Institut für Intermodale
Transport- und
Logistiksysteme

Technische
Universität
Braunschweig

Simulation of mechatronic systems (Simulation mechatronischer Systeme)

Lecture Notes

Jürgen Pannek

April 9, 2026

Jürgen Pannek
Institute for Intermodal Transport and Logistic Systems
Hermann-Blenck-Str. 42
38519 Braunschweig

FOREWORD

In the summer term 2026, I am pleased to offer the module *Simulation of Mechatronic Systems (Simulation mechatronischer Systeme)* at Technische Universität Braunschweig. These lecture notes are designed as a living companion to the course: they will be continuously refined and expanded over the course of the semester. In addition, comments, remarks, and corrections will be incorporated on an ongoing basis to further improve their quality and usefulness.
The aim of the module is to classify simulation techniques from numerical mathematics and apply these to mechatronic case studies. After completing the module, the students shall be able to recall, categorize, apply, select and rate simulation methods to mechatronic use cases. Moreover, students shall be able to describe, explain, evaluate, analyze and assess simulation results. As such, students shall be capable to derive and apply automation procedures for deployment, simulation and testing of digital models.
To this end, we will tackle the subject areas

- dynamical systems,
- software development,
- simulation and testing,
- visualization, and
- process automation

within the lecture. To support students, we utilize university resources such as GitLab to assist students to create a professional software development tool chain and interact with cluster computing technology within the tutorial classes. The module itself is accredited with 5 credits.
An electronic version of this script can be found at

https://www.tu-braunschweig.de/en/itl/teaching/lecture-notes

Literature for further reading

- Dynamical systems
  - DEUFLHARD, P. ; BORNEMANN, F.: *Scientific computing with ordinary differential equations*. Springer, 2012
  - KHALIL, H.K.: *Nonlinear Systems*. Prentice Hall, 2002
  - BULIRSCH, R. ; STOER, J.: *Numerische Mathematik 2*. Springer, 2005
- Software development
  - CHACON, S. ; STRAUB, B.: *Pro Git*. Apress, 2022
  - FARLEY, D.: *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley Professional, 2022
  - IGLBERGER, K.: *C++ Software Design: Design Principles and Patterns for High-Quality Software*. O'Reilly Media, 2022
- Simulation and testing
  - GLÖCKLER, M.: *Simulation mechatronischer Systeme: Grundlagen und technische Anwendung*. Springer, 2014
  - ZIEGLER, B.P. ; MUZY, A. ; KOFMAN, E.: *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. Academic press, 2018
- Visualization
  - WILKE, C.O.: *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O'Reilly Media, 2019
- Process automation
  - COWELL, C. ; LOTZ, N. ; TIMERLAKE, C.: *Automating DevOps with GitLab CI/CD Pipelines*. Packt Publishing, 2023
  - LUNZE, J.: *Automatisierungstechnik*. 5th edition. DeGruyter, 2020

Contents

List of Tables

List of Figures

List of Definitions and Theorems

LIST OF ALGORITHMS

CHAPTER 1

INTENTION

> Theory provides the maps that turn an uncoordinated set of experiments or computer simulations into a cumulative exploration.
>
> David E. Goldberg[1]

This chapter situates the lecture within the broader engineering workflow of developing and analyzing mechatronic systems. The quotation above captures a central message of the course: simulation becomes scientifically and practically valuable only when it is embedded in a sound methodology for modeling, experiment design, interpretation of results, and technical decision-making. Without such a methodological framework, simulation easily degenerates into little more than number crunching.

Against this background, the present chapter introduces the context in which the subsequent material is to be understood. Section 1.1 places simulation within the development process for mechatronic and cyber-physical systems and discusses the main purposes for which simulations are used in practice. Section 1.2 then outlines the software-engineering perspective of the lecture by introducing DevOps, version control, and elementary system architecture concepts that are also addressed in the accompanying tutorials. Finally, Section 1.3 briefly explains why modern simulation workflows increasingly rely on containerization and cluster-based deployment using Docker and Kubernetes.

At the same time, this chapter serves as a bridge to the remainder of the lecture. While the following chapter introduces the fundamental terminology of systems, models, simulators, and simulations, the later chapters build on this foundation to discuss numerical solution methods, the design and analysis of simulation studies, and the testing and automation of digital models.

[1] *1953, American computer scientist

1.1 Aim of lecture

As outlined in the foreword, the lecture combines perspectives from dynamical systems, software development, simulation and testing, visualization, and process automation. The objective is therefore not merely to present isolated numerical methods, but to embed these methods into a coherent engineering workflow that spans from mathematical modeling to implementation, evaluation, and automated deployment. In this sense, the lecture follows the logic of modern mechatronic and cyber-physical system development as reflected, for example, in the V-model according to the VDI/VDE 2206 guideline [16], cf. Figure 1.1.

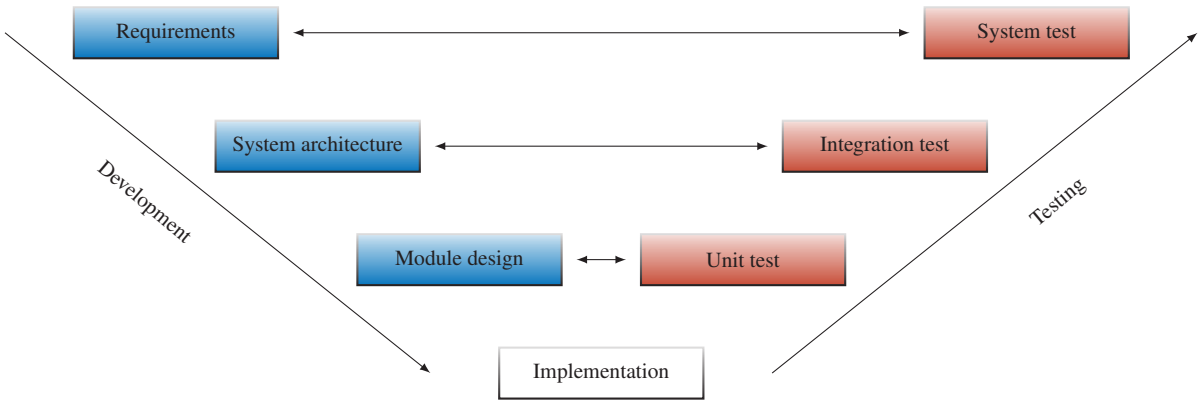

Figure 1.1: V model for system development

The V-model highlights that simulation is not an isolated activity. Rather, it supports several stages of the development process: it helps to analyze design alternatives, to verify expected system behavior, and to prepare testing and validation activities before expensive experiments on real hardware are carried out. At the same time, iterative refinement loops, which are often associated with agile development, remain essential in practical projects. For this reason, the tutorials accompanying the lecture use a contemporary toolchain in order to implement, integrate, test, and automate representative simulation examples.
From a methodological point of view, the role of simulation in engineering practice can be organized into three fundamental classes of application scenarios:

1. Understand a known scenario:
   In this case, the scenario is known and at least some real observations or measurements are available. Typical examples range from the driving behavior of a vehicle and the response of mechanical components to the behavior of software in a concrete operating situation. Here, simulation supports explanation and interpretation. The leading question is: "Why does something happen the way it happens".

2. Optimize a known scenario:
   uilding on the previous case, simulation can also be used to improve a given system with respect to prescribed criteria. Typical examples include timetable design for transport systems, performance studies for engines, material testing under varying operating conditions, or the design of software behavior before deployment. In such settings, real-world experiments are often expensive, time-consuming, or difficult to repeat systematically. Simulation therefore serves as a tool for design exploration and improvement. The leading question is: "How can something be utilized to improve given criteria".

3. Predict unknown scenario:
   A third application domain arises when simulation is used to anticipate behavior in situations for which direct experimentation is impossible, unsafe, or not yet feasible. Examples include climate projections, pollutant dispersion in urban environments, or the assessment of new materials and technical concepts. In such cases, simulation is not primarily used to reproduce a known reality, but to estimate plausible future behavior and to quantify uncertainty. The leading question is: "What can reasonably be expected to happen, and with which degree of confidence?".

This classification also clarifies the place of the lecture within the curriculum. The module accompanies the lecture *Modeling of Mechatronic Systems (Modellierung mechatronischer Systeme)*, in which models and modeling techniques are developed in more detail. The present lecture starts from the assumption that a model is available and focuses on how such models are implemented, simulated, analyzed, tested, and integrated into a professional engineering workflow. In the tutorial sessions, this perspective is deepened by means of a modern full-stack development environment and, where appropriate, by the use of middleware and cluster-based computing infrastructure.

1.2 DevOps, Toolchain and System Architecture

On the implementation side of simulation, contemporary engineering practice is strongly shaped by agile development principles and DevOps-oriented workflows. The central idea is to begin with a working prototype that provides core functionality and then to refine, extend, and stabilize this prototype in short development cycles. In this way, simulation software is not treated as a one-off artifact, but as an evolving technical product that must remain maintainable, testable, and deployable throughout its life cycle. A common representation of this process is shown in Figure 1.2.

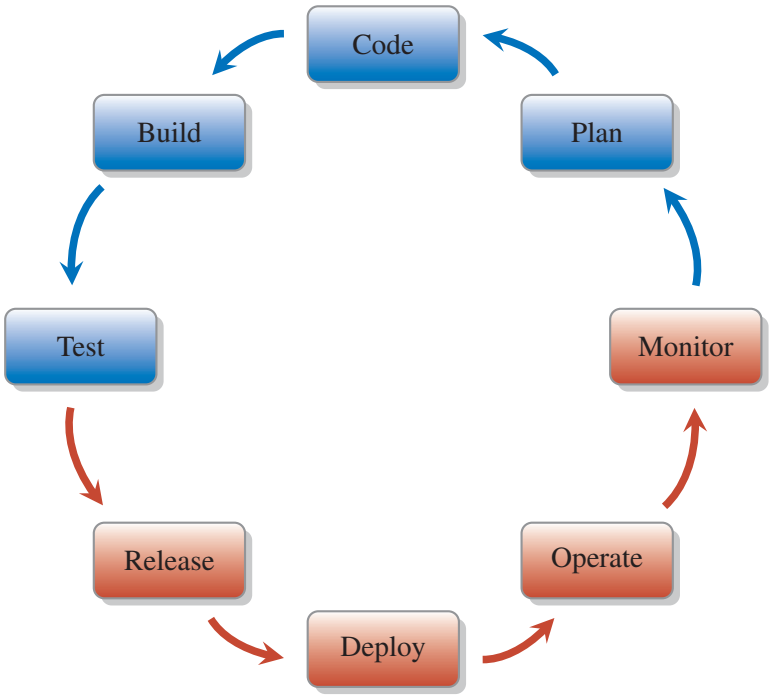

Figure 1.2: Agile development

Remark 1.1
Figure 1.2 shows the extended DevOps cycle, which also includes software that is already deployed and in operation. In early development phases, the steps operate *and* monitor *are often absent or play only a minor role.*

Because DevOps seeks to shorten feedback loops between development and operation, the process is inherently iterative and is often visualized as a continuous cycle or an infinity loop. The individual stages can be interpreted as follows:

- **Plan:** Tasks are collected, structured, prioritized, and scheduled. In agile settings, this is commonly done by means of tickets or user stories that describe the intended functionality, the motivation behind it, and suitable acceptance criteria.
- **Code:** The required functionality is implemented and reviewed. In collaborative projects, this includes code review, branch management, and the preparation of merge requests.
- **Build:** The software is transformed into an executable artifact. Depending on the project, this may include compilation, dependency resolution, packaging, and the automatic execution of preliminary checks.
- **Test:** Automated tests are used to verify correctness and to reduce the risk of introducing regressions. In the context of simulation software, this step is particularly important because

errors in numerical code often remain unnoticed unless they are checked systematically.

- **Release:** A validated software version is prepared for delivery. This creates a reproducible and traceable state that is suitable for downstream deployment.
- **Deploy:** The released version is transferred to the target environment, for example a server, a cluster, or a containerized runtime infrastructure.
- **Operate / configure infrastructure:** The software is executed and maintained in its target environment. This includes configuration, scalability, security, logging, and the reliable provision of required resources.
- **Monitor:** Runtime behavior, logs, and quality indicators are observed in order to identify faults, bottlenecks, or opportunities for improvement. The resulting feedback initiates the next planning cycle.

Within the tutorial sessions of this lecture, we use a GitLab-based toolchain that supports version control, issue tracking, continuous integration and delivery (CI/CD), automated testing, and automated deployment. In this way, students are introduced not only to simulation methods themselves, but also to the software-engineering environment in which such methods are developed and maintained in practice.
A key component of this environment is version control. Git has become the dominant version control system because it combines distributed collaboration, efficient branching and merging, and a robust basis for traceable software development. Platforms such as GitLab build upon Git and extend it towards a broader DevOps ecosystem. Figure 1.3 illustrates a typical branching structure.

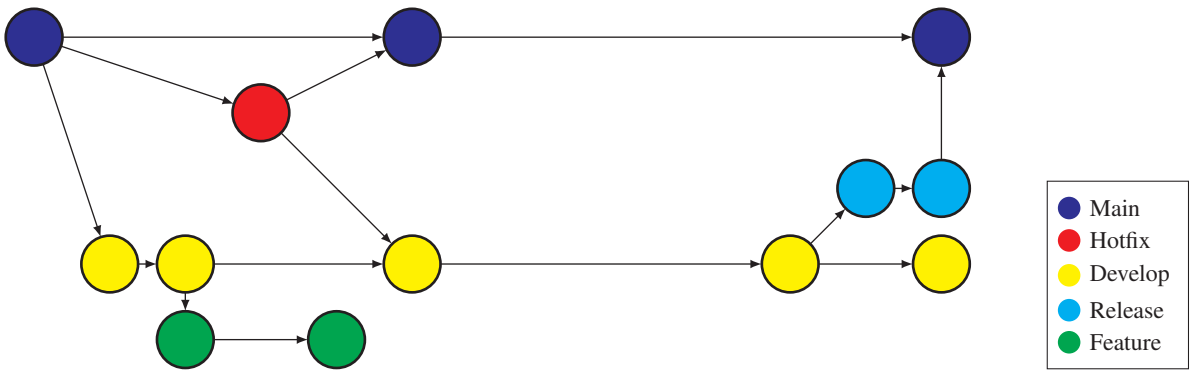

Figure 1.3: Version control including CI/CD pipeline

Within Figure 1.3, the respective colors indicate branches, which allow for

- parallel development and

- structuring the development, test and release process.

A developer may, for example, implement a new feature in a dedicated feature branch, validate it by tests, and then merge it into a development branch before it ultimately becomes part of a release or the main branch. Such a branching strategy makes the evolution of the software transparent and helps to separate experimental work, stabilization, and urgent bug fixing.
Closely related to version control is the internal structure of the software itself. In practice, simulation software and its associated testing code are typically organized into classes with clearly defined interfaces. Such interfaces improve modularity, enable reuse across different applications, and simplify long-term maintenance. Figure 1.4 shows a schematic UML class diagram.

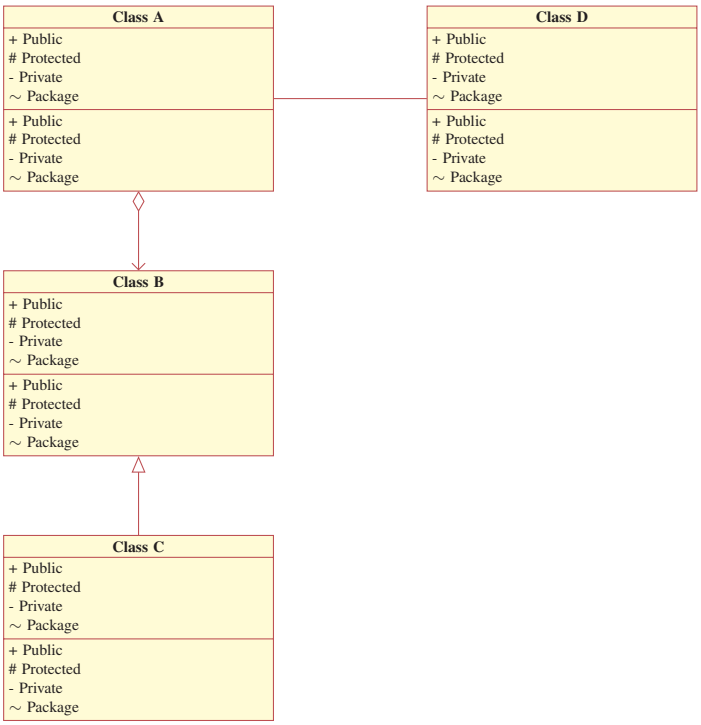

Figure 1.4: Sketch of UML diagram of classes

Each class in Figure 1.4 consists of three main components:

- Name
- Attributes
- Operations

Both attributes and operations may come in one of the four types

- public – can be accessed from the outside
- protected – can be accessed by associated classes
- private – cannot be accessed

- package – are linked to external

indicating their usability.
hese visibility levels structure how software components may interact and thereby support both robustness and maintainability.
Objects are then instantiated from classes. Multiple objects of the same class may exist simultaneously while sharing the same implementation but storing different data. The connecting lines and symbols in UML diagrams further indicate structural relations between classes. In the present context, the most relevant ones are:

- aggregation (indicated between Class A and Class B) – one Class A object deals with possible several Class B objects
- inheritance (indicated between Class B and Class C) – Class C has all properties of Class B and can add new ones or modify existing one
- association (indicated between Class A and Class D) – one Class A object and one Class D object can access their protected attributes/operations

The software perspective outlined above is inseparably linked to the underlying computing infrastructure. For this reason, the next section briefly introduces the deployment technologies that are relevant for modern simulation workflows.

1.3 Kubernetes and Docker

Once simulation software has been developed, it must be executed on actual computing hardware. For small educational examples or early prototypes, this can often be done directly on a local computer by compiling and running the code in a straightforward manner. For collaborative software development, reproducible execution environments, and especially for large-scale simulation studies, however, such an approach quickly reaches its limits.
To address these challenges, modern engineering workflows increasingly rely on technologies such as Docker and Kubernetes. In the lecture and the tutorial sessions, we will only touch the basic ideas behind these tools. Nevertheless, it is important to understand why their combination offers significant advantages over traditional and purely virtualized deployment strategies, as illustrated in Figure 1.5.
Docker is a platform for developing, shipping, and running applications in containers. The key idea is to package an application together with its runtime environment, libraries, and depen-

[2]Source: https://www.docker.com

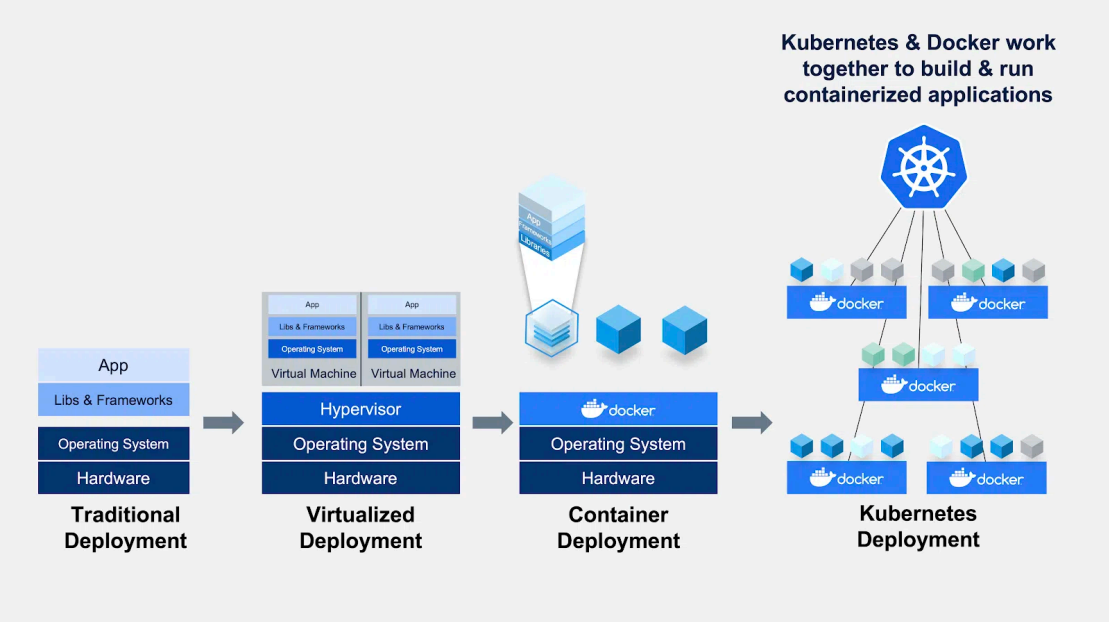

Figure 1.5: Traditional, virtualized and containerized deployment on distributed hardware[2]

dencies such that it can be executed in a portable and reproducible manner across different deployment systems. In this sense, containers help to make applications comparatively lightweight, self-contained, and easier to transfer between development, testing, and production environments. Kubernetes addresses a complementary problem. Rather than packaging a single application, it provides mechanisms for orchestrating and managing containerized applications across multiple computers. From the perspective of the user or developer, the individual machines become nodes of a larger cluster. Kubernetes then automates essential tasks such as deployment, scaling, restart strategies, service discovery, and workload distribution.
Used together, Docker and Kubernetes provide an infrastructure that is portable, scalable, and resilient. They separate application logic from the details of the underlying hardware and thereby support a development process in which simulation software can be reproduced reliably, deployed systematically, and executed efficiently even in distributed environments.

