Wrocław University of Science and Technology
Faculty of Electronics, Photonics and Microsystems

Field of Study: Electronics and Computer Engineering

BACHELOR THESIS

Title of Thesis:
Dynamic System Simulator

Author:
Kacper Różycki

Supervisor:

Dr inż. Robert Muszyński,
Department of Cybernetics and Robotics

WROCŁAW 2025

Abstract

This thesis presents the design and implementation of the *Dynamic System Simulator* (DSS), an educational software environment for modelling, simulating, and visualising continuous-time dynamical systems. The aim is to provide a transparent and extensible tool that links state–space equations directly to observable behaviour through interactive experiments, while remaining readable enough to be used as a learning resource.

The simulator is implemented in Python and follows a layered architecture that separates model definitions, controller logic, closed-loop composition, numerical integration, visualisation, and logging. All systems share a uniform state–space interface, with models defined by $\dot{x} = f(t, x, u)$ and controllers by $u = \pi(t, x)$. When closed-loop operation is required, a wrapper composes the plant and controller into an effective dynamics function integrated using adaptive ODE solvers (`scipy.integrate.solve_ivp`). A Streamlit-based graphical interface provides configuration of parameters, initial conditions, and solver settings, and presents results via time histories, phase portraits, and simple animations.

A representative library of models is included, spanning mechanical, electrical, electromechanical, and purely mathematical dynamics (e.g., pendulum systems, Van der Pol oscillator, DC motor, inverted cart–pole, and the Lorenz system), together with an energy-based swing up strategy and an LQR stabiliser for the cart–pole benchmark. The simulator is evaluated through qualitative behaviour checks, a numerical convergence study, and computational performance measurements, demonstrating reliable integration under practical tolerances and feasibility for interactive educational use.

Keywords: dynamical systems, state–space modelling, numerical integration, simulation, control, educational software, Python, Streamlit.

Streszczenie

Niniejsza praca przedstawia projekt i implementację *Dynamic System Simulator* (DSS) – edukacyjnego środowiska oprogramowania służącego do modelowania, symulacji i wizualizacji układów dynamicznych czasu ciągłego. Celem jest dostarczenie przejrzystego i rozszerzalnego narzędzia, które łączy równania przestrzeni stanów bezpośrednio z obserwowalnym zachowaniem poprzez interaktywne eksperymenty, pozostając jednocześnie wystarczająco czytelnym, by służyć jako materiał dydaktyczny.

Symulator zaimplementowano w języku Python zgodnie z architekturą warstwową, która rozdziela definicje modeli, logikę sterowania, kompozycję pętli zamkniętej, całkowanie numeryczne, wizualizację oraz logowanie danych. Wszystkie systemy współdzielą jednolity interfejs przestrzeni stanów, gdzie modele zdefiniowane są jako $\dot{x} = f(t, x, u)$, a regulatory jako $u = \pi(t, x)$. Gdy wymagana jest praca w pętli zamkniętej, moduł typu wrapper składa obiekt i regulator w wypadkową funkcję dynamiki, całkowaną przy

użyciu adaptacyjnych solverów ODE z biblioteki SciPy (funkcja `solve_ivp`). Graficzny interfejs oparty na bibliotece Streamlit umożliwia konfigurację parametrów, warunków początkowych i ustawień solvera, a wyniki prezentuje poprzez przebiegi czasowe, portrety fazowe oraz proste animacje.

Dołączono reprezentatywną bibliotekę modeli obejmującą dynamikę mechaniczną, elektryczną, elektromechaniczną i czysto matematyczną (m.in. systemy wahadłowe, oscylator Van der Pola, silnik DC, odwrócone wahadło na wózku oraz układ Lorenza), wraz z energetyczną strategią *swing-up* i stabilizatorem LQR dla zadania cart–pole. Symulator poddano ewaluacji poprzez jakościową weryfikację zachowania, badanie zbieżności numerycznej oraz pomiary wydajności obliczeniowej, wykazując wiarygodne całkowanie w ramach praktycznych tolerancji oraz przydatność do interaktywnych zastosowań edukacyjnych.

Słowa kluczowe: układy dynamiczne, modelowanie w przestrzeni stanów, całkowanie numeryczne, symulacja, sterowanie, oprogramowanie edukacyjne, Python, Streamlit.

Contents

Chapter 1

Introduction

Modeling and simulation are among the most important tools in modern engineering. They allow complex physical systems to be analysed, tested and understood before they are implemented in practice. In the domain of dynamics, simulation is particularly valuable because the behaviour of even simple models depends strongly on parameters, initial conditions and nonlinear effects. Numerical trajectories, phase portraits and animations provide an immediate bridge between abstract equations and the time-dependent motion they describe, and therefore form a natural complement to the analytical derivations typically presented in courses on mechanics and control.

At the same time, many educational settings still rely either on purely symbolic work or on closed simulation environments that hide key modelling and numerical details. Students often learn to manipulate formulas without developing an intuition for how changes in parameters translate into qualitative changes of motion, stability, oscillations or chaotic regimes. When interactive tools are available, they are frequently tied to proprietary ecosystems or to fixed libraries of components, which reduces transparency and makes it difficult to treat the simulator itself as an object of study. The motivation for this thesis is to address these limitations by providing a small but complete simulation environment that remains open, readable and easy to extend, while still supporting a range of representative dynamical phenomena.

The objective of the thesis is the development of a *Dynamic System Simulator* that enables experimentation with several mechanical, electrical and electromechanical models through a uniform workflow. The simulator is intended to expose the key ingredients that appear in essentially every continuous-time simulation task: the mathematical model in state–space form, a numerical integration method with controlled accuracy, and a visual layer that turns computed trajectories into plots and animations. From an educational perspective, the goal is not only to compute solutions, but to make the modelling and simulation pipeline explicit and modifiable, so that students can connect equations of motion to observable behaviour by directly changing parameters, initial conditions and (when applicable) controller settings and then comparing the resulting trajectories.

In addition to the engineering goal of implementation, the thesis also includes a small research component: a concise comparison of commonly used simulation environments

and a justification of the technology choices used for the final software. Several popular tools (including MATLAB/Simulink, Mathematica and equation-based environments) offer mature solvers and visual tooling, but they are either commercial, less accessible for students, or less suitable for a transparent code-oriented extension workflow [The25a, The25b, Wol25a, Wol25b]. For this reason Python was selected as the main implementation platform [Pyt25]. The choice is motivated by the open-source character of the ecosystem, the availability of robust scientific libraries for numerical integration and plotting, and the practical advantage that students can install and modify the software without licensing constraints.

The implementation developed in this work is organised around a layered architecture that separates concerns and keeps interfaces stable. Each dynamical system is represented as a model object providing a right-hand side of the form $\dot{x} = f(t, x, u)$ together with auxiliary metadata used for plotting and (when relevant) planar animation. Controllers are represented independently as objects implementing an input law $u = \pi(t, x)$. When a closed-loop experiment is required, a wrapper component combines a plant model and a controller into a single effective dynamics function, so that the numerical solver always integrates an object with the same interface. This design localises coupling logic, keeps the numerical core independent of the physical interpretation of the state, and allows new models and control strategies to be added without changing the solver. A dedicated logging mechanism stores configuration data together with numerical results, which supports reproducibility and makes it possible to repeat experiments under exactly the same settings.

The numerical core of the simulator is built on standard adaptive ODE solvers and therefore supports a controlled accuracy–cost trade-off. For interactive use the solver must be fast enough to update plots in response to parameter changes, while still providing trajectories that are reliable for didactic interpretation. For this reason the evaluation in this thesis does not treat numerical integration as a black box: it includes convergence experiments in which solver tolerances are varied, and performance tests in which runtime and solver work are measured across representative models. The results justify a practical default configuration that achieves a suitable balance between speed and accuracy for typical educational experiments.

An important part of the simulator is the graphical user interface. While scripts remain useful for reproducible batch experiments and figure generation, a GUI provides the most direct educational value because it enables rapid exploration. In this work the GUI is implemented in Streamlit, which allows a complete interactive application to be written in Python and tightly coupled to the simulation core [Str25]. The GUI acts as a thin configuration and presentation layer: it collects user inputs through widgets, constructs configuration structures, dispatches them to runner functions that instantiate the chosen model and controller, runs the solver, and then presents the results through consistent interactive plots and animations.

