.org/10.1017/S096249290100006X

[79] Stefano Massaroli, Michael Poli, Michelangelo Bin, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. 2020. Stable neural flows. arXiv:2003.08063 [cs.LG] (2020).

[80] Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. 2021. Dissecting neural ODEs. arXiv:2002.08071 [cs.LG] (2021).

[81] D. Masti and A. Bemporad. 2018. Learning nonlinear state-space models using deep autoencoders. In *Proceedings of the 2018 IEEE Conference on Decision and Control (CDC'18)*. 3862–3867.

[82] Sparsh Mittal and Shraiysh Vaishay. 2019. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture* 99 (Oct. 2019), 101635. https://doi.org/10.1016/j.sysarc.2019.101635

[83] George Montanez, Saeed Amizadeh, and Nikolay Laptev. 2015. Inertial hidden Markov models: Modeling change in multivariate time series. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.

[84] Mehrdad Moradi, Cláudio Gomes, Bentley James Oakes, and Joachim Denil. 2019. Optimizing fault injection in FMI co-simulation. In *Proceedings of the 2019 Summer Simulation Conference*. 12. https://doi.org/10.5555/3374138.3374170

[85] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA.

[86] Mohammed Kyari Mustafa, Tony Allen, and Kofi Appiah. 2019. A comparative review of dynamic neural networks and hidden Markov model methods for mobile on-device speech recognition. *Neural Computing and Applications* 31, 2 (2019), 891–899.

[87] Oliver Nelles. 2001. *Nonlinear System Identification: From Classical Approaches to Neural Networks and Fuzzy Models*. Springer-Verlag, Berlin, Germany. https://doi.org/10.1007/978-3-662-04323-3

[88] Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. 2020. On second order behaviour in augmented neural ODEs. arXiv:2006.07220 [cs.LG] (2020).

[89] Viktor Oganesyan, Alexandra Volokhova, and Dmitry Vetrov. 2020. Stochasticity in neural ODEs: An empirical study. arXiv:2002.09779 [cs, stat] (June 2020).

[90] Olalekan Ogunmolu, Xuejun Gu, Steve Jiang, and Nicholas Gans. 2016. Nonlinear systems identification using deep dynamic neural networks. arXiv:1610.01439 [cs] (Oct. 2016).

[91] Olalekan P. Ogunmolu, Xuejun Gu, Steve B. Jiang, and Nicholas R. Gans. 2016. Nonlinear systems identification using deep dynamic neural networks. *CoRR* abs/1610.01439 (2016).

[92] Katharina Ott, Prateek Katiyar, Philipp Hennig, and Michael Tiemann. 2020. When are neural ODE solutions proper ODEs? arXiv:2007.15386 [cs, stat] (July 2020).

[93] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, 8026–8037.

[94] Ludovic Pintard, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. 2013. Fault injection in the automotive standard ISO 26262: An initial approach. In *Dependable Computing*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, et al. (Eds.), Vol. 7869. Springer, Berlin, Germany, 126–133. https://doi.org/10.1007/978-3-642-38789-0_11

[95] Alessio Plebe and Giorgio Grasso. 2019. The unbearable shallow understanding of deep learning. *Minds and Machines* 29, 4 (Dec. 2019), 515–553. https://doi.org/10.1007/s11023-019-09512-8

[96] Michael Poli, Stefano Massaroli, Junyoung Park, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. 2020. Graph neural ordinary differential equations. arXiv:1911.07532 [cs.LG] (2020).

[97] Tong Qin, Kailiang Wu, and Dongbin Xiu. 2019. Data driven governing equations approximation using deep neural networks. *Journal of Computational Physics* 395 (Oct. 2019), 620–635. https://doi.org/10.1016/j.jcp.2019.06.042

[98] Meng Qu, Yoshua Bengio, and Jian Tang. 2019. GMNN: Graph Markov neural networks. In *Proceedings of the International Conference on Machine Learning*. 5241–5250.

[99] Alessio Quaglino, Marco Gallieri, Jonathan Masci, and Jan Koutník. 2020. SNODE: Spectral discretization of neural ODEs for system identification. arXiv:1906.07038 [cs.NE] (2020).

[100] R. Rai and C. K. Sahu. 2020. Driven by data or derived through physics? A review of hybrid physics guided machine learning techniques with cyber-physical system (CPS) focus. *IEEE Access* 8 (2020), 71050–71073.

[101] M. Raissi, P. Perdikaris, and G. E. Karniadakis. 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics* 378 (Feb. 2019), 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

[102] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. 2018. Multistep neural networks for data-driven discovery of nonlinear dynamical systems. arXiv:1801.01236 [nlin, physics:physics, stat] (Jan. 2018).

[103] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. 2020. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science* 367, 6481 (Feb. 2020), 1026–1030. https://doi.org/10.1126/science.aaw4741

[104] Syama S. Rangapuram, Matthias W. Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. 2018. Deep state space models for time series forecasting. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, 7785–7794.

[105] Saman Razavi, Bryan A. Tolson, and Donald H. Burn. 2012. Review of surrogate modeling in water resources. *Water Resources Research* 48, 7 (July 2012), 1–32. https://doi.org/10.1029/2011WR011527

[106] Danilo Jimenez Rezende and Shakir Mohamed. 2016. Variational inference with normalizing flows. arXiv:1505.05770 [stat.ML] (2016).

[107] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the International Conference on Machine Learning*. 1278–1286.

[108] David Rolnick, Priya L. Donti, Lynn H. Kaack, Kelly Kochanski, Alexandre Lacoste, Kris Sankaran, Andrew Slavin Ross, et al. 2019. Tackling climate change with machine learning. arXiv:1906.05433 [cs, stat] (Nov. 2019).

[109] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *CoRR* abs/2006.10637 (2020).

[110] Lars Ruthotto and Eldad Haber. 2018. Deep neural networks motivated by partial differential equations. arXiv:1804.04272 [cs, math, stat] (Dec. 2018).

[111] Lars Ruthotto and Eldad Haber. 2020. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision* 62, 3 (April 2020), 352–364. https://doi.org/10.1007/s10851-019-00903-1

[112] Alvaro Sanchez-Gonzalez, Victor Bapst, Kyle Cranmer, and Peter W. Battaglia. 2019. Hamiltonian graph networks with ODE integrators. *CoRR* abs/1909.12790 (2019).

[113] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. 2020. Learning to simulate complex physics with graph networks. *CoRR* abs/2002.09405 (2020).

[114] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin A. Riedmiller, Raia Hadsell, and Peter W. Battaglia. 2018. Graph networks as learnable physics engines for inference and control. *CoRR* abs/1806.01242 (2018).

[115] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[116] Johan Schoukens and Lennart Ljung. 2019. Nonlinear system identification: A user-oriented roadmap. *CoRR* abs/1902.00683 (2019).

[117] M. Schoukens and J. P. Noël. 2017. Three benchmarks addressing open challenges in nonlinear system identification. *IFAC-PapersOnLine* 50, 1 (2017), 446–451. https://doi.org/10.1016/j.ifacol.2017.08.071

[118] Dieter Schramm, Wildan Lalo, and Michael Unterreiner. 2010. Application of simulators and simulation tools for the functional design of mechatronic systems. *Solid State Phenomena* 166–167 (Sept. 2010), 1–14. https://doi.org/10.4028/www.scientific.net/SSP.166-167.1

[119] Kristof T. Schütt, Pieter-Jan Kindermans, Huziel E. Sauceda, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. 2017. SchNet: A continuous-filter convolutional neural network for modeling quantum interactions. arXiv:1706.08566 [stat.ML] (2017).

[120] Elliott Skomski, Jan Drgona, and Aaron Tuor. 2020. Physics-informed neural state space models via learning and evolution. arXiv:2011.13497 [cs.NE] (2020).

[121] Elliott Skomski, Soumya Vasisht, Colby Wight, Aaron Tuor, Jan Drgona, and Draguna Vrabie. 2021. Constrained block nonlinear neural dynamical models. arXiv:2101.01864 [math.DS] (2021).

[122] B. Sohlberg and E. W. Jacobsen. 2008. Grey box modelling—Branches and experiences. *IFAC Proceedings Volumes* 41, 2 (2008), 11415–11420. https://doi.org/10.3182/20080706-5-KR-1001.01934

[123] Heung-Il Suk, Chong-Yaw Wee, Seong-Whan Lee, and Dinggang Shen. 2016. State-space model with deep learning for functional dynamics estimation in resting-state fMRI. *NeuroImage* 129 (2016), 292–307. https://doi.org/10.1016/j.neuroimage.2016.01.005

[124] Peter Toth, Danilo Jimenez Rezende, Andrew Jaegle, Sébastien Racanière, Aleksandar Botev, and Irina Higgins. 2020. Hamiltonian generative networks. arXiv:1909.13789 [cs, stat] (Feb. 2020).

[125] Oliver T. Unke and Markus Meuwly. 2018. A reactive, scalable, and transferable model for molecular energies from a neural network approach based on local information. *Journal of Chemical Physics* 148, 24 (2018), 241708. https://doi.org/10.1063/1.5017898

[126] Oliver T. Unke and Markus Meuwly. 2019. PhysNet: A neural network for predicting energies, forces, dipole moments, and partial charges. *Journal of Chemical Theory and Computation* 15, 6 (2019), 3678–3693. https://doi.org/10.1021/acs.jctc.9b00181

[127] Felipe A. C. Viana, Christian Gogu, and Raphael T. Haftka. 2010. Making the most out of surrogate models: Tricks of the trade. In *Volume 1: 36th Design Automation Conference, Parts A and B*. ASMEDC, Montreal, Quebec, Canada, 587–598. https://doi.org/10.1115/DETC2010-28813

[128] Laura von Rueden, Sebastian Mayer, Katharina Beckh, Bogdan Georgiev, Sven Giesselbach, Raoul Heese, Birgit Kirsch, et al. 2020. Informed machine learning—A taxonomy and survey of integrating knowledge into learning systems. arXiv:1903.12394 [cs, stat] (Feb. 2020).

[129] Laura von Rueden, Sebastian Mayer, Rafet Sifa, Christian Bauckhage, and Jochen Garcke. 2020. Combining machine learning and simulation to a hybrid modelling approach: Current and future directions. *Advances in Intelligent Data Analysis XVIII* 12080 (2020), 548–560. https://doi.org/10.1007/978-3-030-44584-3_43

[130] Jiang Wang, Simon Olsson, Christoph Wehmeyer, Adrià Pérez, Nicholas E. Charron, Gianni de Fabritiis, Frank Noé, and Cecilia Clementi. 2019. Machine learning of coarse-grained molecular dynamics force fields. *ACS Central Science* 5, 5 (2019), 755–767. https://doi.org/10.1021/acscentsci.8b00913

[131] Sifan Wang, Yujun Teng, and Paris Perdikaris. 2020. Understanding and mitigating gradient pathologies in physics-informed neural networks. arXiv:2001.04536 [cs, math, stat] (Jan. 2020).

[132] Sifan Wang, Xinling Yu, and Paris Perdikaris. 2020. When and why PINNs fail to train: A neural tangent kernel perspective. arXiv:2007.14527 [cs, math, stat] (July 2020).

[133] G. Wanner and E. Hairer. 1991. *Solving Ordinary Differential Equations I: Nonstiff Problems*, Vol. 1. Springer-Verlag.

[134] Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. 2017. Visual interaction networks: Learning a physics simulator from video. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, 4542–4550.

[135] Paul Westermann and Ralph Evins. 2019. Surrogate modelling for sustainable building design—A review. *Energy and Buildings* 198 (Sept. 2019), 170–186. https://doi.org/10.1016/j.enbuild.2019.05.057

[136] Hao Wu, Andreas Mardt, Luca Pasquali, and Frank Noe. 2018. Deep generative Markov state models. arXiv preprint arXiv:1805.07601 (2018).

[137] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A comprehensive survey on graph neural networks. *CoRR* abs/1901.00596 (2019).

[138] Winnie Xu, Ricky T. Q. Chen, Xuechen Li, and David Duvenaud. 2021. Infinitely deep Bayesian neural networks with stochastic differential equations. arXiv:2102.06559 [cs, stat] (Aug. 2021).

[139] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, and Weinan E. 2018. Deep potential molecular dynamics: A scalable model with the accuracy of quantum mechanics. *Physical Review Letters* 120, 14 (April 2018), 143001. https://doi.org/10.1103/PhysRevLett.120.143001

[140] Linfeng Zhang, Jiequn Han, Han Wang, Wissam A. Saidi, Roberto Car, and Weinan E. 2018. End-to-end symmetry preserving inter-atomic potential energy model for finite and extended systems. arXiv:1805.09003 [physics.comp-ph] (2018).

[141] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2018. Deep learning on graphs: A survey. *CoRR* abs/1812.04202 (2018).

[142] Yaofeng Desmond Zhong, Biswadip Dey, and Amit Chakraborty. 2019. Symplectic ODE-Net: Learning Hamiltonian dynamics with control. *CoRR* abs/1909.12077 (2019).

[143] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

Received 2 November 2021; revised 23 September 2022; accepted 4 October 2022

Chapter 7

A Universal Mechanism for Implementing Functional Mock-up Units

The paper presented in this chapter has been published in the peer-reviewed conference *International Conference on Simulation and Modeling Methodologies, Technologies and Applications.*

A Universal Mechanism for Implementing Functional Mock-up Units

Christian Møldrup Legaard[1], Daniella Tola[1], Thomas Schranz[2],
Hugo Daniel Macedo[1] and Peter Gorm Larsen[1]
[1]*DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Aarhus, Denmark*
[2]*Graz University of Technology, Graz, Austria*

Keywords: Co-simulation, Functional Mock-up Interface, Functional Mock-up Unit, Tool.

Abstract: Producing independent simulation units that can be used in a Functional Mock-Up Interface (FMI) setting is challenging. In some cases, a modelling tool may be available that provides the exact capabilities needed by exporting such units. However, there may be cases where existing tools are not suitable, or the cost is prohibitive, thus it may be necessary to implement a Functional Mock-up Unit (FMU) from scratch. Correctly implementing an FMU from scratch requires a deep technical understanding of the FMI specification and the technologies it is built upon. A consequence of FMI being a C-based standard is that an FMU must, generally, be implemented in C or a compiled language that offers a binary-compatible with C such as C++, Rust, or Fortran. In this paper we present UniFMU, a tool that makes it possible to implement FMUs in any language, by writing an adapter that can be plugged in to our modular approach. UniFMU also provides both a graphical user interface and command-line interface feature for generating new FMUs from a selection of programming languages. We expect our tool and approach to be useful for the simulation community both when porting simulators written in languages without FMI support, and when implementing or re-implementing such support.

1 INTRODUCTION

When modelling Cyber-Physical Systems (CPSs), it is advantageous to model different parts using different formalisms and tools and then combine the different models as simulation units using co-simulation (Gomes et al., 2018). One of the most popular standards for co-simulation is called the Functional Mock-up Interface (FMI) (Modelica Association, 2019), which defines how different simulators are coupled and a format for packaging simulation units. A co-simulation combines a number of such packaged simulation units, termed Functional Mock-up Units (FMUs), using a master algorithm that combines the simulation of each of the independent simulation units (Thule et al., 2019a; Thule et al., 2019b) into a joint simulation of the system.

A common way to obtain FMUs is to use FMI-enabled modelling tools such as OMEdit (Asghar and Tariq, 2010), Simulink (Simulink09, 2009) or 20-sim (Controllab Products B.V., 2013) to create models interactively using a GUI, which can subsequently be exported as FMUs. While existing tools may cover the needs of most modelling applications, the need for specialized FMUs that can only be implemented by hand will frequently arise. Unfortunately, the process of creating an FMU from scratch is cumbersome and difficult as it requires:

- A deep understanding of the FMI specification.
- The code to be implemented in a C-compatible language.
- Cross-compilation to support multiple platforms.
- Manual creation and synchronization of the modelDescription.xml file.
- Correct packaging of assets as an FMU archive.

Due to the many pitfalls of this process, it is impractical for anyone but experts to produce FMUs by hand.

This paper presents an extendable tool called *Universal Functional Mock-up Unit* (UniFMU) that facilitates the implementation of FMUs in any programming language. Specifically, our contribution is a tool that provides:

- Support for Python, C# and Java FMUs out of the box.

Legaard, C., Tola, D., Schranz, T., Macedo, H. and Larsen, P.
A Universal Mechanism for Implementing Functional Mock-up Units.
DOI: 10.5220/0010577601210129
In *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2021)*, pages 121-129
ISBN: 978-989-758-528-9

- An easy to use extension mechanism to provide support for any language.
- CLI to generate template FMUs using a single command.
- Pre-built binaries for Windows, Linux and macOS, eliminating the need for cross-compilation and complex tool-chain setup.
- Flexible configuration of the execution environment, such as running inside a Docker container or activating a virtual environment.
- A work in progress GUI for modifying FMU's modelDescription.xml files, eliminating the need to modify by hand.

The generated FMUs are fully FMI compliant meaning they can be used in any FMI enabled co-simulation tool, without making any modifications. The tool is freely available and can be accessed in the GitHub repository: *https://github.com/INTO-CPS-Association/unifmu/*.

We expect our tool and approach to be useful for the simulation community. As we describe in Section 5, by implementing one of two available protocols or using one of our language backends a user porting simulators written in languages without FMI support can focus on implementing the FMI standard functionality in his favourite language. Our work also lowers the complexity required to modeling tool providers interested in implementing or improve tools with FMI export capabilities. With our work the tool can rely on a modular approach where UniFMU deals with the C API and the export implementation can focus on providing the model semantics.

In the following, we provide a brief introduction to related work in Section 2. Then, Section 3 provides an introduction to FMI with an emphasis on the implementation of an FMU from scratch. Next, Section 4 demonstrates the alternative implementation facilitated by UniFMU. Next, Section 5 provides the details of our approach that may be useful to adopters interested to generate FMUs using our tool. Following this Section 6 describes how UniFMU executes the authors code inside the FMU as well as how the tool can be extended to support a new language. Section 7 then describes how the generated FMUs can be ran inside a docker container. Finally, Section 8 provide a few concluding remarks including the planned future work.

2 RELATED WORK

The difficulty of authoring FMUs by hand has led to the development of several tools that support the authoring of FMUs. Typically, each tool focuses on supporting a specific language and implements its own workflow. A selection of this type of tool is seen in Table 1.

Table 1: Overview of tools that can: (i) import and simulate FMUs, (ii) export models as FMUs.

Name	Language	Import	Export
FMUSDK	C	x	x
FMI++	C/C++	x	x
FMI4j	Java	x	x
JavaFMI	Java	x	x
JFMI	Java	x	
PythonFMU	Python		x
PyFMU	Python		x
OvertureFMU	VDM		x

A drawback of using language specific tools is that it requires the user to install and learn how to use several different tools. Another issue is that these tools tend to cover only the most popular languages and/or languages where interoperability with C is easy to implement.

In addition to tools that allow authoring of FMUs, some simulation tools allow user written code to be mixed with the code implemented by FMUs. For example (fmp, 2021; Widl et al., 2013) allow FMUs to be imported and simulated in Python. This makes it possible to insert additional Python code in the simulation loop, essentially implementing a *virtual FMU* without the hassle of packaging. Clearly, this approach for mixing in code is rather limited in terms of reuse and the inability to mix different languages in the simulation. Our tool provides a generic way to generate FMUs that can be used in any FMI enabled simulation tool.

More relevant are works that provide a separation between the FMI C-API and the implementation of the FMU such *Proxy-FMU* developed in (Hatledal et al., 2019). Here, the authors propose using *remote procedure calling* (RPC) to allow FMUs to run in a distributed setting, while at the same time removing restrictions on the FMUs implementation language. A further development by the same authors is the *fmu-proxify* tool that allows existing FMUs to be wrapped in a way such that it enables distributed co-simulation in any FMI enabled simulation tool.

Similar to Proxy-FMU our tool also uses RPC as a mechanism to enable the execution of arbitrary code inside an FMU. The main contribution of UniFMU is that the tool ships with support for several languages that for which template FMUs can quickly be generated using the CLI. Another distinction is that our tool makes it easy to run additional commands like selecting a specific virtual environment to run the FMU inside or deploying it in a docker container as described in section 7.

3 CREATING AN FMU IN C

One of the most difficult and time consuming aspects of FMI based co-simulation is implementing models and packaging them as FMUs. In many cases, free or commercial tools are available that automate the conversion of the model into an FMU. However, in practice there are situations where no tool covering your particular needs exists. For instance consider use cases where an FMU would need to:

- Capture input from a operator through a GUI.
- Interact with remote hardware.
- Integrate data-driven components such as neural-networks.

In such cases it may be necessary to implement the FMU from scratch in C. As a running example we introduce a simple *Adder*-FMU that takes the sum of its two inputs to form its output, as shown in fig. 1. We chose this example to illustrate the effort required to implement even the simplest FMU from scratch. Additionally, we omit details of how to implement many of the specialized FMI methods, that require careful considerations to memory management. Hopefully, this will be enough to convince the reader that creating a more complex FMU from scratch would be even more challenging and time consuming.

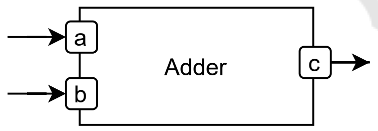


Figure 1: Adder FMU computes c = a + b.

To understand the challenges of generating an FMU we first examine the its structure. In plain terms an FMU is a zip-archive containing a collection of files that together define the interface and behavior of a model. A minimal example of how the contents of the adder FMU zip folder may look like is depicted in fig. 2.

The folder must contain the functional behavior of an FMU, which is realized by a shared library (unifmu.so) located in the `binaries` directory (one for each operating system), and a configuration file, the `modelDescription.xml`, stored in the root of the FMU that provides metadata about the unit. At runtime, the master algorithm dynamically links the binary for the specific platform allowing the master to invoke the appropriate methods to get and set

```
adder_c.fmu
 └─binaries
 │  ├─linux64
 │  │  └─adder.so
 │  ├─windows64
 │  │  └─adder.dll
 │  └─darwin64
 │     └─adder.dylib
 └─modelDescription.xml
```

Figure 2: Directory structure of adder's C implementation.

variables of the model through the *C-API*. To obtain this library several methods declared in the FMI's C-headers must be implemented in a C compatible language and compiled separately for every platform that the FMU is expected to be run on. A small selection of these is seen in listing 1.

```c
typedef struct
{
 fmi2Real a;
 fmi2Real b;
 fmi2Real c;
} Adder;

void *fmi2Instantiate(const char *
    name,
    ...)
{
 Adder *instance = malloc(...);
 return instance;
}

int fmi2DoStep(void *c, ...)
{
 Adder *fmu = c;
 fmu->c = fmu->a + fmu->b;
 return fmi2OK;
}
```

Listing 1: Implementation of Adder in C.