With this engineering context in place, the lecture can now turn to its conceptual core. The next chapter introduces the fundamental terminology of systems, models, simulators, and simulations, on which the numerical, analytical, and automation-oriented methods of the subsequent chapters will be built.

CHAPTER 2
BASICS FOR SYSTEMS AND SIMULATION

> Theory provides the maps that turn an uncoordinated set of experiments or computer simulations into a cumulative exploration.
>
> David E. Goldberg[1]

This chapter introduces the terminology and conceptual foundations that will be used throughout the remainder of the lecture. In particular, we make precise what kind of systems we consider and how simulation is related to them. To this end, Section 2.1 develops the basic notion of a system or process, starting from standard terminology and moving towards a mathematical description. Section 2.2 then introduces models as purpose-oriented representations of such systems. In Section 2.3, we define the notions of simulator, simulation, and assessment that will be used throughout the lecture. Finally, Section 2.4 presents a running example that will accompany us in the subsequent chapters.

2.1 Systems

n this lecture, we follow definitions, terminology and notation that are commonly used in the books by Lunze [14], Khalil [13], and Sontag [15].
We begin with the object that is actually of interest, namely the system or process under consideration. Although the term „system“ is used in many scientific and non-scientific contexts, it is often employed rather informally and on a very general level. In the literature [5], we find the following description of a system (translated from German):

[1] *1953, American computer scientist

> A system is a set of interrelated elements that are viewed as a whole in a particular context and considered as distinct from their environment.
>
> DIN IEC 60050-351 (2014)

Building on this description, a process is given as follows (translated from German):

> A process is the entirety of relations and interacting elements in a system through which matter, energy or information is transformed, transported or stored.
>
> DIN IEC 60050-351 (2014)

Since our aim is simulation, we require a more precise and mathematically usable notion. We therefore introduce the following definition:

Definition 2.1 (System and process).
Consider two sets $\mathcal{U}$ and $\mathcal{Y}$. Then a map $\Sigma : \mathcal{U} \to \mathcal{Y}$ is called a *system* and the application of this map to an input $\mathbf{u} \in \mathcal{U}$ to obtain an output $\mathbf{y} = \Sigma(\mathbf{u}) \in \mathcal{Y}$ is called a *process*.

In particular, the sets $\mathcal{U}$ and $\mathcal{Y}$ are called the input and output sets. An element $\mathbf{u} \in \mathcal{U}$ is called an input. Inputs act from the environment on the system and are not determined by the system itself or by its internal properties, cf. Figure 2.1.

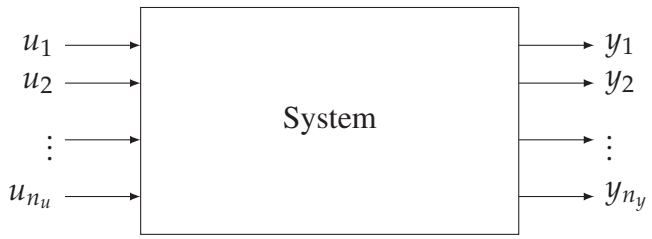

Figure 2.1: Term of a system

We distinguish between inputs, which are used to specifically manipulate (or control) the system, and inputs, which are not manipulated on purpose. We call the first ones *control or manipulation inputs*, and we refer to the second ones as *disturbance inputs*. An element from the output set $\mathbf{y} \in \mathcal{Y}$ is called an output. In contrast to an input, the output is generated by the system and influences the environment.

Remark 2.2
Note that in most cases not all measurable outputs are actually measured. Similarly, in many cases not all manipulable inputs are controlled.

To describe how a system evolves, we next introduce the concept of time:

Definition 2.3 (Time).
A *time set* $\mathcal{T}$ is a subgroup of $(\mathbb{R}, +)$.

Time allows us to describe the evolution of a system. To illustrate this point, consider the two electrical systems shown in Figure 2.2, namely an ideal resistor and an ideal capacitor.

Figure 2.2: Example of static and dynamic systems

The systems in Figure 2.2 are characterized by the input variable $I(t)$, the output variable $U(t)$, and time t. For the resistor R, the output is uniquely determined by the input at every time instant t, that is,

$$y(t) = U(t) = R \cdot I(t) = R \cdot u(t). \tag{2.1}$$

If the output depends only on the input at the same time instant, then the system is called *static*.

In contrast, the voltage $U(t)$ across the capacitor C at time instant t depends on the entire history $I(\tau)$ for $\tau \leq t$, that is,

$$y(t) = U(t) = \frac{1}{C} \int_{-\infty}^{t} I(\tau) d\tau = \frac{1}{C} \int_{-\infty}^{t} u(\tau) d\tau.$$

If we additionally know the voltage $U(t_0)$ at a time instant $t_0 \leq t$, then only the history $t_0 \leq$

$\tau \leq t$ of the current is required, i.e.

$$y(t) = U(t) = \frac{1}{C}\int_{-\infty}^{t} I(\tau)d\tau = \underbrace{\frac{1}{C}\int_{-\infty}^{t_0} I(\tau)d\tau}_{U(t_0)} + \frac{1}{C}\int_{t_0}^{t} I(\tau)d\tau = U(t_0) + \frac{1}{C}\int_{t_0}^{t} u(\tau)d\tau. \quad (2.2)$$

As can be seen from (2.2), the initial value $U(t_0)$ contains all information about the past history $\tau \leq t_0$. For this reason, $U(t_0)$ is naturally interpreted as the internal *state* of the capacitor at time instant t_0. If the output of a system depends not only on the current input but also on its history, then the system is called *dynamic*.

Remark 2.4
Note that by this definition the set of dynamic systems covers the set of static systems.

If for a system according to Figure 2.1 the outputs $y_1(t), \ldots, y_{n_y}(t)$ depend on the history of the inputs $u_1(\tau), \ldots, u_{n_u}(\tau)$ for $\tau \leq t$ only, then the system is called *causal*. As all technically feasible systems are causal, we will restrict ourselves to this case.

The discussion so far allows us to formulate the general notion of a state of a dynamical system:

Definition 2.5 (State).
Consider a system $f : \mathcal{U} \to \mathcal{Y}$. If the output $\mathbf{y}(t)$ uniquely depends on the history of inputs $\mathbf{u}(\tau)$ for $t_0 \leq \tau \leq t$ and some $\mathbf{x}(t_0)$, then the variable $\mathbf{x}(t)$ is called *state* of the system.

Link: For further details on how to design inputs/controls such that system properties regarding outputs can be generated, we refer to the lectures *Control Engineering 1 & 2*.

In contrast to the pure input/output description of a system from Definition 2.1, states allow us to look inside the system. In engineering practice, the input/output view is often referred to as a black-box description, whereas a state description corresponds to a white-box perspective. The states and their evolution over time allow us to analyze and interpret properties such as long-term behavior (for example convergence or stability) as well as short-term phenomena (for example transient behavior or overshoot).

Task 2.6
Which variable represents a state in case of induction?

Solution to Task 2.6: Current through the inductor

In order to understand and make use of a system or process, we require a model of it. Strictly speaking, the mathematical description introduced in Definition 2.1 is already a model. In engineering practice, however, the term usually refers to a deliberately reduced representation of a system or process that captures only those aspects that are relevant for the intended purpose.

2.2 Models and modeling

Abstractly speaking, a model represents exactly those properties or subsystems that are relevant for the intended use. This leads to the following definition:

Definition 2.7 (Model).
Consider a system Σ with input and output sets $\mathcal{U}$ and $\mathcal{Y}$. Then a map $\Sigma_M : \mathcal{U} \to \mathcal{Y}$ is called a *model* of the system Σ if $\Sigma_M \sim \Sigma$.

Whenever a model is used, deviations between model prediction and reality must be expected, both on short and on long time horizons, depending on the chosen level of abstraction. Since a model never captures all aspects of reality, the system or process is conceptually divided into two parts,

- the model, which describes what we are interested in, and
- the remainder, which contains everything else.

Since we cannot tell anything about the remainder (as it is not modeled), interactions between model and remainder can only be interpreted as disturbances.

Remark 2.8
Note that, for a system, a disturbance refers to unknown influences originating for instance from the environment, whereas for a model a disturbance may also represent influences that are simply not included in the chosen level of abstraction.

Link: For further details on modeling, we refer to the lecture *Modeling of mechatronic systems* and for process modeling, identification and handling disturbances, we refer to the lecture *Systemics*.

For the models considered in this lecture, we assume that the following principles are satisfied:

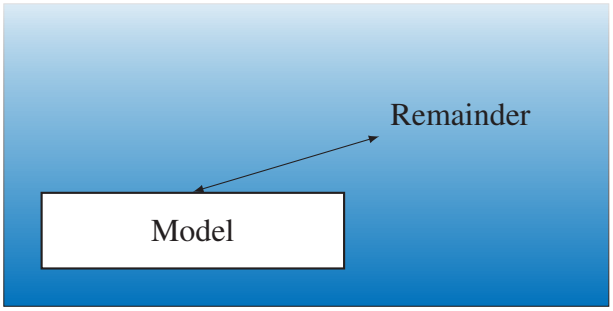

Figure 2.3: Model and remainder

Assumption 2.9 (Principles of modeling)
During the modeling process, six principles need to be met:

1. Principle of Correctness: A model needs to present the facts correctly regarding structure and dynamics (semantics). Specific notation rules have to be considered (syntax).
2. Principle of Relevance: All relevant items have to be modeled. Non-relevant items have to be left out, i.e. the value of the model doesn't decline if these items are removed.
3. Principle of Cost vs. Benefit: The amount of effort to gather the data and produce the model must be balanced against the expected benefit.
4. Principle of Clarity: The model must be understandable and usable. The required knowledge for understanding the model should be as low as possible.
5. Principle of Comparability: A common approach to modeling ensures future comparability of different models that have been created independently from each other.
6. Principle of Systematic Structure: Models produced in different views should be capable of integration. Interfaces need to be designed to ensure interoperability.

At this point, an important observation arises: in practice, the modeler and the model user are often different persons or groups with different objectives and perspectives. As a consequence, a model that is excellent from the modeler's point of view need not be the most useful one for the user. For example, a highly detailed model may reproduce reality very accurately, yet still be too complex for efficient simulation or interpretation. Hence, modeling must always be aligned with its intended use, and the quality of a model is ultimately determined by the degree to which it satisfies the requirements of its user („fitness for use“).

2.3 Simulation

Within this lecture, our intended use is simulation where we are going to evaluate a model over time. Utilizing time allows us to consider not only instances of inputs $\mathbf{u}$ but entire sequences $\mathbf{u}(\cdot)$ where the input is no longer a single value but a map $\mathbf{u} : \mathcal{T} \to \mathcal{U}$. To evaluate such sequences, we require the notion of a simulator to connect points over time:

Definition 2.10 (Simulator).
Consider a model Σ_M, a state $\mathbf{x}_0$ at time t_0 and a terminal time t_f to be given. Suppose that t_0 and t_f are the lower and upper bounds of a set $\mathbf{T} \subset \mathcal{T}$ and an input sequence $\mathbf{u}(t)$, $t \in \mathbf{T}$ to be given. We call a method $\Phi : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}$ a *simulator* if

$$\mathbf{x}(t) = \Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) \tag{2.3}$$

holds for any $t \in \mathbf{T}$.

More intuitively speaking, a simulator evaluates the model — and therefore depends on it — for a given initial value $(t_0, \mathbf{x}_0)$, an input trajectory $\mathbf{u}(\cdot)$, and parameters p, and returns the corresponding state at any future time instant $t \in \mathbf{T}$.

Remark 2.11
A simulator is generally not unique, since different computational approaches may be used. If the model is given by a system of differential equations, then an analytical solution, numerical integrators, or even Koopman-based approaches may serve as simulators.

Task 2.12
Consider the differential equation

$$\dot{\mathbf{x}}(t) = \mathbf{x}(t). \tag{2.4}$$

Define an analytical and a numerical simulator.

Solution to Task 2.12: The analytical solution of (2.4) is given by

$$\mathbf{x}(t) = C \cdot \exp(t)$$

which can be identified using initial values $\mathbf{x}(t_0) = \mathbf{x}_0$ to

$$\mathbf{x}(t) = \mathbf{x}_0 \cdot \exp(t - t_0).$$

Hence, we obtain

$$\Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot)) := \mathbf{x}_0 \cdot \exp(t - t_0).$$

A numerical solution of (2.4) can be derived using the Euler method

$$\mathbf{x}(t) = \mathbf{x}(t_0) + \dot{\mathbf{x}}(t_0) \cdot (t - t_0)$$

which again can be applied for given initial values $\mathbf{x}(t_0) = \mathbf{x}_0$. In this case, we obtain

$$\Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot)) := \mathbf{x}_0 + \dot{\mathbf{x}}(t_0) \cdot (t - t_0)\,.$$

Both are simulators but will reveal different results. For this reason, it is important to select the right solution method for the task at hand.

In contrast to a simulator, which evaluates a single configuration, a simulation considers entire sets of initial values, time intervals, input trajectories, and parameters.

Definition 2.13 (Simulation).
Consider a model Σ_M and a simulator Φ to be given. Suppose sets of initial values $\mathcal{X}_0 \subset \mathcal{X}$, initial times $\mathcal{T}_0$ and terminal times $\mathcal{T}_f$ as well as input sequences sets $\mathcal{U}^{\mathbf{T}} := \{\mathbf{u}(t) \mid t \in \mathbf{T}\}$ and parameters $\mathcal{P}$ to be given. Then we call a map $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ given by

$$\Theta(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) = \left\{ \Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) \mid (t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) \in \mathcal{T}_f \times \mathcal{T}_0 \times \mathcal{X}_0 \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \right\} \tag{2.5}$$

a *simulation*.

Based on the obtained simulation results, an assessment is carried out using so-called key performance indicators, which are characterized on a high level in ISO 22400 [10]:

> A key performance criterion is a function that measures defined information retrieved from the system/process or model against a standard.
>
> ISO 22400 (2014)

Similar to the high level definition of a system/process, in practice we require a more rigorous definition to actually calculate:

Definition 2.14 (Cost function).
We call a key performance criterion given by a function $\ell : \mathcal{X} \to \mathbb{R}_0^+$ a *cost function*.

Task 2.15
Design a cost function to compute the square deviation from a target $\mathbf{x}^{ref}$.

Solution to Task 2.15: Given the target value $\mathbf{x}^{\text{ref}}$, the standard squared deviation is given by

$$\ell(\mathbf{x}) := \left(\mathbf{x} - \mathbf{x}^{\text{ref}}\right)^2.$$

In that case the costs are minimal if the state of the model/system approaches the wanted target value $\mathbf{x}^{\text{ref}}$.

The value of a key performance criterion provides only a snapshot, namely an evaluation at a single time instant $t \in \mathcal{T}$. To assess overall performance, the criterion must be evaluated over the operating period of the system. Since this amounts to defining a function of a function, the resulting object is called a functional.

Definition 2.16 (Cost functional and assessment).
Consider a key performance criterion $\ell : \mathcal{X} \to \mathbb{R}_0^+$. Then we call

$$J(\Theta(\Sigma_M, \Phi)(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p)) := \int_{t_0}^{t} \ell(\mathbf{x}(\tau))d\tau \tag{2.6}$$

cost functional for the tuple $(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p)$. The combination of cost functionals for all tuples of the simulation $J(\Theta(\Sigma_M, \Phi))$ is called assessment.

Task 2.17
Define the mean of a simulation.

Solution to Task 2.17: An assessment using the mean over all simulations in the set $\mathcal{S} := \mathcal{T}_f \times \mathcal{T}_0 \times \mathcal{X}_0 \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ is given by

$$J(\Theta(\Sigma_M, \Phi) := \sum_{(t,t_0,\mathbf{x}_0,\mathbf{u}(\cdot))\in\mathcal{S}} \frac{1}{\sharp\mathcal{S}} J(\Theta(\Sigma_M, \Phi)(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p))$$

which corresponds to evaluating all possible combinations contained in the set $\mathcal{S}$.

Remark 2.18
ote that, in addition to the mean, other statistical quantities such as covariance, quantiles, or statistical tests may also be used for assessment.

Having introduced the basic terminology required for simulation, we next turn to an example that illustrates how these concepts appear in a concrete mechatronic setting.