The scope of the thesis is deliberately focused on continuous-time systems described by ordinary differential equations and on models that are representative in introductory

courses on dynamics and control. The implemented library spans several characteristic behaviours: small oscillations and energy exchange in pendulum models, self-sustained nonlinear oscillations in the Van der Pol system, sensitive dependence on initial conditions in chaotic examples such as the double pendulum and the Lorenz system, and a classical control benchmark in the form of the inverted pendulum on a cart with a combined swing-up and stabilisation workflow. These examples are not chosen merely as demonstrations of software functionality: they form a coherent set of teaching scenarios in which students can observe how modelling assumptions, parameter regimes and control laws change the qualitative behaviour of a system.

The remainder of the thesis is structured to connect theory, design, implementation and validation in a progressive way. Chapter 2 introduces the theoretical foundations used throughout the work, including the state–space representation of dynamical systems, basic properties such as equilibria and stability, and the role of numerical integration in producing computable trajectories. Chapter 3 presents the system design and architecture, formalising the interfaces between models, controllers, wrappers, solver, visualisation and logging. Chapter 4 collects the mathematical models implemented in the simulator and derives their state–space forms in a consistent notation. Chapter 5 describes the concrete software implementation, repository organisation and the Streamlit-based graphical interface. Chapter 6 evaluates the simulator both numerically and educationally, including qualitative behaviour checks, convergence and runtime studies, and example case studies demonstrating interactive use through the GUI. Chapter 7 concludes the thesis by summarising the outcomes, discussing limitations and outlining possible future extensions.

Chapter 2

Theoretical Foundations and Related Work

2.1 Introduction: Why Study Dynamic Systems

Every physical, biological, and engineered process evolves in time. Describing this evolution is the task of dynamic system theory. It provides the language for understanding change, prediction, and control. The same formal principles govern mechanical motion, electrical transients, thermal transfer, and even population growth. By abstracting from material form to mathematical structure, the theory makes it possible to model, analyse, and ultimately reproduce the behaviour of complex systems within computational environments.

This chapter introduces the conceptual and mathematical foundations on which the present work is based. It outlines how real phenomena are represented as dynamic systems, how their temporal behaviour is computed numerically, and how such computations are organised in simulation environments. The discussion remains general and independent of any specific implementation, providing a coherent theoretical framework for the simulator architecture that follows in Chapter 3.

2.2 Dynamic Systems: From Phenomenon to Model

A system is called *dynamic* when its present state determines its future. Formally, there exists a set of variables $x(t)$ describing the internal condition of the system, and a rule that governs how these variables change in time. The fundamental relation can be written as

$$\begin{cases} \dot{x} = f(t, x, u) \\ y = g(t, x, u) \end{cases} . \tag{2.1}$$

Here, u denotes external inputs acting on the system, y denotes the measurable outputs, f is the state-transition function governing $\dot{x}$, and g is the output function mapping the current state and input to observable quantities. Together, the functions f and g form the *state–space representation*, a canonical mathematical model for dynamic systems [Kha02, SK16].

Different categories of systems can be distinguished according to the properties of the function f. If f depends linearly on its arguments, the system is linear; otherwise it is nonlinear. When f does not explicitly depend on time, the system is time–invariant; otherwise it is time–varying. If the future trajectory of $x(t)$ is uniquely determined by its initial condition and input, the system is deterministic; if it also depends on random influences, it is stochastic. Such classifications indicate the mathematical complexity and the choice of analysis tools [Kha02].

The notion of state encapsulates all information necessary for predicting future behaviour. Inputs act as external causes that modify this evolution, while outputs represent the quantities accessible to observation. The mapping from physical variables to these abstract sets is the process of modelling. Once this correspondence is established, the physical system and its mathematical representation become equivalent for analysis [Kha02].

In physics, the governing equations (2.1) arise from fundamental balance relations among forces, fluxes, or potentials. Classical mechanics formulates these relations through Newton's laws of motion, describing the evolution of systems under the influence of forces [TM13].

In the domain of electrical systems, the dynamics are described by potentials and fluxes [NR14]. Let $I_k(t)$ denote signed branch currents incident on a node and $V_\ell(t)$ denote signed voltage drops along a closed loop. Kirchhoff's current law (KCL) states that the algebraic sum of currents at each node is zero, and Kirchhoff's voltage law (KVL) states that the algebraic sum of voltages along any closed loop is zero:

$$\sum_{k=1}^{n} I_k(t) = 0, \tag{2.2}$$

$$\sum_{\ell=1}^{m} V_\ell(t) = 0. \tag{2.3}$$

These balance relations express conservation of electric charge and energy for ideal circuit elements. Together with the constitutive relations of individual components, they lead to systems of differential equations consistent with the general state–space form (2.1).

Defining the Lagrangian [TM13]

$$L(q, \dot{q}, t) = T(q, \dot{q}) - V(q), \tag{2.4}$$

as the difference between kinetic and potential energy, the principle of stationary action with q denoting system configuration vector, yields the Euler–Lagrange equations

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = Q, \tag{2.5}$$

where Q represents generalised external forces or torques. This relation provides a general starting point for deriving the dynamics of mechanical and electromechanical systems [TM13].

Building on (2.5), the motion of many systems can be expressed in the generic second–order form

$$M(q)\,\ddot{q} + B(q,\dot{q}) = \tau, \tag{2.6}$$

where $M(q)$ is the inertia matrix and $B(q,\dot{q})$ collects velocity-dependent, gravitational, and dissipative terms [SK16]. For numerical integration, this second-order equation is rewritten as two first-order relations,

$$\begin{cases} \dot{q} = v, \\ \dot{v} = M(q)^{-1}\big(\tau - B(q,v)\big) \end{cases}, \tag{2.7}$$

which define the evolution of q and $\dot{q}$ [Kha02]. Introducing the state vector $x = [q, \dot{q}]^T$ yields a first-order system consistent with the general state-space structure of Eq. (2.1) [Kha02].

Dynamic systems exhibit several characteristic properties such as equilibria, stability, and energy behaviour. Equilibria satisfy $\dot{x} = 0$, stability describes sensitivity to perturbations, and damping or resistance leads to dissipation. These features together determine whether a system oscillates, converges, or diverges in time [Kha02]. Only the simplest differential equations admit closed-form solutions. For most practical models, analytical integration is impossible, and numerical methods are required [HNW93].

2.3 Numerical Integration: Bridging Model and Computation

Numerical integration replaces continuous time by a discrete sequence of moments and approximates the derivative $\dot{x}$ through finite differences. Given an initial condition $x(t_0) = x_0$ and the system dynamics in Eq. (2.1), the objective is to compute an ordered set of approximations representing the system trajectory. The fundamental idea is that, over sufficiently small intervals, the derivative remains nearly constant [HNW93].

The simplest discretisation assumes

$$x_{k+1} = x_k + h\,f(t_k, x_k), \tag{2.8}$$

where h is the integration step. This explicit Euler method illustrates stepwise propagation but becomes unstable for stiff or oscillatory systems [HNW93].

To reduce these effects, higher–order methods evaluate the derivative at several points within each interval. Runge–Kutta schemes compute intermediate slopes and combine them into a weighted average. Multistep methods use solutions from previous steps. Implicit algorithms evaluate f at the unknown future state x_{k+1}, offering enhanced stability at the cost of solving nonlinear equations [HNW93].

Local and global truncation errors describe how numerical approximations deviate from exact solutions. Stability determines whether such errors remain bounded. Energy-preserving algorithms known as symplectic integrators are essential for long-term simulation of mechanical systems [HLW06].

Numerical integration forms the core of every simulation. The solver iteratively evaluates $f(t, x)$, updates the state, and records the results. This transforms the abstract model into computable trajectories [HNW93].

2.4 Simulation Paradigms and Environments

Simulation is the algorithmic execution of a model in a virtual domain. Equation-based modelling describes systems directly through their differential relations. Block-diagram modelling represents them as networks of interconnected components. Code-based modelling expresses them through programmable structures. These paradigms are standard in engineering modelling and simulation [SK16].

A simulator requires components defining the model, computing its evolution, and presenting results. Modularity, solver independence, transparency, and reproducibility form the methodological backbone of modern simulation environments [SK16].