The code shown in the snippet represents a simplified implementation of the many details and functions that must be implemented, and in addition to it we compile a binary for each architecture including the FMI standard headers as shown in fig. 3. In addition to the sheer number of functions that have to be implemented, low-level programming considerations of memory management and ownership of strings makes it difficult to implement the functions correctly. This has been described as one of the challenges of creating FMUs, as the documentation of the FMI standard is insufficient (Schweiger et al., 2019; Bertsch et al., 2014).

The binaries provides the functional behavior of the model, the `modelDescription.xml` declares the

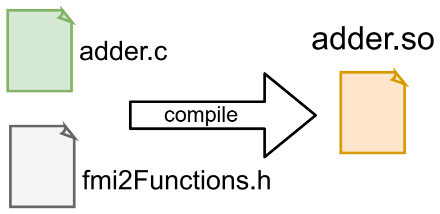


Figure 3: FMI headers and their implementation compiled into a shared library.

interface and capabilities of the model. Among other things, this file declares inputs and outputs of the model, their type and their default value. This information is used by the master algorithm to connect pairs of inputs and outputs of models during the configuration of a co-simulation. It should be stressed that in the general case the binary itself is oblivious to the contents of the `modelDescription.xml` file. As such special care should be taken to ensure that the variables declared in the description are consistent with the implementation. For example if one of the inputs to the adder is declared as an integer in the description rather than a floating point number, the binary would still treat it as a floating point number leading to an incorrect output value. We discuss the issue of ensuring consistency of the `modelDescription.xml` further in section 8.

4 CREATING AN FMU IN UniFMU

UniFMU provides a *Command Line Interface* (CLI) that can be used to author FMUs in several popular languages such as *Python*, *C#* and *Java* as shown in Table 2, and we plan to expand the list of supported languages in the future. We distinguish between a backend and a language, since one language can implement multiple backends. We define a backend as the method of communication between the UniFMU wrapper and the FMU. This is described in more detail in Section 5.

Table 2: List of supported languages and backends, * denotes default backend.

Language	Backends
Python	gRPC*, ZeroMQ
C#	gRPC
Java	gRPC

The CLI is implemented in Python, but it should be stressed that the generated FMUs do not depend on Python during simulation (except for FMUs implemented in Python). The tool can be installed using, *pip*, the de-facto package manager for Python, using a single command:

```
pip install unifmu
```

The package manager installs the CLI as well as any resources needed during the generation and runtime of the FMUs. Alternatively, the tool can be installed from source using the instructions found in the associated GitHub repository[1].

To generate an FMU you invoke the program with the sub-command *generate* with arguments specifying the language and name of the exported FMU. For example to generate an FMU named `python_adder.fmu` in Python the following command can be used:

```
unifmu generate python python_adder.fmu
```

Executing this command creates an FMU with the file structure shown in fig. 4. The generated FMU is fully functional and serves as a template that can be modified to implement the desired behavior for the model.

```
adder.fmu
├── binaries
│   ├── darwin64
│   ├── linux64
│   │   └── unifmu.so
│   └── windows64
├── resources
│   ├── model.py^i
│   ├── backend_grpc.py^b
│   ├── fmi2.py^b
│   └── launch.toml
└── modelDescription.xml
```

Figure 4: Python FMU directory tree. ^{b} denotes backend, ^{i} denotes implementation.

An important difference between an FMU implemented in C and one implemented using UniFMU is that the behavior of the model is not defined by the binaries, but rather by the file(s) stored in the `resources` folder. This makes it possible to reuse the same binaries for all FMUs, independently of the language that they are implemented in. These binaries are pre-compiled and shipped with the tool for Linux, Windows and macOS. Two benefits of this is that the FMU can run on all platforms without any additional effort from the author and that it is not necessary to install a compiler tool-chain.

The `model.py` file shown in fig. 4 implements the functionality of the FMU. Inspecting the code inside of the `model.py` FMU, the most relevant function is `do_step` shown in listing 2.

[1] https://github.com/INTO-CPS-Association/unifmu

```python
class Model(Fmi2FMU):
    def __init__(self):
        self.a = 0.0
        self.b = 0.0
        self._update_outputs()

    def _update_outputs(self):
        self.c = self.a + self.b

    def exit_initialization_mode(
   self):
        self._update_outputs()
        return Fmi2Status.ok

    def do_step(self, ...):
        self._update_outputs()
        return Fmi2Status.ok
```

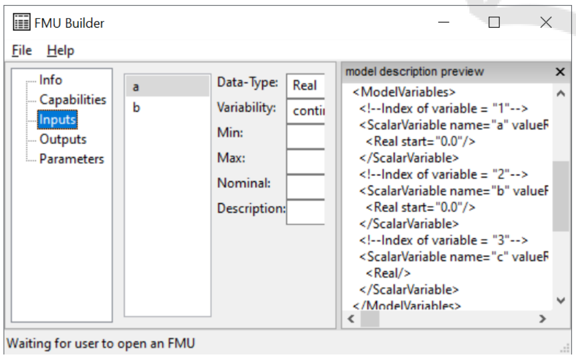
Listing 2: model.py implementation.

It contains the addition operation functionality provided by the FMU, and the variables `a` and `b` are the inputs to the FMU, and the variable `c` is an output variable containing the result of the addition of the two inputs. These variables and their types are all defined in the `modelDescription.xml` file.

In addition to the CLI, UniFMU can be launched as a GUI, seen in fig. 5, using the subcommand:

```
unifmu gui
```

The GUI is currently work in progress, and can so far be used to access the same functionality as the CLI. The vision is to extend this GUI to be able to declare and modify the variables corresponding to the contents of the modelDescription.xml.



a
b

Data-Type: Real
Variability: conti
Min:
Max:
Nominal:
Description:

```
<ModelVariables>
  <!--Index of variable = "1"-->
  <ScalarVariable name="a" valueR
    <Real start="0.0"/>
  </ScalarVariable>
  <!--Index of variable = "2"-->
  <ScalarVariable name="b" valueR
    <Real start="0.0"/>
  </ScalarVariable>
  <!--Index of variable = "3"-->
  <ScalarVariable name="c" valueR
    <Real/>
  </ScalarVariable>
</ModelVariables>
```

Waiting for user to open an FMU

Figure 5: GUI for UniFMU.

5 HOW DOES IT WORK?

The main motivation behind UniFMU is to allow arbitrary code written in any language to be executed within an FMU. The mechanism used by the tool is to provide a generic binary that spawns a separate process for each instantiated slave during runtime. Unlike the binary, the spawned processes aren't restricted to the narrow set of compiled languages that are conventionally used to implement FMUs. This opens the possibility for using interpreted languages or languages that rely on a garbage collector to manage memory.

In order to forward the FMI calls from the master algorithm to the concrete implementation provided *slave processes* a remote procedure call (RPC) based on *gRPC*[2] is used to pass commands from the binary to the slave process. Seen from the perspective of the master algorithm this additional layer of indirection is totally opaque, meaning that FMUs produced by the tool can be used in any FMI compliant simulation tool.

Comparing this with the conventional approach, we add an extra layer between the master algorithm and the FMU itself. The comparison between the conventional and UniFMU approach is illustrated in fig. 6. In both approaches the master algorithm communicates with the binary files through the C API that implements the FMI standard. The difference is that the UniFMU wrapper "translates" these C API calls to messages that can be exchanged through a RPC, such as gRPC or ZeroMQ. Only one of these two backends is necessary to implement when supporting a new language. The specific backend used is defined in the configuration file of the FMU, as shown in the launch.toml file in listing 3. In addition to the backend, the launch.toml file specifies the command used to start the backend process. It is possible to specify different commands based on different OSs, since they may require a different setup.

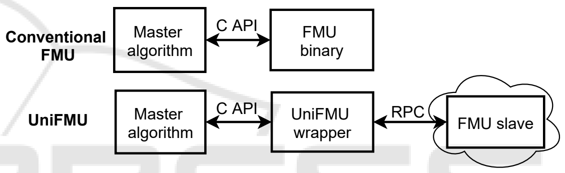


Figure 6: Conventional versus UniFMU approach.

```toml
[grpc]
linux = ["python3", "backend_grpc.py"]
macos = ["python3", "backend_grpc.py"]
windows = ["python", "backend_grpc.py"]
```

Listing 3: launch.toml.

gRPC. Is a RPC that is based on HTTP/2 for transporting messages, and Protocol Buffers (protobuf)[3] for describing the information in the messages as a

[2]https://grpc.io/

[3]https://developers.google.com/protocol-buffers

schema. An example of the `DoStep` schema, defined in the protobuf file, is illustrated in listing 4.

```
service SendCommand{
    rpc Fmi2DoStep(DoStep)
        returns (StatusReturn) {}
}
message DoStep{
    double current_time = 1;
    double step_size = 2;
    bool no_step_prior = 3;
}
```

Listing 4: Structure of DoStep message and Fmi2DoStep call in protobuf schema.

ZeroMQ. ZeroMQ[4] is a networking library that allows messages to be transmitted efficiently across transport layers such as TCP or as Inter Process Communication. Unlike the gRPC backend there is no explicit schema-file that dictates the structure of the messages. Instead, the messages are structured according to a simple protocol described in the developer documentation found in the UniFMU repository. The serialization of the messages is performed by the Serde[5] library. This makes it possible to automatically generate high-performance serialization for several formats such as:

- JSON
- Pickle
- Flatbuffers

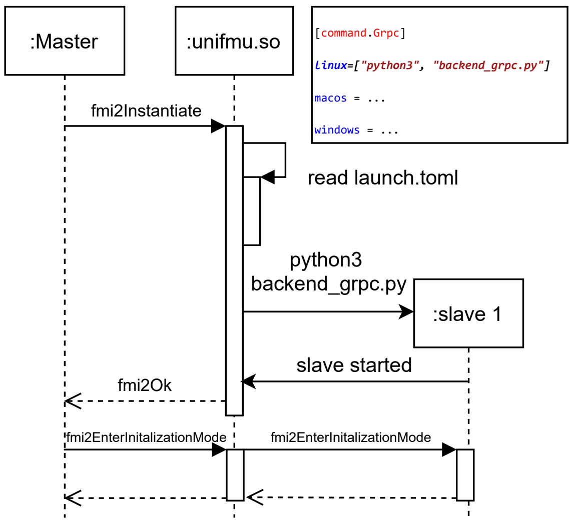
For dynamically typed languages such as Python or JavaScript the schemaless approach may be simpler to implement.

The main difference between these two backends is that gRPC uses a protobuf schema, while ZeroMQ is schemaless. Using a schema to define the messages and calls between the UniFMU wrapper and the FMU allows to declare the types of each message. This reduces the risk of using incorrect types when implementing the functions, easing the process of creating FMUs in statically-typed languages.

To understand the flexibility of this approach, it is useful to examine the process for creating an instance of an FMU, and the forwarding of the FMI commands from the binary to the slave instance, as depicted in fig. 7.

First the master algorithm invokes the `fmi2Instantiate` function defined by the binary. Following this, the wrapper reads the *launch.toml* file, which is present in any FMU generated by UniFMU. This is the configuration file which defines

[4]https://zeromq.org/
[5]https://serde.rs/



```
[command.Grpc]
Linux=["python3", "backend_grpc.py"]
macos = ...
windows = ...
```

Figure 7: Instantiation of slave and forwarding of FMI method calls.

the details about which communication backend is used to communicate with the slave process and even more importantly the specific command used to spawn the process. The *launch.toml* file in fig. 7 shows how the commands defined in the file are used to instantiate a Python FMU. For example an FMU implemented in Python using the gRPC backend would use the following command to launch the slave process:

```
python3 backend_grpc.py
```

Here the backend_grpc.py script serves as an implementation by implementing the `Fmi2DoStep` command, defined in the protobuf schema shown in listing 4. The `DoStep` message declares the parameters used in the `Fmi2DoStep` command. The complete protobuf schema can be found in our GitHub repository.

6 HOW TO EXTEND SUPPORT TO A NEW LANGUAGE?

There may be cases where a user would like to implement an FMU in a language not yet supported by the tool. One of the advantages of UniFMU is that new languages can be added without making any modification to the binaries of the FMU. This section shows the steps followed to extend the UniFMU support to include C# by implementing a gRPC backend. We chose the gRPC option, because a schema based serialization format like the one used by gRPC is especially suitable for compiled languages as it allows messages to be constructed using strongly typed objects. Following our running example, we will use the adder introduced in section 3 as a concrete example.

The implementation of the gRPC backend in C# amounts to implement the remote procedure calls defined in *unifmu_fmi2.proto* file under `schemas` in the repository[6]. The protobuf compiler, *protoc*, is then used to compile the UniFMU protobuf schema into C# code, as illustrated in fig. 8.

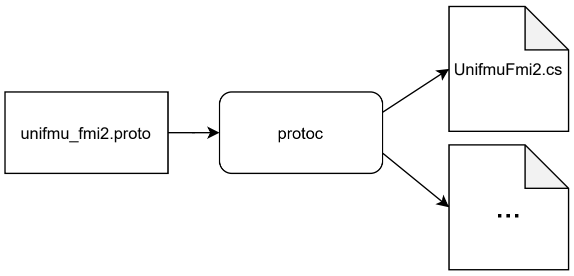


Figure 8: Generate C# files containing the protobuf schema.

The next steps are to create a base class of the FMI2 functions and implement the functionality of the adder. This can easily be done by translating these implementations from Python to C# code. The adder derives functions from the FMI2 base class, thus the specific implementations of the FMI2 functions are implemented in each FMU. After this, the gRPC server is coded, implementing the functions that were generated from the protobuf schema and calling them on the adder. A snippet of the `Fmi2DoStep` function implementation in the gRPC server is shown in **??**.

```
public override Task<StatusReturn>
    Fmi2DoStep(DoStep request,...)
{
    Fmi2Status status =
    this.fmu.DoStep(
        request.CurrentTime,
        request.StepSize,
        request.NoStepPrior);
    ...
}
```

Listing 5: gRPC server implementation of Fmi2DoStep.

The `Fmi2doStep` implementation is the only necessary function to implement in the adder, since this is where the core functionality is described. Listing 6 shows the `Fmi2doStep` implementation of the adder.

[6]https://github.com/INTO-CPS-Association/unifmu/

```
public override Fmi2Status DoStep(
    double currentTime, ...)
{
    this.c = this.a + this.b;
    return Fmi2Status.Ok;
}
```

Listing 6: C# example of fmi2doStep implementation.

One of the last steps is to implement the handshake connection that will be performed between the UniFMU wrapper and the FMU itself. This can be done in the `backend_grpc.cs` file, where the specific FMU to initialize can also be defined. The last step is to define the commands to be called for initialization of the FMU in the `launch.toml` file.

When generating a new C# FMU, the file structure will be as illustrated in fig. 9, where the `adder.cs` file can be exchanged for any other C# file.

```
adder_csharp.fmu
├─binaries
├─resources
│ ├─schemas
│ ├─model.cs^i
│ ├─backend_grpc.cs^b
│ ├─fmi2.cs^b
│ ├─csharp.csproj
│ └─launch.toml
└─modelDescription.xml
```

Figure 9: C# FMU directory tree. ^{b} denotes backend, ^{i} denotes implementation.

As we have demonstrated in this section, there is no need to write any C/C++ code when creating a new backend for UniFMU. The work amounts to implement a gRPC server using the protobuf schema and an abstract class containing the FMI2 function declarations.

7 EXECUTING FMUs INSIDE DOCKER CONTAINER

Invoking code from within an FMU needs the host machine to provide the necessary runtime environment to do so. For example, running python scripts requires that the host machine has a compatible python interpreter and all libraries used in the scripts installed. This limits the portability of the FMUs, especially when shared between different host machines, potentially running different operating systems.

To mitigate this issue runtime dependencies can be packaged into a virtualization environment such as docker[7]. A detailed description of the extension and more advanced topics, such as remote deployment, building and deployment settings can be found in (Schranz et al., 2021). In essence, to dockerize there is no need to change the wrapper code, all of the necessary actions are performed using a short script (run.sh for unix, run.ps1 for Windows), as seen in Listing ??. The script builds a docker image using the FMUs global unique identifier and runs it. Once the wrapper disconnects, the process terminates, the container is stopped and disposed. The actions within the Dockerfile depend on the choice of backend. An adaptation to the framework to support all backends is under development.

To dockerize the FMU, the commands given in the launch.toml can be set to run a shellscript:

```
[grpc]
linux = ["/bin/bash", "run.sh" ]
macos = ["/bin/bash", "run.sh" ]
windows = ["powershell", ".\\run.ps1"]
```

```
#!/bin/sh
...
# build image
docker build -t "$uid" .
# run container
docker run --net=host --rm ...
```

Listing 7: Shell script, run.sh, used to deploy docker container on unix.

8 CONCLUDING REMARKS AND FUTURE WORK

In this paper we have introduced the tool UniFMU that makes it possible to implement FMUs in several languages with built-in support by the tool. We have demonstrated the process of creating an FMU in a supported language, Python, and have demonstrated the process for extending the tool to support C#. This makes it possible to produce FMUs with limited knowledge of the internal workings and with limited knowledge of C. In the future we hope to be able to use a similar Java extension to enable the Overture FMU (Thule et al., 2018) export to be moved over to the Visual Studio Code VDM substantiation (Rask et al., 2020).

One issue not discussed in the paper is performance. Invoking functions of an FMU using RPC instead of calling them directly through the C-ABI incurs a performance cost. As part of (Hatledal et al., 2019) the authors provide the results of experiments where the total simulation time of multiple FMUs is measured for the two approaches. The results seem to indicate that there is an almost constant overhead per RPC call resulting in the largest impact on models that must be simulated with small step sizes. A possible way to reduce the performance overhead is to reduce the total number of RPC calls. For example several FMI calls made in between two `fmi2DoStep` calls could be grouped and sent as a single message, since the outputs would not change in between.

UniFMU provides a way to package and execute arbitrary code inside of an FMU. However, it does not directly provide a way to ensure consistency between the model description and the code. Other works like (Legaard et al., 2020; Hatledal et al., 2020) solve this issue by declaring the interface in the implementation of the FMU and using code generation targeted for specific languages to export the model description. This approach cannot readily be applied for a large number languages without implementing without implementing code generation for each language individually. There is a work in progress GUI, where it is possible to import the FMU, and using the editor manage the input and output variables of the model description, as shown in fig. 5.

In addition to the standalone GUI for the tool, it is possible to bundle the offering in the INTO-CPS Association services, for instance in the front-end used to setup and launch co-simulations using MAESTRO, the INTO-CPS Application (Macedo et al., 2020; Talasila et al., 2020). It is also a possibility to enable the Model-Based Design of Cyber-Physical Systems community to use and make the tool available in the HUBCAP project cloud platform (Larsen et al., 2020; Kulik et al., 2020).

ACKNOWLEDGEMENTS

We acknowledge the funding from the Poul Due Jensen Foundation for funding the project Digital Twins for Cyber-Physical Systems (DiT4CPS).

REFERENCES

(2021). CATIA-Systems/FMPy. CATIA Systems.

Asghar, S. A. and Tariq, S. (2010). Design and implementation of a user friendly OpenModelica graphical connection editor. page 81.

Bertsch, C., Ahle, E., and Schulmeister, U. (2014). The Functional Mockup Interface – seen from an industrial perspective. pages 27–33.

[7]https://www.docker.com/

Controllab Products B.V. (2013). http://www.20sim.com/. 20-sim official website.

Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. (2018). Co-simulation: a Survey. *ACM Comput. Surv.*, 51(3):49:1–49:33.

Hatledal, L., Zhang, H., and Collonval, F. (2020). Enabling python driven co-simulation models with pythonfmu. pages 235–239.

Hatledal, L. I., Styve, A., Hovland, G., and Zhang, H. (2019). A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface. *IEEE Access*, 7:109328–109339.

Hatledal, L. I., Styve, A., Hovland, G., and Zhang, H. (2019). A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface. *IEEE Access*, 7:109328–109339. Conference Name: IEEE Access.

Kulik, T., Macedo, H. D., Talasila, P., and Larsen, P. G. (2020). Modelling the HUBCAP Sandbox Architecture In VDM – a Study In Security. In Fitzgerald, J. S. and Oda, T., editors, *Proceedings of the 18th International Overture Workshop*, pages 20–34. Overture.

Larsen, P. G., Macedo, H. D., Fitzgerald, J., Pfeifer, H., Benedikt, M., Tonetta, S., Marguglio, A., Gusmeroli, S., and Jr., G. S. (2020). An Online MBSE Collaboration Platform. pages 263–270. INSTICC, Proceedings of the 10th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH.

Legaard, C. M., Gomes, C., Larsen, P. G., and Foldager, F. F. (2020). Rapid Prototyping of Self-Adaptive-Systems using Python Functional Mockup Units. SummerSim '20. ACM New York, NY, USA.

Macedo, H. D., Rasmussen, M. B., Thule, C., and Larsen, P. G. (2020). Migrating the INTO-CPS Application to the Cloud. In Sekerinski, E., Moreira, N., Oliveira, J. N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., and Delmas, D., editors, *Formal Methods. FM 2019 International Workshops*, pages 254–271, LNCS 12233. Springer-Verlag.

Modelica Association (2019). Functional Mock-up Interface for Model Exchange and Co-Simulation. https://www.fmi-standard.org/downloads.

Rask, J. K., Madsen, F. P., Battle, N., Macedo, H. D., and Larsen, P. G. (2020). Visual Studio Code VDM Support. In Fitzgerald, J. S. and Oda, T., editors, *Proceedings of the 18th International Overture Workshop*, pages 35–49. Overture.