2.4 Example

In the following, we introduce the main example that will accompany us throughout the lecture. To this end, we consider a quarter-car test bench as depicted in Figure 2.4. The system is sufficiently rich to exhibit nontrivial dynamics, while still remaining simple enough for its parameters and physical interpretation to be understood intuitively.
The test bench is excited by the input $u(\cdot)$, which represents the road profile. Since this input trajectory can in principle be chosen arbitrarily, the class of admissible excitations is extremely large. Consequently, one does not simulate all possible input sequences, which would be impossible, but rather representative scenarios or specifically designed test profiles.
For this test bench, we restrict attention to vertical motion only. The chassis is represented by the mass m_1 at position y_1, while the suspension between chassis and wheel is modeled by the spring–damper pair with parameters s_1 and d_1. Likewise, the wheel and axle assembly is represented by the mass m_2 at position y_2, and the tire-ground interaction is modeled by a second spring–damper pair with parameters s_2 and d_2. Finally, road irregularities are described by the road height function $u(t)$.
To derive the equations of motion using d'Alembert's principle, we first introduce the individual forces

$$m_i \dot{v}_i^m(t) = F_i^m(t), \quad d_i v_i^d(t) = F_i^d(t), \quad s_i y_i^s(t) = F_i^s(t)$$

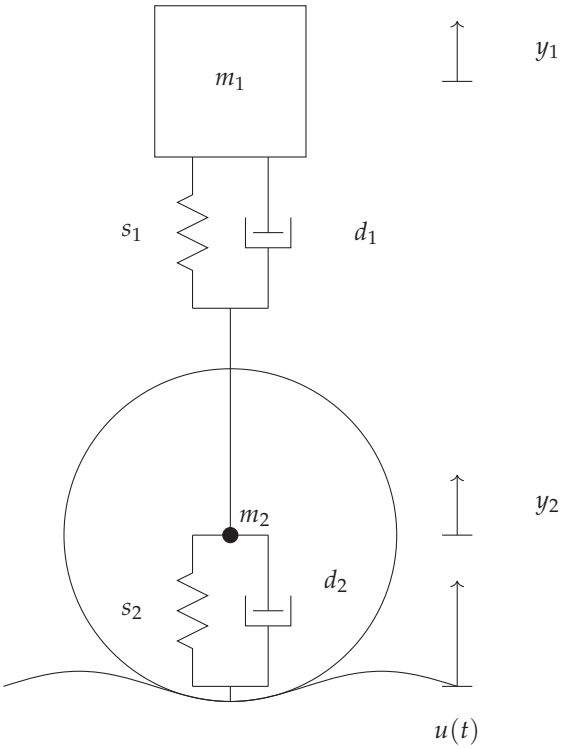

Figure 2.4: Schematic drawing of a quarter car test bench

for $i = 1, 2$ where we used

$$
\begin{aligned}
v_1^m(t) &= \dot{y}_1(t), & v_1^d(t) &= v_1(t) - v_2(t), & y_1^s(t) &= y_1(t) - y_2(t),\\
v_2^m(t) &= \dot{y}_2(t), & y_2^d(t) &= v_2(t) - \dot{u}(t), & y_2^s(t) &= y_2(t) - u(t).
\end{aligned}
$$

We can now combine these relations to obtain the force balance at both masses. For this purpose, the directions of the forces must be treated carefully. At m_1, the force F_1^m points upward. Since m_1 is attached to the upper ends of the spring and damper, the forces F_1^d and F_1^s also act upward. Hence, for m_1 we obtain

$$F_1^m + F_1^d + F_1^s = 0.$$

At m_2, the forces F_1^d and F_1^s act downward, whereas all remaining forces act upward. Hence, we obtain

$$F_2^m - F_1^d - F_1^s + F_2^d + F_2^s = 0.$$

Combined, we have

$$\begin{aligned} 0 &= F_1^m + F_1^d + F_1^s \\ &= m_1 \dot{v}_1^m(t) + d_1 v_1^d(t) + s_1 y_1^s(t) \\ &= m_1 \ddot{y}_1(t) + d_1(\dot{y}_1(t) - \dot{y}_2(t)) + s_1(y_1(t) - y_2(t)) \end{aligned}$$

and

$$\begin{aligned} 0 &= F_2^m - F_1^d - F_1^s + F_2^d + F_2^s \\ &= m_2 \dot{v}_2^m(t) - d_1 v_1^d(t) - s_1 y_1^s(t) + d_2 v_2^d(t) + s_2 y_2^s(t) \\ &= m_2 \ddot{y}_2(t) - d_1(\dot{y}_1(t) - \dot{y}_2(t)) - s_1(y_1(t) - y_2(t)) + d_2(\dot{y}_2(t) - \dot{u}(t)) + s_2(y_2(t) - u(t)). \end{aligned}$$

These equations form a coupled system of two second-order differential equations, which can equivalently be rewritten as a system of four first-order differential equations.

Remark 2.19
Note that also a Lagrangian or Hamiltonian approach to derive the equations of motion can be used, yet the outcome will always be equivalent.

The resulting model is still comparatively simple and can be adjusted by varying the spring and damper parameters. Depending on the required level of fidelity, this may already provide sufficiently accurate results. At the same time, the model is rich enough that, in practice, one would typically resort to a numerical solver for simulation.

In the following chapter, we turn to description and solution methods for such models and show how these lead to concrete simulators.

CHAPTER 3

SOLUTION METHODS FOR DYNAMICAL SYSTEMS

This chapter turns the modeling concepts introduced in the previous chapter into computational procedures for actually generating trajectories. In contemporary engineering and scientific practice, dynamical systems arise in almost every application domain, and their mathematical descriptions may differ substantially with respect to time, space, and amplitude. Figure 3.1 provides a rough overview of these characteristics.

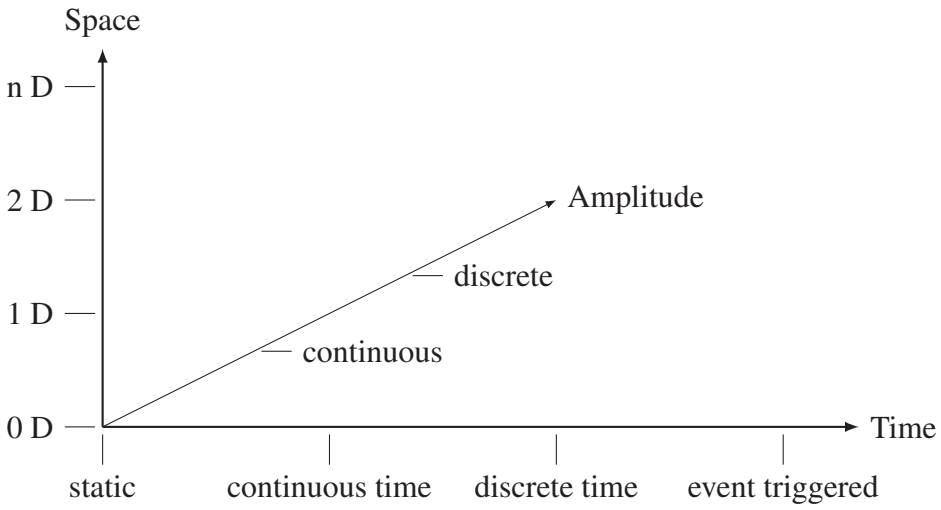

Figure 3.1: Dimensions of model characteristics

Within this lecture, we concentrate on models described by differential equations. Such models arise naturally in mechatronic applications, for example from d'Alembert's principle or from Lagrangian and Hamiltonian formulations. Section 3.1 therefore revisits essential concepts from differential equations before Sections 3.2–3.4 introduce numerical methods for solving the asso-

ciated initial value problems. Finally, Section 3.5 discusses how these methods must be adapted when simulations are embedded into software and deployment settings. In terms of the V-model, this chapter therefore addresses the computational core located at the deepest point of implementation, highlighted in Figure 3.2.

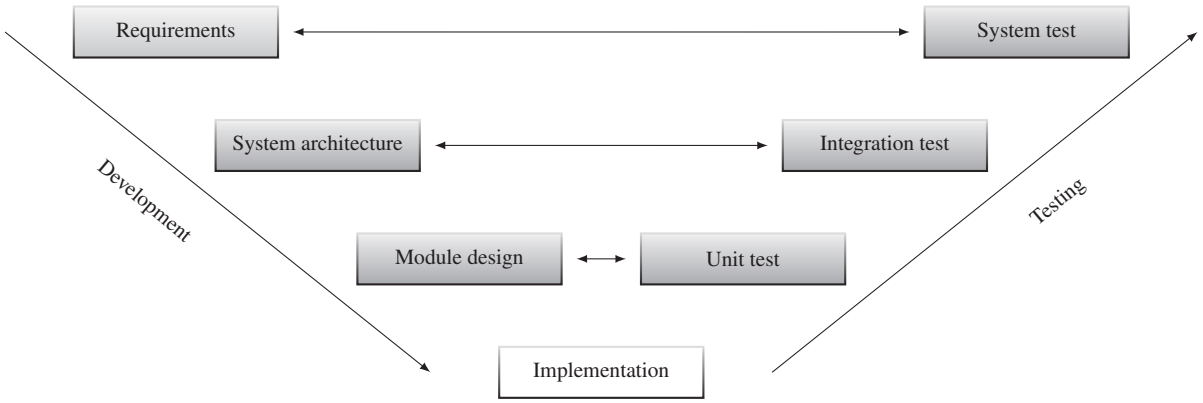

Figure 3.2: V model for system development

For an in-depth understanding, we refer to the books of Deuflhard [4] and Stoer [1].

3.1 Differential equations

As indicated by Figure 3.1, dynamical systems may differ with respect to structural properties such as time, space, and amplitude:

- Regarding time, we start off with static models, which are characterized by the fact that inputs, outputs and measurements of the system are available. In contrast to that, continuous time models exhibit data streams being received continuously. Discrete time models differ from that by the availability of data, which is received at certain, not necessarily equidistant time instances. Last, event triggered models require issues to trigger receiving data.

- Regarding space, models may vary from a simple connection to complex systems.

- Regarding amplitude, models may differ regarding continuous spaces like e.g. mass and discrete spaces such as gear shifts.

In this section, we focus on models Σ_M described by first-order ordinary differential equations in continuous time, that is, with $\mathcal{T} = \mathbb{R}$, high-dimensional state space, and real-valued amplitudes, i.e. $\mathcal{X} = \mathbb{R}^{n_x}$. Such an ordinary differential equation relates the derivative of a function $\mathbf{x} : \mathcal{T} \to \mathcal{X}$ to time and to the function itself. More formally:

Definition 3.1 (Ordinary Differential Equation).
An *ordinary differential equation* in $\mathcal{X} = \mathbb{R}^{n_x}$, $n_x \in \mathbb{N}$, is given by the dynamic

$$\dot{\mathbf{x}}(t) := \frac{d}{dt}\mathbf{x}(t) = f(t, \mathbf{x}(t)) \tag{3.1}$$

where $f : \mathbb{D} \to \mathcal{X}$ is a continuous function with open subset of $\mathbb{D} \subset \mathcal{T} \times \mathcal{X} = \mathbb{R} \times \mathbb{R}^{n_x}$.

Task 3.2
Models derived via the Lagrangian approach typically form ordinary differential equation systems of second order

$$\ddot{\tilde{\mathbf{x}}}(t) = \tilde{f}(t, \tilde{\mathbf{x}}(t)). \tag{3.2}$$

Reform the latter in the form (3.1).

Solution to Task 3.2: ystem (3.2) can be rewritten as a first-order system by introducing the state vector $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)^\top$ with $\mathbf{x}_1 = \tilde{\mathbf{x}}$ and $\mathbf{x}_2 = \dot{\tilde{\mathbf{x}}}$. This yields

$$\dot{\mathbf{x}}(t) = \begin{pmatrix} \dot{\mathbf{x}}_1(t) \\ \dot{\mathbf{x}}_2(t) \end{pmatrix} = \begin{pmatrix} \mathbf{x}_2 \\ \tilde{f}(t, \mathbf{x}_2(t)) \end{pmatrix} = f(t, \mathbf{x}(t))$$

where $\mathbf{x}_2 = \dot{\tilde{\mathbf{x}}}$.

A solution of (3.1) is a continuously differentiable function $\mathbf{x} : \mathcal{T} \to \mathcal{X}$ that satisfies the differential equation. Throughout the lecture notes, we adopt the following conventions:

- The independent variable t is referred to as time, although other interpretations are possible.
- For the time derivative $\frac{d}{dt}\mathbf{x}(t)$ we use the abbreviation $\dot{\mathbf{x}}(t)$.
- The function $\mathbf{x}(t)$ is called *solution* at time t and the entirety $\mathbf{x}(\cdot)$ is called *trajectory*.
- If the function f is independent of t, i.e. $\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$, then the differential equation is called *autonomous*.

An ordinary differential equation typically possesses infinitely many solutions, cf. Task 2.12. To single out one particular trajectory, an additional constraint is required, namely an initial value condition. Combined with the differential equation (3.1), this yields the so-called initial value problem:

Definition 3.3 (Initial Value Problem).
Consider a function $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ together with values $t_0 \in \mathcal{T}$ and $\mathbf{x}_0 \in \mathcal{X}$ to be given. Then the *initial value problem* is to find the solution satisfying the differential equation

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}(t)) \tag{3.1}$$

and the initial value condition

$$\mathbf{x}(t_0) = \mathbf{x}_0. \tag{3.3}$$

The time $t_0 \in \mathcal{T}$ is called the *initial time*, and the value $\mathbf{x}_0 \in \mathcal{X}$ is called the *initial value*. Together, the pair $(t_0, \mathbf{x}_0)$ and the constraint (3.3) specify the *initial conditions*.

Remark 3.4
A continuously differentiable function $\mathbf{x} : \mathbb{D} \to \mathcal{X}$ *solves the initial value problem* (3.1), (3.3) *for some* $t_0 \in \mathcal{T}$ *and* $\mathbf{x}_0 \in \mathcal{X}$ *if and only if for each* $t \in \mathcal{T}$ *the integral equation*

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t} f(\tau, \mathbf{x}(\tau))\, d\tau \tag{3.4}$$

holds. This follows directly by integrating (3.1) *with respect to time or, conversely, by differentiating* (3.4) *with respect to* t *and applying the fundamental theorem of calculus. In particular, continuity of* $\mathbf{x}(t)$ *on the right-hand side of* (3.4) *implies continuous differentiability of the resulting expression. Hence, every continuous function* $\mathbf{x}(t)$ *satisfying* (3.4) *is automatically continuously differentiable.*

Under suitable assumptions, existence and uniqueness of a solution to the initial value problem from Definition 3.3 can be established. A standard sufficient assumption is Lipschitz continuity in the state argument:

Definition 3.5 (Lipschitz Condition).
Consider a function $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. Then f is called *Lipschitz* in its second argument, if for each compact set $K \subset \mathcal{T} \times \mathcal{X}$ there exists a constant $L > 0$ and

$$\|f(t, x) - f(t, y)\| \leq L\|x - y\| \tag{3.5}$$

holds for all $t \in \mathcal{T}$ and all $x, y \in \mathcal{X}$ with $(t, x), (t, y) \in K$.

Task 3.6
Show that any function $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ is Lipschitz if it is continuously differentiable in its second argument.

Solution to Task 3.6: Since f is continuously differentiable, we have that there exists $K \in \mathbb{R}_0^+$ such that

$$\lim_{x \to y} \frac{\|f(t,x) - f(t,y)\|}{\|x - y\|} \leq K$$

holds. Now we can set $L := K$ and multiply both sides by $\|x - y\|$ showing the assertion.

Using this property, we can show the following:

Theorem 3.7 (Existence and Uniqueness).
Consider a differential equation (3.1) *with $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. Moreover, f is considered to be continuous and Lipschitz continuous in the second argument. Then for each initial condition $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$, there exists a unique solution $\mathbf{x}(t; t_0, \mathbf{x}_0)$ of the initial value problem* (3.1), (3.3). *This solution is defined for all t from an open maximal interval of existence $I_{t_0,\mathbf{x}_0}$ with $t_0 \in I_{t_0,\mathbf{x}_0}$.*

Note that by Task 3.6, the following holds:

Corollary 3.8 (Simplified Existence and Uniqueness).
Consider a differential equation (3.1) *with $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. If f is continuously differentiable in its second argument, then the assertion of Theorem 3.7 holds.*

It is useful to distinguish between the *flow* induced by the dynamics and an individual *trajectory*. The flow describes the local direction field generated by the dynamics, whereas a trajectory is associated with a specific initial value and input signal. Figure 3.3 illustrates this distinction: the coloring indicates the intensity of the flow, the arrows indicate its direction, and the highlighted trajectory follows this field for one particular initialization.
Note that at the boundary of the interval of existence $I_{t_0,\mathbf{x}_0}$ the solution ceases to exist. If the interval is bounded, then there are two possible reasons for that: For one, the solution may diverge, or secondly the solution converges to a boundary point of $\mathcal{T} \times \mathcal{X}$. In the remainder of this script, we will always assume that the assumptions of Theorem 3.7 are met without explicitly stating it.

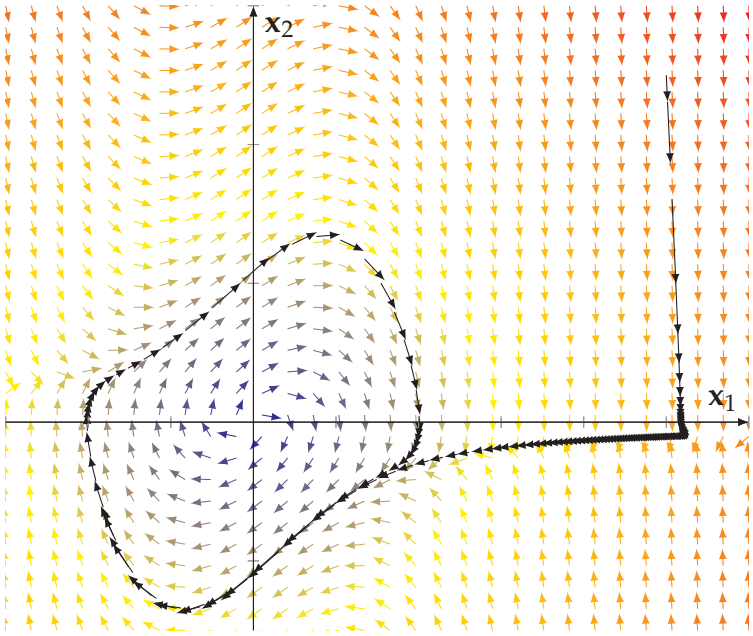

Figure 3.3: Sketch of a dynamic flow and a trajectory

Remark 3.9

1. A consequence of Theorem 3.7 is the so-called cocycle property. This property states that for $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$ *and two time instances* $t_1, t \in \mathbb{R}$*, we have*

$$\mathbf{x}(t; t_0, \mathbf{x}_0) = \mathbf{x}(t; t_1, \mathbf{x}_1) \tag{3.6}$$

with $\mathbf{x}_1 = \mathbf{x}(t_1; t_0, \mathbf{x}_0)$ *given that all terms are defined according to Theorem 3.7.*

2. Another consequence is that two solutions cannot intersect, as they would have to coincide for all times.

3. Some ordinary differential equations can be solved analytically via various methods. In general, this is not true and numerical methods must be used for this purpose. Yet, one typically not only applies numerical methods, but tries to show certain properties of the solution analytically.

Table 3.1: Advantages and disadvantages of differential equations

Advantage	Disadvantage
✓ Allows to model dynamics	✗ May require solvers
✓ Provides unique solutions	✗ Requires identification

Based on ordinary differential equations and the initial value problem, we next introduce numer-

ical methods to compute a solution of the initial value problem. These methods can later be used as simulators in the sense of Definition 2.10 in simulations.

Remark 3.10
Besides numerical methods, analytical solution techniques are also available. Among the best known are separation of variables (sometimes also referred to as the Fourier method), as well as direct integration, substitution, and variation of parameters. These approaches lie outside the scope of this lecture. Although they can provide exact solutions, they are in general not broadly applicable to the classes of problems relevant here.

In the following, we will concentrate on a particular class of numerical solution methods, which is already indicated in the class diagram sketched in Figure 3.4.

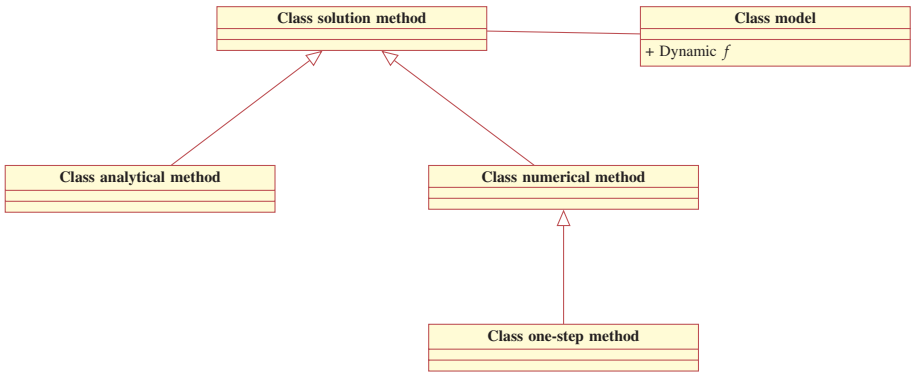

Figure 3.4: Sketch of UML diagram of class solution method

We emphasize that a solution method is conceptually separate from the model itself. Although not every method is suitable for every class of models, numerical methods are designed to approximate solutions for broad classes of differential equations rather than for one single concrete system. This separation between model and solution method will become particularly important in Chapter 4, where we turn to simulation workflows and the processing of results.