Modern computing ecosystems allow models, solvers, and visualisation tools to be implemented in unified frameworks. Their interoperability supports the complete simulation workflow efficiently [SK16].

Chapter 3

System Design and Architecture

3.1 Choice of Simulation Environment

Several computational environments were considered for implementing the simulator, including MATLAB/Simulink, Mathematica and Modelica [The25a, The25b, Wol25a, Mod25]. Each of these offers advanced numerical solvers and mature visualization capabilities. However, they are either commercial or rely on proprietary formats, which makes them less accessible for students and educational use.

To ensure openness and portability, the simulator was implemented in Python, which is free and open-source and supports object-oriented design [Pyt25]. Python offers a mature ecosystem of scientific libraries, including NumPy [HMvdW+20], SciPy [V+20] and Matplotlib [Hun07]. Although developing a custom simulator in Python may lead to reduced numerical efficiency compared to specialized tools, this trade-off is acceptable given the educational nature of the project.

In summary, Python was chosen as the implementation platform for its openness, extensibility, and alignment with the didactic objectives of the work. The following section outlines the functional and non-functional requirements that guided the design of the simulator within this environment.

3.2 System Requirements and Design Goals

Given the selection of Python as the implementation environment, the simulator was designed around a set of clear objectives that define its expected functionality and quality attributes.

At the functional level, the system must be capable of representing a broad class of dynamic models expressed in state-space form, integrating them numerically over time, and presenting the results in a graphical form. The user should be able to adjust parameters, change initial conditions, and observe how these modifications influence the system's behavior. The simulator should therefore allow new models or controllers to be introduced easily through well-defined interfaces, without modification to the underlying solver or visualization code.

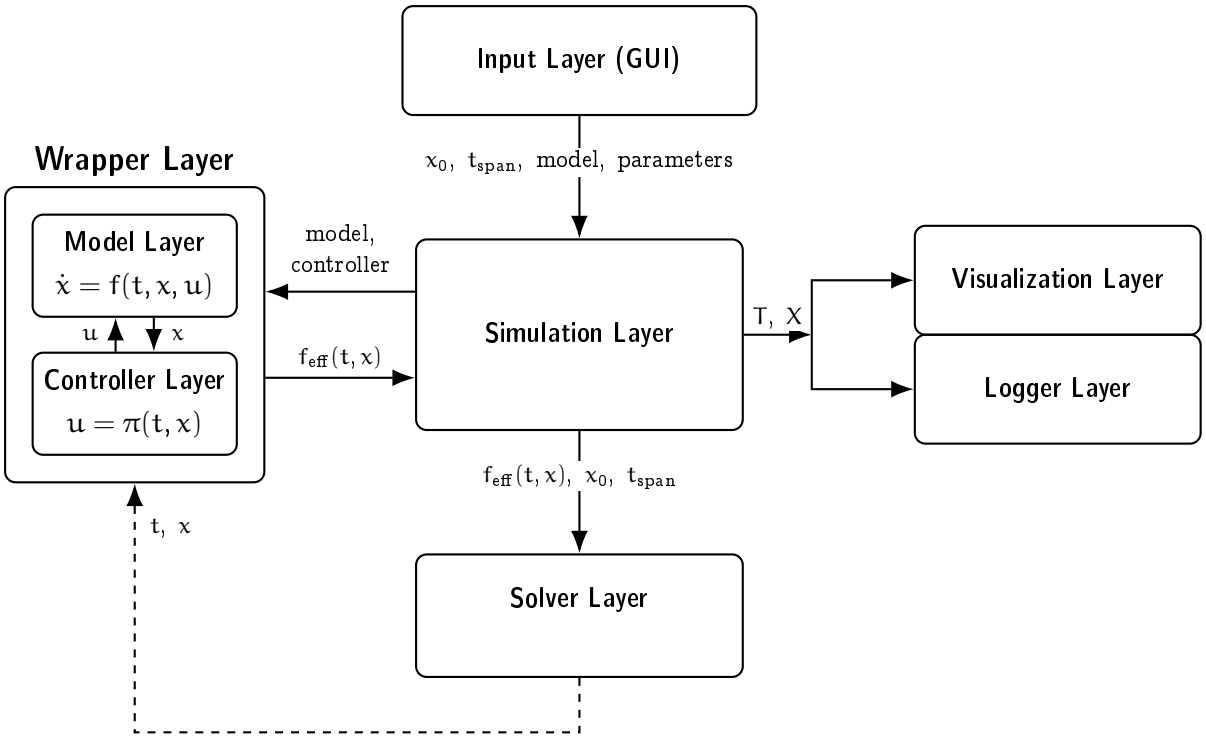


Figure 3.1 Overall architecture of the *Dynamic System Simulator*

Beyond functionality, several non-functional goals were established. The simulator is intended as an educational tool; therefore, simplicity and transparency were prioritized over computational performance. The code must remain readable and self-explanatory so that students can follow the logic of simulation step by step. Reproducibility is another core requirement: all simulation parameters and results should be logged automatically, allowing each experiment to be repeated exactly. Finally, the software must remain portable and open-source, relying solely on freely available Python packages.

3.3 Architecture of the Simulator

The workflow of the *Dynamic System Simulator*, illustrated in Figure 3.1, follows a sequence of stages aligned with its software layers. A simulation begins in the *Input Layer (GUI)*, where the user specifies the model type, controller type, parameters, initial state x_0, and simulation horizon t_{span}. These settings are passed to the *Simulation Layer*, which constructs the *Model Layer* (implementing $\dot{x} = f(t, x, u)$) and the *Controller Layer* (implementing $u = \pi(t, x)$), and bundles them into the *Wrapper Layer*. The wrapper links the two components into a single closed-loop unit and provides a uniform interface for evaluating the system dynamics. Composite physical models, such as a DC motor coupled with a pendulum, are assembled before being wrapped together with their controller.

During execution, the Simulation Layer supplies the solver with the closed-loop dynamics $f_{\text{eff}}(t, x)$, together with x_0 and t_{span}. As integration proceeds, the *Solver Layer* advances time and, at each step, requests a new value of $f_{\text{eff}}(t, x)$ by returning the current time t and state x. In implementation this feedback is routed through the Simulation Layer, but in Figure 3.1 it is shown as a dashed line directed toward the Wrapper Layer to emphasise the conceptual flow. Upon receiving (t, x), the wrapper invokes the controller to compute $u = \pi(t, x)$, applies this input to the model to obtain $f(t, x, u)$, and returns the result to the Simulation Layer, which reconstructs $f_{\text{eff}}(t, x)$ and passes it to the solver. Meanwhile, the Simulation Layer streams the accumulated trajectory (T, X) to the *Visualization Layer* and *Logger Layer*, which handle rendering and data storage.

3.4 Core Components and Their Interaction

Input Layer

The Input Layer defines all simulation parameters and initial conditions through either a graphical or script interface. It generates a structured configuration object that separates user interaction from internal computation and guarantees reproducibility of results.

Simulation Layer

The Simulation Layer serves as the central coordinator of the simulation process. It initializes the model and controller, constructs the wrapper, and obtains from it the effective dynamics function $f_{\text{eff}}(t, x)$, which represents the closed-loop behaviour of the system. This function, together with the initial state x_0 and simulation time span, is passed to the solver for numerical integration. During execution, the Simulation Layer mediates communication between the wrapper and solver: it forwards the solver's current time t and state x to the wrapper and returns the computed derivatives for integration. After the simulation concludes, the layer transfers the resulting trajectories to the visualization and logging modules. By managing data exchange among components, it defines the simulator's operational sequence while keeping all subsystems independent.

Model Layer

The Model Layer encapsulates the dynamics of a physical or electromechanical system. Each model defines how its state evolves in time in response to the current configuration, velocity, and external or control inputs. The internal mathematical formulation, which may come from Newton–Euler laws, the Euler–Lagrange equations, or reduced first-order models, is handled entirely within the layer and remains independent of the simulation architecture.

For consistency, every model exposes a standard interface of the form

$$\dot{x} = f(t, x, u),$$

which corresponds to the general state-space equation introduced in Eq. (2.1). This allows the Simulation Layer and Solver Layer to treat all systems in a uniform way.