Schranz, T., Alfalouji, Q., Falay, B., Legaard, C., Wilfling, S., and Schweiger, G. (2021). Coupling physical and machine learning models: Case study of a residential building. In *14th International Modelica Conference (Submitted Manuscript)*.

Schweiger, G., Gomes, C., Engel, G., Hafner, I., Schöggl, J.-P., Posch, A., and Nouidui, T. (2019). Functional Mock-up Interface: An empirical survey identifies research challenges and current barriers.

Simulink09 (2009). Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/.

Talasila, P., Sanjari, A., Villadsen, K., Thule, C., Larsen, P. G., and Macedo, H. D. (2020). Introducing Test Driven Development and Upgrades to the INTO-CPS Application. In Cleophas, L. and Massink, M., editors, *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops*, pages 311–317, Cham. Springer International Publishing.

Thule, C., Lausdahl, K., Gomes, C., Meisl, G., and Larsen, P. G. (2019a). Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45–61.

Thule, C., Lausdahl, K., and Larsen, P. G. (2018). Overture FMU: Export VDM-RT Models as Tool-Wrapper FMUs. In Pierce, K. and Verhoef, M., editors, *The 16th Overture Workshop*, pages 23–38, Oxford. Newcastle University, School of Computing. TR-1524.

Thule, C., Palmieri, M., Gomes, C., Lausdahl, K., Macedo, H. D., Battle, N., and Larsen, P. G. (2019b). Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks. In *Co-Sim-19 workshop*.

Widl, E., Müller, W., Elsheikh, A., Hörtenhuber, M., and Palensky, P. (2013). The FMI++ library: A high-level utility package for FMI for model exchange. In *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, pages 1–6.

Chapter 8

Rapid Prototyping of Self-Adaptive-Systems using Python Functional Mock-up Units

The paper presented in this chapter has been published in the peer-reviewed conference of *SummerSim: Summer Simulation Multi-Conference*

RAPID PROTOTYPING OF SELF-ADAPTIVE-SYSTEMS USING PYTHON FUNCTIONAL MOCKUP UNITS

Christian Møldrup Legaard
Cláudio Gomes
Peter Gorm Larsen

DIGIT, Department of Engineering, Aarhus University
Finlandsgade 22, 8200 Aarhus N, Denmark
{cml,claudio.gomes,pgl}@eng.au.dk

Frederik Forchhammer Foldager

Agrointelli
Agro Food Park 13, 8200 Aarhus N
Denmark
ffo@agrointelli.com

ABSTRACT

During the development of Cyber-Physical Systems (CPSs), it is crucial to enable efficient collaboration between different disciplines. Co-simulation plays a key role in this by allowing the system as a whole to be simulated by composing simulations of its parts. The ability to do this coupling relies on the models adhering to a well-defined interface. The Functional Mockup Interface (FMI) defines this interface and the models that implemented it are called Functional Mockup Units (FMUs). While a wealth of specialized simulation tools can generate FMUs, they are often commercial and do not support of complex software prototypes. Rather than implement these as FMUs from scratch (FMI requires expertise in C), losing valuable time, the contribution presented in this paper is a tool that allows FMUs to be implemented rapidly in Python. The advantages of this approach are demonstrated in an industrial use case, where a tracking simulator is implemented as an FMU.

Keywords: Python, prototyping, self-adaptive system, co-simulation, FMI

1 INTRODUCTION

The global competition for intelligent physical products is increasing and as a consequence, the time to market becomes of prime importance. Different disciplines have different traditions for how they can model and analyse the properties that are most important for them. Cyber-Physical Systems (CPSs) are heterogeneous and involve many different disciplines and thus facilitating optimal collaboration between such disciplines is essential (Lee 2008). Here co-simulations can be an effective technique to couple the different models together at the early stages of development, such that it is possible to analyse the overall CPS performance based on the complex interactions between its components (Gomes et al. 2018).

Since the different models typically make use of different branches of mathematics, the interoperability between them is typically ensured using different kinds of standards. Thus each individual model needs to be represented as a stand-alone simulation unit with a standardised interface. In this work we make use of the Functional Mockup Interface (FMI) version 2.0 for co-simulation (FMI v. 2.0 2014) where such units implementing the FMI standard are called Functional Mockup Units (FMUs). Numerous legacy modeling and simulation tools can export their models as FMUs. The contribution presented here complements such tools enabling faster production of FMUs using the Python programming language to support faster collaboration between different disciplines.

We argue that this prototyping technology is particularly valuable in the early stages of co-simulation (as in agile methodologies) but also for prototyping monitoring and self-adaptive systems (Weyns 2019). The main advantage of our tool is that it does not require compilation, and is platform-independent. Furthermore, Python has a vast ecosystem of numerical libraries, making it the ideal language for prototyping for the aforementioned systems.

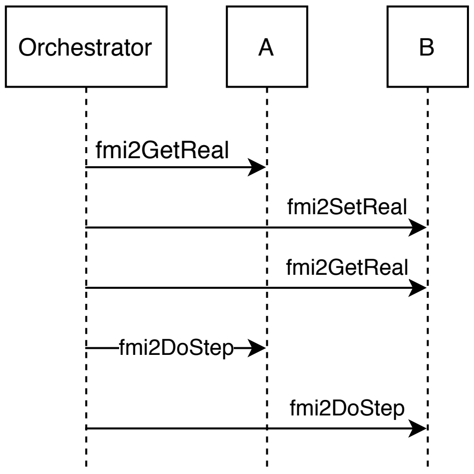
After this introduction, Section 2 provides the necessary background to be able to understand the main contribution of this paper which is presented in Section 3. Afterward, Section 4 presents a case study using the new contribution to produce an online tracking simulator of an unmanned agricultural robot. Finally, Sections 5 and 6 provide an overview of related work and a few concluding remarks about this work respectively.

2 BACKGROUND

2.1 Co-simulation

Co-simulation is a technique that enables the simulation of an entire system by simulating its individual parts (Kübler and Schiehlen 2000). Given the wide range of Modelling and Simulation (M&S) tools, standardized interfaces have been developed to enable these different tools to communicate their simulation results. One such interface is the Functional Mockup Interface (FMI) (Blochwitz et al. 2012).

The FMI defines a C API that binaries exported from each M&S have to implement. Each binary exported from a M&S tool corresponds to a subsystem that has inputs, internal behavior, and outputs. Other software can communicate with such a binary by setting inputs, asking it to compute the internal behavior, and querying its outputs, as illustrated in Figure 1a. To convey this information, FMI standardizes an XML representation of the inputs, outputs and internal structure of the binary. The binary and metadata are packaged in a zip file with the `fmu` extension. The FMI denotes these packages as Functional Mockup Units (FMUs). Figure 1b illustrates the typical structure of an FMU.



(a) Example master algorithm.

```
SineGenerator.fmu/
├── binaries
│   ├── linux64
│   │   └── pyfmu.so
│   └── win64
│       └── pyfmu.dll
├── modelDescription.xml
└── resources
    ├── sinegenerator.py
    └── slave_configuration.json
```

(b) File structure a FMU.

Figure 1: Use of FMI functions and FMU structure.

2.2 Self-Adaptive Systems

Self-adaptation is increasingly recognized as important not just for software systems (Weyns 2019) but also for cyber-physical systems (Zhou et al. 2019, Kritzinger et al. 2018). In its essence, a self-adaptive system can perform well in an environment that is difficult to model at the design time of the system. For example, the case study we introduce in Section 4 is a robot that functions in a wide range of soil types, often different than the ones available for testing at design time.

The implementation of a self-adaptive system comprises techniques such as calibration of models (to enable remote sensing of the system's state), and optimization (to determine the best course of action). We refer to the cited surveys for more details. We argue that prototyping such a system requires access not just to detailed models of the physical dynamics of the system, but also to a high-level language, that provides plenty of numerical libraries. FMI provides easy access to FMUs and therefore to the dynamics of the system, but, to the best of our knowledge, there are not many M&S tools that have a vast library for numerical computing.

Despite the simple interface that the FMI provides (a factor that has contributed to its wide adoption), if one needs to have a custom FMU, one needs to program it in the C language. We argue that C is not the right level of abstraction to prototype self-adaptation processes.

In the next section, we describe the implementation of a generation tool that produces FMUs that are specified in Python, while at the same time implementing the FMI standard. Such FMUs can be changed without requiring recompilation of the binaries, and allow the use of any Python libraries for the computation (provided these are either packaged with the FMU, or available in the execution platform).

3 CONTRIBUTIONS

The contribution of this paper is *PyFMU* (https://github.com/INTO-CPS-Association/pyfmu), a tool which enables rapid development of FMUs using Python. The process of exporting the FMU is fully automated by the tool and requires little knowledge of the FMI Standard and no knowledge of C. The goal of the tool is to enable rapid prototyping of FMUs for a wide range of applications. First, we demonstrate the process of creating an FMU from scratch, in Section 3.1. Following this, we describe the mechanism the tool uses to export and execute the Python code, in Section 3.2.

The FMUs and code used to produce the results can be found as an attachment to the repository's releases: https://github.com/INTO-CPS-Association/pyfmu/releases/tag/0.0.4

3.1 Creating an FMU

A simple sine wave generator is used as an example. The generator has a single output y, and three parameters: amplitude a, frequency ω and phase θ. The output is determined as follows:

$$y = a\sin(\omega t + \theta).$$

The process of creating an FMU using the tool is split into 3 steps, as seen in Figure 2. The first and last steps are carried out by a command-line interface (CLI) supplied with the tool. To create a new project, the sub-command *generate* is used:

```
pyfmu generate -p SineGenerator
```

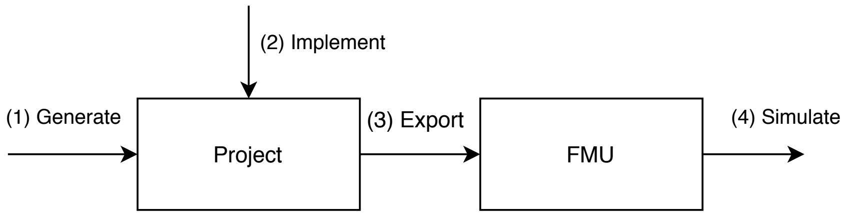


Figure 2: This figure illustrates the typical workflow of when using the tool. The first step is to generate a project. Next, the FMU is implemented in Python. Finally, the project is exported as an FMU, which can then be used as part of a co-simulation.

The command creates a new directory named *SineGenerator* referred to as a *project*. By default, the tool generates the templates of the files needed to start implementation. In this case, the project structure is:

```
SineGenerator/
├── project.json
└── resources
    └── sinegenerator.py
```

The Python script is referred to as the *slave script*. Inside the slave script, the class *SineGenerator* is defined, which is referred to as the *slave class*. When the FMU is instantiated it is this class that acts as its implementation. The purpose of the project.json file is to specify which class to instantiate if multiple scripts are present.

SineGenerator implementation

```python
class SineGenerator(Fmi2Slave):
        def __init__(self):

                author = "Christian Møldrup Legaard"
                modelName = "SineGenerator"
                description = "A single output sine-wave generator"

                super().__init__(
                modelName=modelName,
                author=author,
                description=description)

                self.register_variable('y','real','output')
                self.register_variable('w','real','parameter','fixed',start=1)
                self.register_variable('a','real','parameter','fixed',start=1)
                self.register_variable('p','real','parameter','fixed',start=0)

        def setup_experiment(self, start_time, end_time, tolerance):
                self._start_time = start_time

        def exit_initialization(self):
                self.y = self.a * sin(self.w * self._start_time + self.p)

        def do_step(self, current_time, step_size, no_step_prior):
                self.y = self.a * sin(self.w * current_time + self.p)
```

Figure 3: Typical FMU structure. The highlights show the lines that were added to the generated code.

The tool also generates the initial content of these files, which minimizes the amount of code the user has to write. For reference see Figure 3 which shows a full implementation of the SineGenerator. To implement the FMU, one needs to define its inputs, outputs, and parameters. In FMI standard these entities are generalized as *variables*. To declare a new variable the *register_variable* function is used. For example the output *y*, is defined as: `self.register_variable('y','real','output')`.

The SineGenerator is peculiar, in the sense that it does not take any inputs from other FMUs. Had this not been the case an input *x*, could easily be defined as `self.register('x','real','input',start=0)`. Note that start values must be declared for inputs according to the FMI specification. PyFMU performs validation on the variables to ensure that the model adheres to the FMI specification, even when executed as pure Python code.

Next, the behavior of the FMU must be implemented. This is done by defining special methods in the slave class, corresponding to the methods of the FMI interface. These are invoked whenever the matching function on the FMI interface is called. For example, to define the equivalent of the *fmi2DoStep* function of the FMI standard (recall Figure 1a), the `do_step` method is defined as in Figure 3. The correspondence between the FMI function and the Python counterparts is shown in Figure 4c.

The final stage is to export the project, producing in an FMU with a structure as shown in Figure 1b. Exporting the project as an FMU is done using the *export* command:

```
pyfmu export -p /SineGenerator -o somedir/SineGenerator
```

3.2 Implementation

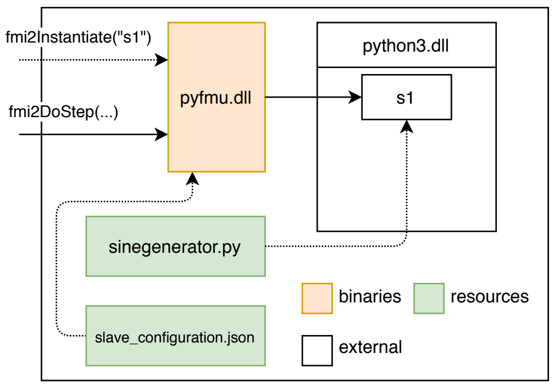
Recall the SineGenerator example, its structure is seen in Figure 1b. PyFMU uses an approach where generic binaries `pyfmu.so/dll` are pre-compiled and used by all exported FMU. It acts as a "wrapper" around the Python code implementing the FMU. Whenever a call is made to FMI interface the call is passed to a Python interpreter which is running inside the FMU, as illustrated in Figure 4a. The tool ships with pre-compiled binaries for several platforms, ensuring that an FMU produced on one platform, will also run on the others. The main advantage of this approach is that it eliminates the need for the user to ever deal with FMU compilation issues.

The Python interpreter is included in the binary using an approach referred to as *embedding*. That is, the Python interpreter is embedded into the C/C++ program, by including its headers and linking the library. This makes it possible to programmatically interact with Python, for example by importing modules, instantiating objects and calling functions. Figure 4b shows how the *_doStep* method is called on a Python object from C.

The same mechanism is used for creating instances of the slave class within the Python interpreter. To instantiate the objects within the interpreter the binary reads the name of the slave script and class from the *slave_configuration.json* file. The process of generating the *modelDescription.xml* file is fully automated by the tool. Rather than going through the trouble of reading and parsing the script as a text file, the tool simply creates an instance of the slave class. The tool then inspects the declared variables in an object-oriented fashion and creates the appropriate XML elements in the model description. This approach has the added benefit that the full flexibility of Python can be used to generate complex FMU interfaces with ease. For example, Figure 5 shows how the SineGenerator may be modified to add multiple outputs.

4 CASE STUDY: PROTOTYPING A TRACKING SIMULATOR

In the process of moving towards unmanned agricultural operations, novel monitoring systems are needed to account for unforeseen events occurring during field operations. For example, the soil surface characteristics may vary locally based on topography, soil composition, and moisture content. The variations can reduce the



(a) Illustration of the process used to instantiate Python slave inside the FMU and how subsequent calls are propagated to the slave. When fmi2Instantiate is called for the first time the binary starts a Python interpreter. Following this, the binary reads the name of the script and which class to instantiate from the slave_configuration.json file. This information is used to create an instance of the class inside the interpreter. After this calls to the FMI component are propagated to the newly instantiated Python object.

```
#include <Python.h>
...
PyObject* f = PyObject_CallMethod(
    pInstance_,
    "_do_step",
    "(ddO)",
    currentTime,
    stepSize,
    noSetPrior);
```

(b) Invoking the slave's doStep from the wrapper.

fmi2Functions.h	SineGenerator
fmi2DoStep	do_step
fmi2SetupExperiments	setup_experiments
fmi2ExitInitializationMode	exit_initialization

(c) Examples of FMI function mapping to the corresponding slave functions.

Figure 4: Mechanism used to wrap execute Python code within the FMUs.

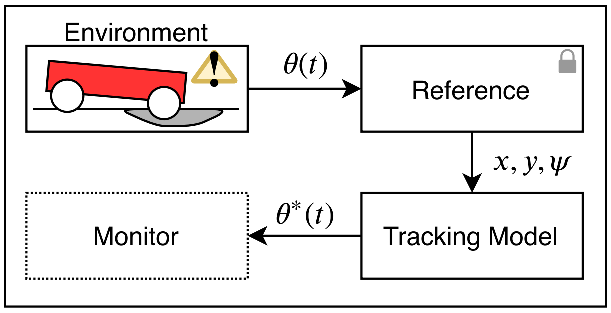
```
def __init__(self):
    ...
    n_outputs = 3
    for i in range(n_outputs):
        self.register_variable(f'y{i}','real','output')
```

Figure 5: Programmatic variable declaration.

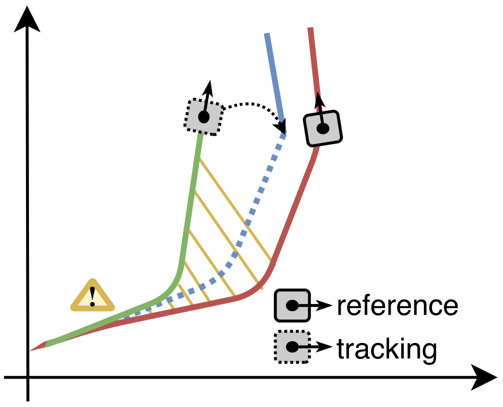
traction of the wheels and consequently affect motion and maneuverability. With better monitoring systems, the vehicle control system can adapt the steering despite these variations. In this section, we describe how PyFMU allowed us to leverage the full flexibility of Python to quickly prototype a tracking simulator of an agricultural autonomous vehicle, called Robotti. Initial modeling efforts of the system were described in (Foldager et al. 2018).

The goal of the tracking simulator is to detect changes in the reference system's dynamics caused by changes in the environment, during a single simulation, as shown in Figure 6a. Asides from detecting that a change has happened, the tracking simulator attempts to find a new set of parameters θ^* which would explain the observed change in behavior. This is made possible by the fact that the tracking simulator itself contains a simplified dynamical model of the Robotti, referred to as the *tracking model*. During co-simulation, both models are run in parallel producing two separate trajectories as shown in Figure 6b.

Undisturbed, the two trajectories will evolve in an almost similar fashion. However, the moment that the reference system parameters change the two trajectories will start to diverge and the tracking simulator will trigger a re-calibration. We first give a brief description of Robotti and how it is modeled in this case study. Then we describe the implementation of the tracking simulator using PyFMU.



(a) Co-simulation scenario of tracking simulator. Each box corresponds to an FMU and the arrows identify connections between them. The reference represents a black-box model of the physical system, which is influenced by a set of parameters controlled by the environment θ, which vary over time. The goal of the tracking model is to match the behavior of the reference as close as possible. It does so by finding the set of parameters θ^* which results in an trajectory as close as possible to the reference system.



(b) Illustration of the online calibration process. The reference and tracking models share the same trajectory until an event occurs which causes a change in the reference model. After some time the discrepancy between the two models has grown sufficiently large to trigger a recalibration. The recalibration process finds the parameters θ^* which results in the trajectory closest to the reference (dotted blue line), within a bounded interval of time (e.g., since the discrepancy occurred). Following this the tracking model resumes form the end of the optimal trajectory (solid blue line).

Figure 6: Tracking simulator overview.

4.1 Robotti

The agricultural robotic vehicle is a four-wheeled Ackermann steered autonomous system. The vehicle is designed as a generic platform applicable for various agricultural operations such as weeding, spraying, or cultivation. The navigation and steering control are performed on-board on a ROS-based system and is equipped with various sensors such as RTK-GPS, LiDAR, IMU and encoders. A photo of Robotti is shown in Figure 8.

For the tracking simulator, we use a bicycle dynamic model adapted from (Kong et al. 2015, Rajamani 2011), and given as follows. We assume the longitudinal velocity $\dot{x}$ is constant and model the lateral dynamics. The lateral and rotational acceleration were obtained by summing up the forces and moments at the center of the vehicle. The orientation and position were obtained by numerically integrating Equations (8) and (9) twice. F_{c_f} is the sum of the local tire forces on the front and rear wheels calculated by the product of the slip angle α and the tire stiffness coefficient. The slip angles are calculated as a function of the orientation of the vehicle ψ, the velocity components $\dot{x}$ and $\dot{y}$ and the steering angle on each of the front wheels $\delta_{l/r}$ where subscripts indicate left and right. l_f and l_r are the distances between the center of gravity of the vehicle and the front and rear wheels in the longitudinal direction as shown in Figure 7. The equations of the model are summarized in Equations (1) to (9), and parameters and their order of magnitude are shown in Table 1

Front tire slip angle	$\alpha_f = \delta_f - (\dot{y} + l_f \dot{\psi})/\dot{x}$	(1)
Rear tire slip angle	$\alpha_r = (\dot{y} - l_r \dot{\psi})/\dot{x}$	(2)
Lateral tire force at a front wheel	$F_{c_f} = C_{\alpha_f} \alpha_f$	(3)
Lateral tire force at a rear wheel	$F_{c_r} = C_{\alpha_r}(-\alpha_r)$	(4)
Longitudinal acceleration	$\ddot{x} = \dot{\psi}\dot{y} + a$	(5)
Lateral acceleration	$\ddot{y} = -\dot{\psi}\dot{x} + (1/m)(F_{c_f}\cos(\delta_f) + F_{c_r})$	(6)
Yaw acceleration	$\ddot{\psi} = (1/I_{zz})(l_f F_{c_f} - l_r F_{c_r})$	(7)
Velocity in the global frame	$\dot{X} = \dot{x}\cos(\psi) - \dot{y}\sin(\psi)$	(8)
Velocity in the global frame	$\dot{Y} = \dot{x}\sin(\psi) + \dot{y}\cos(\psi)$	(9)

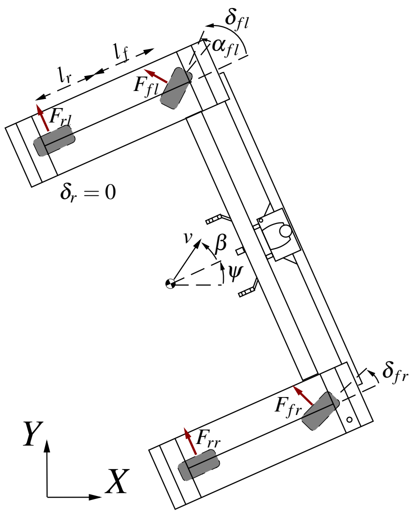


Figure 7: Sketch of the Robotti

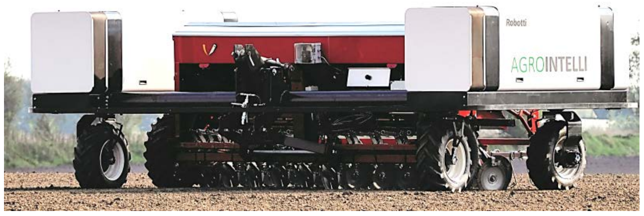


Figure 8: Robotti in the field. Photo: Agrointelli.