3.2 Explicit methods

The most prominent and also the simplest class of numerical methods are one-step methods. Their defining feature is that each new approximation is computed solely from the approximation obtained in the previous step. Multistep methods, by contrast, also use several previously computed values. Both classes typically operate on fixed step sizes, that is, on sampled time instants. Using Definition 2.3, we obtain:

Definition 3.11 (Sampled time).
Consider a time set $\mathcal{T} = \mathbb{R}$, a base time $t_0 \in \mathcal{T}$ and a number $T \in \mathbb{R}$. Then we call the set

$$\mathbb{T} := \{t_i \in \mathcal{T} \mid t_i = t_0 + i \cdot T,\ i \in \mathbb{Z}\} \tag{3.7}$$

sampled time set or *time grid*. Moreover, we call T *sampling period*.

Accordingly, the basic idea of a one-step method is to compute the approximation at one time instant from the approximation at the previous time instant. This leads to the following definition:

Definition 3.12 (One-step method).
Suppose a map $\Psi : \mathbb{R} \times \mathcal{X} \to \mathcal{X}$ and a parameter $h \in \mathbb{R}$ to be given. Then we call

$$\mathbf{x}(t_{i+1}) = \Psi(h, \mathbf{x}(t_i)) \tag{3.8}$$

a *one-step method* and h *step size*.

Technically, a one-step method describes only a single transition from t_i to t_{i+1} and not the computation of an entire trajectory. A numerical solution method is therefore the time-extended version of this concept:

Definition 3.13 (Numerical solution method).
Consider a map $\Psi : \mathbb{R} \times \mathcal{X} \to \mathcal{X}$, initial conditions $(t_0, \mathbf{x}(t_0)) \in \mathbb{R} \times \mathcal{X}$ and a terminal time $\overline{T} \in \mathcal{T}$, $\overline{T} > t_0$ to be given. Then we call a map $\Psi : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ defining

$$\mathbf{x}(\overline{T}) = \Psi(\overline{T}, t_0, \mathbf{x}(t_0)) \tag{3.9}$$

a *numerical solution method*.

Remark 3.14
Note that in many cases step size and sampling period are identical, i.e. $h = T$.

Example 3.15
Given a differential equation (3.1)*, the simplest one-step methods are the so-called Euler method*

$$\Psi(h, \mathbf{x}(t_i)) = \mathbf{x}(t_i) + h \cdot f(t_i, \mathbf{x}(t_i)),$$

and the Heun method

$$\Psi(h, \mathbf{x}(t_i)) = \mathbf{x}(t_i) + \frac{h}{2} \cdot \left(f(t_i, \mathbf{x}(t_i)) + f\left(t_i + h, \mathbf{x}(t_i) + h \cdot f(t_i, \mathbf{x}(t_i))\right)\right).$$

To apply any numerical approximation method for differential equations, we must make sure that the approximation error stays bounded with respect to the chosen sampling period h. To this end, we want the approximation to converge to the true solution if the sampling period is reduced, i.e. $h \to 0$. To ensure convergence, two properties are required:

Definition 3.16 (Consistency).
Consider a one-step method $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. We call the method *consistent* if there exist $C, p \in \mathbb{R}_0^+$ such that

$$\|\Psi(h, \mathbf{x}_0) - \mathbf{x}(t_0 + h; t_0, \mathbf{x}_0)\| \leq C \cdot h^{p+1} \tag{3.10}$$

holds for all initial values $t_0 \in \mathcal{T}$, $\mathbf{x}_0 \in \mathcal{X}$ and all at least p times continuously differentiable models $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and all $0 < h \leq h^\star$. We call p the *order of consistency*.

Task 3.17
Assess whether consistency is a local or global property with respect to a trajectory.

Solution to Task 3.17: By definition, consistency means that the one-step method is bounded in each single step. For this reason, it represents a local error and therefore a local property.

In the autonomous case, that is, when the dynamics f are independent of time, consistency can be checked by means of the following criterion:

Lemma 3.18 (Consistency check).
Consider a one-step method $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and suppose the initial value problem to be autonomous. Then the method is consistent if

$$\lim_{h \to 0} \left\| \frac{\Psi(h, \mathbf{x}) - \mathbf{x}}{h} - f(\mathbf{x}) \right\| = 0. \tag{3.11}$$

Task 3.19
Show that the Euler method is consistent.

Solution to Task 3.19: We directly obtain

$$\left\| \frac{\Psi(h,\mathbf{x}) - \mathbf{x}}{h} - f(\mathbf{x}) \right\| = \|f(\mathbf{x}) - f(\mathbf{x})\| = 0.$$

Having dealt with the single step error, we have to include that any one-step method by definition will use the possibly error prone value to continue calculating. To cope with this source, we define the following:

Definition 3.20 (Stability).
Suppose a one-step method $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and a constant $M > 0$ to be given. Then we call Ψ to be stable if

$$\|\Psi(h,\mathbf{x}_1) - \Psi(h,\mathbf{x}_2)\| \leq (1 + h \cdot M) \cdot \|\mathbf{x}_1 - \mathbf{x}_2\| \tag{3.12}$$

holds for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}$ and all $0 < h \leq h^\star$.

Task 3.21
Show that the Euler method is stable.

Solution to Task 3.21: Similar to consistency, we directly obtain

$$\begin{aligned} \|\Psi(h,\mathbf{x}_1) - \Psi(h,\mathbf{x}_2)\| &= \|\mathbf{x}_1 + h \cdot f(\cdot,\mathbf{x}_1) - \mathbf{x}_2 - h \cdot f(\cdot,\mathbf{x}_2)\| \\ &= \|\mathbf{x}_1 - \mathbf{x}_2\| + h \cdot \|f(\cdot,\mathbf{x}_1) - f(\cdot,\mathbf{x}_2)\| \leq (1 + h \cdot L) \cdot \|\mathbf{x}_1 - \mathbf{x}_2\| \end{aligned}$$

where L is the Lipschitz constant of f and we get $M = L$.

Now, these two properties can be combined to obtain convergence of the method:

Theorem 3.22 (Convergence).
Suppose a one-step method $\Psi : \mathbb{R} \times \mathcal{X} \to \mathcal{X}$ of order p, which is consistent and stable and let $(t_0, \mathbf{x}_0)$ be given initial conditions. Then for each $\overline{h} \in \mathbb{R}$ there exists a constant $K(\overline{h}) > 0$ such that

$$\left\| \mathbf{x}(t_i) - \Psi^i(h, \mathbf{x}_0) \right\| \leq K(\overline{h}) \cdot h^p \tag{3.13}$$

holds for all $i \in \mathbb{N}$ with $i \cdot h \leq \overline{h}$ and all at least p times continuously differentiable models $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$.

Convergence can be seen as the global error of a one-step method. What we can take from Theorem 3.22 together with Lemma 3.18 is that Lipschitz plus consistency (of order p) gives us convergence (of order $p - 1$).

Table 3.2: Advantages and disadvantages of one-step methods

Advantage	**Disadvantage**
✓ Provides universal method for ODEs	✗ May diverge quickly for large h
✓ Distinguishes local/global error	✗ Order diminishes by 1 along trajectory

The idea of the Heun method can be continued to derive methods of higher order. Similar as for the Heun method, these methods require additional evaluations of the dynamic f. These methods are combined in the class of so-called Runge–Kutta methods:

Definition 3.23 (Explicit Runge–Kutta class methods).
Suppose a map $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and a step size $h \in \mathbb{R}$ to be given. Let

$$k_1 = f(t_j + c_1 \cdot h, \mathbf{x}(t_j)) \tag{3.14}$$
$$k_2 = f(t_j + c_2 \cdot h, \mathbf{x}(t_j) + \alpha_{21} \cdot h \cdot k_1) \tag{3.15}$$
$$\vdots \tag{3.16}$$
$$k_m = f(t_j + c_m \cdot h, \mathbf{x}(t_j) + \alpha_{m1} \cdot h \cdot k_1 + \ldots + \alpha_{mm-1} \cdot h \cdot k_{m-1}). \tag{3.17}$$

Then we call

$$\Psi(h, \mathbf{x}(t_j)) = \mathbf{x}(t_j) + h \cdot (\beta_1 \cdot k_1 + \ldots + \beta_m \cdot k_m) \tag{3.18}$$

an m-step *explicit Runge–Kutta* class method.

The Runge–Kutta class methods are defined uniquely by the coefficients α_{ij} and β_i.

Remark 3.24
In order to reveal a convergent (consistent and stable) behavior, the coefficients α_{ij} and β_i must satisfy certain conditions, which are outside the scope of this lecture.

Note that the number m refers to the required evaluations of the dynamics f and is in general not identical with the order of the method. In short, the methods are given by so-called *Butcher tableaux*.

c_1					
c_2	α_{21}				
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_m	α_{m1}	α_{m2}	$\cdots$	α_{mm-1}	
	β_1	β_2	$\cdots$	β_{m-1}	β_m

Table 3.3: Butcher tableaux of explicit Runge–Kutta class methods

Task 3.25
Assemble the Butcher tableaux of the Euler and Heun method given in Example 3.15.

Solution to Task 3.25: The Euler and Heun method are given by

0	
	1

0		
1	1	
	$\frac{1}{2}$	$\frac{1}{2}$

The classical Runge–Kutta method is a four-stage special case of this class; the corresponding Butcher tableau is shown in Table 3.4.
From an algorithmic point of view, explicit one-step methods are very simple to implement:

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

Table 3.4: Butcher tableaux of the classical Runge–Kutta method

Algorithm 1 Explicit Runge–Kutta methods

Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$

Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$

Input: Step size $h \in \mathbb{R}_0^+$

Input: Terminal time $\overline{T}$

Input: Butcher tableaux of explicit m step Runge–Kutta class method Ψ

procedure CLASS EXPLICIT RUNGE–KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
  $N \leftarrow (\overline{T} - t_0)/h$
  for $j = 0, \ldots, N-1$ **do**
    for $i = 1, \ldots, m$ **do**
      $k_i = f(t_j + c_i \cdot h, \mathbf{x}(t_j) + \alpha_{i1} \cdot h \cdot k_1 + \ldots + \alpha_{ii-1} \cdot h \cdot k_{i-1})$
    end for
    $\mathbf{x}(t_{j+1}) \leftarrow \mathbf{x}(t_j) + h \cdot (\beta_1 \cdot k_1 + \ldots + \beta_m \cdot k_m)$
    $t_{j+1} = t_j + h$
  end for
end procedure

Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$

Although the method is straightforward to implement, deriving suitable coefficients is highly nontrivial. Moreover, higher convergence and consistency orders come at a cost: in general, a higher-order method requires more function evaluations, cf. Table 3.5.

Table 3.5: Order of consistence vs. number of function evaluations for one-step methods

Order of consistence p	1	2	3	4	5	6	7	8	≥ 9
Minimal steps m	1	2	3	4	6	7	9	11	$\geq p+3$

In general, the following holds:

Theorem 3.26 (Bounded order of consistency for explicit methods).
Consider an explicit Runge–Kutta method with m steps. Then the order of consistency p of this method is bound by $p \leq m$.

Remark 3.27
his observation must be interpreted with care. On the one hand, Theorem 3.26 shows that higher consistency orders, and thus potentially higher accuracy, are achievable. On the other hand, this gain comes at the price of additional stages m, and therefore of additional evaluations of the dynamics f. In general, there is no universally best solver; the appropriate choice depends on the structure of the concrete problem at hand.

From an implementation point of view, the explicit Runge–Kutta method can be understood as a specialization of a general one-step method. Figure **??** sketches this relation and already separates the class `model`, which contains the dynamics of the underlying system.

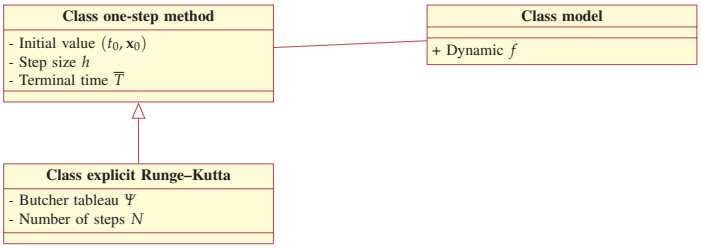

Figure 3.5: Sketch of UML diagram of class explicit Runge–Kutta

Remark 3.28
Note that Figure 3.5 is not complete and only a sketch of a possible class diagram.

Regarding explicit one-step methods in general, we see the advantages and disadvantages outlined in Table 3.6.

Table 3.6: Advantages and disadvantages of explicit Runge–Kutta class methods

Advantage	Disadvantage
✓ Allows for simple implementation	✗ Requires multiple evaluations
✓ Utilizes intermediate steps	✗ Propagates intermediate error

3.3 Implicit methods

In contrast to explicit methods, implicit methods use a fully coupled coefficient structure. The purpose of this additional coupling is to increase the achievable order of consistency and thereby to obtain more accurate approximations for a given number of stages. More formally, we define the following:

Definition 3.29 (Implicit Runge–Kutta class methods).
Suppose a map $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and a step size $h \in \mathbb{R}$ to be given. Let

$$k_i = f\left(t_i + c_i \cdot h, \mathbf{x}(t_i) + \sum_{j=1}^{m} \alpha_{ij} k_j\right) \tag{3.19}$$

define intermediate steps for $i = 1, \ldots, m$. Then we call

$$\Psi(h, \mathbf{x}(t_i)) = \mathbf{x}(t_i) + h \cdot \sum_{i=1}^{m} \beta_i \cdot k_i \tag{3.20}$$

an m-step *implicit Runge–Kutta* class method.

The method is called implicit as the values of k_i, $i = 1, \ldots, m$ is not longer a definition but requires the solution of a $m \cdot n_x$ dimensional nonlinear equation system. Similar to the explicit case, the Butcher tableau can be defined as in Table 3.7.

c_1	α_{11}	α_{12}	$\cdots$	α_{1m}
c_2	α_{21}	α_{22}	$\cdots$	α_{2m}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	
c_m	α_{m1}	α_{m2}	$\cdots$	α_{mm}
	β_1	β_2	$\cdots$	β_m

Table 3.7: Butcher tableaux of implicit Runge–Kutta class methods

For these methods, we can show the following:

Theorem 3.30 (Bounded order of consistency for implicit methods).
Consider an implicit Runge–Kutta class method with m steps. Then the order of consistency p of this method is bound by $p \leq 2 \cdot m$.

Again similar to the explicit case, the implementation of implicit methods is a straight forward procedure as displayed in Algorithm 2.

Algorithm 2 Implicit Runge–Kutta methods

Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}_0^+$
Input: Terminal time $\overline{T}$
Input: Butcher tableaux of implicit m step Runge–Kutta class method Ψ

procedure CLASS IMPLICIT RUNGE–KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
  $N \leftarrow (\overline{T} - t_0)/h$
  for $j = 0, \dots, N-1$ **do**
    $k \leftarrow$ SOLUTION NONLINEAR EQUATION SYSTEM($f, t_j, \mathbf{x}(t_j), h, \Psi, \varepsilon$)
    $\mathbf{x}(t_j) \leftarrow \mathbf{x}(t_j) + h \cdot \sum_{i=1}^{m} \beta_i \cdot k_i$
    $t_{j+1} = t_j + h$
  end for
end procedure

Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$

Although implicit methods can substantially improve consistency and convergence properties, Algorithm 2 also shows their main computational challenge: the auxiliary variables k_i are obtained from a nonlinear equation system. A natural approach is to solve this system by Banach fixed-point iteration, which in turn requires the associated map to be a contraction. For this purpose, we introduce the abbreviations

$$k := \begin{pmatrix} k_1 \\ \vdots \\ k_m \end{pmatrix} \qquad \text{and} \qquad F(k) := \begin{pmatrix} f\left(t_j + c_1 \cdot h, \mathbf{x}(t_j) + \sum_{j=1}^{m} \alpha_{1j} k_j\right) \\ \vdots \\ f\left(t_j + c_m \cdot h, \mathbf{x}(t_j) + \sum_{j=1}^{m} \alpha_{mj} k_j\right) \end{pmatrix}.$$

In order to be a contraction, we require

$$\|F(k^j) - F(k^i)\| \leq K \|k^j - k^i\|$$

to hold. Here, we have the following:

Theorem 3.31 (Contraction of implicit Runge–Kutta methods).
Consider an implicit Runge–Kutta class method and suppose the dynamics f to be Lipschitz with

Lipschitz constant L. Then the constant K in

$$\|F(k^j) - F(k^i)\| \leq K\|k^j - k^i\| \tag{3.21}$$

can be given by $K = h \cdot L$. *If the step size is chosen such that* $K = h \cdot L < 1$, *then inequality* (3.31) *is a contraction.*

Given that the assumptions made in Theorem 3.31 hold, the following Algorithm 3 can be applied to solve the nonlinear equation system (3.19)

Algorithm 3 Full step iteration

Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}_0^+$
Input: Butcher tableaux of implicit m step Runge–Kutta class method Ψ
Input: Stopping threshold $\varepsilon \in \mathbb{R}^+$
1: **procedure** SOLUTION NONLINEAR EQUATION SYSTEM($f, t_0, \mathbf{x}_0, h, \Psi, \varepsilon$)
2: $i = 1$ and $k^{i-1} \leftarrow 1$, $k^i \leftarrow 0$
3: **while** $\|k^i - k^{i-1}\| \geq \varepsilon$ **do**
4:
$$k^{i+1} = \begin{pmatrix} k_1^{i+1} \\ \vdots \\ k_m^{i+1} \end{pmatrix} \leftarrow \begin{pmatrix} f\left(t_0 + c_1 \cdot h, \mathbf{x}_0 + \sum_{j=1}^{m} \alpha_{1j} k_j^{i+1}\right) \\ \vdots \\ f\left(t_0 + c_m \cdot h, \mathbf{x}_0 + \sum_{j=1}^{m} \alpha_{mj} k_j^{i+1}\right) \end{pmatrix} = F(k^i)$$
5: $i \leftarrow i + 1$
6: **end while**
7: **end procedure**
Output: Approximation k of solution of nonlinear equation system

Similar to the explicit case, on the implementation side we can inherit similar items from the class one-step method. Internally, the implicit methods require the stopping threshold and a function to solve nonlinear equation systems as in Algorithm 3. Figure 3.6 provides a respective sketch of the class diagram.

It is important to note that the order of consistency of an implicit Runge–Kutta method also depends on the accuracy with which the nonlinear stage equations are solved. In particular, the local error bound (3.16) must still hold, which makes the choice of ε dependent on the step size h.

Combined, we restate facts for implicit Runge–Kutta class methods outlined in Table 3.8.

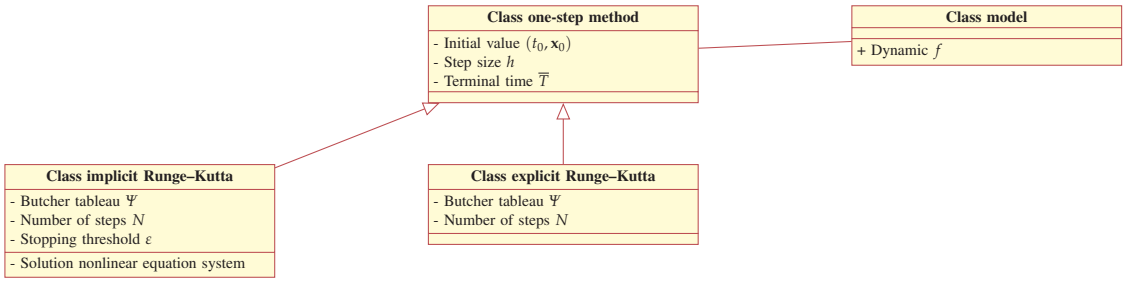

Figure 3.6: Sketch of UML diagram of class implicit Runge–Kutta

Table 3.8: Advantages and disadvantages of implicit Runge–Kutta class methods

Advantage	Disadvantage
✓ Allows higher order of convergence	✗ Consistency depends on chosen error
✓ Reduces number of evaluations per order of consistency	✗ Requires solution of nonlinear equation system
	✗ Limits step size

3.4 Adaptive step size methods

So far, we have assumed that the step size h is constant. In practice, however, solutions of differential equations may contain intervals with rapidly changing behavior as well as intervals with comparatively smooth behavior. A constant step size is therefore often inefficient, because it has to be chosen according to the most demanding part of the trajectory.