Controller Layer

The Controller Layer implements the regulation law that generates control signals from the current state:

$$u(t) = \pi(t, x).$$

Controllers may be feedback, feedforward, or hybrid in structure. They are fully decoupled from the model's internal formulation and interact with it only through this standard interface, which the wrapper accesses at each solver step.

Wrapper Layer

The Wrapper Layer fuses the model and controller into a single callable function:

$$f_{\text{eff}}(t, x) = f\big(t, x, \pi(t, x)\big).$$

At each solver request, it invokes the controller to compute $u(t, x)$ and passes this input to the model to evaluate $f(t, x, u)$. The resulting $f_{\text{eff}}(t, x)$ is returned to the Simulation Layer for integration. When composite systems are used, the wrapper maintains consistent variable ordering and parameter mapping, preserving modularity without altering system behaviour.

Solver Layer

The Solver Layer performs numerical integration of $\dot{x} = f_{\text{eff}}(t, x)$ using adaptive step sizes. It monitors local error and stability while remaining independent of the physical interpretation of the equations. Because it operates on the general function $f_{\text{eff}}(t, x)$, different numerical schemes can be used interchangeably without affecting higher-level components.

Visualization Layer

The Visualization Layer reconstructs trajectories, control signals, and derived quantities from the time vector T and state matrix X. It provides clear graphical representations of system behaviour for qualitative analysis and remains independent of the computational core.

Logger Layer

The Logger Layer records configuration data, parameters, and results in structured, portable files. Its outputs enable precise replication and comparison of simulations. Together with visualization, it completes the information flow on the right side of Figure 3.1, combining immediate feedback with documentation.

Chapter 4

Mathematical Models and System Dynamics

This chapter presents the mathematical models implemented in the *Dynamic System Simulator*. Its purpose is to connect the general framework of dynamic systems introduced in Chapter 2 with the specific equations used by the software. The models cover a range of mechanical, electrical and electromechanical systems. Each system is expressed in state–space form and is consistent with the simulation architecture described in Chapter 3.

4.1 General Modelling Framework

All systems used in the simulator are described by the state–space representation (2.1). In this work the simulator is concerned primarily with the internal behaviour of the systems, and all components of the state are available for inspection. For this reason the output function is chosen as the identity, obtaning the system

$$\begin{cases} \dot{x} = f(t, x, u) \\ y = x \end{cases}, \tag{4.1}$$

which is a special case of (2.1).

Mechanical models are derived from the Lagrangian [TM13]

$$L(q, \dot{q}) = T(q, \dot{q}) - V(q), \tag{4.2}$$

where q is the vector of system generalised coordinates, T is the system kinetic energy and V is its potential energy. Computing the Euler-Lagrange equation (2.5), for many systems the resulting equations of motion can be written in the generic second-order form (2.6), which corespond to first-order relation (2.7). Introducing the state vector $x = (q^T, v^T)^T$ yields a first–order representation consistent with (4.1). In the following sections this framework is applied to all the systems.

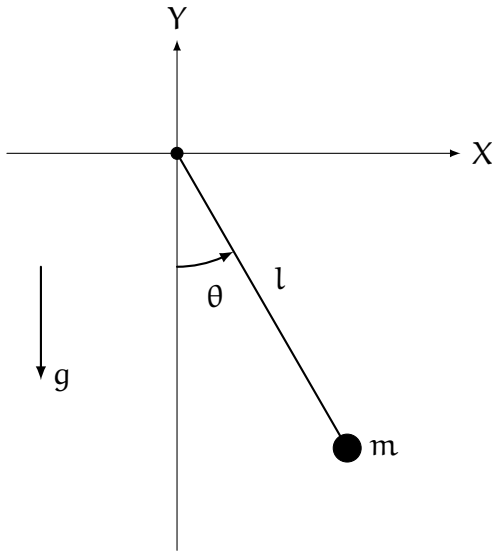


Figure 4.1 Geometry of the single pendulum model

4.2 Single Pendulum

The single pendulum is one of the simplest nonlinear mechanical systems [Wik25g]. It serves as an independent oscillatory model and as a subsystem in more complex configurations used by the simulator. Figure 4.1 illustrates its geometry: a point mass m is attached to a rigid, massless rod of length l and hinged at one end. The angle θ is measured from the downward vertical, so that $\theta = 0$ corresponds to the stable hanging equilibrium.

Under the point–mass assumption the moment of inertia of the pendulum about the pivot is

$$I = ml^2. \tag{4.3}$$

The kinetic energy is therefore

$$T(\theta, \dot{\theta}) = \frac{1}{2} I \dot{\theta}^2 = \frac{1}{2} ml^2 \dot{\theta}^2. \tag{4.4}$$

The potential energy is

$$V(\theta) = -mgl \cos\theta, \tag{4.5}$$

where g denotes the magnitude of gravitational acceleration; gravity acts along the Y axis opposite to its positive orientation, which is reflected in the sign of V.

The Lagrangian is defined as

$$L(\theta, \dot{\theta}) = T(\theta, \dot{\theta}) - V(\theta) = \frac{1}{2} ml^2 \dot{\theta}^2 + mgl \cos\theta. \tag{4.6}$$

Computing (2.5) and substituting (4.3) for the single coordinate θ yields

$$ml^2 \ddot{\theta} + mgl \sin\theta = Q_\theta, \tag{4.7}$$

where Q_θ is the generalised torque acting at the pivot.

In the simulator this torque has three contributions: an externally applied control torque, viscous damping and Coulomb friction and it is modelled as

$$Q_\theta = \tau(t) - b\dot{\theta} - c\ \mathrm{sgn}(\dot{\theta}), \tag{4.8}$$

where $\tau(t)$ denotes the control torque, $b \geqslant 0$ is the viscous damping coefficient, $c \geqslant 0$ is the Coulomb friction coefficient and $\mathrm{sgn}(\cdot)$ is the sign function representing dry friction. Substituting (4.8) into (4.7) leads to the second–order equation

$$ml^2\ddot{\theta} + b\dot{\theta} + c\ \mathrm{sgn}(\dot{\theta}) + mgl\sin\theta = \tau(t). \tag{4.9}$$

This equation is a scalar instance of the generic second–order structure (2.6). It can be written as

$$M(\theta)\,\ddot{\theta} + B(\theta,\dot{\theta}) = \tau(t), \tag{4.10}$$

with constant inertia

$$M(\theta) = ml^2, \tag{4.11}$$

and remaining terms collected in

$$B(\theta,\dot{\theta}) = mgl\sin\theta + b\dot{\theta} + c\ \mathrm{sgn}(\dot{\theta}). \tag{4.12}$$

This interpretation makes the relation to the general form (2.6) explicit and will be mirrored in the multi–link case.

For numerical simulation the second–order equation (4.9) is converted to a first–order system using the conversion (2.7). The state vector is defined as

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \theta \\ \dot{\theta} \end{pmatrix}. \tag{4.13}$$

The corresponding state–space model $\dot{x} = f(t,x,u)$ has the form

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} x_2 \\ \dfrac{1}{ml^2}\left(\tau(t) - mgl\sin x_1 - bx_2 - c\ \mathrm{sgn}(x_2)\right) \end{pmatrix}. \tag{4.14}$$

This equation fits directly into the framework of (4.1) with state $x = (\theta,\dot{\theta})^T$ and input $u = \tau$ and is the form implemented in the simulator.