Parameter	Magnitude	Description
l_f	10^0	Distance COG to front wheel (m).
m	10^3	Mass of the vehicle.
I_{zz}	10^3	Rotational inertia.
$C_{\alpha_{f/r}}$	10^2	Tire cornering stiffness.

Table 1: Parameter and their magnitudes (specific values are omitted to protect company property)

For simplicity, the surface-tire interaction was modeled by a linear relationship between the tire cornering stiffness and the tire slip angle. In the actual model of the Robotti, this interaction is modeled by non-linear relationships that include the surface friction coefficient, the tire normal load, and the steering angle. Moreover, the dynamics of Robotti were derived for a four-wheel vehicle as schematically shown in Figure 7. The actual Robotti model was implemented using the 20-sim tool (Broenink 1997, Kleijn 2009) and exported as an FMU.

4.2 Results And Discussion

The purpose of the prototyped tracking simulator is to show when the model proposed in Equations (1) to (9) fails to accurately represent reality. This is achieved by constantly monitoring the behavior of the system.

When discrepancies exceed a tolerance value, a new calibration is started, that tries to find new parameters for the model that explain the measured data (recall Figure 6b).

Our tracking simulator was prototyped in Python and exposed as an FMU through PyFMU. To produce the simulation results, the tracking simulator uses the model in Equations (1) to (9) and the Python numerical library SciPy (Eric Jones and Travis Oliphant and Pearu Peterson and others 2001). The actual data comes as inputs to the PyFMU, and the recalibration process is implemented using the SciPy library.

To validate the tracking simulator, we used a co-simulation with the FMU produced from the actual model of the Robotti, created in 20-sim. To more easily trigger the recalibration process, we change the C_{α_f} parameter of the actual Robotti FMU during the co-simulation. These results are explained in Figure 10.

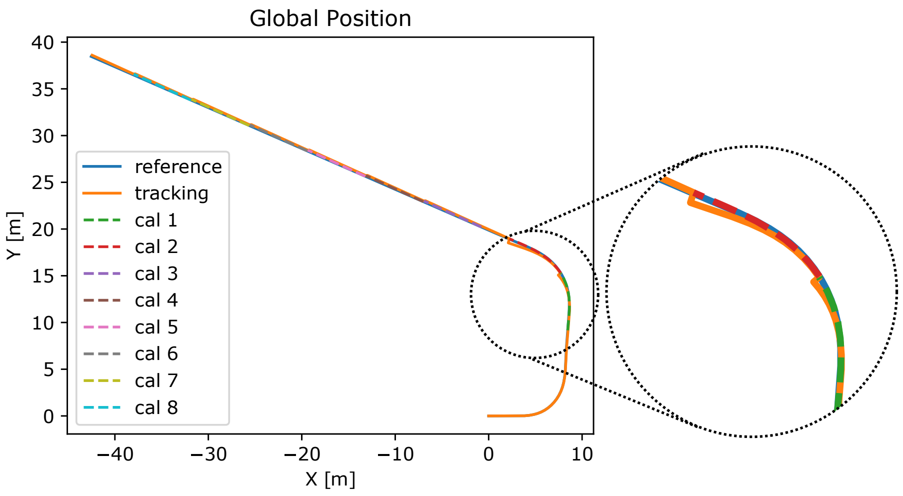


Figure 9: Global position of the robot. The orange line represents the simulation while the blue line represents the values of the actual Robotti FMU. Until the first bend, the simulation matches the actual Robotti FMU. On the second bend, however, there is divergence. This triggers the first re-calibration. The dashed line represents the interval of actual data used to find a new C_{α_f} parameter. Interleaved by a brief cool-down period, there are two subsequent re-calibrations, as the error is still above the tolerance.

Detecting the local surface characteristics during operation is a problem involving the surface (e.g., soil type and water content), the tires, and the vehicle dynamical system prescribing the traction and maneuverability. The soil-tire interaction is typically modeled using classical terramechanics models (Wong et al. 1984) which is a semi-empirical model that is often used as a basis for modeling off-road applications. In this work, we use a bike model to track the motion of a more complex vehicle model in an FMU.

The design of a tracking simulator requires advanced domain knowledge and a careful choice of the parameters used in the recalibration process. The recalibration process problem is no different than a model identification problem. What makes the design of a tracking simulator unique is that each recalibration process is a slightly different model identification problem. For instance, the first re-calibration may succeed, while the second one may not. Developing a tracking simulator that works every time is outside the scope of this paper and the subject of ongoing work.

These challenges are better understood by prototyping and experimentation activities, which are enabled by tools such as PyFMU. This has allowed us to grasp some of the trade-offs between the different parameters that go into the design of a tracking simulator. For instance, a larger re-calibration interval is not always better, as it may lead to diverging optimization.

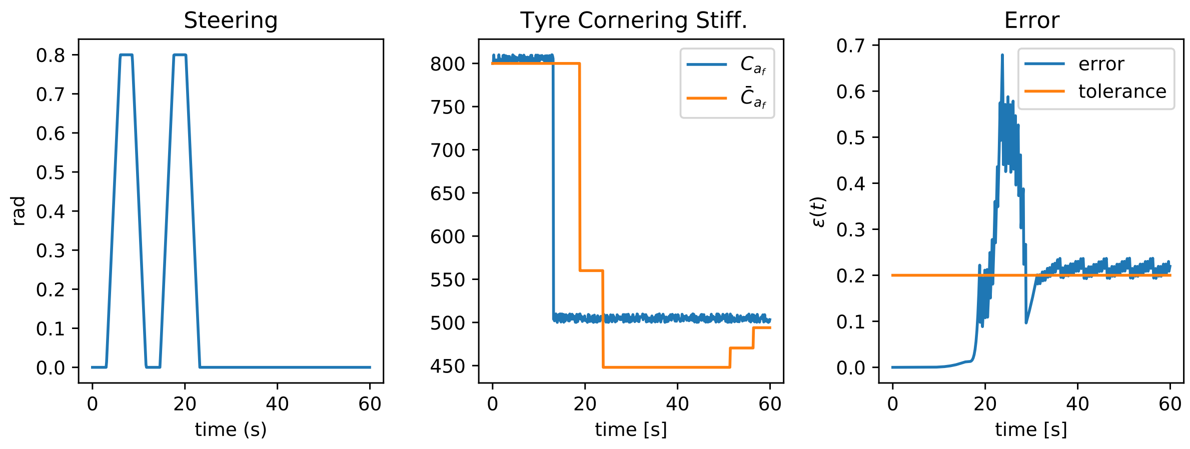


Figure 10: Steering angle, tyre stiffness and error of reference and adaptive model.

5 RELATED WORK

We restrict our scope to approaches to the implementation of FMUs for co-simulation. We therefore exclude other co-simulation interfaces (e.g., HLA (IEEE 2014)). Additionally, we consider only general approaches that are not restricted to modelling a specific domain such as fluid dynamics or power systems. To the authors knowledge no survey exists that provides a comprehensive comparison of tools capable of producing FMUs. Rather than attempt this, several works are presented highlighting the two main approaches which are used for exporting FMUs.

The first approach of generating FMUs is using a compiled language such as C/C++. Aslan et al. (2015) describes an object-oriented framework for the implementation of FMUs for co-simulation. New FMUs are developed by providing C++ code that extends a given base class. There is the possibility that a user specifies the model equations in C++, and the code provides some standard solvers to solve it. FMI (2015) proposes a similar object-oriented framework but supports only FMI1. The strength of this approach is the efficiency of the model and portability once compiled for a given platform.

The alternative to compiling FMUs is the approach used by PyFMU referred to as "tool-coupling" by Widl and Müller (2017). Thule et al. (2018) uses this approach for interfacing with models written in the language VDM-RT (Verhoef et al. 2006). Gomes et al. (2018) describes a method and a tool for the modification of existing FMUs, by wrapping them into new FMUs, according to the needs of specialized master algorithms. The new FMUs need to be recompiled, and may modify the inputs provided (e.g., implementing different input approximation functions), the outputs requested, or the way time stepping is performed.

More related to the tool is PythonFMU (https://github.com/NTNU-IHB/PythonFMU) which we originally planned to use. However, at the time we encountered issues with the generated model description files. This, among other limitations, spawned the work on PyFMU as a separate project. The authors of PythonFMU has since fixed several of these issues.

6 CONCLUDING REMARKS

Being able to rapidly prototype and simulate different designs in the early design stages of CPSs is extremely valuable, especially in the context of self-adaptive system development. The PyFMU tool makes it possible to implement FMUs with ease, using the popular Python programming language and its extensive set of numerical libraries.

The tracking simulator implemented in the case study demonstrates the advantages of this approach; as it would have been very time consuming to implement and refine the optimization algorithm using an existing approach. This is because the convergence of the recalibration process of a tracking simulator depends on when it started, which portion of the real trajectory is taken into account, and on several other parameters of the optimization engine. Having a prototype enables developers to more easily understand the different parameters involved and ultimately fine-tune the system. For instance, this prototype allowed us to conclude that a larger re-calibration interval is not always better, as it may fail to converge.

Another important perspective is that the content of the PyFMU is *pure* Python code. This means it can be developed, tested, and debugged, independently of the FMU where it resides. The ability to verify changes to the models quickly using a debugger was a big advantage to us.

Our tool opens the possibility of co-simulation to the many people who know Python but do not have sufficient knowledge of FMI or C to implement one from scratch. The tool currently supports the core functionality of FMI, however, some optional features such as providing the derivatives of the FMU and declaring physical units are subject of ongoing work. An continuously updated list of supported features and capabilities can be found on the front page of the code repository.

REFERENCES

2015. “FMI++”. https://sourceforge.net/projects/fmipp/.

Aslan, M., U. Durak, and K. Taylan. 2015, July. “MOKA: An Object-Oriented Framework for FMI Co-Simulation”. In *Conference on Summer Computer Simulation*, pp. 1–8. Chicago, Illinois, Society for Computer Simulation International San Diego, CA, USA.

Blochwitz, T., M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. 2012, November. “Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models”. In *9th International Modelica Conference*, pp. 173–184. Munich, Germany, Linköping University Electronic Press.

Broenink, J. F. 1997. “Modelling, Simulation and Analysis with 20-Sim”. *Journal A Special Issue CACSD* vol. 38 (3), pp. 22–25.

FMI v. 2.0 2014. “Functional Mock-up Interface for Model Exchange and Co-Simulation”.

Foldager, F., O. Balling, C. Gamble, P. G. Larsen, M. Boel, and O. Green. 2018, July. “Design Space Exploration in the Development of Agricultural Robots”. In *AgEng conference*. Wageningen, The Netherlands.

Gomes, C., B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, and P. De Meulenaere. 2018. “Semantic Adaptation for FMI Co-Simulation with Hierarchical Simulators”. *SIMULATION* vol. 95 (3), pp. 1–29.

Gomes, C., C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. 2018. “Co-Simulation: A Survey”. *ACM Computing Surveys* vol. 51 (3), pp. Article 49.

IEEE 2014. “IEEE 1516 High Level Architecture.”.

Eric Jones and Travis Oliphant and Pearu Peterson and others 2001. “SciPy: Open source scientific tools for Python”. [Online; accessed 31 March 2020].

Kleijn, C. 2009. *20-Sim 4.1 Reference Manual*. Getting Started with 20-sim.

Kong, J., M. Pfeiffer, G. Schildbach, and F. Borrelli. 2015, June. “Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design”. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1094–1099. Seoul, South Korea, IEEE.

Kritzinger, W., M. Karner, G. Traar, J. Henjes, and W. Sihn. 2018. “Digital Twin in Manufacturing: A Categorical Literature Review and Classification”. *IFAC-PapersOnLine* vol. 51 (11), pp. 1016–1022.

Kübler, R., and W. Schiehlen. 2000. “Two Methods of Simulator Coupling”. *Mathematical and Computer Modelling of Dynamical Systems* vol. 6 (2), pp. 93–113.

Lee, E. A. 2008. “Cyber Physical Systems: Design Challenges”. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369.

Rajamani, R. 2011. *Vehicle Dynamics and Control*. Springer Science & Business Media.

Thule, C., K. Lausdahl, and P. G. Larsen. 2018, July. “Overture FMU: Export VDM-RT Models as Tool-Wrapper FMUs”. In *The 16th Overture Workshop*, edited by K. Pierce and M. Verhoef, pp. 23–38. Oxford, Newcastle University, School of Computing. TR-1524.

Verhoef, M., P. G. Larsen, and J. Hooman. 2006. “Modeling and Validating Distributed Embedded Real-Time Systems with VDM++”. In *FM 2006: Formal Methods*, edited by J. Misra, T. Nipkow, and E. Sekerinski, Lecture Notes in Computer Science 4085, pp. 147–162, Springer-Verlag.

Weyns, D. 2019. “Software Engineering of Self-Adaptive Systems”. In *Handbook of Software Engineering*, edited by S. Cha, R. N. Taylor, and K. Kang, pp. 399–443. Cham, Springer International Publishing.

Widl, E., and W. Müller. 2017, July. “Generic FMI-Compliant Simulation Tool Coupling”. In *The 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, pp. 321–327.

Wong, J. Y., M. Garber, and J. Preston-Thomas. 1984. “Theoretical Prediction and Experimental Substantiation of the Ground Pressure Distribution and Tractive Performance of Tracked Vehicles”. *Proceedings of the Institution of Mechanical Engineers, Part D: Transport Engineering* vol. 198 (4), pp. 265–285.

Zhou, P., D. Zuo, K. Hou, Z. Zhang, J. Dong, J. Li, and H. Zhou. 2019, February. “A Comprehensive Technological Survey on the Dependable Self-Management CPS: From Self-Adaptive Architecture to Self-Management Strategies”. *Sensors* vol. 19 (5), pp. 1033.

ACKNOWLEDGEMENTS

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins. We also acknowledge EU for funding the INTO-CPS project (grant agreement number 644047) which was the original source of funding for the INTO-CPS co-simulation and the initial modeling of Robotti. Finally, we thank Daniella Tola for proofreading drafts of this manuscript.

AUTHOR BIOGRAPHIES

CHRISTIAN MØLDRUP LEGAARD is a PhD student at the Department of Engineering at Aarhus University. His research is centered on a combination of co-simulation and machine learning. His email address is cml@eng.au.dk.

CLÁUDIO GOMES is a Post-Doc at the Department of Engineering at Aarhus University. His research is centered on co-simulation and digital twins. His email address is claudio.gomes@eng.au.dk.

FREDERIK FORCHHAMMER FOLDAGER is an industrial PhD student at the Department of Engineering at Aarhus University and he is employed at Agrointelli. His research is centered on modeling of soil–machine interaction. His email address is ffo@agrointelli.com.

PETER GORM LARSEN is a Full Professor at the Department of Engineering at Aarhus University. His research is centered on digital twins, co-simulation and tool building. His email address is pgl@eng.au.dk.

Chapter 9

Neuromancer Framework

NeuroMANCER: Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations.

- Extended the codebase to handle arbitrary *Numpy* [54] operators, based on lazily evaluated computation graph.
- Improve packaging and distribution of the library

Chapter 10

Portable runtime environments for Python-based FMUs: Adding Docker support to UniFMU

The paper presented in this chapter has been published in the peer-reviewed conference *International Modelica Conference*.

Portable runtime environments for Python-based FMUs: Adding Docker support to UniFMU

Thomas Schranz[1] Christian Møldrup Legaard[2] Daniella Tola[2] Gerald Schweiger[1]

[1]Graz University of Technology, Austria, {thomas.schranz,gerald.schweiger}@tugraz.at
[2]DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Denmark, {cml,dt}@ece.au.dk

Abstract

Co-simulation is a means to combine and leverage the strengths of different modeling tools, environments and formalisms and has been applied successfully in various domains. The Functional Mock-Up Interface (FMI) is the most commonly used standard for co-simulation. In this paper we extend UniFMU, a tool that allows users to build Functional Mock-Up Units (FMUs) in virtually any programming language, to support execution within Docker. As a result the generated FMUs can be distributed in an environment containing all runtime dependencies. To describe the process of creating Dockerized FMUs using UniFMU, we show how to model and co-simulate a robotic arm and a controller using two Python-based FMUs.

Keywords: FMI, Co-Sim, Python, Tool-Coupling, Docker

1 Introduction

Complex, heterogeneous systems can be found throughout all fields of science and industry. Due to increasing complexity, market competition and specialization, system evaluation and simulation-based analysis has become more and more difficult (G. Schweiger et al. 2019). However, there often exist partial models for different parts of these systems, albeit in different domains and developed using different tools (Gomes et al. 2018). Co-simulation is a means to combine and leverage the strengths of different modeling tools, environments and formalisms (Cremona et al. 2019) and has been applied successfully in various domains (Gerald Schweiger et al. 2018; Pedersen et al. 2017; Nageler et al. 2018). The Functional Mock-Up Interface (FMI) was found to be the most promising standard for continuous time, discrete event, and hybrid co-simulation in a survey by (G. Schweiger et al. 2019). FMI is maintained by the Modelica association (Modelica Association 2021); it can be used to co-simulate components packaged as Functional Mock-Up Units (FMUs), each of which can be built using a different FMI-enabled modeling tool.

1.1 Co-Simulation Tools

With Open Modelica (Asghar and Tariq 2010), Simulink[1] or 20-sim[2] users can generate FMUs based on common modeling languages such as Modelica or MATLAB/Simulink using a graphical interface. The Universal Functional Mock-up Unit (UniFMU) (Legaard et al. 2021) tool allows users to build FMUs from arbitrary code in any programming language; it supports Python, C# and Java out-of-the-box. It uses a precompiled binary wrapper that implements the methods specified in the FMI standard's C-headers to spawn a process that executes the FMU's actual code. This way the FMU can be built from code written in an interpreted language or a language that uses automatic garbage collection. However, this setup, allowing for this kind of flexibility, requires the host machine to provide the process with all runtime dependencies which limits portability, especially between different host machines, and potentially necessitates a complicated setup procedure.

There exists a number of distributed, FMI-based co-simulation tools, many of which were analyzed in (Hatledal et al. 2019). However, all of them require a tight coupling between the co-simulation components and the master algorithm. ProxyFMU, a tool developed by the authors of (Hatledal et al. 2019) decouples the FMUs, in a way that they become independent of the master algorithm in a client/server solution that supports JavaScript, Python, C++ and the JVM on the client side.

The authors of (Hinze et al. 2018) propose a method for running FMUs inside Docker containers by placing the entire FMU archive inside the container and extending the master algorithm with a *remote procedure call* protocol. A distinction between their work and our approach is that FMUs generated using UniFMU work with any FMI-enabled master algorithm without the need to implement any additional protocols.

1.2 Contributions

In this paper we extend UniFMU using the virtualization environment Docker[3], such that the FMUs can be shipped with all runtime dependencies. We provide a general

[1]http://www.mathworks.com/products/simulink
[2]http://www.20sim.com
[3]https://www.docker.com

mechanism that can be leveraged for all languages supported by the tool. The resulting FMUs have nearly the same portability as compiled FMUs (except for the dependency on Docker) and require no language-specific setup procedure, but still allow the use of non-compiled languages and languages that use automatic garbage collection. To explain the process of creating Dockerized FMUs using UniFMU, we show how to model and co-simulate a robotic arm and a controller using two Python-based FMUs. The UniFMU tool (with the extensions for Dockerization) is available on Github[4]. The FMUs described in this paper can be found in a separate repository[5].

The rest of the paper is structured as follows. First, section 2 introduces a robotic arm and a controller which is used as a case study throughout the paper. Next, section 3 provides an introduction to UniFMU and describe how the robotic arm and controller can be implemented as FMUs using the tool. Then, section 4 describes the extension of UniFMU that allows FMUs and their dependencies to be deployed inside Docker containers. Afterwards, section 6 provides a discussion of the results and outlines future work on the tool. Finally, section 7 provides concluding remarks.

2 Case Study

To exemplify the process of using UniFMU we consider the case of modeling a robotic arm coupled to a controller as depicted in Figure 1. The example is chosen to highlight how different modeling formalisms can be realized by the tool. Specifically, the robotic arm described in subsection 2.1 is inherently continuous, whereas the controller described in subsection 2.2 is discrete.

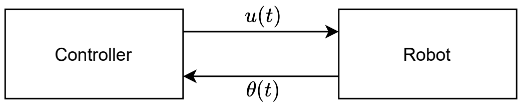


Figure 1. Connection between controller and robot model.