Adaptive step-size methods aim to reduce this drawback. The basic idea is straightforward: first, one-step with step size h is carried out and the numerical error is estimated. If this error exceeds a prescribed threshold, the step size is reduced and the step is repeated. If the error is sufficiently small, the current step is accepted and a new step size is computed for the next iteration.

To estimate the error and derive an updated step size, we combine two one-step methods of different consistency order. By construction, the higher-order method is typically more reliable and can be used to estimate the error of the lower-order method. Since naively evaluating two separate methods would require many function evaluations, one usually employs methods that reuse the same stage evaluations — so-called embedded methods.

Definition 3.32 (Embedded one-step method).
Suppose two convergent one-step methods $\Psi^1, \Psi^2 : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ of different order. Furthermore

let the intermediate steps

$$k_i = f\left(t_i + c_i \cdot h, \mathbf{x}(t_i) + \sum_{j=1}^{m} \alpha_{ij} k_j\right) \tag{3.22}$$

be identical for both methods for $i = 1, \ldots, m$. Then the combination Ψ^1, Ψ^2 is called an embedded one-step method.

Remark 3.33
By (3.22)*, the values α_{ij} and c_i are identical for both methods. Since the methods are of different order, we obtain that β_j are different for these methods.*

Two commonly used embedded methods are the Runge–Kutta $4(3)$ and the Dormand–Prince $5(4)$ method (in MATLAB called ode45) shown in Tables 3.9 and 3.10.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
1	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	
Ψ^1	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	0
Ψ^2	$\frac{1}{6}$	$\frac{2}{6}$	0	$\frac{1}{6}$

Table 3.9: Butcher tableaux of the Runge–Kutta $4(3)$ method

Before combining the latter to an algorithm, we first discuss and analyze the three described steps.

Error estimation

Since the step size can only be adapted for the current and for future iterations, it does not make sense to elaborate on errors, which already occurred in past iterations. Hence, we can focus on stability, i.e. one single step, and ignore convergence, i.e. the long run which is already given by construction of both one-step methods.
Since two one-step methods of different order are used, our aim is to estimate the error of the lower-order method. To this end, we compare the true solution with the approximation produced

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
Ψ^1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
Ψ^2	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

Table 3.10: Butcher tableaux of the Dormand-Prince $5(4)$ method

by the lower-order method Ψ^1

$$\varepsilon^1 := \mathbf{x}(t_{j+1}) - \Psi^1(h, \mathbf{x}(t_j))$$

where h is the current step size. Similarly, we obtain the error of the solution of the higher order method Ψ^2

$$\varepsilon^2 := \mathbf{x}(t_{j+1}) - \Psi^2(h, \mathbf{x}(t_j))$$

and the difference of both

$$\varepsilon := \Psi^2(h, \mathbf{x}(t_j)) - \Psi^1(h, \mathbf{x}(t_j)).$$

Since ε is the only quantity that can actually be computed, it must serve as the basis for the step-size adaptation. Moreover, because Ψ^2 is of higher order, we obtain

$$\lim_{h \to 0} \frac{\|\varepsilon^2\|}{\|\varepsilon^1\|} = 0.$$

Defining $\theta := \|\varepsilon^2\| / \|\varepsilon^1\|$, we can reformulate the latter fraction to

$$\frac{\|\varepsilon^2\|}{\|\varepsilon^1\|} = \frac{\|\varepsilon^1 - \varepsilon\|}{\|\varepsilon^1\|} = \theta \quad \Longleftrightarrow \quad \|\varepsilon^1 - \varepsilon\| = \|\varepsilon^1\| \cdot \theta.$$

Applying the triangle inequalities

$$\|\varepsilon^1\| - \|\varepsilon\| \leq \|\varepsilon^1 - \varepsilon\| \leq \|\varepsilon^1\| + \|\varepsilon\|$$

we obtain

$$\frac{1}{1+\theta}\|\varepsilon\| \leq \|\varepsilon^1\| \leq \frac{1}{1-\theta}\|\varepsilon\|.$$

Hence, for $h \to 0$ we have $\theta \to 0$ and the approximation

$$\|\varepsilon\| \approx \|\varepsilon^1\|,$$

holds. Therefore, we can use $\|\varepsilon\|$ as approximation of the true error $\|\varepsilon^1\|$ of the lower order method. This reveals

Theorem 3.34 (Approximated error for adaptive step size methods).
Consider two convergent one-step methods $\Psi^1, \Psi^2 : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ of different order. If the step size h is sufficiently small, then the error of the lower order method can be approximated via the difference of the local errors of Ψ^1, Ψ^2

$$\varepsilon^1 \approx \varepsilon := \Psi^2(h, \mathbf{x}(t_j)) - \Psi^1(h, \mathbf{x}(t_j)) \tag{3.23}$$

for each iteration instant $t_j = t_0 + j \cdot h$.

Computation of step size

Based on the approximated error ε, an adaptation of the step size shall be computed. To this end, a threshold $\bar{\varepsilon}$ for the error needs to be predefined for the adaptive step size method.
From consistency, we know that the lower order method satisfies

$$\|\varepsilon^1\| \leq C \cdot h^{p+1}$$

where p is the known order of the method and C is unknown. Hence, in the worst case, the latter holds with equality. Now we can use our approximation $\varepsilon \approx \varepsilon^1$ and the worst case to obtain

$$\|\varepsilon\| \approx \|\varepsilon^1\| = C \cdot h^{p+1}$$

and can identify

$$C \approx \frac{\|\varepsilon\|}{h^{p+1}}.$$

The new step size h_{new} shall satisfy

$$C \cdot h_{\text{new}}^{p+1} \leq \bar{\varepsilon}$$

where we can use the identified constant C to obtain

$$C \cdot h_{\text{new}}^{p+1} \approx \frac{\|\varepsilon\|}{h^{p+1}} \cdot h_{\text{new}}^{p+1} \leq \bar{\varepsilon} \quad \Longleftrightarrow \quad h_{\text{new}} \approx \sqrt[p+1]{\frac{\bar{\varepsilon}}{\|\varepsilon\|}} \cdot h.$$

Since this formula is still based on an approximation, one typically introduces an additional safety factor, for example $\eta = 0.9$, when computing the new step size. This leads to the following result:

Theorem 3.35 (Adaptive step size computation).
Consider two convergent one-step methods $\Psi^1, \Psi^2 : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ of different order. Let p be the lower order of both methods and suppose the step size $h > 0$ to be sufficiently small for Theorem 3.34 to hold. To satisfy a given error threshold $\bar{\varepsilon} > 0$, the step size can be adapted using

$$h_{new} = \eta \cdot \sqrt[p+1]{\frac{\bar{\varepsilon}}{\|\varepsilon\|}} \cdot h. \tag{3.24}$$

with safety constant $\eta \in (0,1)$.

Note that the threshold is guaranteed for the lower-order method. At the same time, the higher-order approximation is already available and is typically more accurate. For this reason, practical implementations usually advance the iteration with the higher-order solution.

Remark 3.36
Regarding the next iteration step, an identical computation as in Theorem 3.35 can be used. In contrast to the current iteration step, however, not only a reduction of the step size but also an increase may occur.

Integrating all of the above, we obtain Algorithm 4.

Algorithm 4 Adaptive step size method

Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Butcher tableaux of embedded method Ψ^1, Ψ^2
Input: Error threshold $\bar{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$

procedure CLASS ADAPTIVE STEP SIZE($f, t_0, \mathbf{x}_0, \overline{T}, \Psi^1, \Psi^2, \bar{\varepsilon}, \eta$)
  $h \leftarrow \overline{T} - t_0$
  while $t_j \neq \overline{T}$ **do**
    if $t_j + h > \overline{T}$ **then**
      $h \leftarrow \overline{T} - t_j$
    end if
    $t_{j+1} \leftarrow t_j + h$
    $\Psi^1(h, \mathbf{x}(t_j)) \leftarrow$ CLASS EXPLICIT RUNGE–KUTTA($f, t_j, \mathbf{x}(t_j), h, t_{j+1}, \Psi^1$)
    $\Psi^2(h, \mathbf{x}(t_j)) \leftarrow$ CLASS EXPLICIT RUNGE–KUTTA($f, t_j, \mathbf{x}(t_j), h, t_{j+1}, \Psi^2$)
    $\varepsilon \leftarrow \Psi^2(h, \mathbf{x}(t_j)) - \Psi^1(h, \mathbf{x}(t_j))$
    $h_{\text{new}} \leftarrow \eta \cdot \sqrt[p+1]{\frac{\bar{\varepsilon}}{\|\varepsilon\|}} \cdot h$
    if $\varepsilon > \bar{\varepsilon}$ **then**
      $h \leftarrow h_{\text{new}}$
    else
      $\mathbf{x}(t_{j+1}) := \Psi^2(h, \mathbf{x}(t_j))$
      $h \leftarrow h_{\text{new}}$
      $j \geq j + 1$
    end if
  end while
end procedure

Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$

Using Algorithm 4, we can refine our schematic view of the class one-step method as shown in Figure 3.7. In particular, the adaptive step-size method does not require a separate reimplementation of explicit Runge–Kutta methods; rather, it reuses them as building blocks.

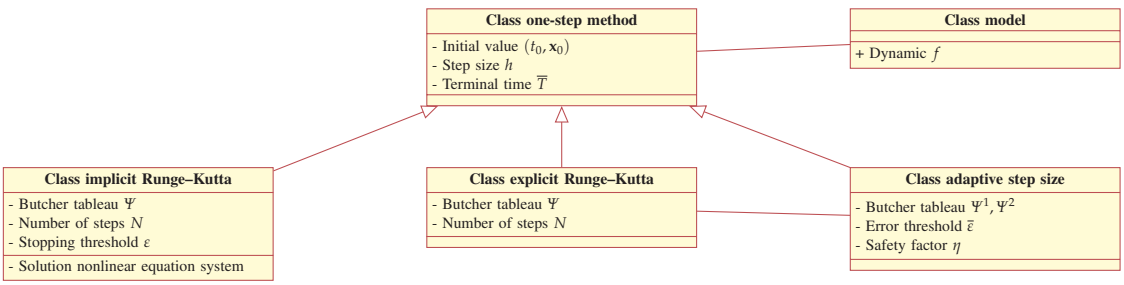

Figure 3.7: Sketch of UML diagram of class one-step method

Regarding adaptive step size methods, Table 3.11 summarizes advantages and disadvantages of such methods.

Table 3.11: Advantages and disadvantages of adaptive step size methods

Advantage	Disadvantage
✓ Adapts step size to dynamics	✗ Consistency depends on lower order
✓ Reduces number of evaluations	✗ Requires two one-step methods
✓ Reuses intermediate results	✗ Requires embedding of methods

The classes introduced so far in Algorithms 1, 2, and 4 can be unified under the general class of one-step methods. This is precisely the abstraction already suggested by the class diagrams in Figure 3.7. Algorithm 5 formalizes this viewpoint.

Algorithm 5 One-step methods

Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}^+$
Input: One-step method Ψ
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\bar{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0, 1)$

procedure CLASS ONE-STEP METHOD($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
  if Ψ is explicit Runge–Kutta method **then**
    $\mathbf{x}(\overline{T}) \leftarrow$ CLASS EXPLICIT RUNGE–KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
  else if Ψ is implicit Runge–Kutta method **then**
    $\mathbf{x}(\overline{T}) \leftarrow$ CLASS IMPLICIT RUNGE–KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
  else if Ψ is adaptive step size method **then**
    $\mathbf{x}(\overline{T}) \leftarrow$ CLASS ADAPTIVE STEP SIZE($f, t_0, \mathbf{x}_0, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
  else if ... **then**
  end if
end procedure

Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$

We can already observe from the algorithms above that the interface for the adaptive step size method slightly differs from the explicit and implicit methods. Within an implementation, the respective function of the derived objects must therefore be modified.

3.5 Adaptation for deployment

As the previous sections have shown, one-step methods provide a flexible framework for numerically solving differential equations. At the same time, we distinguished from the outset between the sampling time T of the physical system or process and the numerical step size h of the method. If no additional constraints are present, the simplest choice is to set $h = T$ and work on a common grid. In practical deployment scenarios, however, this assumption is often too restrictive.

Remark 3.37
Upon separating step size and sampling time, there are three generic cases:

- *Unified grid: All systems are evaluated using a superset of grids. In simulation, this is a common approach despite inducing high computational load. In practice, however, measurement values are missing and the approach cannot be utilized.*
- *Splitting grids: Sampling and step size are considered separately and intermediate points are neglected on a vice versa basis. Again, this is possible in simulations. The computational load is greatly reduced compared to a unified grid, yet now at minimum two programs are required, which additionally need to be synchronized using, e.g., thread blockers. Similar due to unavailability of measurements, this is not realizable in practice.*
- *Debouncing: Here, zero order hold is used for measurements, i.e. held constant until a new measurement is available. Using this idea, the grids can be split but additionally a realization in practical applications is possible.*

If constraints on the sampling time T are present, a so-called multiscale time grid can be used to compute approximate solutions of the differential equation (3.1).

Definition 3.38 (Multiscale time grid).
Consider sampling time $T \in \mathbb{R}_{>0}$ to be given defining a time grid $\mathbb{T}$ according to (3.7). Then for each $i \in \mathbb{N}$ the step size $h := T/i$ defines a multiscale time grid $\mathbb{T}_h$ satisfying $\mathbb{T} \subset \mathbb{T}_h$.

This concept is particularly useful in practice. A multiscale time grid allows numerical solution methods to be applied to systems whose physical components operate on a slower sampling scale T than the internal integration method. Algorithm 6 outlines a corresponding implementation.
With the class diagrams discussed so far in mind, we observe that the class `multiscale` provides the step size h, the sampling time T, the terminal time $\overline{T}$, and the initial values $(t_0, \mathbf{x}_0)$

Algorithm 6 Multiscale application of one-step methods

Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}^+$
Input: Sampling time $T \in \mathbb{R}^+$
Input: One-step method Ψ
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\bar{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0, 1)$
1: **procedure** CLASS MULTISCALE($f, t_0, \mathbf{x}_0, h, T, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
2: $N_T \leftarrow (\overline{T} - t_0)/T$
3: **for** $i = 0, \ldots, N_T$ **do**
4: $\mathbf{x}(t_{i+1}) \leftarrow$ CLASS ONE-STEP METHOD($f, t_i, \mathbf{x}(t_i), h, t_i + T, \Psi, \bar{\varepsilon}, \eta$)
5: **end for**
6: **end procedure**
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$

used by the one-step method. Accordingly, these quantities are shifted in the sketch shown in Figure 3.8.

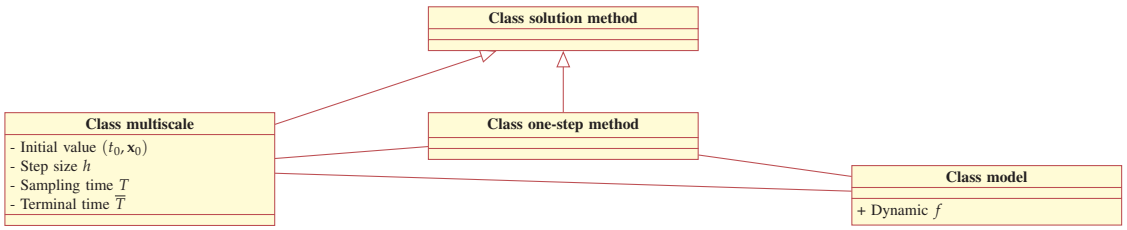

Figure 3.8: Sketch of UML diagram of class multiscale method

In addition, the multiscale formulation allows us to exploit the cocycle property of differential equations, cf. Remark 3.9, and even to switch solution methods during runtime if required.

Table 3.12: Advantages and disadvantages of multiscale methods

Advantage	Disadvantage
✓ Decouples step size and sampling	✗ Requires additional grid
✓ Utilizes cocycle property	✗ May require asynchronous runs
✓ Integrates previous methods	

Remark 3.39
Here, we restrict attention to the case of a fixed sampling time T. In practice, one also encounters settings with fixed sampling time T and only bounded step size h. Since such situations need not admit a common multiscale structure, sampling and numerical integration may have to run asynchronously. Corresponding solutions are beyond the scope of this lecture.

CHAPTER 4

SIMULATION

In Chapter 3, we developed numerical methods for computing approximate solutions of differential equations. In the present chapter, we build on these methods to analyze models through the behavior generated by their trajectories. This step marks the transition from solving equations to performing simulations. Conceptually, the distinction between a *model* and a *simulator* lies in their role: a model represents the relevant dynamics of a system, whereas a simulator uses this model together with numerical procedures and parameters to generate behavior. In terms of the V-model, we therefore move from the computational core of implementation towards the use of these components within simulation workflows, as indicated in Figure 4.1.

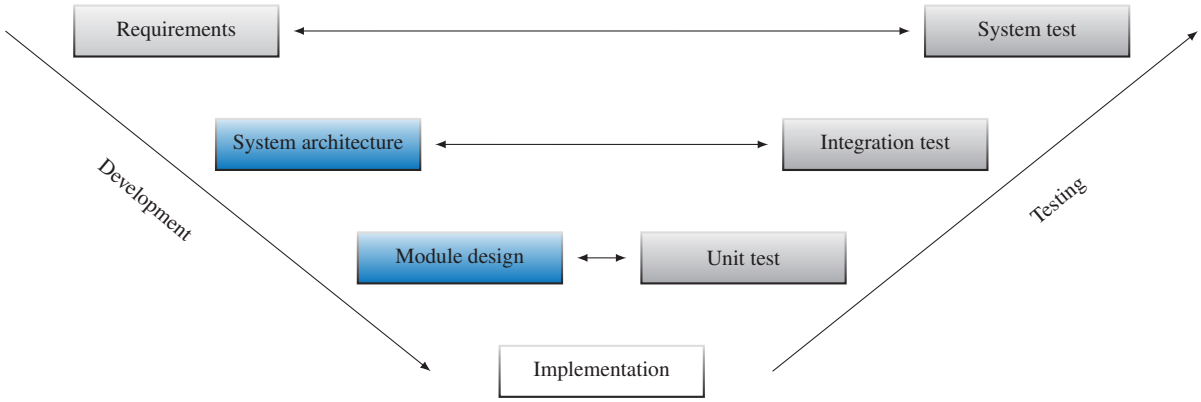

Figure 4.1: V model for system development

As discussed in the previous chapter, one-step methods can be applied to models provided that the assumptions underlying the numerical method are satisfied. This immediately reveals two main classes of parameters that influence the generated behavior: parameters associated with the numerical solution method and parameters contained in the model itself. In both cases, only those

variations are admissible that do not destroy the structural assumptions on which the simulation is based, for example convergence properties of the numerical method or the order and regularity of the underlying differential equation.
Accordingly, this chapter first introduces the simulation problem itself and discusses how parametrization shapes its structure in Section 4.1. Section 4.2 then turns to the processing of simulation results from the perspectives of accessibility, statistics, and visualization. Based on this, Section 4.3 studies relations between inputs and outputs by means of sensitivity analysis. In this way, the chapter prepares the transition to Chapter 5, where simulation results are connected to requirements, testing, and automation.