4.3 Double Pendulum

The planar double pendulum extends the single pendulum to two links connected in series and exhibits rich nonlinear behaviour [Wik25b]. In the simulator it serves as an example of a simple two–link mechanism. Figure 4.2 shows the configuration: the first link of length l_1 is hinged at a fixed pivot, and the second link of length l_2 is attached to its end. The link masses m_1 and m_2 are concentrated at the ends of the rods, which

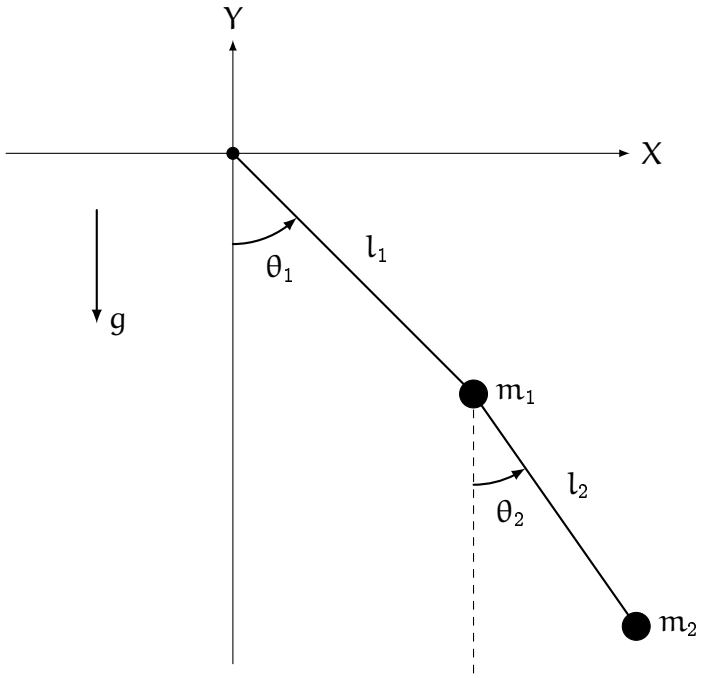


Figure 4.2 Geometry of the planar double pendulum model

are treated as massless. The angles θ_1 and θ_2 are measured from the downward vertical, in analogy with the single pendulum.

The generalised coordinates and velocities are

$$q = \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} \qquad \dot{q} = \begin{pmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{pmatrix}. \tag{4.15}$$

Under the point–mass assumption, the squared magnitudes of the velocities of the two masses are

$$v_1^2 = l_1^2 \dot{\theta}_1^2, \qquad v_2^2 = l_1^2 \dot{\theta}_1^2 + 2 l_1 l_2 \cos\Delta\, \dot{\theta}_1 \dot{\theta}_2 + l_2^2 \dot{\theta}_2^2, \tag{4.16}$$

with $\Delta = \theta_1 - \theta_2$.

The total kinetic energy is therefore

$$T = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2, \tag{4.17}$$

which, after substitution of (4.16), gives

$$T(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) = \frac{1}{2}\Big((m_1 + m_2) l_1^2\, \dot{\theta}_1^2 + 2 m_2 l_1 l_2 \cos\Delta\, \dot{\theta}_1 \dot{\theta}_2 + m_2 l_2^2\, \dot{\theta}_2^2 \Big). \tag{4.18}$$

The potential energy can be written as

$$V(\theta_1, \theta_2) = -(m_1 + m_2) g l_1 \cos\theta_1 - m_2 g l_2 \cos\theta_2, \tag{4.19}$$

where g denotes the magnitude of gravitational acceleration; gravity acts along the Y axis opposite to its positive orientation, which is reflected in the sign of V.

Consquently, the Lagrangian is $L(q,\dot{q}) = T(q,\dot{q}) - V(q)$. From the kinetic energy (4.18) one obtains the inertia matrix $M(q)$ by matching the quadratic form

$$T(q,\dot{q}) = \frac{1}{2}\dot{q}^{T} M(q)\dot{q}, \tag{4.20}$$

which yields

$$M(q) = \begin{bmatrix} (m_1+m_2)l_1^2 & m_2 l_1 l_2 \cos\Delta \\ m_2 l_1 l_2 \cos\Delta & m_2 l_2^2 \end{bmatrix}. \tag{4.21}$$

The gravity vector follows from (4.19) as

$$G(q) = \begin{bmatrix} (m_1+m_2) g l_1 \sin\theta_1 \\ m_2 g l_2 \sin\theta_2 \end{bmatrix}. \tag{4.22}$$

In addition to gravity, the equations of motion contain velocity–dependent coupling terms and dissipative effects. From Euler-Lagrange equation we obtain centrifugal forces

$$B_{cc}(q,\dot{q}) = \begin{bmatrix} m_2 l_1 l_2 \sin\Delta\, \dot{\theta}_2^2 \\ -m_2 l_1 l_2 \sin\Delta\, \dot{\theta}_1^2 \end{bmatrix}, \tag{4.23}$$

with $\Delta = \theta_1 - \theta_2$. Viscous damping at the joints is modelled by the term

$$B_v(\dot{q}) = \begin{bmatrix} b_1 & 0 \\ 0 & b_2 \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} = \begin{bmatrix} b_1\dot{\theta}_1 \\ b_2\dot{\theta}_2 \end{bmatrix}, \tag{4.24}$$

where $b_1, b_2 \geqslant 0$ are damping coefficients. Coulomb friction is represented by

$$B_c(\dot{q}) = \begin{bmatrix} c_1\ \mathrm{sgn}(\dot{\theta}_1) \\ c_2\ \mathrm{sgn}(\dot{\theta}_2) \end{bmatrix}, \tag{4.25}$$

with non–negative coefficients c_1, c_2. The centrifugal terms together with gravity and damping are collected in the vector

$$B(q,\dot{q}) = \begin{bmatrix} B_1(q,\dot{q}) \\ B_2(q,\dot{q}) \end{bmatrix}. \tag{4.26}$$

For the point–mass double pendulum these components take the explicit form

$$\begin{aligned} B_1(q,\dot{q}) &= (m_1+m_2) g l_1 \sin\theta_1 + \big(B_{cc}(q,\dot{q})\big)_1 + b_1\dot{\theta}_1 + c_1\ \mathrm{sgn}(\dot{\theta}_1), \\ B_2(q,\dot{q}) &= m_2 g l_2 \sin\theta_2 + \big(B_{cc}(q,\dot{q})\big)_2 + b_2\dot{\theta}_2 + c_2\ \mathrm{sgn}(\dot{\theta}_2). \end{aligned} \tag{4.27}$$

Thus $B(q,\dot{q})$ combines gravitational effects (4.22), the centrifugal couplings (4.23) and both viscous and Coulomb friction (4.24), (4.25) in a single term. Let

$$\tau(t) = \begin{pmatrix} \tau_1(t) \\ \tau_2(t) \end{pmatrix} \tag{4.28}$$

denote the generalised torques applied at the joints. By the Euler–Lagrange equation (2.5) the double pendulum satisfies the standard second–order form (2.6) with the matrices $M(q)$ and $B(q, \dot{q})$ defined above. Written componentwise, the equations of motion for the angles are

$$\begin{cases} (m_1 + m_2) l_1^2 \ddot{\theta}_1 \; + \; m_2 l_1 l_2 \cos\Delta \, \ddot{\theta}_2 \; + \; B_1(q, \dot{q}) = \tau_1(t) \\ m_2 l_1 l_2 \cos\Delta \, \ddot{\theta}_1 \; + \; m_2 l_2^2 \ddot{\theta}_2 \; + \; B_2(q, \dot{q}) = \tau_2(t) \end{cases} . \tag{4.29}$$

At each time instant the simulator solves this 2×2 linear system for $\ddot{\theta}_1$ and $\ddot{\theta}_2$. In compact notation this can be written as

$$\ddot{q} = M(q)^{-1} \big(\tau(t) - B(q, \dot{q}) \big), \tag{4.30}$$

which is the form used in the implementation.

For numerical integration the second–order model is converted to a first–order state–space system. The state vector is defined as

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{pmatrix} . \tag{4.31}$$

The corresponding state–space model $\dot{x} = f(t, x, u)$ takes the form

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \\ \ddot{\theta}_1(x, t) \\ \ddot{\theta}_2(x, t) \end{pmatrix} , \tag{4.32}$$

where $\ddot{\theta}_1(x, t)$ and $\ddot{\theta}_2(x, t)$ are obtained by substituting $q = (x_1, x_2)^T$ and $\dot{q} = (x_3, x_4)^T$ into (4.30). This representation has the same general form as (4.1) and is used directly by the simulator.

4.4 Inverted Pendulum on a Cart

The inverted pendulum on a cart extends the pendulum models to a classical underactuated system [Wik25d]. In the simulator it serves as a benchmark for nonlinear control: the task is to stabilise a rigid rod mounted on a horizontally moving cart. The mechanical configuration is shown in Figure 4.3: a cart of mass M moves along a horizontal track, and a massless rod of length l carries a point mass m at its end. The angle θ is measured from the upward vertical, so that $\theta = 0$ corresponds to the upright position, an unstable pendulum equilibrium.