2.1 Robotic Arm

The robotic arm is modeled as a controlled inverted pendulum. The states of the system are its angle θ, the angular velocity ω and the current running through the coils of the electrical motor i. The dynamics of the robotic arm are described by Equation 1. Note that contrary to the visualization shown in Figure 7 the model only considers a single joint that rotates around a single axis.

$$f(x) = \begin{bmatrix} \dot{\theta} \\ \dot{\omega} \\ \dot{i} \end{bmatrix} = \begin{bmatrix} \omega \\ \frac{K \cdot i - b \cdot \omega - m \cdot g \cdot l \cdot cos(\theta)}{J} \\ \frac{u \cdot V_{abs} - R \cdot i - K \cdot \omega}{L} \end{bmatrix} \quad (1)$$

[4]https://github.com/INTO-CPS-Association/unifmu

[5]https://github.com/Daniella1/robot_unifmu

where:

The derivative of the angle $\dot{\theta}$ is, per definition, equal to the velocity of the arm ω. The derivative of the angular velocity $\dot{\omega}$ is determined by the torque coefficient $K = 7.45\ s^{-2}A^{-1}$, the current i, the motor-shaft friction $b = 5.0\ kg \cdot m^2 \cdot s^{-1}$ and the gravity acting on the arm, denoted by $m \cdot g \cdot l \cdot \cos(\theta)$, with $m = 5.0\ kg$, $g = 9.81 ms^{-2}, l = 1.0\ m$. The change in current is determined by the input from the controller u, the voltage across the coils $V_{abs} = 12.0\ V$, the resistance $R = 0.15\ \Omega$ and the motor's inductance $L = 0.036\ H$.

2.2 Controller

A proportional-integral-derivative (PID) controller (Åström and Murray 2010) is used to generate the control signals sent to the robotic arm. The continuous formalization of the controller is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t) \quad (2)$$

where $e(t)$ is a measure of the error of the variable being controlled and K_p, K_i and K_d are coefficients used to tune how the proportional and derivative terms are weighted. In case of the robot, the controller is trying to minimize the error between the desired angle $\theta^*(t)$ and the true angle $\theta(t)$. Thus, the error is defined as $e(t) = \theta^*(t) - \theta(t)$.

In practice, most controllers are implemented digitally, which means that derivatives and integrals must be replaced by discrete approximations. There are several ways to do this, the simplest being to replace derivatives by first-order differences

$$\dot{e}(t_k) \approx \dot{e}_k = \frac{e_k - e_{k-1}}{T},$$

and integrals by sums

$$\int_0^{t_k} e(t_k) \approx E_k = \sum_{n=1}^{N} e_{k_n} \cdot T$$

where $e_k = e(t_k)$, T is the sampling time and $N = t_k / T$ is the number of samples between time 0 and t_k. After replacing the continuous definitions in Equation 2 we obtain an equation that can be implemented on a discrete controller

$$u_k = K_p e_k + K_i E_k + K_d \dot{e}_k \quad (3)$$

This discretization scheme is simple to implement

3 Modeling

In this section, we describe how UniFMU is installed and how it is used to generate an FMU. We provide a brief overview of the resulting FMU's structure and method of operation. Subsequently, we describe the FMUs used to model the robotic arm and the controller. For illustrative purposes both FMUs are implemented in Python, however in the general case they can be implemented in a mix of languages.

3.1 Creating an FMU using UniFMU

UniFMU is a command line interface (CLI) that can be installed through Python's package installer `pip` or from source following the instructions in the official repository. Installation through `pip` uses a single command:

```
pip install unifmu
```

It should be noted that the FMUs generated with UniFMU do not require Python during runtime, unless the FMUs themselves are implemented in Python. To generate an FMU the tool has to be invoked with the subcommand `generate`, and supplied with the language the FMU is implemented in and the name it should have; for a Python-based FMU with the name `robot` this looks like:

```
unifmu generate python robot
```

This generates an FMU with the structure shown in Figure 2. The `binaries` directory contains a precompiled wrapper for Windows, Linux and MacOS that implements the methods specified in the FMI's C-headers and relays them to the actual implementation of the FMU found in the `resources` directory. `model.py` defines a class that declares a set of methods that correspond to the methods in the FMI standard, such as FMI's `fmi2DoStep` which is implemented by the Python method `do_step`. The actual overwrite used to model the robotic arm can be seen in Listing 1.

```
robot.fmu
├── binaries
│   ├── darwin64
│   ├── linux64
│   │   └── unifmu.so
│   └── win64
├── resources
│   └── model.py
└── modelDescription.xml
```

Figure 2. The directory structure for a Python FMU. Note that several files generated by the tool are omitted for simplicity.

3.2 Robotic Arm FMU

The robotic arm FMU is implemented in Python using a numerical solver provided by the *SciPy* (Virtanen et al. 2020) package. The general procedure for solving an ODE using Scipy is to define a function which evaluates the derivative for a given combination of state and time. Using Equation 1 as a reference the function $f(\cdot)$ can be defined as shown in Listing 1.

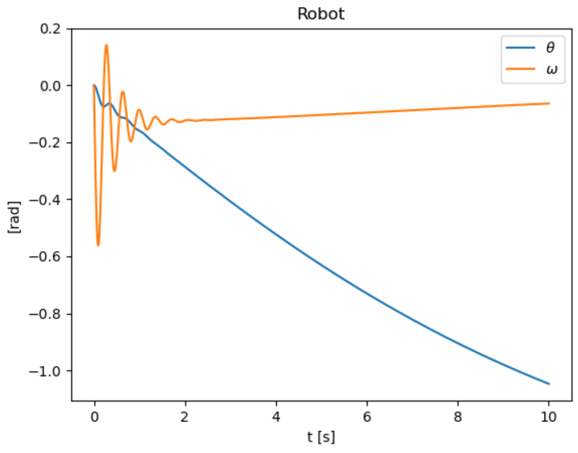


Figure 3. Standalone test of robotic arm, with values $\theta_0 = \omega_0 = i_0 = u(t) = 0$.

```
def do_step(self,current_time,step_size,no_step_prior):
    def f(t, y):
        theta, omega, i = y
        tau=self.k1*np.cos(theta)
        domega=(self.K*i-self.b*omega-tau)/self.J
        di=(self.V-self.R*i-self.K*omega)/self.L
        dtheta=omega
        return dtheta, domega, di
    res=solve_ivp(f,(current_time,current_time+step_size),y0)
    self.theta,self.omega,self.i=res.y[:,-1]
    return Fmi2Status.ok
```

Listing 1. Implementation of the `fmi2DoStep` method for the robotic arm FMU.

Given the definition of the derivative, the `solve_ivp` function can be used to obtain the solution for the next step of the FMU and allows users to choose between solvers. However, it is also possible to use any other Python library providing numerical solvers or to implement a custom solver. This is a very flexible solution as it allows users to choose the type of solver that is suitable for the particular ODE. After solving the ODE, the newly estimated state is assigned to the instance, where it can be accessed from other methods and the FMI.

To test the dynamics of the robotic arm FMU, a small test program is written in Python which invokes the *do_step* several times. The results are shown in Figure 3 for initial state and input $\theta_0 = \omega_0 = i_0 = u(t) = 0$. We see that the angle of the robot decreases from 0 to -1 over 10 seconds.

3.3 Controller FMU

The controller-FMU implements a simple control algorithm that determines the signal sent to the motor based on the difference between the desired and the actual current angle. Similarly to how Equation 1 was translated into a

Python function describing the derivative of the state, we use the control policy Equation 3 as a reference to implement the expression shown in Listing 2.

```
def do_step(self, current_time, step_size,
    no_step_prior):
    err=self.setpoint_t-self.measured_t
    self.I=self.I+err*step_size
    D=(err-self.p_err)/step_size
    self.u=self.Kp*err+self.Ki*self.I+self.Kd
    *D
    self.p_err = err
    return Fmi2Status.ok
```

Listing 2. Implementation of the `fmi2DoStep` method for the controller-FMU.

In most practical situations, controllers are implemented on a processing unit where updates to the output would happen at a fixed update rate determined by the controller's clock frequency and the number of operations needed at each update.

An implicit assumption of our model is that the step-size used by the solver matches the update rate of the controller. For small step sizes, the discrete approximation implemented by the model remains relatively accurate. However, for larger step sizes the accuracy of the discretization scheme is reduced, which may ultimately cause the closed-loop system to become unstable. A solution to mitigate the discretization error is to use a more sophisticated discretization scheme, such as Tustin's method (Franklin, Powell, and Emami-Naeini 2020).

As in the robotic arm FMU, we can also use any external library for modeling the controller. For instance, a package such as *python-control*[6], can be used to evaluate the performance of different controllers.

The functionality of the controller can be verified by writing a small test program in Python that invokes the *do_step* method of the FMU. To examine the closed-loop behavior, the robot is replaced with a simple linear model described by the ODE $\dot{\theta} = u$. Executing the Python test program, we obtain the step-response of the closed-loop system (with surrogate model) as depicted in Figure 4.

4 Docker Support

A key contribution of this paper is extending UniFMU so that the generated FMUs can be executed within a virtualization environment using Docker. To create a Dockerized FMU the user can append the *–dockerize* switch to UniFMU's `generate` subcommand:

```
unifmu generate python --dockerize robot
```

The functionality is available for Linux and macOS and all languages that the tool supports. Windows support is under development, but is held back by limitations of Docker's networking capabilities when running on Windows.

[6]https://python-control.readthedocs.io/en/0.8.3/index.html

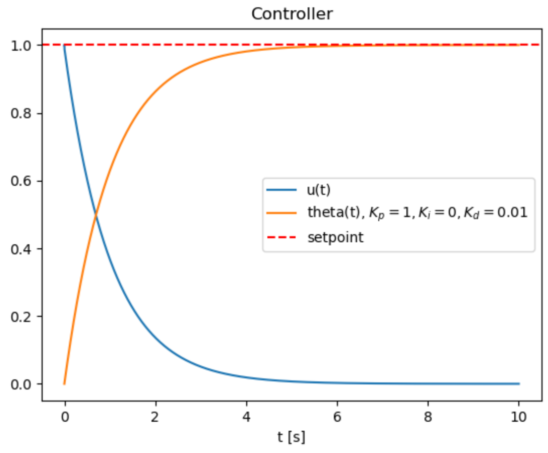


Figure 4. Standalone test of the controller FMU with using linear model for plant $\dot{\theta} = u$, step size = 0.001, setpoint = 1.0

4.1 Setting up the image

A configuration file, referred to as the `Dockerfile`, provides instructions to build the environment on any host machine. An excerpt from the `Dockerfile` used by the robotic arm FMU can be seen in Listing 3. The first line declares that the image for the FMU is assembled ontop a pre-built Python 3.8. image from the Docker container library. The second line invokes the package manager `pip` to install packages required by the model. For simplicity, the three dots represent the dependencies required by the Python backend to communicate with the binary. The third line instructs Docker to copy the `container_bundle` directory into the image. The `container_bundle` contains all files that are needed during runtime, such as the actual model implementation and all user-generated files and dependencies.

```
FROM python:3.8
RUN pip install ... scipy
    roboticstoolbox-python matplotlib
COPY container_bundle resources
...
```

Listing 3. Dockerfile used to assemble the image used by FMU instances.

4.2 Instantiating a Dockerized FMU

The process of creating an instance of a Dockerized FMU is depicted in Figure 5. The steps are as follows: First, the binary will ensure that the image declared in the Dockerfile has been built. If this is not the case, it will automatically invoke the `Dockerfile` to build the image. Next, from the image a container is created. The container has access to all dependencies listed in the `Dockerfile`, such as Python packages that were installed through `pip` and everything inside the `container_bundle`. Note that each instance of an FMU is executed within its own container and removed after use. This ensures that no instances of an FMU share any state or influence each other directly.

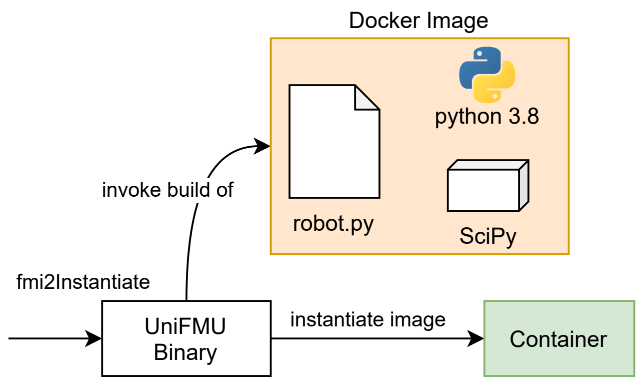


Figure 5. Deployment of a model inside the Docker container.

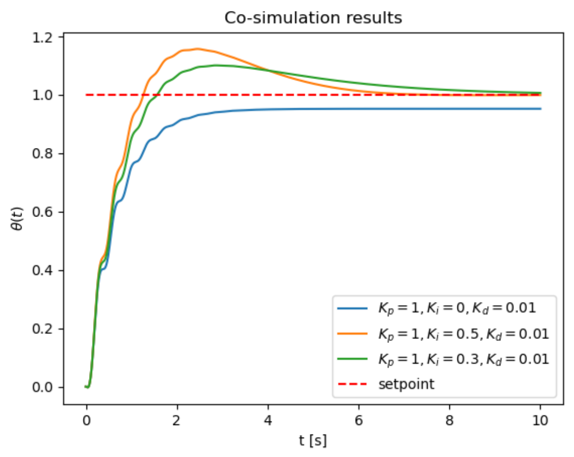


Figure 6. Co-simulation of the controller and the robot with the $setpoint = 1$. Experiments of varying the controller parameters are shown.

5 Results

A co-simulation was configured and run using the two FMUs with the INTO-CPS tool-chain (Larsen et al. 2016). We used a fixed step-size solver with a step-size of 0.001 seconds, set the desired angle to 1 radian and plotted θ as a function of time for various values of K_p, K_i and K_d. The corresponding plot can be seen in Figure 6. Fig. 7 shows the robotic arm at an angle of 1 radian. The controller with $K_i = 0$ exhibits a substantial steady-state error, whereas the ones with an integral term converge within 10 seconds. Besides, it can be seen that controllers with an integral term cause the system to overshoot the setpoint. Tweaking the coefficients of the controller allows us to balance the tendency to overshoot and the steady-state error, such that they meet the requirements of the application. Methods based on heuristics exist for tuning PID controllers, which could be applied to tune the controller for the robotic arm. However, we considered applying these to be beyond the scope of this example use case.

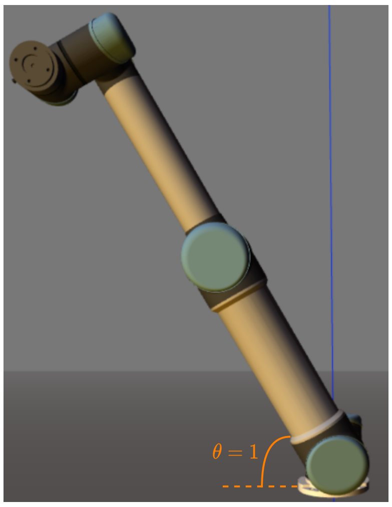


Figure 7. A visualization of the robot during the co-simulation. The measured θ is equal to the $setpoint = 1\ rad$, when controlled with the PID-controller.

6 Discussion

A central objective of the FMI standard is to facilitate the exchange of models generated by different tools. To do so, FMI requires communication through a C-API, which complicates implementing models in languages that cannot be compiled into a C-compatible binary. UniFMU circumvents this issue by providing a generic C-binary that handles all communication between FMI calls to the FMU and the FMU's actual code. Being able to use high-level programming languages such as Python allows developers to leverage a large ecosystem of scientific libraries and thus implement models quickly and efficiently, especially in contrast to writing everything from scratch. Consider the implementation of the `do_step` method for the robotic arm shown in Listing 1. The ODE is declared and solved in eight lines of code. We believe that this has the potential to simplify co-simulation for more modeling applications and engage more developers.

Another aspect to this approach is that the resulting FMUs can be verified and debugged using the development tools of the FMU's language. For instance, it allowed us to write small test programs for verifying the FMUs before performing the co-simulation of the system. In our experience, the ability to effectively test the individual models greatly reduces the number of issues encountered when integrating the models.

Using FMUs that require runtime dependencies to be handled manually is counter-intuitive to the idea of simple, standardized model exchange. Consequently, in this work we addressed this issue by providing a way to automatically virtualize the runtime environment with all dependencies inside a Docker container rather than requiring the host machine to provide a suitable environment. The way this Dockerization was implemented did not affect UniFMU's precompiled binaries and all changes to the language-specific backends are simply additional configuration options instead of hard dependencies on Docker itself. The latter might be reused in the future to implement remote deployment.

7 Conclusion

Co-simulation is a key research interest. The FMI standard is among the most popular interfaces for model exchange and co-simulation. There are various tools to generate FMI-compliant FMUs. UniFMU is one such tool that allows users to build FMUs from arbitrary code written in any language. We used UniFMU to generate two Python-based FMUs in order to co-simulate a robotic arm and a controller. However, the resulting FMUs required the host machine to provide a Python runtime environment with all dependencies preinstalled, effectively limiting portability and ease of deployment. To address this issue we extended UniFMU using the virtualization tool Docker. With our extension, UniFMU is able to generate FMUs shipped with a `Dockerfile` that automatically builds a runtime environment inside a container. This way the FMUs are almost as portable as compiled FMUs (except for the dependency on Docker) but still support the use of non-compiled languages. Our extension is not limited to Python but can be reused for other languages as well. Besides, the changes we implemented can help in developing a configuration for remote deployment of FMUs in the future.

Acknowledgements

The authors would like to thank Thomas Schwengler for their support with designing the Docker-integration.

The reported research was conducted within the project NextHyb2 (881150) and project DigitalEnergyTwin (873599), which received funding in the framework of "Stadt der Zukunft" and "Energieforschung", a research and technology program of the Austrian Ministry for Transport, Innovation and Technology (BMVIT).

References

Asghar, Syed Adeel and Sonia Tariq (2010). *Design and Implementation of a User Friendly OpenModelica Graphical Connection Editor*. eng.

Åström, Karl Johan and Richard M Murray (2010). *Feedback Systems: An Introduction for Scientists and Engineers*. In English. ISBN: 978-1-4008-2873-9.

Cremona, Fabio et al. (2019-06). "Hybrid Co-Simulation: It's about Time". en. In: *Software & Systems Modeling* 18.3, pp. 1655–1679. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-017-0633-6.

Franklin, Gene F., J. David Powell, and Abbas Emami-Naeini (2020). *Feedback Control of Dynamic Systems*. eng. Eighth edition, global edition. Harlow, United Kingdom: Pearson Education Limited. ISBN: 978-1-292-27452-2.

Gomes, Cláudio et al. (2018-07). "Co-Simulation: A Survey". en. In: *ACM Computing Surveys* 51.3, pp. 1–33. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3179993.

Hatledal, Lars Ivar et al. (2019-02). "FMU-proxy: A Framework for Distributed Access to Functional Mock-up Units". In: pp. 79–86. DOI: 10.3384/ecp1915779. URL: https://ep.liu.se/en/conference-article.aspx?series=ecp&issue=157&Article_No=8 (visited on 2021-05-09).

Hinze, Christoph et al. (2018). "Towards Real-Time Capable Simulations with a Containerized Simulation Environment". In: *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pp. 1–6. DOI: 10.1109/M2VIP.2018.8600827.

Larsen, Peter Gorm et al. (2016). "Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project". In: *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pp. 1–6. DOI: 10.1109/CPSData.2016.7496424.

Legaard, Christian Møldrup et al. (2021). "A Universal Mechanism for Implementing Functional Mock-up Units". In: *11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. SIMULTECH 2021. Virtual Event, to appear.

Modelica Association (2021). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. https://www.fmi-standard.org/downloads.

Nageler, P. et al. (2018-08). "Novel method to simulate large-scale thermal city models". en. In: *Energy* 157, pp. 633–646. ISSN: 03605442. DOI: 10.1016/j.energy.2018.05.190. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360544218310363 (visited on 2021-05-06).

Pedersen, Nicolai et al. (2017). "Distributed Co-Simulation of Embedded Control Software with Exhaust Gas Recirculation Water Handling System using INTO-CPS:" in: *Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. Madrid, Spain: SCITEPRESS - Science and Technology Publications, pp. 73–82. ISBN: 9789897582653. DOI: 10.5220/0006412700730082. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006412700730082 (visited on 2021-05-06).

Schweiger, G. et al. (2019-09). "An empirical survey on co-simulation: Promising standards, challenges and research needs". en. In: *Simulation Modelling Practice and Theory* 95, pp. 148–163. ISSN: 1569190X. DOI: 10.1016/j.simpat.2019.05.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S1569190X1930053X (visited on 2021-05-06).

Schweiger, Gerald et al. (2018-12). "District energy systems: Modelling paradigms and general-purpose tools". en. In: *Energy* 164, pp. 1326–1340. ISSN: 03605442. DOI: 10.1016/j.energy.2018.08.193. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360544218317274 (visited on 2021-05-06).

Virtanen, Pauli et al. (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17, pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

Chapter 11

Coupling physical and machine learning models: case study of a single-family house

The paper presented in this chapter has been published in the peer-reviewed conference *International Modelica Conference.*

Coupling physical and machine learning models: case study of a single-family house

Basak Falay[1] Sandra Wilfling[2] Qamar Alfalouji[2] Johannes Exenberger[2] Thomas Schranz[2]
Christian Møldrup Legaard[3] Ingo Leusbrock[1] Gerald Schweiger[2]

[1]AEE-Institue for Sustainable Technologies, Austria `b.falay@aee.at`
[2]Institute of Software Technology, Technical University of Graz, Austria `gerald.schweiger@tugraz.at`
[3]DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Denmark, `cml@ece.au.dk`

Abstract

The emergence of Cyber-Physical Systems poses new challenges for traditional modelling and simulation techniques. We need to combine white, grey, and black box models as well as different tools developed for specific subsystems and domains. Co-simulation is a promising approach to modeling and simulating such systems. This paper presents a case study where a physical model of a building's heating system implemented in Modelica is co-simulated with a machine learning model of a stratified hot water tank implemented in Python. The Python model is exported as Functional Mock-up Unit using UniFMU.