4.1 Parametrization and Processing

In Chapter 3, the numerical methods were formulated for the initial value problem from Definition 3.3. In simulation, however, we are typically not interested in a single isolated trajectory alone. Rather, we wish to generate and compare system behavior for different choices of model and method parameters.
We therefore begin by making explicit which parameters may be varied on the model side. To this end, we introduce the notion of a parametrized control system:

Definition 4.1 (Parametrized control system).
We call a map $f : \mathcal{T} \times \mathcal{X} \times \mathcal{U} \times \mathcal{P} \to \mathcal{X}$ given by

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}(t), \mathbf{u}(t), p) \tag{4.1}$$

a *parametrized control system*. We also refer to (4.1) as parametrized model Σ_M.

This definition distinguishes between time-dependent parameters, represented here by $\mathbf{u}(\cdot)$, and time-independent parameters, represented by p. The quantity $\mathbf{u}(\cdot)$ typically denotes an external input to the system or model and is therefore often used to influence the system behavior, for example in order to stabilize the dynamics or to excite certain operating regimes. Throughout this lecture, we simply refer to $\mathbf{u}(\cdot)$ as a time-dependent parameter.
The parametrized control system thus makes explicit which aspects of the dynamics can be modified when a simulator, in the sense of Definition 2.10, is executed.

Remark 4.2
Strictly speaking, the notion of a model introduced in Definition 2.7 is best understood as a map from inputs to outputs, whereas the parametrized model (4.1) *describes the evolution of the state.*

To recover an explicit input–output description, one may supplement the state equation by an output map of the form

$$\mathbf{y}(t) = h(t, \mathbf{x}(t), \mathbf{u}(t), p).$$

Recalling the definitions of *simulator* and *simulation*, the simulator itself may additionally be parametrized by the chosen numerical method as well as by quantities such as step size and sampling time. This allows us to combine the parametrized model with the numerical solution method in a common software-oriented view; see Figure 4.2.

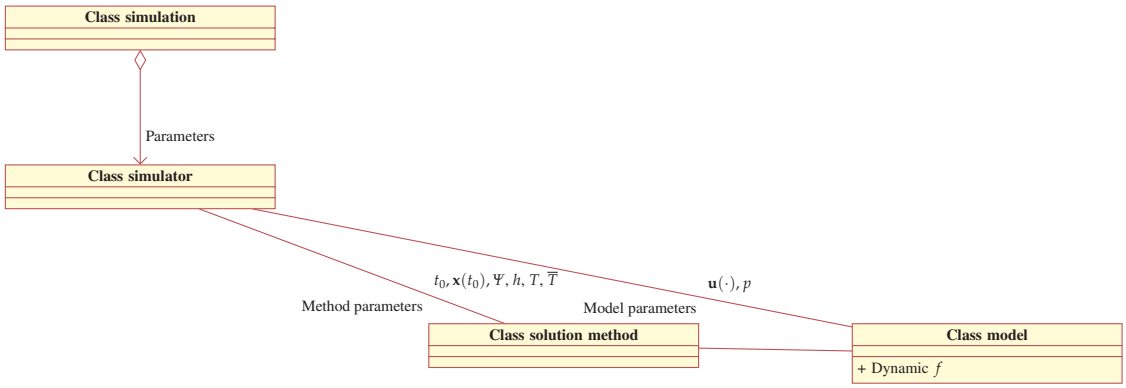

Figure 4.2: Sketch of UML diagram of class simulator

We emphasize that the connection from class simulation to class simulator is modeled as an aggregation, that is, one instance of class simulation may contain several instances of class simulator. A direct practical consequence is that a simulation can orchestrate several simulator runs, for example for different parameter combinations or scenarios. From an algorithmic point of view, this leads to the straightforward prototypes shown in Algorithms 7 and 8.
At the implementation level, the central technical artifact used to realize a simulation is a computer program.

Definition 4.3 (Computer program).
A sequence or set of instructions to be executed by a computer is called a *computer program*. Moreover, a computer program, or a component thereof, is called a *user interface* or *HMI (human–machine interface)* if it allows data to be transferred to and from the program.

Algorithm 7 Prototype of simulator

Input: Model Σ_M
Input: Parameters $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$
Input: Step size $h \in \mathbb{R}^+$
Input: Sampling time $T \in \mathbb{R}^+$
Input: One step method Ψ
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\bar{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
1: **procedure** CLASS SIMULATOR($\Sigma_M, t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p, h, T, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
2: $\quad\Sigma_M \leftarrow$ CLASS MODEL($\mathbf{u}(\cdot), p$)
3: $\quad\mathbf{x}(\overline{T}) \leftarrow$ CLASS SOLUTION METHOD($\Sigma_M, t_0, \mathbf{x}(t_0), h, T, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
4: **end procedure**
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$

Algorithm 8 Prototype of simulation

Input: Model Σ_M
Input: Parameter sets $\mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$
Input: Step size $h \in \mathbb{R}^+$
Input: Sampling time $T \in \mathbb{R}^+$
Input: One step method Ψ
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\bar{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
1: **procedure** CLASS SIMULATION($\Sigma_M, \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}, h, T, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
2: $\quad$**for all** $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ **do**
3: $\quad\quad$**while** $t_0 < \overline{T}$ **do**
4: $\quad\quad\quad\mathbf{x}(t_0 + T) \leftarrow$ CLASS SIMULATOR($\Sigma_M, t_0, \mathbf{x}(t_0), h, T, \overline{T}, \Psi, \bar{\varepsilon}, \eta$)
5: $\quad\quad\quad t_0 \leftarrow t_0 + T$
6: $\quad\quad$**end while**
7: $\quad$**end for**
8: **end procedure**
Output: Trajectory $\mathbf{x}(\cdot)$

Table 4.1: Advantages and disadvantages of splitting simulation/simulator

Advantage	Disadvantage
✓ Separate test from evaluation	✗ Induces structural overhead
✓ Allows parallel runs	✗ Requires parameter separation

Having established how trajectories are generated for given parameter sets, we now turn to the question of how the resulting data can be processed and interpreted.

4.2 Processing of results

Simulation results are processed in order to understand system behavior, to assess performance, and to support predictions for relevant scenarios. To this end, the generated data is typically used to

- design and organize replications,
- estimate performance metrics, for example by statistical means, and
- analyze the system and the experimental setup, for example by textual or graphical reporting.

Remark 4.4
The purpose of replication design is to obtain as much deterministic or statistical information as possible from simulation runs at the lowest reasonable computational cost. In practice, this often means reducing either the number of replications or the run length of each experiment.
Performance metrics aim at point estimates, interval estimates, or other quantitative summaries of relevant properties of the simulated system. In this context, sample size and independence of observations are often crucial.
System analysis and experimentation seek to understand the behavior of the system, generate predictions for different scenarios, and reveal design trade-offs. Typical examples include parameter studies and scenario comparisons.

The starting point of any processing step is the data generated by the simulation itself. A first essential distinction concerns the time horizon of the simulation, since this directly affects how the resulting data can be handled.

Definition 4.5 (Terminating/non-terminating simulation).
Consider a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. If the terminal time is finite, i.e. $\overline{T} < \infty$, then we refer to Θ as *terminating simulation*. If $\overline{T} = \infty$, then Θ is called *non-terminating simulation*.

he distinction between these two cases has a major impact on result processing. In the case of a terminating simulation, several runs can be executed sequentially and the results can subsequently be combined *a posteriori*, that is, after the runs have finished. For non-terminating simulations, by contrast, one cannot wait for termination; data has to be processed during runtime, and several runs must typically be considered in parallel rather than sequentially. We therefore begin by formalizing the outcome of a simulation:

Definition 4.6 (Replication).
Given a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ with terminal time $\overline{T} < \infty$ and inputs $\{(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)_{i \in \mathcal{I}}\} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Then the set of outputs $\mathcal{R} := \{(\mathbf{x}_j(t_i))_{i,j \in \mathcal{I}}\}$ is called *replication*.

Remark 4.7
Since a simulation may call several simulators with different parameter choices, the index $\mathbf{x}_j$ is used to distinguish simulator outputs, whereas t_i denotes the simulation time index.

Once a replication has been generated, many different processing steps become possible. The first and most basic one is to make the replication accessible. A fundamental mechanism for doing so is the so-called data dump, which captures the essential idea of storing simulation output.

Definition 4.8 (Data dump).
Suppose a replication $\mathcal{R}$ to be given by a computer program. Then we call a transfer of the replication to another computer program a *data dump*.

Although this definition is deliberately general, typical implementations include export functions that store data on persistent media, for example as `csv` or `json` files, or routines that transfer tables between databases, for example via SQL. Note that the receiving computer program may also simply be the operating system.
Accordingly, a data dump is typically performed after termination of a terminating simulation. While it is in principle also possible to execute a data dump during runtime, this is often avoided because it introduces additional overhead. During runtime, other forms of output are usually more appropriate.

Definition 4.9 (Widget).
Given a replication $\mathcal{R}$ within a computer program. Then we call a transfer of the replication to a user interface a *widget*.

A widget therefore denotes a textual or graphical interface through which simulation results are exposed to a user. In this sense, even a console output can be interpreted as a widget. Based on the last two definitions, the UML sketch of the simulation architecture can now be extended as shown in Figure 4.3.
Apart from accessibility, several other steps for data processing exist:

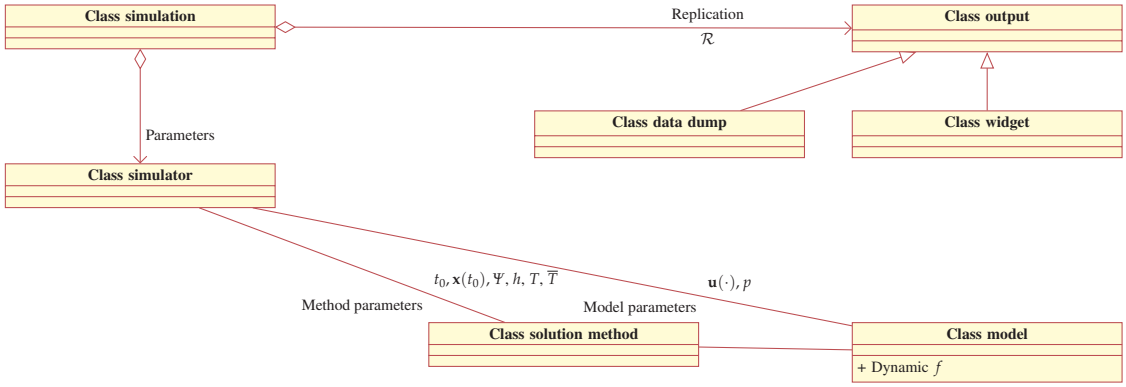

Figure 4.3: Sketch of UML diagram of class simulator with data processing

Definition 4.10 (Data processing).
Consider a replication $\mathcal{R}$ to be given by a computer program. Then possible processing steps include

- Saving — making a replication accessible,
- Validation — ensuring correctness and relevance of data,
- Sorting — arranging data according to an order or in sets,
- Summation — reducing data to their statistical properties,
- Aggregation — combining specific data,
- Classification — separating data by properties, and
- Reporting — listing data or results of processing steps.

TThese data-processing operations are quite fundamental, but they form the basis for the broader field of so-called data analytics, whose aim is to extract insight from data in support of technical, social, or business processes.

Definition 4.11 (Data analytics).
Given a replication $\mathcal{R}$ by a computer program, data analytics refers to steps including

- Inspecting — detecting corrupt / inaccurate / incomplete data,
- Cleaning — deleting, replacing, modifying or completing data,
- Transforming — converting data into formats / structures, and
- Modeling — defining and analyzing data requirements.

Within this lecture, we restrict ourselves to basic methods of data processing. As indicated by the preceding definitions, both data processing and data analytics operate on replications and are therefore naturally associated with the output of a simulation, cf. Figure 4.4.

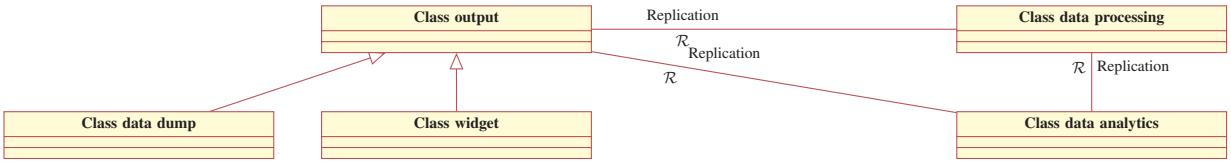

Figure 4.4: Sketch of UML diagram connecting output and data processing/analytics

There are several options to sort and report data. The most simple one is a table, cf. Table 4.2.

Table 4.2: Example of a data dump

t	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$
16	−6	−2	11
17	−6	−3	10
18	−6	−3	9
13	−5	−2	12
14	−5	−2	11
15	−5	−2	11
19	−5	−3	8
20	−5	−3	7
21	−5	−3	6
11	−4	−2	13
12	−4	−2	12
22	−4	−3	6
10	−2	−2	13
23	−2	−3	5
24	−2	−3	5
25	−2	−4	6
9	−1	−1	13
3	0	0	8
2	1	2	8
1	2	3	7
4	2	0	10

Table 4.2 – continued from previous page

6	2	−1	12
7	2	−1	12
8	2	−1	13
5	3	0	11
0	5	5	5

If we exploit additional structural knowledge—for example that t denotes time and is therefore a natural sort key—we can apply the data-processing step of *sorting* to Table 4.2 and obtain the ordered results shown in Table 4.3.

Table 4.3: Sorted data dump from Table 4.2 with sort key t

t	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$
0	5	5	5
1	2	3	7
2	1	2	8
3	0	0	8
4	2	0	10
5	3	0	11
6	2	−1	12
7	2	−1	12
8	2	−1	13
9	−1	−1	13
10	−2	−2	13
11	−4	−2	13
12	−4	−2	12
13	−5	−2	12
14	−5	−2	11
15	−5	−2	11
16	−6	−2	11
17	−6	−3	10
18	−6	−3	9
19	−5	−3	8
20	−5	−3	7

Table 4.3 – continued from previous page			
21	−5	−3	6
22	−4	−3	6
23	−2	−3	5
24	−2	−3	5
25	−2	−4	6

The same data can also be arranged geometrically as points in a so-called point cloud.

Definition 4.12 (Point cloud).
Given a replication $\mathcal{R}$, a *point cloud* is a widget aligning selected data as dots in a coordinate system.

Task 4.13 (Point cloud)
Use the state coordinates $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ *from the replication* $\mathcal{R}$ *given in Table 4.2 to plot a point cloud in Cartesian coordinates.*

Solution to Task 4.13: The respective point cloud is given in Figure 4.5.

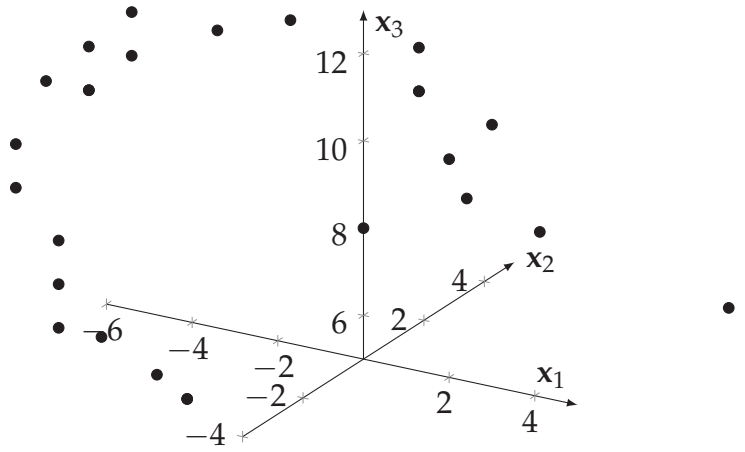

Figure 4.5: Point cloud of three time series from Figure 4.6

While both the table and the point cloud already provide useful reports, neither representation explicitly captures temporal order. A first step towards more informative aggregation is therefore to include time directly in the visualization.

Definition 4.14 (Time series / trajectories).
Given a replication $\mathcal{R}$ that includes time, a widget that aligns time t with the corresponding data $\mathbf{x}(t)$ is called a *time series*. Connecting the points of such a time series in temporal order is called a *trajectory plot*.

Task 4.15 (Time series)
Use the state coordinates $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ *from the replication* $\mathcal{R}$ *given in Table 4.2 to plot the time series.*

Remark 4.16
A time series may correspond to a single component of the state vector $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_{n_x})$*, but it may equally well represent the complete simulator output* $\mathbf{x}$ *itself.*

Solution to Task 4.15: The respective time series and trajectory plots are given in Figure 4.6.

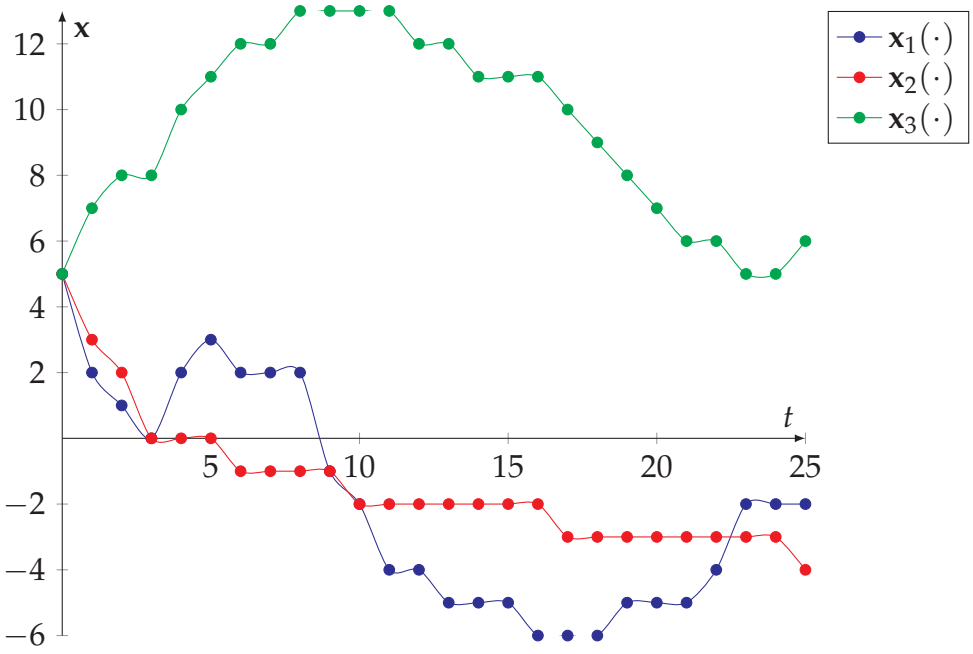

Figure 4.6: Sketch of three time series

In the present example, the different time series appear to evolve towards different levels within their respective subspaces. To analyze such behavior more systematically, for example with re-

spect to convergence tendencies or variability, one may introduce basic statistical quantities such as the mean, the sample variance, and confidence intervals. These belong to the aggregation and classification aspects of data processing.

Definition 4.17 (Mean, sample variance and confidence interval).
Consider a replication $\mathcal{R} = \{(\mathbf{x}_j)_{j\in\mathcal{I}}\}$ to be given. Then we call

$$\mathrm{E}(\mathcal{R}) := \frac{\sum\limits_{j\in\mathcal{I}} \mathbf{x}_j}{\sharp\mathcal{I}} \tag{4.2}$$

mean or *expected value* of $\mathcal{R}$. Moreover, we call

$$\mathrm{S}(\mathcal{R}) := \frac{\sum\limits_{j\in\mathcal{I}} \big(\mathbf{x}_j - \mathrm{E}(\mathcal{R})\big)}{\sharp\mathcal{I} - 1} \tag{4.3}$$

sample variance and

$$\mathrm{I}(\mathcal{R}) := \left[\mathrm{E}(\mathcal{R}) \pm t(\sharp\mathcal{I} - 1, 1 - \alpha/2)\frac{\mathrm{S}(\mathcal{R})}{\sqrt{\sharp\mathcal{I}}}\right] \tag{4.4}$$

confidence interval where $t(\sharp\mathcal{I} - 1, 1 - \alpha/2)$ is the critical value of the t-distribution (or z-distribution) with confidence level $1 - \alpha/2$. We call

$$Q_1(\mathcal{R}) := \mathrm{E}\left(\{\mathbf{x}_j\} \mid \mathbf{x}_j \text{ is in the lowest 25\% of } \mathcal{R}\right) \tag{4.5}$$

$$Q_3(\mathcal{R}) := \mathrm{E}\left(\{\mathbf{x}_j\} \mid \mathbf{x}_j \text{ is in the highest 25\% of } \mathcal{R}\right) \tag{4.6}$$

first and third quartile. Last, we call any element of $\mathcal{R}$ satisfying

$$\mathbf{x}_j \notin [Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)] \tag{4.7}$$

for $k > 0$ an *outlier*.