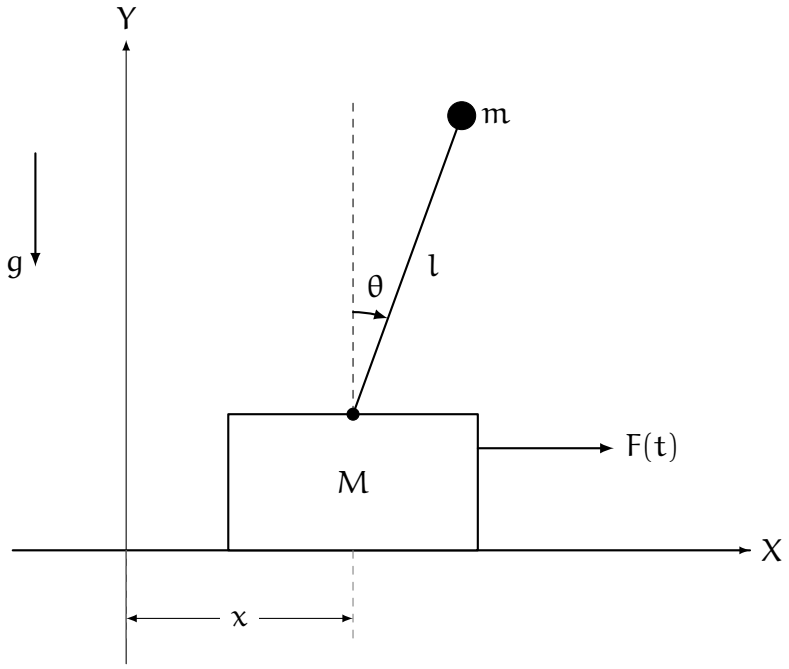


Figure 4.3 Geometry of the inverted pendulum on a cart

The generalised coordinates and velocities are

$$q = \begin{pmatrix} x \\ \theta \end{pmatrix}, \qquad \dot{q} = \begin{pmatrix} \dot{x} \\ \dot{\theta} \end{pmatrix}, \tag{4.33}$$

where x denotes the horizontal cart position and θ is the pole angle from the upward vertical. The gravitational acceleration is denoted by g.

Under the point–mass assumption the cart has purely translational kinetic energy

$$T_{\text{cart}} = \frac{1}{2} M \dot{x}^2. \tag{4.34}$$

The point mass at the rod tip is placed at the position

$$\begin{cases} x_m = x + l \sin\theta \\ y_m = l \cos\theta \end{cases}, \tag{4.35}$$

and thus has velocity components equal to

$$\begin{cases} \dot{x}_m = \dot{x} + l\dot{\theta} \cos\theta \\ \dot{y}_m = -l\dot{\theta} \sin\theta \end{cases}, \tag{4.36}$$

so that its kinetic energy is

$$T_{\text{pend}} = \frac{1}{2} m \left(v_x^2 + v_y^2 \right). \tag{4.37}$$

The total kinetic energy is therefore

$$T(q, \dot{q}) = T_{\text{cart}} + T_{\text{pend}} = \frac{1}{2}(M + m)\, \dot{x}^2 + ml \cos\theta\, \dot{x}\dot{\theta} + \frac{1}{2} m l^2 \dot{\theta}^2. \tag{4.38}$$

Taking the upright position as the reference level and the vertical coordinate of the point mass, (4.35) the gravitational potential energy can be written as

$$V(\theta) = mg\big(l\cos\theta - l\big) = mgl(\cos\theta - 1), \tag{4.39}$$

where g denotes the magnitude of gravitational acceleration; gravity acts along the Y axis opposite to its positive orientation, which is reflected in the sign of V.

The Lagrangian is

$$L(q, \dot{q}) = T(q, \dot{q}) - V(q). \tag{4.40}$$

Applying the Euler–Lagrange equation (2.5) to the coordinates x and θ yields the coupled equations of motion

$$\begin{cases} (M+m)\ddot{x} \;+\; ml\cos\theta\,\ddot{\theta} \;-\; ml\sin\theta\,\dot{\theta}^2 = Q_x \\ ml\cos\theta\,\ddot{x} \;+\; ml^2\ddot{\theta} \;+\; mgl\sin\theta = Q_\theta \end{cases}, \tag{4.41}$$

where Q_x and Q_θ denote the generalised forces associated with the cart translation and pendulum rotation, respectively.

In the simulator the cart is actuated by a horizontal force $F(t)$, and both the cart and the pendulum are subject to viscous and Coulomb friction. Thus the generalised forces are modelled as

$$Q_x = F(t) - b_{\text{cart}}\dot{x} - F_{c,\text{cart}}\ \operatorname{sgn}(\dot{x}), \tag{4.42}$$

$$Q_\theta = -b_{\text{pend}}\dot{\theta} - T_{c,\text{pend}}\ \operatorname{sgn}(\dot{\theta}), \tag{4.43}$$

where $b_{\text{cart}}, b_{\text{pend}} \geqslant 0$ are viscous damping coefficients and $F_{c,\text{cart}}, T_{c,\text{pend}} \geqslant 0$ represent Coulomb friction in the cart translation and at the pendulum pivot, respectively.

Equations (4.41) together with (4.42)–(4.43) can be written in the generic second–order form (2.6),

$$M(q)\,\ddot{q} + B(q, \dot{q}) = \tau(t), \tag{4.44}$$

with inertia matrix

$$M(q) = \begin{bmatrix} M+m & ml\cos\theta \\ ml\cos\theta & ml^2 \end{bmatrix}, \tag{4.45}$$

input vector

$$\tau(t) = \begin{pmatrix} F(t) \\ 0 \end{pmatrix}, \tag{4.46}$$

and remaining terms collected in

$$B(q, \dot{q}) = \begin{pmatrix} B_1(q, \dot{q}) \\ B_2(q, \dot{q}) \end{pmatrix}, \tag{4.47}$$

where

$$\begin{aligned} B_1(q, \dot{q}) &= -ml\sin\theta\,\dot{\theta}^2 + b_{\text{cart}}\dot{x} + F_{c,\text{cart}}\ \operatorname{sgn}(\dot{x}), \\ B_2(q, \dot{q}) &= mgl\sin\theta + b_{\text{pend}}\dot{\theta} + T_{c,\text{pend}}\ \operatorname{sgn}(\dot{\theta}). \end{aligned} \tag{4.48}$$

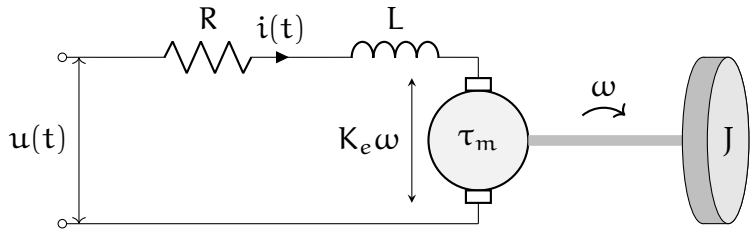


Figure 4.4 Schematic diagram of the DC motor electromechanical model

This makes the relation to the general form (2.6) explicit and parallels the interpretation used for the double pendulum.

For numerical integration the second–order system (4.44) is converted to a first–order state–space model using the relation (2.7). The state vector is defined as

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x \\ \theta \\ \dot{x} \\ \dot{\theta} \end{pmatrix}, \tag{4.49}$$

and the control input is identified with the cart force,

$$u(t) = F(t). \tag{4.50}$$

At each time instant the simulator solves the 2×2 linear system (4.44) for $\ddot{x}$ and $\ddot{\theta}$. In compact notation this can be written as

$$\ddot{q} = M(q)^{-1}\big(\tau(t) - B(q, \dot{q})\big), \qquad \ddot{q} = \begin{pmatrix} \ddot{x} \\ \ddot{\theta} \end{pmatrix}, \tag{4.51}$$

which is the form used in the implementation.

The resulting first–order model is

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \\ \ddot{x}(t, x, u) \\ \ddot{\theta}(t, x, u) \end{pmatrix}, \tag{4.52}$$

where $\ddot{x}(t, x, u)$ and $\ddot{\theta}(t, x, u)$ are obtained by substituting $q = (x_1, x_2)^T$, $\dot{q} = (x_3, x_4)^T$ and $u(t) = F(t)$ into (4.51). This representation has the same general form as (4.1) and is used directly in the simulator.