Keywords: Co-Simulation, Functional Mock-Up Interface, Modelling, Machine Learning

1 Introduction

Future intelligent and integrated energy systems must have a high degree of flexibility and efficiency to ensure reliable and sustainable operation (Lund et al. 2017). Along with the rapid expansion of renewable energy, this degree of flexibility and efficiency can be achieved by overcoming the clear separation between different sectors and by increasing connectivity and the associated data availability through the integration of sensors and edge/fog computing (Vatanparvar and Faruque 2018). All of these developments drive the transition towards so-called Cyber-Physical Energy Systems (Palensky, Widl, and Elsheikh 2013). Cyber technologies (sensors, edge/fog computing, IoT networks, etc.) can monitor the physical systems, enable communication between different subsystems, and control them. Thus, the emergence of Cyber-Physical Systems poses new challenges for traditional modelling and simulation approaches.

One of these challenges is that models need to combine computational systems and data communication networks with physical systems. Furthermore, recent studies showed that pure white-box models based on first principles deal with drawbacks such as time-consuming development, validation problems or low computational speed (Li and Wen 2014). Consequently, these approaches have limited use for complex systems such as intelligent buildings outside of academia (Schweiger, Nilsson, et al. 2020). Black-box approaches examine the system from the outside using input/output relations. Depending on the approach, they are computationally efficient but compared to white-box approaches they lack in generalizability and extensibility (Thieblemont et al. 2017). Beside white-box and black-box models, grey-box models fall in between (Harish and Kumar 2016). Several papers highlighted the importance of combining white-, grey-, and black-box models for analyzing and optimizing Cyber-Physical Systems (O'Dwyer et al. 2019; Killian and Kozek 2016; Thilker, Madsen, and Jørgensen 2021).

There are two options to simulate the interactions between subsystems; (i) the entire system can be modelled and simulated with a single tool referred to as monolithic, (ii) already established models for the respective subsystems are coupled in co-simulation (Gomes et al. 2018). A recent survey discussed the advantages, disadvantages, and challenges of co-simulation approaches (Schweiger, Engel, et al. 2018). This survey showed that experts consider the Functional Mock-Up Interface (FMI) standard to be the most promising standard for continuous-time, discrete-event, and hybrid co-simulation.

In this paper, the physical parameters of a subsystem (stratified storage tank) are not available. In this situation, model calibration and parameter estimation approaches can be used depending on availability of the measurement data. On the other hand, machine learning models can be as well used to mimic the behavior of the system by construct relationships between input and outputs without being dependent on the components parameters. Artificial Neural Networks was used to model the stratified storage tank in (Géczy-Vig and Farkas 2010). In this work, Random Forest (RF) was used to model the temperatures in each layer of the stratified storage tank. Since the states of the other components influence the state of the stratified storage, we have created a co-simulation workflow where the machine learning and physical models can be coupled. Physical and machine learning models are available at `https://github.com/tug-cps/NextHyb2` . Unfortunately, we cannot publish the data due to data privacy policy. Therefore, we have additionally generated a synthetic, open-source data set.

2 Method

2.1 Heating System of Single-family House

A single-family house with $180m^2$ floor area, located in Austria with an annual energy consumption of 7500 kWh was analyzed in this work. Figure 1 gives an overview of the main components of the single-family house heating system. The house, equipped with a floor heating system, has three different heat sources: (i) a solar collector with $46m^2$ flat plate area, (ii) a stove which directly heats the house, and the excess heat feeds the storage tank and (iii) an air-to-water heat pump. Additionally, an estimated $3m^3$ storage tank bridges these three heat sources in order to increase the efficiency of the heating system. The house has an indoor pool ($24m^3$), which is heated by the hot water storage tank or directly by the solar collector.

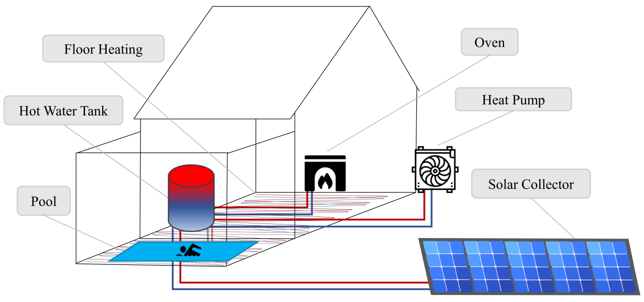


Figure 1. Overview of the single-family house heating system. Red line represents the supply and blue line represents the return temperatures.

The following rule-based control strategy of the heating system is given below.

- The priority of the solar collector is to maintain the temperature of top layer of the storage tank at 52°. If this condition is satisfied, then the excess heat from the solar collectors heats the indoor pool to 35°.
- If the solar collector cannot meet the heating demand of the indoor pool and if the top layer temperature of storage tank is higher than 52°, the storage tank heats the pool.
- If these conditions don't satisfy or the temperature of the bottom layer of the storage tank drops below 35°, the heat pump turns on.
- If the temperature of the stove is higher than 40°, the excess heat is fed into the storage tank.

2.2 Measurement Data

In Appendix, Figure A.1 shows an overview of the heating system components and the locations of the heat meters. Temperatures are represented in ($T_{component}$, mass flow rate in $dm_{component}$. Four temperature sensors from top to bottom respectively $T_{Storage,1}, T_{Storage,2}, T_{Storage,3}, T_{Storage,4}$ are located at the storage tank. The measured data from the heat meters is between 01.02.2019 and 31.01.2020, with a temporal resolution of 1 minute. Figure 2 gives an overview of the data quality of the measurement data. White lines represent the missing data points and corresponding periods. 4% of the measurement data is missing; 65% of them falls into the period between November 2019 and January 2020, 25% of them falls into September 2019. In addition to missing values, there are wrong measurements between November 2019 and January 2020 due to the failures in the meters.

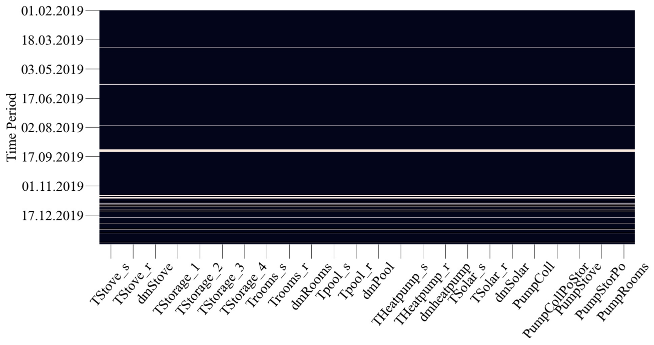


Figure 2. Missing data periods for the given measurement data

The missing parts of the data were imputed by taking the profile of the previous day. Figure 3 demonstrates the imputation of missing data points for four days in a row, given in dashed lines. The measurement data was ignored after November 2019 due to the bad quality of data. The whole data set was resampled to 15-minute values to avoid the over fitting the predictions of the ML model. After post-processing, the dataset had 27840 datapoints. The resampled data was later used for training and testing for the ML model.

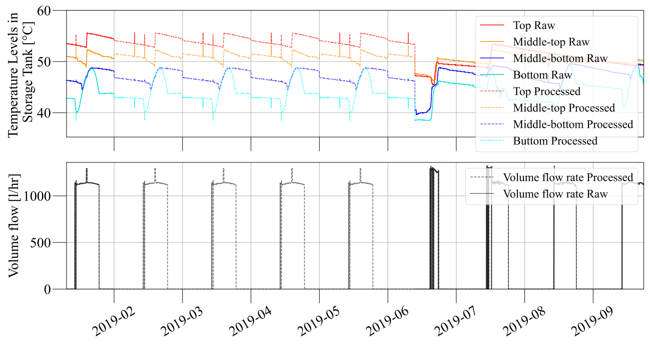


Figure 3. Imputation of the missing data

One of the most critical features in the dataset is mass flow rates from each component. Figure 4 shows the sparsity of the mass flow rates from each components. The y-axis represent the total data points (27840) after preprocessing, the x-axis represents the mass flow values of the components. The black points in the figure show the values that are not zero and the gaps between the black points represent the zero values. Mass flow rate in the stove has the highest percentage 99.8% of zero values and the mass flow rate in the *CollectortoPool* has the lowest percentage with 86%.

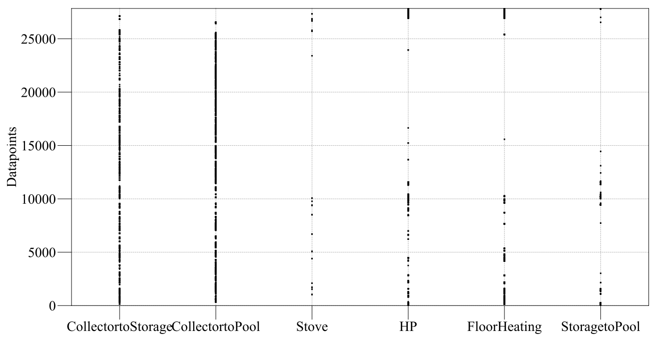


Figure 4. Visualization of data sparsity in mass flow rates of each component

2.3 Physical Models

The physical models were implemented in the Modelica language (Fritzson and Engelson 1998). Modelica is an open source, a-causal, object-oriented and multi-domain modelling language. A discussion of limitations and promising approaches of the Modelica language can be found (Schweiger, Nilsson, et al. 2020). All the models used in this system are based on the Modelica IBPSA Project 1 library (Wetter, Treeck, et al. 2019) and the Buildings Library (Wetter, Zuo, et al. 2014). Dymola was used to simulate Modelica models (Brück et al. 2002). The following sub - implemented in Modelica are: Solar system (`Buildings.Fluid.SolarCollectors.EN12975`), heat pump (`Buildings.Fluid.HeatPumps.CarnotTCon`) and the indoor pool (`Modelica.Fluid.Vessels.ClosedVolume`). The energy demand of the house and the heat supply profile of the stove were taken from the measurement data instead of modelling these components. Since there was no weather profile acquired within the given data period, Typical Meteorological Year 3 (TMY3) for Austria were generated from Meteonorm.

2.4 Machine Learning Model

There was no available information of the system parameters of the storage tank such as the insulation material and the thickness, the wall thickness, the height or the locations of the temperature sensors. Therefore, the storage tank was modelled based on RF. RF is a combination of tree predictors which splits nodes based on a best split of random subsets of the features, thus reducing the variance of the tree model and increasing the overall predictive power of the model.

The RF model predicted the four temperature layers of the storage tank. An overview of the input features for the model is given in Figure 5. The static input features are temperatures, T_i, and mass flow rates, dm_i, from the solar collector, heat pump, floor heating and stove. The

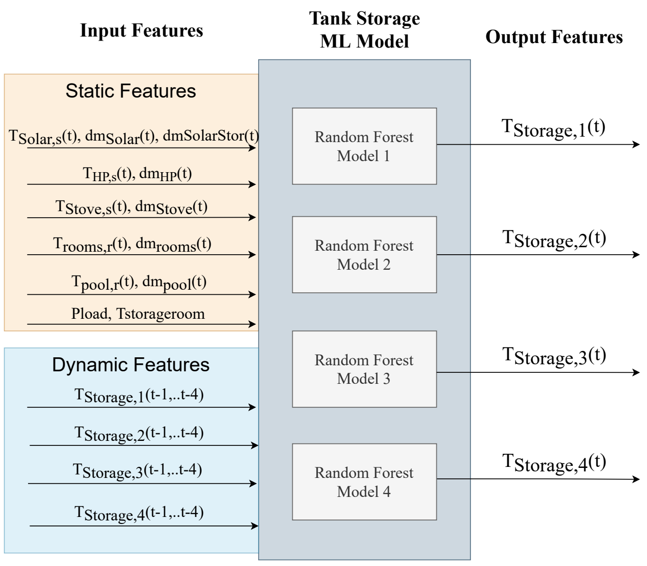


Figure 5. Input/Output features of the applied machine learning model of the storage tank.

dynamic features are the four temperature layers of the storage tank with a 1-hour look-back time with interval 15 minutes and 15 minutes prediction horizon, see Figure 6. The measured data was split randomly into training (80%, 50 epochs) and testing (20%). The model hyperparameters are n_estimators = 100 which represents the number of decision trees that achieves the best trade-off between the accuracy and efficiency; max_depth that has been set to an unlimited value so the nodes can expand automatically; and min_samples_split = 2. The implementation was done using the Python framework presented in (Schranz et al. n.d.) based on Scikit-learn.

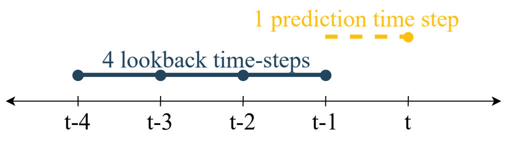


Figure 6. At time t, four look-back time-steps are used to predict one time-step in future with each step = 15 minutes.

2.4.1 Model Performance Analysis

Two criteria were selected to evaluate the performance of the RF model: the coefficient of variation of the Root mean square error (CVRMSE) and mean absolute percentage error (MAPE) given in Equation 1 and Equation 2.

$$CV(RMSE) = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2}}{Y} * 100 \tag{1}$$

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{|Y_i - \hat{Y}_i|}{Y_i}\right) * 100 \tag{2}$$

where Y is the true value, $\hat{Y}$ is predicted value, $\bar{Y}$ is the average of the true values over N test samples.

2.5 Co-Simulation

To integrate the ML model of the storage tank into the simulation environment, the ML model must be exported as an FMU. The UniFMU tool (Legaard et al. 2021) was used to generate a Python-based FMU. Therefore a template of the FMU was generated using the command `unifmu generate python name`.

To specify the behavior the dummy example in Listing 1, implemented by the generated FMU, is replaced with the components of the ML model. A benefit of this is that the `scikit-learn` code can be reused and integrated into the FMU gradually. This makes sure that no breaking changes occur. A crucial part of the FMUs implementation is the `fmi2DoStep` method which instructs the model to simulate forward in time for an amount of time corresponding to the time step. For the storage-tank FMU, this is equivalent to running one or more inference steps of the trained model.

Listing 1. Implementation of `fmi2DoStep` by storage model.

```
from sklearn.ensemble import
    RandomForestRegressor
from sklearn.datasets import
    make_regression
...
def do_step(current_time,step_size,
    no_step_prior):
     self.temp_next=self.forrest(self.
         temp_prevs)
return Fmi2Status.ok
```

3 Results and Discussion

3.1 Validation of the ML model

Table 1 shows the model performance on the test data set. The ML model is imported as FMU in Dymola. Testing of the FMU-ML model with the measurement data is performed in Dymola environment, see Figure 7. $T_{Storage,4}$ and $T_{Storage,3}$ are the worst predicted target value according to the CVRMSE and MAPE. The discussion of Table 1 is supported with the results of the testing.

Table 1. Performance metrics of predicting the four target temperature values: $T_{Storage,1}$, $T_{Storage,2}$, $T_{Storage,3}$ and $T_{Storage,4}$ using random forest models

	CVRMSE	*MAPE*
$T_{Storage,1}$	0.0097	2.3056
$T_{Storage,2}$	0.011	2.2656
$T_{Storage,3}$	0.0157	4.9064
$T_{Storage,4}$	0.0281	6.0215

Winter, spring and summer periods were chosen, aiming to represent different boundary conditions. The only difference in each test period was the initial values set for the FMU-ML. These initial values of the static and dynamic input features were chosen from the measurement data based on each period starting time. Since the stratified hot water storage tank is a short term storage, the daily predictions are representative. In these three figures, 9 days period for each season was chosen to show the model prediction based on interactions of all heating supplies. In Figure 8, Figure 9 and Figure 10, the first subplot represents the comparison between the true and the predicted temperature values of each 4 layers of the storage tank. The true temperature values of the top layer, middle-top layer, middle bottom layer and the bottom layer are respectively red, dark orange, blue and cyan dashed lines. The predicted temperature values are represented the same color code but in straight lines. In the second subplot, the mass flow rates from different components are given. The mass flow rates stand for when the specific component is turned on/off.

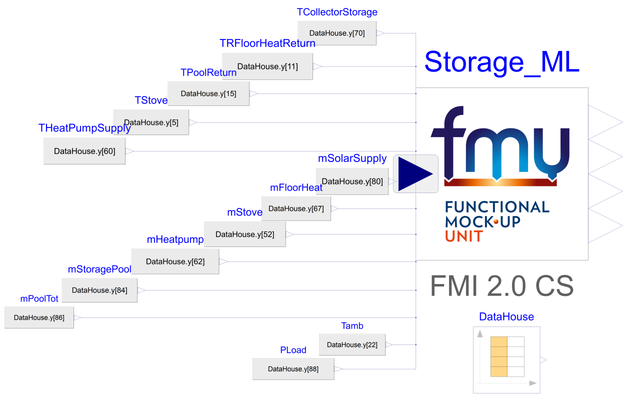


Figure 7. Dymola layout of testing FMU-ML storage model

Figure 8 represents the frequently running components; solar collector and pool heating for summer period. Based on these components inputs, the ML model shows good aggrement with the measurement data during the summer period. One of the static input features, "PLoad", which indicates whether at least one component in the system is on, is introduced to capture the cooling behavior of the storage tank. It is observed in the summer period during the night when there is no load, the four temperature values of the storage tank decrease.

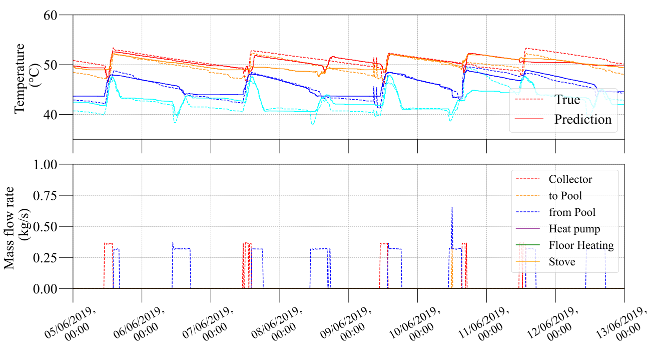


Figure 8. Storage temperature levels predicted vs measurement in summer period

The cold return water from the floor heating is fed into the storage tank from the bottom and the middle layer. These temperatures are expected to decrease as in the measurement data. In Figure 9, between 18th and 19th March, when the floor heating starts, the predictions of the temperature of the bottom layer fails. On the 19th March, when the heat pump is on, represented in the purple line, the middle bottom temperature shows an increasing behavior. However, it cannot reach to the values of the measurement data. On 20th March, only two components are on as in the summer period. On this day, the prediction of the temperature values can catch the measurement data.

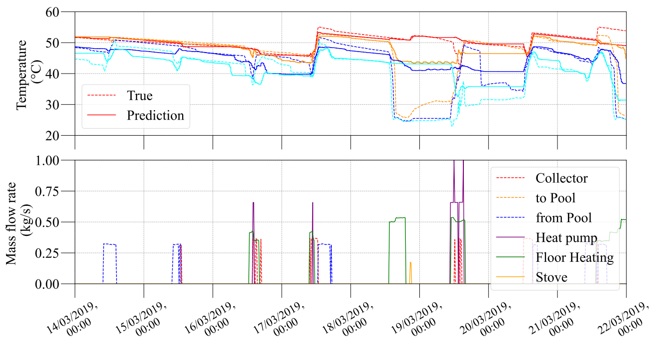


Figure 9. Storage temperature levels predicted vs measurement in spring period

The winter period is selected between February and March due to lack of winter representation from the data, explain in Section 2.2. The bottom temperature layer of the storage tank shows a decreasing behavior due to the floor heating as in the measurement data. However, due to the control strategies when the heat pump turns on, the middle bottom temperature of the storage tank doesn't increase as in the measurement data profile.

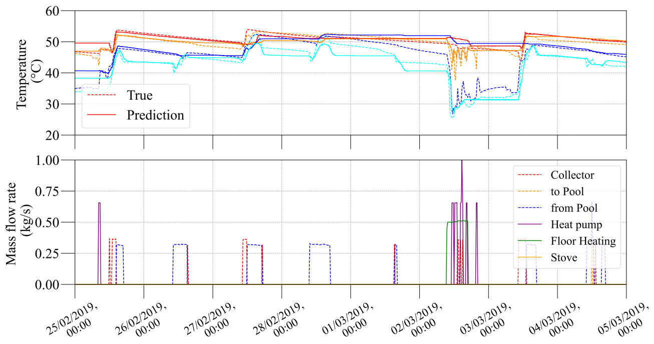


Figure 10. Storage temperature levels predicted vs measurement in winter period

From these three testing periods of the FMU-ML model, it is observed that the temperature values of the storage tank is predicted better when there is only collector component is on. In Figure 4, data that represent collector to storage and collector to pool is denser than the other components data. Therefore, these static input features can dominate the predictions more than the other static features which are sparse. Additionally, the dynamic features of the past predicted temperature values of the storage tank are as well input features. Once these values are predicted wrong, the error accumulates to the further time steps. Results from these tests also show the the performance of the $T_{Storage,4}$ and $T_{Storage,3}$ are worse than the other predicted target values.

3.2 System simulation

In Appendix, Figure A.2 shows the system implementation in Dymola. All heating components explained in use-case are framed with dashed lines in the figure. The figure is visually simplified by hiding the source/sink component inside of the 'StorageML_FMU' component where the supply or return fluid from each component is fed to storage tank. All the simulations are run in a virtual Ubuntu environment with 188 GB RAM and the Intel Xeon Silver 4215R CPU @ 3.2GHz CPUs. Dymola 2021 FD01 with Dassl solver and 10e-6 tolerance was used during this study. The system simulation with the FMU-ML was run 10 times. The averages of the CPU times for the 32 days simulation with 15 minutes interval is 228 seconds. The CPU-time taken to calculate one grid interval highly depends on how the ML algorithm is implemented, number of inputs features, number of processors.

The translated model statistics are given in Table 2. The originally described system has 1966 non-trivial DAEs, after translation it is reduced to an ODE system with 42 continuous time states.

Table 2. Model statistics:Translated model statistics of the single-family house with the FMU-ML storage component.