Remark 4.18
There is no common definition of an outlier. Here, we use the so called Tukey's fence.

These quantities can be combined in a so-called box plot.

Definition 4.19 (Statistical plots).
Given a replication $\mathcal{R}$, a *boxplot* is a five number summary containing median, minimum and maximum as lines and first and third quartile marking a box. Outlier may be added as points.

Task 4.20 (Boxplot)
Compute boxplots for the states $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ *from the replication* $\mathcal{R}$ *given in Table 4.2.*

Solution to Task 4.20: The respective boxplots are given in Figure 4.7.

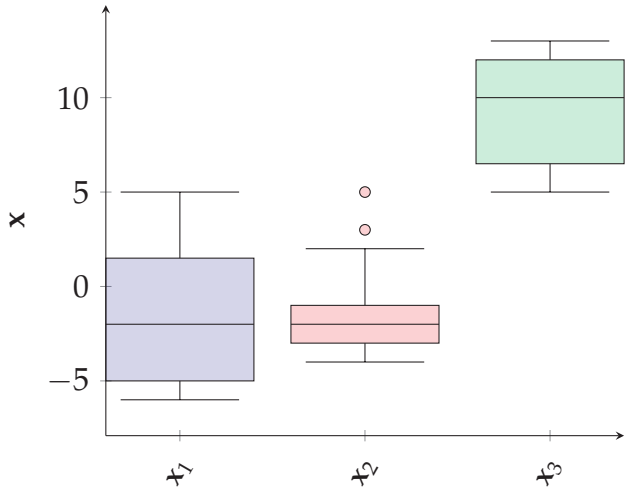

Figure 4.7: Box plot of three time series from Figure 4.6

As a last idea of data processing, we include visualization using a representation of the system at hand. For these cases, the results from the simulation $\mathcal{R}$ must be mapped to factors contained in the graphical models. The advantage of such a visualization is that the implications on reality, and in particular of dependencies of parameters and scenarios is intuitively accessible. Figure 4.8 provides an overview.
However, many relevant dependencies remain hidden in such visualizations alone. To quantify them more directly, we next introduce sensitivity analysis.

4.3 Sensitivity

In many applications, it is essential to understand how variations of the inputs affect the outputs. For continuously variable inputs, this dependence can be assessed analytically by means of

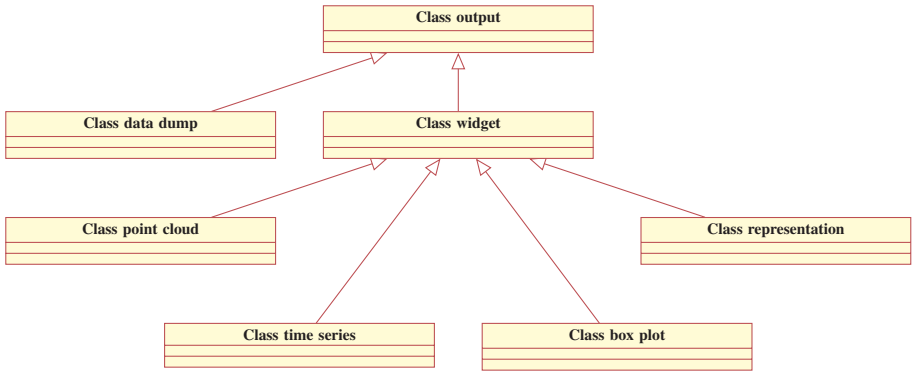

Figure 4.8: Sketch of UML diagram of class output

sensitivities.

Definition 4.21 (Sensitivity).
Consider a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Furthermore suppose the inputs $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ to be fixed and the sets $\mathcal{T}, \mathcal{X}, \mathcal{U}^{\mathbf{T}}$ and $\mathcal{P}$ to be continuous. Then we call

$$\frac{\partial \mathbf{x}_j}{\partial t_0}(t_i), \quad \frac{\partial \mathbf{x}_j}{\partial \mathbf{x}(t_0)}(t_i), \quad \frac{\partial \mathbf{x}_j}{\partial \mathbf{u}(\cdot)}(t_i), \quad \text{and} \quad \frac{\partial \mathbf{x}_j}{\partial p}(t_i) \tag{4.8}$$

sensitivity of the output $\mathbf{x}_j(t_i)$ with respect to the inputs.

The purpose of sensitivities is to quantify the impact of input changes on the output, for example whether a variation decreases the output, leaves it essentially unchanged, increases it, induces structural changes, or even causes bifurcation phenomena.

Task 4.22
Describe cases for input/output changing effects.

Solution to Task 4.22: A decreasing effect is encountered, for example, for dissipative elements such as dampers. An unchanged output indicates that the considered output is independent of the respective input. An increasing effect may occur for amplifying components, for example in an electrical amplifier. Structural changes may arise if parameters influence pole locations and shift them from the open left half-plane into the open right half-plane. Finally, bifurcation may occur when a parameter variation changes the equilibrium structure so that the solution converges to a qualitatively different equilibrium.

Remark 4.23
If $\mathcal{T}$, $\mathcal{X}$, $\mathcal{U}^{\mathbf{T}}$, or $\mathcal{P}$ are not continuous, the corresponding derivatives do not exist in the classical sense, and this approach cannot be applied directly.

By definition, a sensitivity is tied to the specific operating point determined by the chosen inputs $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Nevertheless, it can be interpreted either pointwise, that is, for fixed t_i, or along an entire time series $(t_i)_{i\in\mathcal{I}}$. In practice, we usually do not compute these derivatives analytically but approximate them numerically. For this purpose, at least two simulation outputs corresponding to perturbed inputs are required.

Definition 4.24 (Numerical sensitivity).
Consider a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ and suppose two inputs $(t_{0,1}, \mathbf{x}_1(t_{0,1}), \mathbf{u}_1(\cdot), p_1), (t_{0,2}, \mathbf{x}_2(t_{0,2}), \mathbf{u}_2(\cdot), p_2) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ to be given where $\mathcal{T}$, $\mathcal{X}, \mathcal{U}^{\mathbf{T}}$ and $\mathcal{P}$ are continuous. Then we can use

$$\frac{\partial \mathbf{x}_j}{\partial t_0}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{t_{0,1} - t_{0,2}} \tag{4.9}$$

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{x}(t_0)}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{\mathbf{x}_1(t_{0,1}) - \mathbf{x}_2(t_{0,2})} \tag{4.10}$$

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{u}(\cdot)}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{\mathbf{u}_1(\cdot) - \mathbf{u}_2(\cdot)} \tag{4.11}$$

$$\frac{\partial \mathbf{x}_j}{\partial p}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{p_1 - p_2} \tag{4.12}$$

as an approximation of the sensitivities provided the distances between the elements of respective inputs are small enough and the remaining inputs are equivalent.

Remark 4.25
Note that sensitivities have got nothing to do with statistics, i.e. the results is purely deterministic and not stochastic.

For the processing of sensitivities, one may again use either a data dump or a two-dimensional point cloud. In the latter case, the abscissa represents the respective input $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)$, while the ordinate represents the corresponding sensitivity.
Based on the numerical approximation of sensitivities, we can now discuss the possible outcomes of such an analysis in more detail. In applications, the cases mentioned above may have very different implications. Here, we merely highlight their interpretation; a deeper analysis lies beyond the scope of this lecture:

- Decreasing or unchanged output can be seen as robust behavior and, to some extent, allows us to ignore changes in these inputs.
- Increasing output indicates that the corresponding input variation must be limited, although moderate changes may still be tolerated as long as no structural or bifurcation effects occur.
- Structural changes indicate that qualitative properties of the system behavior are altered. In simulation, it is therefore of particular interest to identify those input values that form the boundary between two structures.
- Bifurcation changes indicate changes in properties of the system, not its behavior. Similar to structural changes, the switching points are of interest.

Building on these considerations, we finally introduce the notion of scenario analysis.

Definition 4.26 (Scenario analysis).
A scenario analysis is given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ and three inputs reflecting

- the worst case,
- the best case, and
- the base case which reflects the most likely scenario.

Scenario analysis is thus concerned with identifying those switching values that change the behavior from the base case towards the worst or best case, and with quantifying the tolerable size of deviations.

Both scenario analysis and sensitivity analysis can be implemented using the simulation prototype from Algorithm 8. To this end, the parameters $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)$ are selected so as to represent the scenarios under consideration, while for sensitivity analysis appropriate parameter perturbations must be supplied. The systematic automation of such definitions is part of Chapter 5.

CHAPTER 5

TESTING AND AUTOMATION

After discussing simulation and the processing of simulation results in the previous chapter, we now turn to validation and verification, thereby returning to the original purpose of simulation within the development process. To this end, we focus on requirements, their precise formulation, and the testing of these requirements in order to complete the V-model perspective, cf. Figure 5.1.

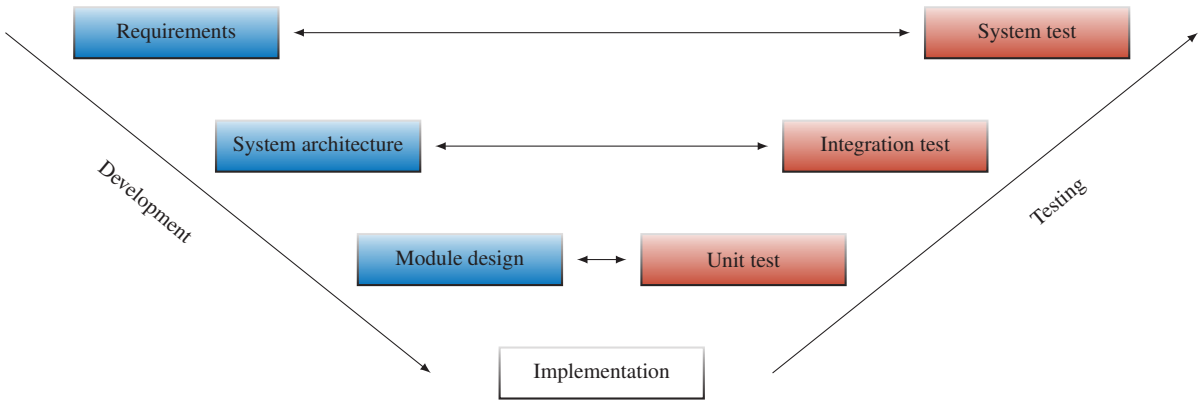

Figure 5.1: V model for system development

We begin with a brief recap of those parts of the lecture that were covered in Chapters 3 and 4. Taking one step back, we then introduce the general concept of a requirement, including its three key elements: component, specified conditions, and quality, in Section 5.1. Section 5.2 subsequently discusses how test cases can be derived in order to structure both the process and the objective of testing. To this end, we introduce a prototype test and the concept of grids. Building on this foundation, Section 5.3 develops a hierarchy of tests that allows us to address integration and system-level issues. Finally, Section 5.4 considers how testing efforts can be automated and organized efficiently.

5.1 Requirements and testing

Previously, we considered

- differential equation solvers and models in Chapter 3, as well as
- simulator, simulation and data processing in Chapter 4

and encapsulated these in modules which led us to the system architecture sketched in Figure 5.2.

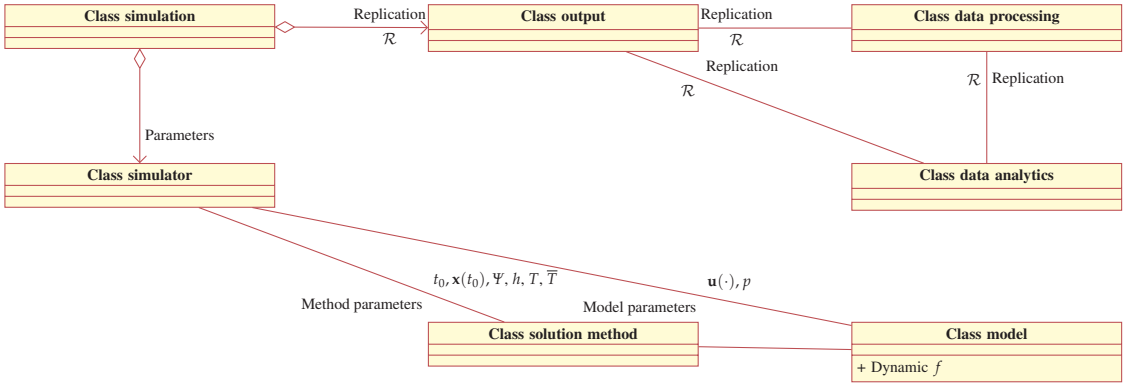

Figure 5.2: Sketch of UML diagram of class simulator with data processing

To complete Figure 5.1, we first need to clarify what is meant by the terms *requirement* and *test*. Within this lecture, we follow ISO 9000:2005, ISO/IEC DIS 25000:2014, ISO 29148:2011, and related standards in order to introduce the necessary terminology, cf. [9, 11, 12] for details. The most fundamental concept in this context is the so-called *(information) need*:

> An (information) need is an insight necessary to manage objectives, goals, risks, and problems.
>
> ISO/IEC 15939 (2007)

Within this lecture, the distinction between objectives, goals, risks and problems are outside our scope. Instead, we generically consider a need to be without intent. Utilizing the term of a need, a requirement can be formulated:

> A requirement is a need or expectation that is stated, generally implied or obligatory.
>
> ISO 9000 (2005)

Remark 5.1
We emphasize that the difference between a need and an expectation lies in necessity: a need refers to an insight that is necessary, whereas an expectation need not be necessary. Consequently, every need is an expectation, but not every expectation is a need. Moreover, an expectation is implied if it is not explicitly stated but nevertheless relevant. Finally, an expectation is obligatory if it is both implied and necessary, that is, if it has the character of a need.

ISO 9000 does not prescribe a unique linguistic form in which a requirement has to be stated. As a consequence, the variety of formulations encountered in practice can be very large, even for the same task. In many cases, requirements are expressed using recurring sentence patterns, templates, or user stories.

Task 5.2
Consider a system Σ to be given. Check whether

1. *The system Σ is described by a model Σ_M.*
2. *The model Σ_M can be evaluated using a simulator Φ.*

are requirements for the system.

Solution to Task 5.2: For 1, an expectation for system Σ is given by „description by a model Σ_M“. Regarding 2, the expectation is not linked to the system and is therefore not a requirement of the system.

From Task 5.2(2) we can additionally observe that the formulation is not sufficiently precise. For a given simulator Φ, the assertion „can be evaluated“ is either true or false. However, even if it is false for one simulator, there may still exist another simulator for which the statement becomes true. For this reason, it is advisable to use requirement patterns or templates that lead to unambiguous questions and answers.
Moreover, Task 5.2 shows that each requirement must ultimately be associated with an answer criterion. In our context, this criterion is captured by the notion of *quality*. To formalize this, we introduce the reference object of a component, which allows us to decide whether we consider a single function, an integration step, or the overall system.

Definition 5.3 (Component).
Consider the system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Then we call any subset of this architecture $\gamma : \mathcal{U} \to \mathcal{Y}$ with input set $\mathcal{U}$ and output set $\mathcal{Y}$ a *component*.

Note that, while this is not technically necessary, components are typically chosen as connected subsets of the architecture in practical applications. Based on these components, we can transfer the concept of quality to our setting.

> Quality is the capability of a component to satisfy stated and implied needs when used under specified conditions.
>
> ISO/IEC DIS 25000 (2014)

Adapted to our simulation setting, this leads to the following definition:

Definition 5.4 (Quality).
Consider a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Then we call $Q : \mathcal{Y} \to \mathbb{R}_0^+$ given by

$$Q(\gamma(\mathbf{u})) = Q(\mathbf{y}) \tag{5.1}$$

quality of a component.

Definition 5.5 (Specified conditions).
Given a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ and a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of Θ. Then we call $\mathcal{C} := \{(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)\} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ *test set* or *specified conditions*.

In particular, we define the following:

Definition 5.6 (Requirement).
Given a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ together with quality criteria $Q : \mathcal{Y} \to \mathbb{R}_0^+$ and specified conditions $\mathcal{C} := \{(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)\} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Then we call the triple $R := (\gamma, Q, \mathcal{C})$ a *requirement*.

As outlined before, requirements are often formulated verbally. Still, the quality criteria defined in Definition 5.4 together with the specified conditions from Definition 5.5 are considered to uphold certain properties [9].

Assumption 5.7 (Properties of quality criteria and specified conditions)
A requirement shall be

- complete — each requirement must fully describe the required and to-be-delivered functionality
- consist – a requirement must not contradict itself and other requirement
- agreed – all stakeholders accept the validity of a requirement
- unambiguous – a requirement is interpreted only one way by all readers
- verifiable – a requirement can be proven by a test or measurement
- traceable – the origin of the requirement, its implementation and the relationship to other documents can be retraced
- necessary – in the event of failure, a requirement shall cause a deficit
- understandable – a requirement contains only terms with fixed meaning
- feasible — a developer can point out specific facts regarding implementation and costs of a requirement

Within the system architecture shown in Figure 5.2, a typical component is the class `model`, while a typical test set consists of initial conditions for the model or the overall simulation.

Remark 5.8
We emphasize that quality is restricted to stated and implied needs and therefore captures only a subset of the overall requirements imposed on a system. As an assertion, quality is related to the concept of a key performance indicator, cf. Definition 2.16, but it is oriented toward a testable pass/fail-type assessment.

Although the normative definition is restricted to needs, the concept of quality can also be applied to requirements more broadly. In order to evaluate the quality of a component with respect to a requirement, we first need to obtain data or measurements from that component.

> A measurement is set of operations having the object of determining a value of a measure.
>
> ISO/IEC 15939 (2007)

Observe that we implicitly already used the notion of a measurement in Definition 5.4 through the relation $\mathbf{y} = \gamma(\mathbf{u})$. More formally, we obtain:

Definition 5.9 (Measurement).
Given a component $\gamma : \mathcal{U} \to \mathcal{Y}$ we call $\mathbf{y} \in \mathcal{Y}$ given by

$$\mathbf{y} = \gamma(\mathbf{u}) \tag{5.2}$$

measurement of the component.

> **Task 5.10**
> *State a measurement of a Runge-Kutta and a one-step method.*

> **Solution to Task 5.10**: The output of the Runge-Kutta method, i.e. endpoint of trajectory $\mathbf{x}(\overline{T})$, is the measurement obtained by executing Algorithm 1 or more generically Algorithm 5 for a one-step method.

Based on measurements, we can further introduce the notion of an evaluation method:

> An evaluation method is procedure describing actions to be performed by the evaluator in order to obtain results for the specified measurement applied to component.
>
> ISO/IEC DIS 25000 (2014)

Finally, we introduce the notion of quality evaluation.

> A quality evaluation is a systematic examination of the extent to which a component is capable of satisfying stated and implied needs.
>
> ISO/IEC DIS 25000 (2014)

Hence, the assessment of a component can be formalized by means of a quality evaluation. This leads to the following conclusion.