4.5 DC Motor

The DC motor model represents a separately excited motor driving a rigid rotor [Wik25a]. Figure 4.4 shows the schematic used in the derivation. The electrical part

is characterised by the armature resistance R, inductance L and back electromotive force constant K_e. The mechanical part consists of a rotor with moment of inertia J and viscous friction coefficient b_m. The motor torque is proportional to the armature current i with torque constant K_t, and the shaft is subject to an external load torque $\tau_{\text{load}}(t, \omega)$.

The state variables are chosen as the armature current i and the rotor angular velocity ω. With $u(t)$ denoting the applied armature voltage, Kirchhoff's voltage law for the armature circuit gives

$$u(t) = R\,i(t) + L\,\dot{i}(t) + K_e\,\omega(t), \tag{4.53}$$

where $K_e\omega(t)$ represents the back electromotive force. Solving for $\dot{i}(t)$ yields the electrical dynamics

$$L\,\dot{i}(t) = u(t) - R\,i(t) - K_e\,\omega(t). \tag{4.54}$$

Balancing torques on the rotor according to Newton's second law for rotation,

$$J\,\dot{\omega}(t) = \tau_{\text{motor}}(t) - \tau_{\text{fric}}(t) - \tau_{\text{load}}(t, \omega), \tag{4.55}$$

with $\tau_{\text{motor}} = K_t i$ and viscous friction $\tau_{\text{fric}} = b_m \omega$, leads to the mechanical dynamics

$$J\,\dot{\omega}(t) = K_t\,i(t) - b_m\,\omega(t) - \tau_{\text{load}}(t, \omega). \tag{4.56}$$

Equations (4.54)–(4.56) form a coupled electromechanical system of second order. In contrast to the pendulum models, which are obtained from the Lagrangian form (2.5) and the generic structure (2.6), the DC motor is derived directly from circuit and torque balance laws. The resulting dynamics, however, still fit the state–space representation (2.1).

Introducing the state vector

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} i \\ \omega \end{pmatrix}, \tag{4.57}$$

the first–order state–space model $\dot{x} = f(t, x, u)$ can be written as

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} \dfrac{1}{L}\bigl(u(t) - R x_1 - K_e x_2\bigr) \\[2ex] \dfrac{1}{J}\bigl(K_t x_1 - b_m x_2 - \tau_{\text{load}}(t, x_2)\bigr) \end{pmatrix}. \tag{4.58}$$

This is a nonlinear, time–invariant system of the form (2.1) with state $x = (i, \omega)^{\mathsf{T}}$, input $u(t)$ equal to the armature voltage and output $y = x$ as in (4.1). This state–space form is used directly in the simulator.

4.6 Van der Pol Oscillator

The Van der Pol oscillator in the simulator is implemented as a simple electrical circuit: a capacitor C and an inductor L connected in parallel with a nonlinear current

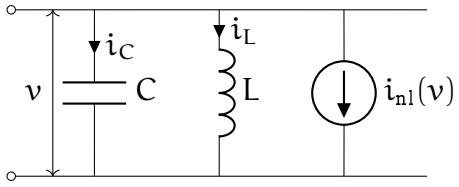


Figure 4.5 Van der Pol oscillator implemented as a parallel LC circuit with a nonlinear current source

source [Wik25h]. The circuit schematic is shown in Figure 4.5. The state variables are chosen as the capacitor voltage v and the inductor current i_L. The nonlinearity is modelled by a current–voltage characteristic of the form

$$i_{nl}(v) = \mu \left(\frac{v^3}{3} - v \right), \tag{4.59}$$

where $\mu > 0$ is a scalar parameter which controls the strength of the nonlinear damping.

Applying Kirchhoff's current law at the node where the three elements meet yields

$$C\,\dot{v}(t) + i_L(t) + i_{nl}\big(v(t)\big) = 0, \tag{4.60}$$

while the inductor voltage–current relation is

$$L\,\dot{i}_L(t) = v(t). \tag{4.61}$$

Equations (4.60)–(4.61) describe the dynamics of the circuit in terms of physically meaningful variables. Unlike the mechanical models, which are derived from the Lagrangian form (2.5) and the generic second–order structure (2.6), the Van der Pol oscillator is obtained directly from network laws. Nevertheless, the resulting dynamics are still represented in the state–space framework (2.1).

Solving (4.60) for $\dot{v}(t)$ with the nonlinearity (4.59) substituted gives

$$C\,\dot{v}(t) = -\,i_L(t) - \mu \left(\frac{v(t)^3}{3} - v(t) \right), \tag{4.62}$$

while from (4.61) one obtains

$$\dot{i}_L(t) = \frac{1}{L} v(t). \tag{4.63}$$

For numerical simulation the state vector is defined as

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} v \\ i_L \end{pmatrix}, \tag{4.64}$$

so that the first–order system $\dot{x} = f(t, x)$ takes the form

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} \dfrac{-x_2 - i_{nl}(x_1)}{C} \\[2ex] \dfrac{x_1}{L} \end{pmatrix}, \tag{4.65}$$

with $i_{nl}(x_1)$ given by (4.59). This is a nonlinear, autonomous, time–invariant system in the sense of the classification in Section 2.2 and constitutes a special case of the general state–space representation (2.1) with state $x = (v, i_L)^T$, no external input ($u \equiv 0$) and output chosen as $y = x$ according to (4.1). This is the form used in the implementation.

4.7 Lorenz System

In addition to the physical models the simulator includes the Lorenz system, a classical three–dimensional nonlinear model exhibiting chaotic behaviour [Wik25f]. It serves as a purely mathematical test case for numerical integration and for visualising trajectories in phase space.

The state space–vector is

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \tag{4.66}$$

and the dynamics are given by the three coupled differential equations

$$\begin{cases} \dot{x}_1 = \sigma(x_2 - x_1) \\ \dot{x}_2 = x_1(\rho - x_3) - x_2 \\ \dot{x}_3 = x_1 x_2 - \beta x_3 \end{cases}, \tag{4.67}$$

where $\sigma > 0$, $\rho > 0$ and $\beta > 0$ are fixed parameters. For the classical chaotic regime one often uses $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$, but in the simulator these constants can be adjusted by the user.

The Lorenz system is autonomous and has no external input, so $u(t) \equiv 0$. In the notation of (4.1) it can be written as

$$\dot{x} = f(t, x, u), \qquad u \equiv 0, \qquad y = x, \tag{4.68}$$

with

$$f(t, x, u) = \begin{pmatrix} \sigma(x_2 - x_1) \\ x_1(\rho - x_3) - x_2 \\ x_1 x_2 - \beta x_3 \end{pmatrix}, \tag{4.69}$$

so that the simulator treats it as a special case of the general state–space representation (2.1). The implementation uses (4.67) directly to generate trajectories and phase portraits.

4.8 Energy–Based Swing–Up Controller

For the cart–pole model (4.52) the simulator implements an energy–based swing–up controller [Wik25c]. Its role is to drive the system from the vicinity of the stable bottom position ($\theta \approx \pi$) to a neighbourhood of the upright position ($\theta \approx 0$), where a linear stabilising controller can take over.

The controller operates on the full state vector (4.49) and uses the same geometric convention as the plant: θ is measured from the upward vertical. For convenience it is useful to introduce wrapped angles

$$\begin{cases} \theta_{\text{up}} = \text{wrap}_{(-\pi,\pi]}(\theta) \\ \theta_{\text{down}} = \text{wrap}_{(-\pi,\pi]}(\theta_{\text{up}} - \pi) \end{cases}, \tag{4.70}$$

where $\mathrm{wrap}_{(-\pi,\pi]}$ maps any angle into $(-\pi,\pi]$. Thus $\theta_{\mathrm{up}} = 0$ corresponds to the upright position and $\theta_{\mathrm{down}} = 0$ to the bottom position.