	FMU-ML Model
Constants	1733
Parameters depending	654
Continuous time states	42
Time varying variables	777
Alias variables	1342
Sizes of linear system of equations	{5}
Sizes after manipulation of the linear system of equations	{0}
Sizes of nonlinear system of equations	{6, 5, 3, 1, 1}
Sizes after manipulation of the nonlinear system of equations	{1, 1, 1, 1, 1}
Number of numerical Jacobians	0

Figure 11 shows the results of coupling ML and physical models of the single-family house heating system. The first subplot in Figure 11 shows the temperature levels of the tank for a 32 days period. The second subplot shows which component is switched on and the third shows the load condition for the storage tank. Despite the same real-world control strategies implemented into system, weather profile that represents the the measurement data is not available. The TMY3 from Meteonorm is used

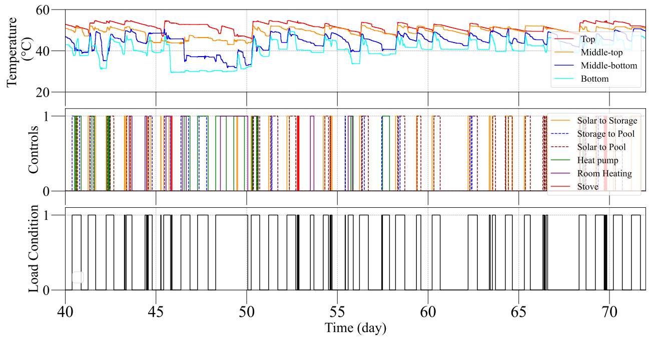


Figure 11. Storage temperature levels according to the controls

for the system simulation and solar collector provides different outputs than the measurement data. And all outputs based on the control strategies changes. Also the physical model parameters of the storage tank have not been adjusted to give a good comparison. Therefore, the results from FMU-ML system simulation cannot be compared with the measurement data.

Acknowledgements

The presented research is part of the project “NextHyb2” (FFG project number 881150) financed by the Austrian Research Promotion Agency FFG. We are grateful to Klima Energie Fonds for sharing the data. We also would like to acknowledge the contributions and the fruitful discussions of Carles Ribas Tugores and Walter Becke in the context of the use case study.

FFG Forschung wirkt. klima+ energie fonds

References

Brück, Dag et al. (2002). “Dymola for multi-engineering modeling and simulation”. In: *Proceedings of modelica*. Vol. 2002. Citeseer.

Fritzson, Peter and Vadim Engelson (1998). “Modelica—A unified object-oriented language for system modeling and simulation”. In: *European Conference on Object-Oriented Programming*. Springer, pp. 67–90.

Géczy-Vig, P. and I. Farkas (2010). “Neural network modelling of thermal stratification in a solar DHW storage”. In: *Solar Energy* 84, pp. 801–806. ISSN: 2261-236X. DOI: 10.1051/matecconf/201822501015.

Gomes, Cláudio et al. (2018). “Co-simulation: a survey”. In: 51.3. ISSN: 0360-0300. DOI: 10.1145/3179993.

Harish, VSKV and Arun Kumar (2016). “A review on modeling and simulation of building energy systems”. In: *Renewable and sustainable energy reviews* 56, pp. 1272–1292.

Killian, Michaela and Martin Kozek (2016). “Ten questions concerning model predictive control for energy efficient buildings”. In: *Building and Environment* 105, pp. 403–412.

Legaard, Christian Møldrup et al. (2021). “A Universal Mechanism for Implementing Functional Mock-up Units”. In: *11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. SIMULTECH 2021. Virtual Event, to appear.

Li, Xiwang and Jin Wen (2014). “Review of building energy modeling for control and operation”. In: *Renewable and Sustainable Energy Reviews* 37, pp. 517–537.

Lund, Henrik et al. (2017). “Smart energy and smart energy systems”. In: *Energy* 137.2, pp. 556–565. DOI: 10.1016/j.energy.2017.05.123.

O’Dwyer, Edward et al. (2019). “Smart energy systems for sustainable smart cities: Current developments, trends and future directions”. In: *Applied energy* 237, pp. 581–597.

Palensky, Peter, Edmund Widl, and Atiyah Elsheikh (2013). “Simulating cyber-physical energy systems: Challenges, tools and methods”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 44.3, pp. 318–326. DOI: 10.1109/TSMCC.2013.2265739.

Schranz, Thomas et al. (n.d.). “Energy Prediction under Changed Demand Conditions:Robust Machine Learning Models and Input Feature Combinations”. In: *Building Simulation 2021. International Building Performance Simulation Association*.

Schweiger, Gerald, Georg Engel, et al. (2018). “Co-simulation an empirical survey: applications, recent developments and future challenges”. In: *MATHMOD 2018 Extended Abstract Volume*, pp. 125–126.

Schweiger, Gerald, Henrik Nilsson, et al. (2020). “Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms”. In: *Applied Mathematics and Computation* 365, p. 124713.

Thieblemont, Hélène et al. (2017). “Predictive control strategies based on weather forecast in buildings with energy storage system: A review of the state-of-the art”. In: *Energy and Buildings* 153, pp. 485–500.

Thilker, Christian Ankerstjerne, Henrik Madsen, and John Bagterp Jørgensen (2021). “Advanced forecasting and disturbance modelling for model predictive control of smart energy systems”. In: *Applied Energy* 292, p. 116889.

Vatanparvar, Korosh and Mohammad Abdullah Al Faruque (2018). “Control-as-a-Service in Cyber-Physical Energy Systems over Fog Computing”. In: *Fog Computing in the Internet of Things: Intelligence at the Edge*. Ed. by Amir M. Rahmani et al. Cham: Springer International Publishing, pp. 123–144. ISBN: 978-3-319-57639-8. DOI: 10.1007/978-3-319-57639-8_7. URL: https://doi.org/10.1007/978-3-319-57639-8_7.

Wetter, Michael, C van Treeck, et al. (2019-09). “IBPSA Project 1: BIM/GIS and Modelica framework for building and community energy system design and operation – ongoing developments, lessons learned and challenges”. In: vol. 323. IOP Publishing, p. 012114. DOI: 10.1088/1755-1315/323/1/012114. URL: https://doi.org/10.1088/1755-1315/323/1/012114.

Wetter, Michael, Wangda Zuo, et al. (2014). “Modelica Buildings library”. In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506. URL: https://doi.org/10.1080/19401493.2013.765506.

A Appendix

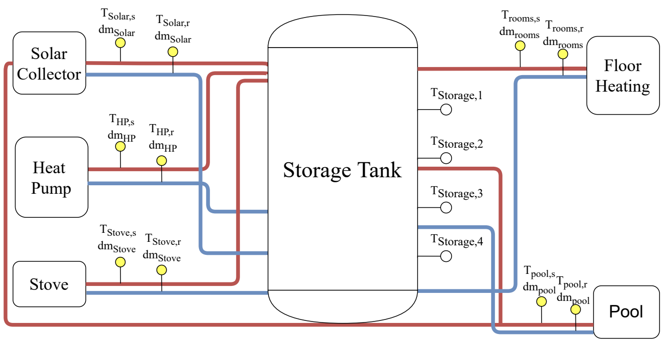


Figure A.1. Overview of the system hydraulic flow. The supply pipe is represented in red, return pipe in blue. In each pipe, the temperature and mass flow rates are measured.

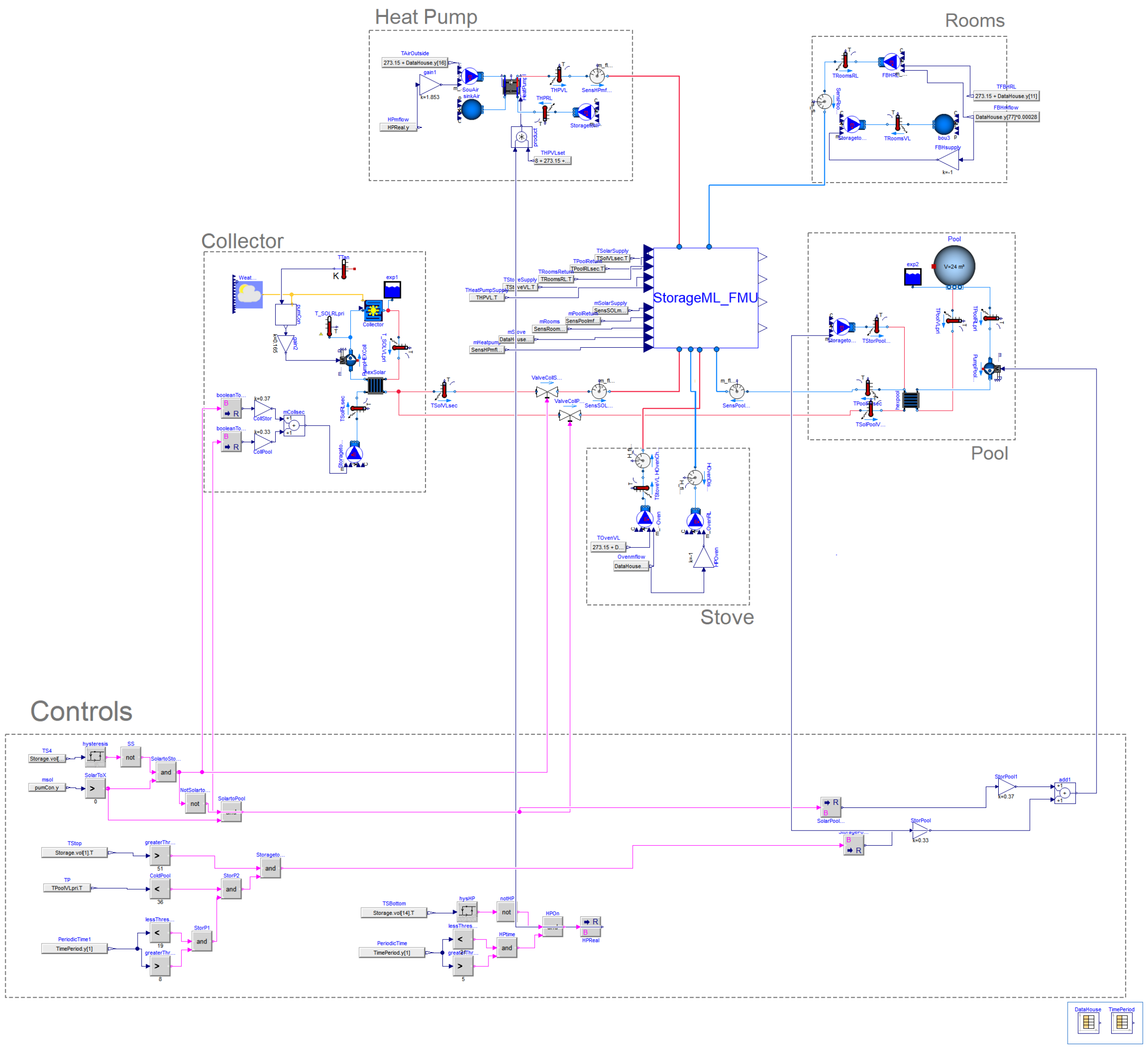


Figure A.2. Dymola layout of the single-family house heating system

Chapter 12

Energy Prediction under Changed Demand Conditions: Robust Machine Learning Models and Input Feature Combinations

The paper presented in this chapter has been published in the peer-reviewed conference of *Proceedings of Building Simulation 2021: 17th Conference of IBPSA*

Energy Prediction under Changed Demand Conditions: Robust Machine Learning Models and Input Feature Combinations

Thomas Schranz[1], Johannes Exenberger[1], Christian Møldrup Legaard[2], Ján Drgoňa[3], Gerald Schweiger[1]
[1]Graz University of Technology, Graz, Austria
[2]Aarhus University, Aarhus, Denmark
[3]Pacific Northwest National Laboratory, Richland, WA, USA

Abstract

Deciding on a suitable algorithm for energy demand prediction in a building is non-trivial and depends on the availability of data. In this paper we compare four machine learning models, commonly found in the literature, in terms of their generalization performance and in terms of how using different sets of input features affects accuracy. This is tested on a data set where consumption patterns differ significantly between training and evaluation because of the Covid-19 pandemic. We provide a hands-on guide and supply a Python framework for building operators to adapt and use in their applications.

Key Innovations

With this paper, we contribute to the state of the art in building energy forecasting by assessing the performance and the robustness of four machine learning algorithms (linear regression, random forest, fully-connected neural network, and recurrent neural network) with various sets of input features. We analyze models in terms of their ability to predict short-term energy demand in a building where the consumption patterns differ significantly between training and test data because of the Covid-19 pandemic.

We provide guidelines for practitioners by

- examining how different lookback and prediction horizons influence the accuracy and robustness of the machine learning models for single-step energy demand prediction
- benchmarking models using additional input features, such as weather data, against models predicting future energy demand from past consumption values only.
- examining the potential of integrating water consumption data for data-driven energy prediction.

In order not to bias the comparison, the models were not hyper-parameter-tuned to our use case. Instead, the Python machine learning framework, as well as the data used in the experiments described here is published on Github (*Link will be supplied after review*). This allows researchers and practitioners to reproduce the results presented in this paper, adapt the framework for their purposes and/or develop and improve models to match their requirements (e.g. in terms of accuracy).

Practical Implications

- Use features engineered from date and time (time of day, weekday, holiday), as it efficiently increases model performance and robustness.
- Random forest provides a simple solution for prediction tasks with adequate accuracy also in scenarios of changed demand patterns.
- Integrating water consumption is not generally recommended to increase robustness, as it is only beneficial in specific forecasting scenarios.

Introduction

The European Union proposed a comprehensive set of measures to drastically cut greenhouse gas emissions by 2030 and become the first climate-neutral continent by 2050 (European Commission and Climate Action DG, 2019). With a share of 78%, the energy sector is the largest contributor of emissions in the European Union (EU) and Iceland (European Environmental Agency, 2020). Driven by socioeconomic changes, such as the trend towards larger homes and the broad availability of energy-consuming entertainment, the buildings sector has become the largest contributor to the global energy demand (Allouhi et al., 2015), accounting for 40% of the energy consumption in the EU (European Commission, 2016).

It is apparent that building designers and operators play a principal role in the reduction of energy-related greenhouse gas emissions. Besides traditional energy saving measures, such as improving thermal insulation or retrofitting heating systems, hot-water boilers and lighting systems, Chwieduk (2003) proposed the introduction of environmentally-friendly energy technologies in the form of automation and data analysis to control, optimize and reduce energy demand as early as 2003. Other strategies include increasing the share of volatile renewable energy sources (Mathiesen et al., 2015), (Chwieduk, 2003), reducing energy demand by encouraging active user participation (Schweiger et al., 2020), (Schranz et al., 2020), and the use of energy services such as demand response (Meyabadi and Deihimi, 2017) or model predictive control (Mariano-Hernández et al., 2021). Many of these strategies rely on accurate prediction of future energy demand, which remains a key research interest.

Advances in embedded computing and cloud technologies provide researchers with large amounts of operational data such as detailed historical energy consumption. Among others, Rätz et al. (2019) have highlighted the capabilities of data-driven modeling techniques and their applications to building control and building energy optimization.

In this paper, we compare the performance of four different data-driven models (linear regression, a random forest regression ensemble, a fully-connected neural network, and a recurrent neural network) that predict future energy demand from past observations in a scenario where measures against the Covid-19 pandemic drastically changed consumption patterns. We benchmark models predicting from past energy consumption values only against models with additional input features, such as water consumption data or weather information in terms of performance and generalization ability. Besides, we examine if the models are able to provide reasonable predictions for the time during the pandemic even when they were trained on consumption data from before the pandemic.

We aim to provide general observations and guidelines for practitioners to help them develop data models for energy forecasting, rather than present another set of models that provide good results for the data set used in the paper but might not work well for their particular use case or data set. We test models that provide straightforward implementation without the necessity for complex hyper-parameter tuning. We use what we consider "default" hyper-parameter settings, that are typically found in the literature, and/or default choices in the Python libraries we use. Consequently, we focus on identifying the most relevant input features for prediction performance and robustness, and how different models are affected by the choice of feature sets, lookback and prediction horizon.

Building Energy Forecasting: Related Work

There exists a large body of literature in the domain of data-driven load forecasting for building energy prediction. Latest reviews of state of the art approaches can be found in Wei et al. (2018), Fathi et al. (2020), and Sun et al. (2020). Machine learning models, such as decision trees, support vector machines (SVM) (Jain et al., 2014), nonlinear regression (Wei et al., 2019), and deep neural networks (Somu et al., 2021), all of which have been applied successfully in the domain of energy forecasting for more than a decade now (Zhao and Magoulès, 2012), are among the most popular approaches.

The majority of models found in the literature use meteorological data, features engineered from date and time, occupancy and historical energy consumption as inputs. Models described in recent studies predict the energy demand for a myriad of building types, such as educational buildings, offices, commercial and industrial complexes as well as residential buildings on various levels of aggregation, ranging from single buildings over city blocks to whole districts. Key objectives are the prediction of total energy demand, electric power consumption or heating/cooling demand. Even when focusing on use cases that are comparable in methodology and objective to the present paper, i.e. the application of data-driven methods to predict electric energy demand of a single office building (academic/mixed-use), a substantial amount of contributions can be identified.

Wang et al. (2018) apply a random forest ensemble to predict hourly energy usage of two university buildings and validate the performance against a regression tree and a SVM model. The model input consists of weather data, including temperature, wind speed and solar radiation, information about the time of day, the weekday, etc. and estimated occupancy data. Hourly occupancy is approximated using class schedules and the number of students registered for each class, the number of staff members working in the building and time tables. The authors find that occupant behavior is a key contributor to uncertainty and that the significance of input variables, likely because of changing operational conditions, varies for different semesters.

Walker et al. (2020) compare the performance of seven machine learning algorithms for time series analysis of energy consumption, including random forests, SVMs and artificial neural networks. The authors use the models to predict hourly energy demand on individual building level and on a building cluster consisting of 47 commercial buildings. They base their choice of features on their understanding of building operation and choose weather information, categorical date and time features (day of week, time of day) and autoregressive past consumption values (past day and past week) for day ahead predictions and acknowledge that the feature selection has a non-trivial influence on model accuracy.

Gaussian process regression, a machine learning prediction approach with relatively low computational complexity is presented in Zeng et al. (2020). The authors use standardized regression coefficients as well as what they call domain knowledge of energy computation to chose weather conditions and information from occupancy schedules to predict energy

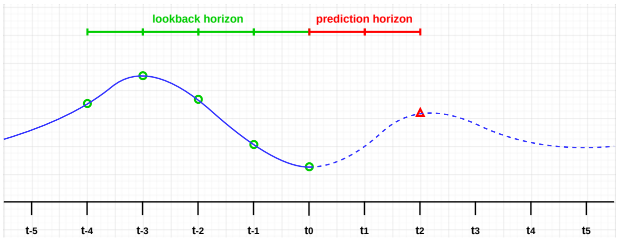


Figure 1: Lookback of four and prediction horizon of two. Four past values (marked in green) are used to predict one value two timesteps ahead (marked in red).

consumption in six large commercial buildings. The potential of transfer learning in artificial neural networks for 24h ahead building energy demand prediction is investigated in Fan et al. (2020), using data of 507 non-residential buildings including offices, schools and university facilities. The authors conclude that transfer learning can be a useful approach when insufficient training data is available.

Method

In this section we describe the use case and its boundary conditions and approximate mathematical formulation of the prediction process. We characterize the data, its sources and the preprocessing steps we applied to it. We explain the data models, including their inputs and outputs, outline the training process for the neural networks and list the metrics we used.

Use Case Description

In this paper we investigate approaches to predict hourly electric energy usage of a five-floor, mixed-use academic building at the Graz University of Technology (TUG) accommodating offices, seminar rooms, laboratories and a lecture hall.

The data models we developed predict a single energy consumption value from a finite set of previously observed values, i.e. the forecasting problem is formulated as a generic regression problem (Bontempi et al., 2013). There are two parameters to this process: i) the lookback horizon, i.e. the number of past observations the prediction is based on and ii) the prediction horizon, i.e. how far the model predicts into the future.

In Figure 1 the blue graph represents the time series development, with the solid line representing past and the dashed line representing future values (with respect to the current time t_0). In this example, four values from the past in conjunction with the current value (marked with green circles) constitute the basis for projecting the value (marked as a red triangle) of time series two steps into the future. This corresponds to a lookback horizon of four and a prediction horizon of two.

In the baseline models, predictions are inferred from a single type of previously observed values. In the dynamical systems approach, this is considered as estimating the future system state from previous and current system states - i.e. predictions of the hourly energy consumption are based on previous consumption values only. Each input vector $\mathbf{x}$ is of size $[n, 1]$, where n = lookback horizon l + one current state. The output y is a single scalar value, i.e. the expected energy consumption c^i at time step $t+p$, where p is the prediction horizon.

$$\mathbf{x} = \begin{bmatrix} c^{t-l} \\ \vdots \\ c^{t} \end{bmatrix}, \qquad y = c^{t+p}$$

To improve prediction accuracy it is possible to use additional information that is expected to be correlated with the energy consumption to describe the system state at any given point in time. Examples are information about the outside temperature, whether it was day or night, the hourly water consumption in the building, or the number of registration for a lecture hall or seminar room located within the building. Using multiple characteristics to describe the system state turns the scalar components in the input vector $\mathbf{x}$ into vectors. Consequently, the input vector turns into an input matrix $\mathbf{X}$ of size $[n, m]$, where m corresponds to the number of characteristics, subsequently denoted features f_k^i. The model output y remains unchanged, as we are still only interested in the future energy consumption.

$$\mathbf{X} = \begin{bmatrix} f_0^{t-l} & \dots & f_m^{t-l} \\ \vdots & & \\ f_0^{t} & \dots & f_m^{t} \end{bmatrix}, \qquad y = c^{t+p}$$

Recurrent neural networks are designed to handle two dimensional inputs with a time axis and a feature axis. For the other models, i.e. linear regression, random forest and fully-connected network, this matrix has to be flattened to a vector $\mathbf{x}$, effectively removing any distinction between different features of the same time step and same features of different time steps.

$$\mathbf{x} = \begin{bmatrix} f_0^{t-l} & \dots & f_m^{t-l} f_0^{t} & \dots & f_m^{t} \end{bmatrix}, \quad y = c^{t+p}$$

Be aware that the energy consumption values may or may not be part of the state description. In the subsequent section we outline the data available to us.

Data

In this section we describe the nature and sources of the data we used in the experiments. The Buildings and Technical Support (BATS) department at the TUG manages buildings and infrastructure on three campuses. In the course of their operations they capture real time data from various types of smart sensors (Schranz et al., 2020). For the experiments described here, the BATS department provided us with consumption sequences from smart water and smart energy meters located in a mixed-use academic/office building.