Corollary 5.11 (Test).
Consider a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Furthermore suppose specified conditions $\mathcal{C}$ to be fixed and the quality $Q : \mathcal{Y} \to \mathbb{R}_0^+$ to be given. Then the triple $(\gamma, \mathcal{C}, Q)$ is called a test.

The connections among these terms, together with their consequences for the definition of a test, are illustrated in Figure 5.3.

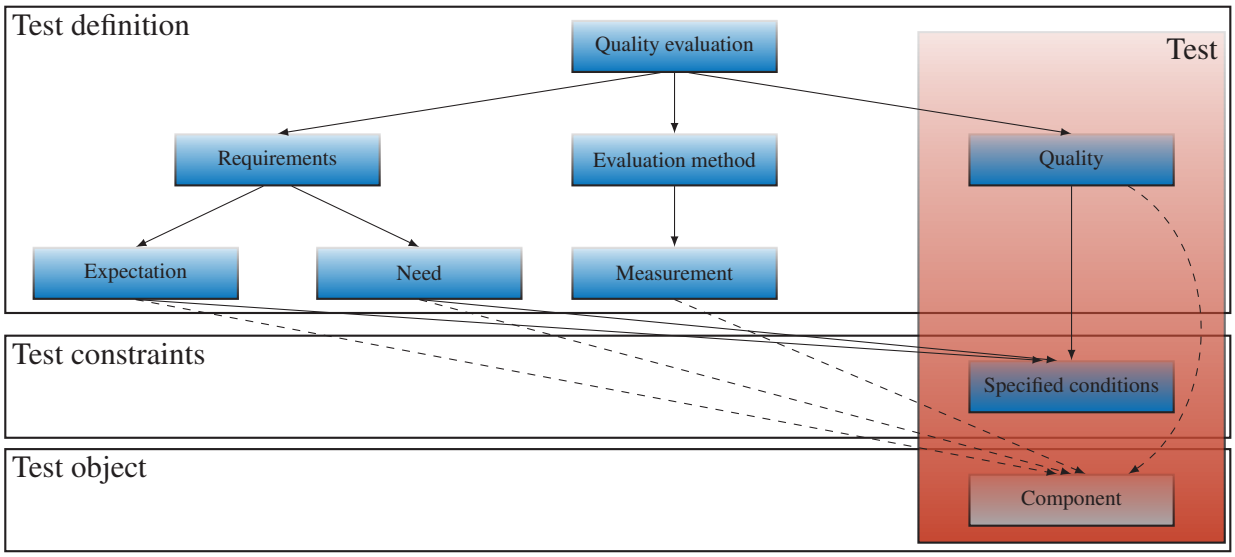

Figure 5.3: Connections of requirement terms

Returning to the right-hand side of Figure 5.1, we distinguish between unit tests, integration tests, and system tests. To formalize this distinction, we introduce the following definition:

Definition 5.12 (Unit).
Consider a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. If there exists no component $\gamma_2 \subsetneq \gamma$, then γ is called a *unit*.

Based on units, we can define:

Definition 5.13 (Unit/integration/system test).
Consider a test $(\gamma, \mathcal{C}, Q)$.

- If γ is a unit, then the test is called *unit test*.
- If $\gamma = \gamma_1 \cup \gamma_2$ where γ_1, γ_2 are units with $\gamma_1 \subsetneq \gamma$, $\gamma_2 \subsetneq \gamma$, then the test is called *integration test*.
- If $\gamma = \Theta$, then the test is called *system test*.

At this point, it is useful to highlight the following distinction regarding the outcome of tests, cf. ISO 9000 (2005):

- Test confirming (and providing objective evidence) that the requirements for a specific intended use or application have been fulfilled, are called *validation*.
- Verification is confirmation (including objective evidence) that specified requirements have been fulfilled.

Hence, in order to verify a system, it may be necessary to extend test cases derived from intended use or application to unintended use or misuse. This leads us to the derivation of test cases.

5.2 Derivation of test cases

To derive test cases, we start at the unit level. Recalling the definition of a unit, we observe that it maps inputs $\mathbf{u} \in \mathcal{U}$ to outputs $\mathbf{y} \in \mathcal{Y}$. Accordingly, a unit test amounts to checking whether the corresponding outputs are correct for the relevant inputs. A typical procedure comprises the following steps:

- Identify units: Units may be functions, methods, classes, or even smaller components depending on the chosen granularity.
- Understand requirements: Making needs and expectations clear helps to define the scope and purpose of a unit test.
- Define test: For each requirement, there should be exactly one test with one test set and one quality.
- Determine inputs/outputs: Collecting information on the unit help to limit the ranges of inputs and outputs.

- Write test code: Utilizing testing frameworks allows to efficiently test units.
- Execute tests: Run to validate correctness of unit.
- Analyze result: The outcome allows to check validity and identify errors/failures to investigate causes.
- Refine: Make the step towards verification.
- Maintain coverage: Unit test reflect standard behavior checks as the code gets larger.

In practice, the number of possible combinations of specified conditions $\mathcal{C}$ may be infinite, even in the simple case in which the input is just a scalar $\mathbf{u} \in [0,1] \subset \mathbb{R}$. To make testing tractable, the specified conditions can therefore be restricted by means of grids.

Definition 5.14 (Grid).
Consider specified conditions $\mathcal{C} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Suppose $n_{\mathcal{T}}, n_{\mathcal{X}}, n_{\mathcal{U}^{\mathbf{T}}}, n_{\mathcal{P}} \in \mathbb{N}$ to be number of sampling points for each set. Then we call

$$\mathcal{C} := \mathbb{T} \times \mathbb{X} \times \mathbb{U} \times \mathbb{P} \tag{5.3}$$

grid where

$$\begin{aligned}
\mathbb{T} &:= \{t_i \in \mathcal{T} \mid i = 1, \ldots, n_{\mathcal{T}}\} \\
\mathbb{X} &:= \{\mathbf{x}_i \in \mathcal{X} \mid i = 1, \ldots, n_{\mathcal{X}}\} \\
\mathbb{U} &:= \{\mathbf{u}_i \in \mathcal{U}^{\mathbf{T}} \mid i = 1, \ldots, n_{\mathcal{U}^{\mathbf{T}}}\} \\
\mathbb{P} &:= \{p_i \in \mathcal{P} \mid i = 1, \ldots, n_{\mathcal{P}}\}
\end{aligned}$$

are grids on the specified conditions.

A prototype of a test considering a grid of specified conditions is given by Algorithm 9.
We observe that the test prototype operates analogously to the simulation prototype by iterating over all specified conditions, that is, over all selected parameter combinations.

Remark 5.15
If the specified conditions are given by a finite set, then Algorithm 9 can also be used for full enumeration.

At the unit level, many tests are required in order to check whether even very simple functions such as GETVARIABLE(), SETVARIABLE(), or the initialization of storage and objects work as

Algorithm 9 Prototype of test

Input: Component γ
Input: Specified condition $\mathcal{C}$
Input: Quality Q
1: **procedure** CLASS TEST($\gamma, \mathcal{C}, Q$)
2: **for all** $\mathbf{u} \in \mathcal{C}$ **do**
3: $Q(\mathbf{u}) \leftarrow Q\,(\text{COMPONENT } \gamma(\mathbf{u}))$
4: **end for**
5: **end procedure**
Output: Quality $Q(\cdot)$

intended. For such functions, the test description $(\gamma, \mathcal{C}, Q)$ is usually much simpler. Nevertheless, these tests form the basis of what is commonly referred to as *continuous integration* and *continuous delivery*.

5.3 Continuous integration and delivery

Continuous integration, often abbreviated as CI, is a software development practice in which code changes from multiple developers are integrated regularly into a shared repository. The primary goal of CI is to detect integration issues early in the development process and to keep the software in a releasable state at all times. Before discussing continuous integration in more detail, we first clarify the associated terminology:

Definition 5.16 (Shared repository / version control system).
A *repository* is a storage location, which manages and tracks changes to files over time, provides shared access, collaboration and the option to revert to previous versions if needed. The management tool to work with a shared repository is called *version control system*.

In the context of continuous integration, Git-based toolchains allow the practical implementation of these principles. To this end, we define the notion of *releasable* as follows.

Definition 5.17 (Releasable).
We call a component $\gamma : \mathcal{U} \to \mathcal{Y}$ *releasable* if all related tests $(\gamma, \mathcal{C}, Q)$ satisfy the quality thresholds

$$Q(\gamma(\mathbf{u})) \geq \underline{Q} \qquad \forall \mathbf{u} \in \mathcal{C}. \tag{5.4}$$

To ensure the latter, continuous integration incorporates the following principles:

Definition 5.18 (Principles of continuous integration).
Aiming at code stability through revision loops while allowing for collaboration, continuous integration follows the principles of

- Maintaining a single source repository: Continuous integration emphasizes the use of a single, central version control repository where all developers commit their changes. This allows for easy and frequent integration of code changes and ensures that everyone is working with the latest version of the codebase.
- Automating the build process: The build process, which includes compiling the code and generating necessary artifacts, can be automated, e.g., via commit triggers. This ensures that the code is consistently built and reduces the chance of human error.
- Build in a clean environment: To avoid unexpected issues caused by inconsistent development environments, the build process should be performed in a clean and controlled environment. This helps to eliminate dependencies and ensures reproducibility.
- Keep builds fast: Obtaining a quick feedback on their changes quickly, developers can catch integration issues and errors early, enabling prompt resolution.
- Run automated tests: Automated testing is critical and includes unit tests, integration tests, and regression tests. These tests verify the correctness and quality of the codebase and help catch issues before they propagate further.
- Fail fast: If a build or test fails, it should be identified and communicated to promptly fix it.
- Provide feedback: Feedback helps developers to understand the impact of their changes and facilitates collaboration within the team. It enables quick identification and resolution of issues.
- Practice continuous integration: Regularly integrating changes helps to identify integration issues early and prevents code branches from diverging too much, reducing the effort required to merge them later.

As can already be seen from the preceding definition, continuous integration primarily concerns integration tests rather than isolated unit tests. However, meaningful integration testing presupposes that the involved units function correctly. This naturally leads to a hierarchy of tests, as depicted in Figure 5.4.

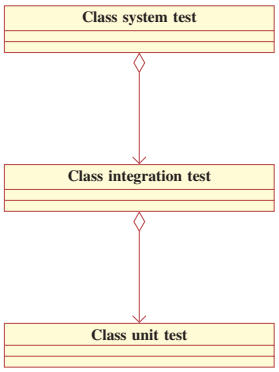

Figure 5.4: Sketch of UML diagram of test classes

Task 5.19
Apply the hierarchy of tests to the simulation setting from Figure 5.2 without output.

Solution to Task 5.19: We obtain the unit/integration/system tests sketched in Figure 5.5.

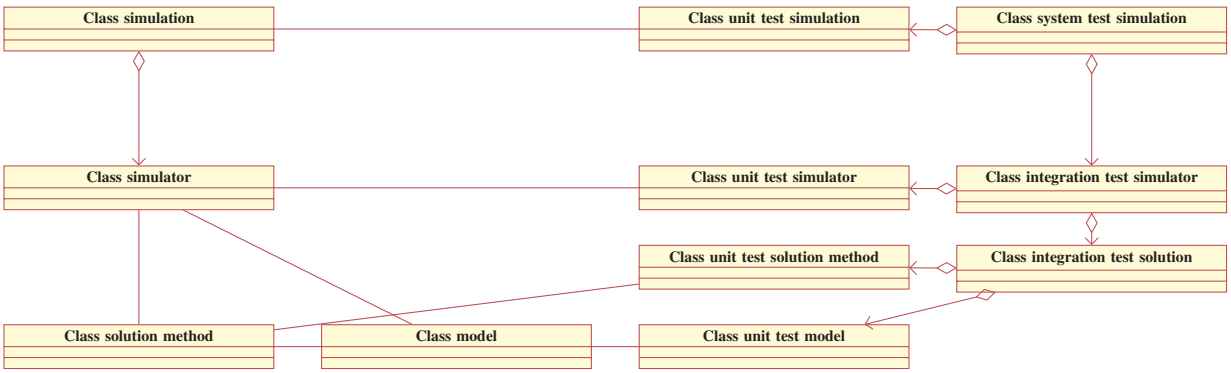

Figure 5.5: Sketch of UML diagram of class simulator with testing

The second part of the section title addresses the handover from development to operations and is commonly referred to as continuous delivery or continuous deployment. As discussed earlier in the context of DevOps, deployment requires releasability. Releasability, in turn, presupposes that all relevant tests have been passed, which motivates the following notion of *test-ready*.

Definition 5.20 (Test ready).
We call a component $\gamma : \mathcal{U} \to \mathcal{Y}$ *test ready* if the tests $(\gamma, \mathcal{C}, Q)$ cover needs and expectations and are practicable.

As can be seen, this notion combines two conditions: practicability and coverage. Practicability refers to the ability to execute tests within given constraints such as time, resources, and budget. It therefore concerns whether the testing objectives can realistically be achieved in the given context. Coverage, by contrast, refers to the extent to which a unit is exercised by a set of tests. It quantifies how much code, functionality, or how many requirements have been tested and thereby indicates the thoroughness and effectiveness of the testing process.

5.4 Automation of testing

Finally, we observe that once the test-ready property has been established, all tests required for determining releasability are available. In such a situation, these tests can be collected and executed in an automated manner.

Algorithm 10 Automation of test

Input: Simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$
Input: Tests $\{(\gamma_i, \mathcal{C}_i, Q_i)\}_{i=1,\dots,n_\gamma}$
1: **procedure** CLASS AUTOMATED TEST($\Theta, \{(\gamma_i, \mathcal{C}_i, Q_i)\}_{i=1,\dots,n_\gamma}$)
2: **for** $i = 1, \dots, n_\gamma$ **do**
3: $Q(\mathbf{u}_i) \leftarrow$ CLASS TEST$(\gamma_i, \mathcal{C}_i, Q_i)$
4: **end for**
5: **end procedure**
Output: Quality $Q(\cdot)$

Remark 5.21
Note that as indicated in Figure 5.5, each integration test aggregates unit tests and completes them with integration test sets, and each system test aggregates integration tests complemented by linking system test sets.

We emphasize that Algorithm 10 proceeds in an unstructured manner and simply executes one test after another. Nevertheless, if the automation is test ready, then the resulting test outcomes allow us to determine whether the simulation is releasable. This result is of particular importance in the context of continuous integration and continuous delivery. In practice, tools such as GitLab and related automation mechanisms make it possible to decide whether developers need to continue improving the simulation or whether it can be handed over to operations as part of a release.
To improve the automation, it is usually beneficial to distinguish explicitly between unit, integration, and system tests. One possible structure is outlined in Algorithm 11.
Algorithm 11 incorporates several ideas that help accelerate the assessment of releasability:

Algorithm 11 Automation of tests using test structures

Input: Simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$
Input: Tests $\{(\gamma_i, \mathcal{C}_i, Q_i)\}_{i=1,\ldots,n_\gamma}$
procedure CLASS AUTOMATED STRUCTURED TEST(Θ, $\{(\gamma_i, \mathcal{C}_i, Q_i)\}_{i=1,\ldots,n_\gamma}$)
  for $i = 1, \ldots, n_\gamma$ **do**
    if γ_i is unit **then**
      $S_{\text{unit}} \leftarrow \{(\gamma_i, \mathcal{C}_i, Q_i)\}$
    else if γ_i is integration **then**
      $S_{\text{integration}} \leftarrow \{(\gamma_i, \mathcal{C}_i, Q_i)\}$
    else
      $S_{\text{system}} \leftarrow \{(\gamma_i, \mathcal{C}_i, Q_i)\}$
    end if
  end for
  for all $(\gamma_i, \mathcal{C}_i, Q_i) \in S_{\text{unit}}, S_{\text{integration}}, S_{\text{system}}$ **do**
    $Q(\mathbf{u}_i) \leftarrow$ CLASS TEST$(\gamma_i, \mathcal{C}_i, Q_i)$
    if $Q(\mathbf{u}_i) < \underline{Q(\mathbf{u}_i)}$ **then return**
    end if
  end for
end procedure
Output: Quality $Q(\cdot)$

- To avoid unnecessary tests, we included the test fail criterion. Hence, if a test is not passed, then the algorithm will automatically terminate and not consider the remainder of tests as in Algorithm 10.

- The tests are structured to consider unit tests first. The idea behind this sequencing is that if a unit test is not passed, then it does not make sense to perform the integration test or system test.

Altogether, this lecture has shown that the simulation of mechatronic systems is far more than the numerical computation of trajectories. Starting from the role of simulation in the development process, we introduced formal system and model descriptions, derived numerical methods for the construction of simulators, and discussed how simulation results can be processed, interpreted, and analyzed in a meaningful way. Building on these foundations, requirements, test cases, and automated test structures complete the path from mathematical modeling to reliable engineering practice. In this sense, simulation becomes a systematic and reproducible tool for understanding, evaluating, and validating mechatronic systems throughout their development life cycle.

BIBLIOGRAPHY

[1] BULIRSCH, R. ; STOER, J.: *Numerische Mathematik 2*. Springer, 2005

[2] CHACON, S. ; STRAUB, B.: *Pro Git*. Apress, 2022

[3] COWELL, C. ; LOTZ, N. ; TIMERLAKE, C.: *Automating DevOps with GitLab CI/CD Pipelines*. Packt Publishing, 2023

[4] DEUFLHARD, P. ; BORNEMANN, F.: *Scientific computing with ordinary differential equations*. Springer, 2012

[5] DEUTSCHES INSTITUT FÜR NORMUNG E.V.: *Internationales Elektrotechnisches Wörterbuch Teil 351: Leittechnik (DIN IEC 60050-351:2014-09)*. Beuth, 2014

[6] FARLEY, D.: *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley Professional, 2022

[7] GLÖCKLER, M.: *Simulation mechatronischer Systeme: Grundlagen und technische Anwendung*. Springer, 2014

[8] IGLBERGER, K.: *C++ Software Design: Design Principles and Patterns for High-Quality Software*. O'Reilly Media, 2022

[9] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Systems and software engineering – Life cycle processes – Requirements engineering (ISO 29148:2011)*. ISO, 2011

[10] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Automation systems and integration – Key performance indicators (KPIs) for manufacturing operations management (ISO 22400:2014)*. ISO, 2014

[11] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE (ISO/IEC 25000:2014)*. ISO, 2014

[12] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Quality management systems — Fundamentals and vocabulary (ISO 9000:2015)*. ISO, 2015

[13] KHALIL, H.K.: *Nonlinear Systems*. Prentice Hall, 2002

[14] LUNZE, J.: *Automatisierungstechnik*. 5th edition. DeGruyter, 2020

[15] SONTAG, E.D.: *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, 1998

[16] VEREIN DEUTSCHER INGENIEURE E.V.: *VDI/VDE 2206 „Entwicklung mechatronischer und cyber-physischer Systeme“ (VDI/VDE 2206:2019)*. VDI, 2019

[17] WILKE, C.O.: *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O'Reilly Media, 2019

[18] ZIEGLER, B.P. ; MUZY, A. ; KOFMAN, E.: *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. Academic press, 2018

Jürgen Pannek
Institute for Intermodal Transport and Logistic Systems
Hermann-Blenck-Str. 42
38519 Braunschweig

During summer term 2026 I give the lecture to the module *Simulation of mechatronic systems (Simulation mechatronischer Systeme)* at the Technical University of Braunschweig. To structure the lecture and support my students in their learning process, I prepared these lecture notes.

The aim of the module is to classify simulation techniques from numerical mathematics and apply these to mechatronic case studies. After completing the module, the students shall be able to recall, categorize, apply, select and rate simulation methods to mechatronic use cases. Moreover, students shall be able to describe, explain, evaluate, analyze and assess simulation results. As such, students shall be capable to derive and apply automation procedures for deployment, simulation and testing of digital models.