Energy and water consumption data is available from May, 5 2019 to July, 21 2020 in one-hour intervals. It is worth noting the special character of this timeframe as the months from March 2020 to July 2020 coincide with the Covid-19 pandemic. Due to the measures imposed by the Austrian federal government and the university directorate access to the academic facilities was restricted, classroom teaching suspended and staff members were directed to work from home whenever possible. This provides the opportunity to investigate how successful the models are in predicting the energy demand given the significant change in consumption behavior because of the restrictions. This is of particular interest because the boundary conditions stay the same, i.e. training and test data are still from the same building but occupancy and occupant behavior are different. All models were trained using data recorded prior to the measures coming into effect and tested on data from periods where the restrictions were in place.

Occupancy data is approximated through schedules exported from the TUG resource management system. The dataset contains the dates and registrations for all events and courses that take place in the lecture hall located in the building.

Weather data is obtained through a web API (http://at-wetter.tk) that provides access to the open data collection published by the Austrian "Zentralanstalt für Meteorology und Geodynamik" (ZAMG). The data was captured at the Graz Airport, which is located about 8 km from the campus site and contains, among other metrics, hourly temperature measurements and the time of sunrise and sunset.

The **date and time features** are engineered from the timestamps the energy consumption data is indexed with. Plots show that energy consumption amplitude correlates with the weekday, the occurrence of public holidays and that consumption is, on average, notably higher during the semester. For each time step we calculate the weekday, a Boolean feature encoding whether it coincides with a public holiday, a Boolean feature whether it is during the semester and the time of day. The time of day and the numeric representation of the weekday assume periodic values, 0 to 23 (because consumption is sampled hourly) and 0 to 6, respectively. To capture periodicity both features are encoded with a sine/cosine pair (Drezga and Rahman, 1998).

Models

We developed four data models, two statistical models (linear regression and a decision tree ensemble) and two neural networks (a fully-connected sequential model and a recurrent network). The entire framework (for data preparation, preprocessing, training, benchmarking and plotting) and all models were implemented using Python 3. For the statistical models we used the machine learning library sklearn (https://scikit-learn.org/stable) and for the neural networks we used the TensorFlow (https://www.tensorflow.org) implementation of the Keras API (https://keras.io).

The **linear regression (LR)** model is relatively simple and requires no parameter settings. As the input samples have to be provided as one-dimensional vector, multi-feature input matrices are flattened along the time axis in a pre-processing step.

The **random forest (RF)** is built from 100 estimators, with no restrictions on maximum depth. A minimum number of two samples is required for splitting internal nodes, at each leaf node there has to be at least one sample. The mean squared error (MSE) is used as splitting criterion and all input features are used in the split. The random forest requires input samples to be vectors, i.e. multi-feature inputs have to be flattened along the time dimension.

The **neural networks** are built as a stack of layers with inputs of variable size. They accept either a vector (only one scalar value per lookback step) or a $[n, m]$ matrix of multiple features for multiple time steps (although the fully-connected network flattens it internally) and produce variable size outputs. With this architecture, it is possible to either predict multiple system characteristics at one time step or one characteristic at multiple time steps in the future. However, to obtain comparable results to the ones generated by the statistical methods, we used exactly one output, i.e. the hourly energy consumption at one single point in the future defined by the prediction horizon.

All networks are trained on batches of 72 samples using an RMSprop optimizer with a learning rate of 0.001, ρ and ϵ parameters of 0.9 and $1e^{-7}$ respectively. We set the maximum number of training epochs to 200 but implemented early stopping to avoid overfitting. The training data is split such that the last 20% of the samples (subsequently called validation data) in the set are used to monitor convergence. Training is stopped when the mean squared error on the validation data does not decrease for 30 consecutive training epochs.

The **fully-connected neural network (NN)** flattens the input along the time dimension, i.e. it uses the same input as the two statistical methods. The flattened input passes through two layers consisting of 64 rectified linear units (ReLU) and is output in a single dense layer with one unit.

The **recurrent neural network (RNN)** uses an architecture designed specifically for time series prediction. The network sequentially applies transformations along the input's time axis, remembering infor-

Table 1: Errors of energy demand prediction for all models using different feature combinations. The bold numbers indicate the best performing models for every feature combination, while the gray cells indicate the overall best performing combination of model and features.

Features		Baseline Model: Energy consumption					Energy consumption, weather				
Prediction horizon		1h	3h	6h	12h	24h	1h	3h	6h	12h	24h
LR	CV-RMSE	0.14	0.24	0.28	0.30	0.31	0.14	0.23	0.27	0.29	0.32
	MAPE	6.5	12.1	16.7	20.7	23.0	6.8	13.0	17.8	21.0	22.7
	R^2	0.78	0.40	0.16	0.07	-0.03	0.78	0.44	0.22	0.10	-0.04
RF	CV-RMSE	**0.14**	0.23	**0.25**	0.27	0.28	0.14	0.22	0.24	**0.25**	**0.27**
	MAPE	**5.8**	**9.5**	12.8	16.5	19.0	**5.5**	**8.7**	**10.7**	**14.2**	**15.5**
	R^2	**0.78**	0.44	**0.33**	0.24	0.16	0.79	0.47	0.39	**0.32**	**0.22**
NN	CV-RMSE	0.15	**0.22**	0.26	**0.27**	0.29	0.14	**0.21**	**0.24**	0.29	0.29
	MAPE	6.4	10.4	**12.5**	**15.3**	**16.2**	6.4	9.7	12.5	20.3	18.3
	R^2	0.77	**0.51**	0.30	**0.25**	0.13	0.79	**0.55**	**0.41**	0.11	0.15
RNN	CV-RMSE	0.15	0.24	0.28	0.29	**0.28**	**0.14**	0.22	0.25	0.30	0.29
	MAPE	5.9	10.7	17.3	21.6	18.0	6.1	11.6	12.8	18.3	20.3
	R^2	0.77	0.39	0.18	0.13	**0.20**	**0.79**	0.48	0.34	0.08	0.11
Features		Energy consumption, datetime					Energy consumption, water				
Prediction horizon		1h	3h	6h	12h	24h	1h	3h	6h	12h	24h
LR	CV-RMSE	0.14	0.22	0.25	0.26	0.27	0.14	0.23	0.28	0.30	0.32
	MAPE	6.7	12.5	16.0	16.8	16.4	6.6	12.4	17.7	22.2	25.1
	R^2	0.79	0.47	0.32	0.28	0.27	0.79	0.43	0.20	0.08	-0.08
RF	CV-RMSE	0.14	**0.19**	**0.21**	**0.20**	0.23	0.15	0.23	0.25	**0.24**	0.29
	MAPE	**5.5**	**7.7**	**9.0**	**9.9**	**11.9**	**5.9**	9.3	**11.8**	**15.2**	19.4
	R^2	0.79	**0.61**	**0.55**	**0.57**	0.44	0.78	0.45	0.34	**0.37**	0.12
NN	CV-RMSE	**0.14**	0.22	0.25	0.27	0.28	**0.14**	**0.21**	**0.25**	0.27	**0.29**
	MAPE	6.9	14.3	16.7	20.3	16.6	6.0	**9.1**	12.6	15.5	**16.0**
	R^2	**0.80**	0.49	0.35	0.23	0.20	**0.79**	**0.55**	**0.35**	0.26	**0.14**
RNN	CV-RMSE	0.14	0.20	0.23	0.29	**0.22**	0.14	0.24	0.27	0.27	0.30
	MAPE	6.2	8.9	13.2	19.9	12.5	6.2	11.4	16.2	18.2	23.8
	R^2	0.79	0.56	0.43	0.11	**0.51**	0.78	0.38	0.21	0.22	0.06

mation about each time step. In our configuration, this information is used to produce one single output, i.e. the projected hourly energy consumption. The network consists of 32 long short-term memory (LSTM) units (Hochreiter and Schmidhuber, 1997) with hyperbolic tangent activation, sigmoid recurrent activation and no dropout. The LSTM layer outputs a single value into a dense layer with one unit.

Metrics

To assess and compare model performance we use three error metrics: the coefficient of variance of the root mean square error (CV-RMSE), the mean absolute percentage error (MAPE) and the coefficient of determination (R^2). All three metrics are frequently applied to evaluate model performance for building energy prediction in the literature (Sun et al., 2020). Besides, all of them are independent of the magnitude of the predicted target values, which means the results are more comparable in different settings (e.g other buildings). Additionally, the CV-RMSE is featured in the ASHRAE Guideline for measurement of energy and demand savings (ASHRAE Standards Committee, 2002).

Results

All models were trained separately using six different input feature combinations and lookback horizons of 12, 48 and 72 hours to predict hourly energy demand one, three, six, 12 and 24 hours into the future. The following feature combination were tested:

- energy demand
- energy demand + weather conditions
- energy demand + date/time features
- energy demand + water consumption
- energy demand + occupancy
- combination of all features above

Results show that models using either occupancy or a combination of all features perform poorly. It appears that occupancy data, approximated by the number of registrations for courses and events held in the building does not properly reflect the actual number of occupants. Especially, because the registration managements system does not seem to be reflecting the changes caused by the Covid-19 restrictions. Consequently, these two combinations are omitted in the subsequent analysis. Table 1 contains the results for all other feature combinations for all models with a lookback horizon of 24 hours.

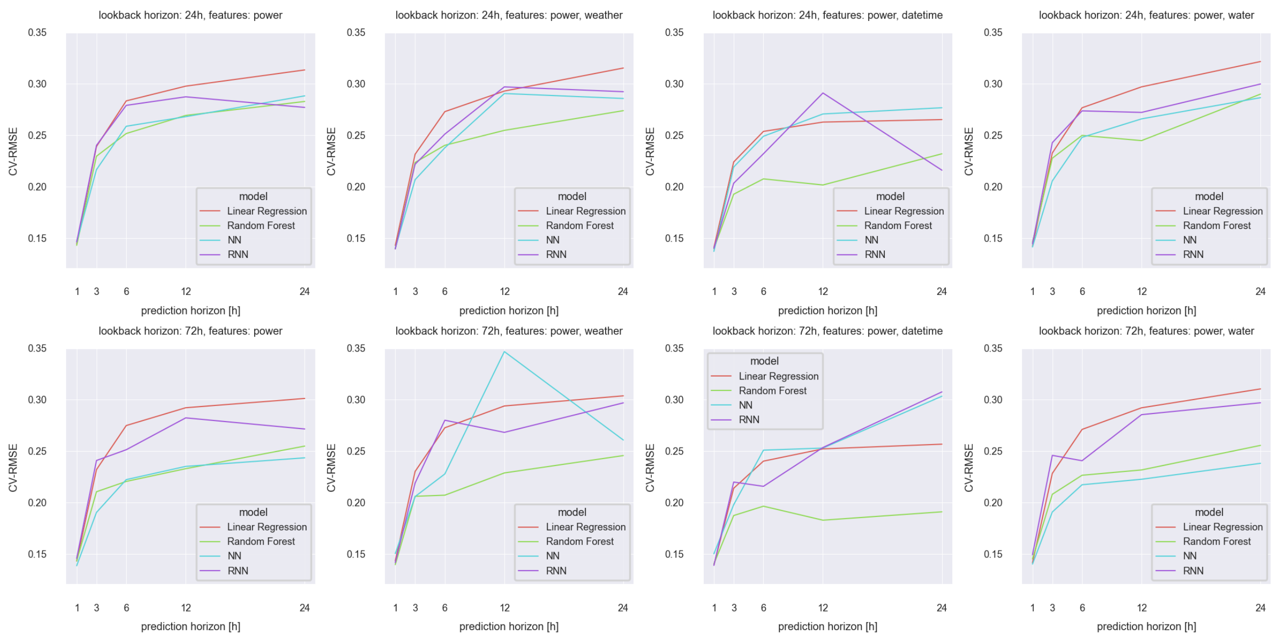


Figure 2: Comparison of the CV-RMSE for all model and feature combinations.

Figure 2 shows the prediction accuracy on the test data for models with a lookback horizon of 24 hours in the first row and 72 hours in the second row. It can be seen that for one-hour-ahead prediction all models perform reasonably well for any choice of input features and lookback horizon. Conversely, 24-hour ahead prediction seems to be difficult for all models. For the three- to 12-hour-ahead prediction, differences in performance between the models and the feature choices are most noticeable. Changing the lookback horizon from 24 to 72 hours does not generally improve accuracy, although the RF seems to benefit slightly from a longer lookback when it uses energy consumption and the date/time features as input. Including water consumption causes no significant change in accuracy for neither model or choice of lookback.

Discussion

Experiments show that the choice of input features and lookback horizon has a varied influence on the different models. We subsequently discuss the findings for each model in detail. There are however, some general observations from all experiments:

- Lookback should be at least 24h.
- All models/features show similar performance for one-hour-ahead prediction.
- Using weather data or water consumption data does not increase accuracy in general (except for fully-connected NN).

Linear regression:

- Using date/time features significantly increases prediction performance compared to using previously observed energy consumption only.
- Changing the lookback from 24 to 72 hours causes no noticeable improvement.

Random forest:

- Using date/time features significantly increases prediction performance compared to using previously observed energy consumption only.
- A longer loockback horizon improves performance.
- Results indicate that RF has the best generalization ability of all models we tested and that it adapts well to the changes in the energy demand patterns.
- RF is the model that is least sensitive to the choice of input features.

Fully-connected neural network:

- With a longer lookback horizion, using hourly water consumption slightly increases prediction performance compared to using previously observed energy consumption only.
- When using previous consumption values alone the NN shows similar performance to the RF model.
- The choice of feature combination and lockback horizon interact with each other.
- The NN does not adapt well to changed demand patterns.
- The NN is very sensitive to the choice of input features.

Recurrent neural network:

- Date/time features significantly increase performance if the lookback horizon and the prediction horizon are 24 hours.

- Performance does not increase with longer lookback horizons.
- The RNN does not adapt well to changed demand patterns.
- The RNN is very sensitive to the choice of input features.

Conclusion

With the development in machine learning algorithms for time series analysis, practitioners are provided with a myriad of choices. Deciding on a suitable algorithm for energy demand prediction in a building is non-trivial and depends on the availability of data. In this paper we compared four machine learning models, commonly found in the literature, in terms of their generalization performance and in terms of how using different sets of input features affects accuracy.

We evaluated the models on three metrics, the coefficient of variance of the root mean square error (CV-RMSE), the mean absolute percentage error (MAPE) and the coefficient of determination (R^2), all of which are widely used to assess building energy demand prediction performance in the literature.

Besides previous consumption values, we used features engineered from date and time (time of day, weekday, holiday), weather data (outside temperature and daylight hours), estimates for occupancy and water consumption data as model inputs. We trained all models on data captured between May 2019 and March 2020, before the onset of the Covid-19 pandemic in Austria, and tested them on data recorded between March and July 2020, where strict measures imposed by the federal government gravely affected energy consumption patterns. The energy and water consumption data was recorded in an academic office building in hourly intervals. We benchmarked different lookback and prediction horizons for the models, ranging from 12 to 72 hours for the lookback and 1 to 24 hours for prediction.

Results show that using features engineered from date and time affects prediction performance most significantly, regardless of the choice of model and lookback horizon. Additionally, we found that simple models, such as linear regression and random forests perform very well both in terms of generalization ability and robustness with respect to the choice of input features and lookback horizon. Especially the random forests showed exceptional generalization performance for all choices of input features. Conversely, the neural networks performed well when predicting from previous consumption values alone, but were sensible to the choice of inputs. It stands to reason that this issue could be addressed with regularization techniques, such as dropout layers, L1 or L2 regularization. Besides, we found that the choice of input features and the choice of lookback horizons for the neural networks interacted with each other. Consequently, we did not find neural networks models to adequately fulfill the requirement of working well without extensive testing and tweaking. Using water consumption data to predict energy demand seemed to improve performance of the neural networks, however, we did not find the results to be conclusive. The benefit of integrating water consumption data into energy prediction models has to be investigated in more detail in follow-up research.

References

Allouhi, A., Y. El Fouih, T. Kousksou, A. Jamil, Y. Zeraouli, and Y. Mourad (2015, December). Energy consumption and efficiency in buildings: current status and future trends. *Journal of Cleaner Production 109*, 118–130.

ASHRAE Standards Committee (2002). ASHRAE Guideline: Measurement of Energy and Demand Savings. *ASHRAE Guideline 14-2002*.

Bontempi, G., S. Ben Taieb, and Y.-A. Le Borgne (2013). Machine Learning Strategies for Time Series Forecasting. In M.-A. Aufaure and E. Zimányi (Eds), *Business Intelligence: Second European Summer School, eBISS 2012, Brussels, Belgium, July 15-21, 2012, Tutorial Lectures*, Lecture Notes in Business Information Processing, pp. 62–77. Berlin, Heidelberg: Springer.

Chwieduk, D. (2003, September). Towards sustainable-energy buildings. *Applied Energy 76*(1-3), 211–217.

Drezga, I. and S. Rahman (1998). Input variable selection for ann-based short-term load forecasting. *IEEE Transactions on Power Systems 13*(4), 1238–1244.

European Commission (2016). Proposal for a directive of the European Parliament and of the council amending Directive 2010/31/EU on the energy performance of buildings. *COM(2016) 765 final*.

European Commission and Climate Action DG (2019). *Going climate-neutral by 2050: a strategic long-term vision for a prosperous, modern, competitive and climate-neutral EU economy.* OCLC: 1140133232.

European Commission, DG Climate ActionEuropean Environment Agency (2020). *Annual European Union greenhouse gas inventory 1990–2018 and inventory report 2020: Submission under the United Nations Framework Convention on Climate Change and the Kyoto Protocol*.

Fan, C., Y. Sun, F. Xiao, J. Ma, D. Lee, J. Wang, and Y. C. Tseng (2020). Statistical investigations of transfer learning-based methodology for short-term building energy predictions. *Applied Energy 262*, 114499.

Fathi, S., R. Srinivasan, A. Fenner, and S. Fathi (2020). Machine learning applications in urban building energy performance forecasting: A systematic review. *Renewable and Sustainable Energy Reviews 133*, 110287.

Hochreiter, S. and J. Schmidhuber (1997, November). Long Short-Term Memory. *Neural Computation 9*(8), 1735–1780.

Jain, R. K., K. M. Smith, P. J. Culligan, and J. E. Taylor (2014). Forecasting energy consumption of multi-family residential buildings using support vector regression: Investigating the impact of temporal and spatial monitoring granularity on performance accuracy. *Applied Energy 123*, 168 – 178.

Mariano-Hernández, D., L. Hernández-Callejo, A. Zorita-Lamadrid, O. Duque-Pérez, and F. Santos García (2021). A review of strategies for building energy management system: Model predictive control, demand side management, optimization, and fault detect & diagnosis. *Journal of Building Engineering 33*, 101692.

Mathiesen, B., H. Lund, D. Connolly, H. Wenzel, P. Østergaard, B. Möller, S. Nielsen, I. Ridjan, P. Karnøe, K. Sperling, and F. Hvelplund (2015). Smart energy systems for coherent 100% renewable energy and transport solutions. *Applied Energy 145*, 139 – 154.

Meyabadi, A. and M. Deihimi (2017). A review of demand-side management: Reconsidering theoretical framework. *Renewable and Sustainable Energy Reviews 80*, 367 – 379.

Rätz, M., A. P. Javadi, M. Baranski, K. Finkbeiner, and D. Müller (2019). Automated data-driven modeling of building energy systems via machine learning algorithms. *Energy and Buildings 202*, 109384.

Schranz, T., K. Corcoran, T. Schwengler, L. Eckersdorfer, and G. Schweiger (2020). Mobile application for active consumer participation in building energy systems. In Monsberger, Michael, Hopfe, Christina Johanna, Krüger, Markus, and Passer, Alexander (Eds), *BauSIM 2020 - 8th Conference of IBPSA Germany and Austria, Proceedings*, pp. 281–287.

Schranz, T., G. Schweiger, S. Pabst, and F. Wotawa (2020). Machine Learning for Water Supply Supervision. In H. Fujita, P. Fournier-Viger, M. Ali, and J. Sasaki (Eds), *Trends in Artificial Intelligence Theory and Applications. Artificial Intelligence Practices*, Lecture Notes in Computer Science, Cham, pp. 238–249. Springer International Publishing.

Schweiger, G., L. V. Eckerstorfer, I. Hafner, A. Fleischhacker, J. Radl, B. Glock, M. Wastian, M. Rößler, G. Lettner, N. Popper, and K. Corcoran (2020). Active consumer participation in smart energy systems. *Energy and Buildings 227*, 110359.

Somu, N., G. Raman M R, and K. Ramamritham (2021). A deep learning framework for building energy consumption forecast. *Renewable and Sustainable Energy Reviews 137*, 110591.

Sun, Y., F. Haghighat, and B. C. M. Fung (2020). A review of the-state-of-the-art in data-driven approaches for building energy prediction. *Energy and Buildings 221*, 110022.

Walker, S., W. Khan, K. Katic, W. Maassen, and W. Zeiler (2020). Accuracy of different machine learning algorithms and added-value of predicting aggregated-level energy performance of commercial buildings. *Energy and Buildings 209*, 109705.

Wang, Z., Y. Wang, R. Zeng, R. S. Srinivasan, and S. Ahrentzen (2018). Random Forest based hourly building energy prediction. *Energy and Buildings 171*, 11–25.

Wei, N., C. Li, X. Peng, F. Zeng, and X. Lu (2019). Conventional models and artificial intelligence-based models for energy consumption forecasting: A review. *Journal of Petroleum Science and Engineering 181*, 106187.

Wei, Y., X. Zhang, Y. Shi, L. Xia, S. Pan, J. Wu, M. Han, and X. Zhao (2018). A review of data-driven approaches for prediction and classification of building energy consumption. *Renewable and Sustainable Energy Reviews 82*, 1027–1047.

Zeng, A., H. Ho, and Y. Yu (2020). Prediction of building electricity usage using Gaussian Process Regression. *Journal of Building Engineering 28*, 101054.

Zhao, H. and F. Magoulès (2012). A review on the prediction of building energy consumption. *Renewable and Sustainable Energy Reviews 16*(6), 3586–3592.