Under the point–mass assumption the mechanical energy of the pendulum measured from the bottom equilibrium is defined as

$$E(\theta,\dot{\theta}) = \frac{1}{2} m l^2 \dot{\theta}^2 + mgl\big(1 - \cos\theta_{\mathrm{down}}\big), \tag{4.71}$$

where m, l and g are the same parameters as in the cart–pole model. The desired energy level corresponding approximately to the upright position is chosen as

$$E_{\mathrm{des}} = 2mgl. \tag{4.72}$$

The swing–up law has two operating regimes. Outside a small neighbourhood of the upright position it pumps energy into the pendulum while damping the cart motion. In this *energy pumping* region the control force is

$$u_{\mathrm{pump}}(x) = k_e\Big(E(x_2,x_4) - E_{\mathrm{des}}\Big)\ \mathrm{sgn}\big(x_4\cos x_2\big) - k_v x_3 - k_x x_1, \tag{4.73}$$

where $k_e > 0$ is the energy gain, $k_v \geqslant 0$ damps the cart velocity and $k_x \geqslant 0$ introduces a mild centring of the cart position to prevent unbounded drift.

Near the upright position, defined by $|\theta_{\mathrm{up}}| < \theta_{\mathrm{soft}}$ for some small angle $\theta_{\mathrm{soft}} > 0$, the controller switches to a linear “soft” regime that stabilises the cart near the origin without significantly changing the pendulum energy. In this region the force is

$$u_{\mathrm{soft}}(x) = -k_{x,\mathrm{soft}}\, x - k_{v,\mathrm{soft}}\,\dot{x}, \tag{4.74}$$

where $k_{x,\mathrm{soft}}, k_{v,\mathrm{soft}} \geqslant 0$ are position and velocity gains chosen for gentle centring.

In both regimes the command force is saturated to a prescribed limit $|u(t)| \leqslant F_{\max}$ and passed through a simple rate limiter to avoid very fast changes of the input, which could lead to stiff behaviour in the numerical integration [HNW93]. The overall swing–up controller realises a static state–feedback law

$$u(t) = K_{\mathrm{sw}}(x(t)), \tag{4.75}$$

where K_{sw} is defined piecewise by (4.73)–(4.74). Combined with the cart–pole dynamics (4.52) this yields a closed–loop nonlinear system consistent with the general framework (4.1).

4.9 LQR Controller for the Inverted Pendulum

Once the pendulum has been brought close to the upright position, a linear quadratic regulator (LQR) is used to stabilise the cart–pole around this unstable equilibrium [Wik25e]. The starting point is a linearisation of the nonlinear state–space model (4.52) around the equilibrium $(x,\dot{x},\theta,\dot{\theta}) = (0,0,0,0)$ with zero input.

Denoting small deviations from the equilibrium by $\delta x = (x, \theta, \dot{x}, \dot{\theta})^{\mathsf{T}}$ and $\delta u = u$, the linearised dynamics can be written in the form

$$\delta \dot{x}(t) = A\,\delta x(t) + B\,\delta u(t), \tag{4.76}$$

where the matrices $A \in \mathbb{R}^{4\times 4}$ and $B \in \mathbb{R}^{4\times 1}$ are obtained by differentiating the right–hand side of (4.52) with respect to the state and the input and evaluating the result at the upright equilibrium. Assuming viscous cart damping b and viscous pivot damping c, the matrices are

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\dfrac{m g}{M} & -\dfrac{b}{M} & \dfrac{c}{M l} \\ 0 & \dfrac{(M+m) g}{M l} & \dfrac{b}{M l} & -\dfrac{(M+m) c}{m M l^{2}} \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 0 \\ \dfrac{1}{M} \\ -\dfrac{1}{M l} \end{bmatrix}. \tag{4.77}$$

The LQR design seeks a state–feedback law

$$\delta u(t) = -K\,\delta x(t), \tag{4.78}$$

that minimises the quadratic performance index

$$J = \int_{0}^{\infty} \left(\delta x(t)^{\mathsf{T}} Q\, \delta x(t) + \delta u(t)^{\mathsf{T}} R\, \delta u(t) \right) dt, \tag{4.79}$$

with weighting matrices $Q \succcurlyeq 0$ and $R > 0$. In the simulator the diagonal matrix Q and the scalar R are chosen using Bryson's rule. Given admissible amplitudes $x_{\max}$, $\theta_{\max}$, $\dot{x}_{\max}$, $\dot{\theta}_{\max}$ and $u_{\max}$, one sets

$$Q = \operatorname{diag}\!\left(\frac{1}{x_{\max}^{2}}, \frac{1}{\theta_{\max}^{2}}, \frac{1}{\dot{x}_{\max}^{2}}, \frac{1}{\dot{\theta}_{\max}^{2}} \right), \qquad R = \left[\frac{1}{u_{\max}^{2}} \right]. \tag{4.80}$$

The optimal feedback gain K is then obtained by solving the continuous–time algebraic Riccati equation associated with (4.76)–(4.79).

In the nonlinear simulator the LQR law is applied to the full state (4.49) with angle wrapping around the upright position. Let

$$\theta_{\mathrm{err}} = \operatorname{wrap}_{(-\pi,\pi]}(\theta), \tag{4.81}$$

and define the error state

$$\delta x = \begin{pmatrix} x - x_{\mathrm{ref}} \\ \theta_{\mathrm{err}} - \theta_{\mathrm{ref}} \\ \dot{x} \\ \dot{\theta} \end{pmatrix}, \tag{4.82}$$

where x_{ref} and θ_{ref} are reference values (typically $x_{\text{ref}} = 0$, $\theta_{\text{ref}} = 0$). The control force applied to the cart is

$$u(t) = -K\,\delta x(t), \tag{4.83}$$

and is additionally saturated to $|u(t)| \leqslant u_{\max}$ to reflect actuator limits. Together, the linear feedback law (4.83) and the nonlinear cart–pole dynamics (4.52) form a locally stabilising closed–loop system around the upright equilibrium, consistent with the general state–space framework (4.1).

Chapter 5

Software Implementation and Graphical Interface

5.1 Purpose and relation to previous chapters

This chapter documents the implementation of the Dynamic System Simulator (DSS), building on the architectural design in Chapter 3 and the state–space formulation in Chapter 2. The focus is the *single execution path* used to run simulations: how the graphical interface (or scripts) constructs a configuration bundle `cfg`, how the simulation core composes an effective closed-loop system (model–controller–wrapper), how numerical integration is performed, and how results are returned in a standard output bundle `out` for visualisation and logging.

The intent is to provide an auditable mapping between (i) the repository structure, (ii) the runtime workflow, and (iii) the extension mechanism (adding new models and controllers), while keeping the description aligned with the actual execution path used in experiments in Chapter 6.

5.2 Implementation overview and design principles

The simulator is implemented in Python [Pyt25]. NumPy arrays are used for state vectors and trajectories [HMvdW+20]. Numerical integration is performed using SciPy (`scipy.integrate.solve_ivp`) [V+20], while the GUI is implemented in Streamlit [Str25] and uses Plotly dashboards [Plo25] and lightweight 2D animations for interactive exploration. The high-level repository organisation is shown in Figure 5.1. Detailed installation steps and usage instructions are provided in Appendix A.

The implementation workflow is summarised in Figure 5.2 and mirrors the layered architecture introduced in Chapter 3. All systems are integrated through a solver-facing callable compatible with `solve_ivp`, while optional inputs are introduced by wrappers that form an effective dynamics function. Each run is defined by a reusable configuration `cfg`; the core library `dss/` is independent from the GUI layer `apps/streamlit/`, and offline experiments are provided in `tools/`.

```
DynamicSystemSimulator/
├── dss/ ..................................... reusable simulation library
│   ├── core/ ................... contracts, pipeline, solver, output bundle
│   ├── models/ ................................... physical models + registry
│   ├── controllers/ ................................ control laws (optional)
│   └── wrappers/ ................................ open/closed-loop wrappers
├── apps/streamlit/ ..................... GUI layer (inputs + visualisation)
│   ├── assets/ ..................................... CSS and static assets
│   ├── components/ ................................ reusable UI + dashboards
│   ├── runners/ ................................... per-model execution glue
│   └── systems/ ..................................... per-model UI views
├── tools/ ..................... offline experiments (Chapter 6 plots/tables)
├── docs/ ..................... installation + usage + reproducibility notes
└── streamlit_app.py ................................ main GUI entry point
```

Figure 5.1 High-level repository structure

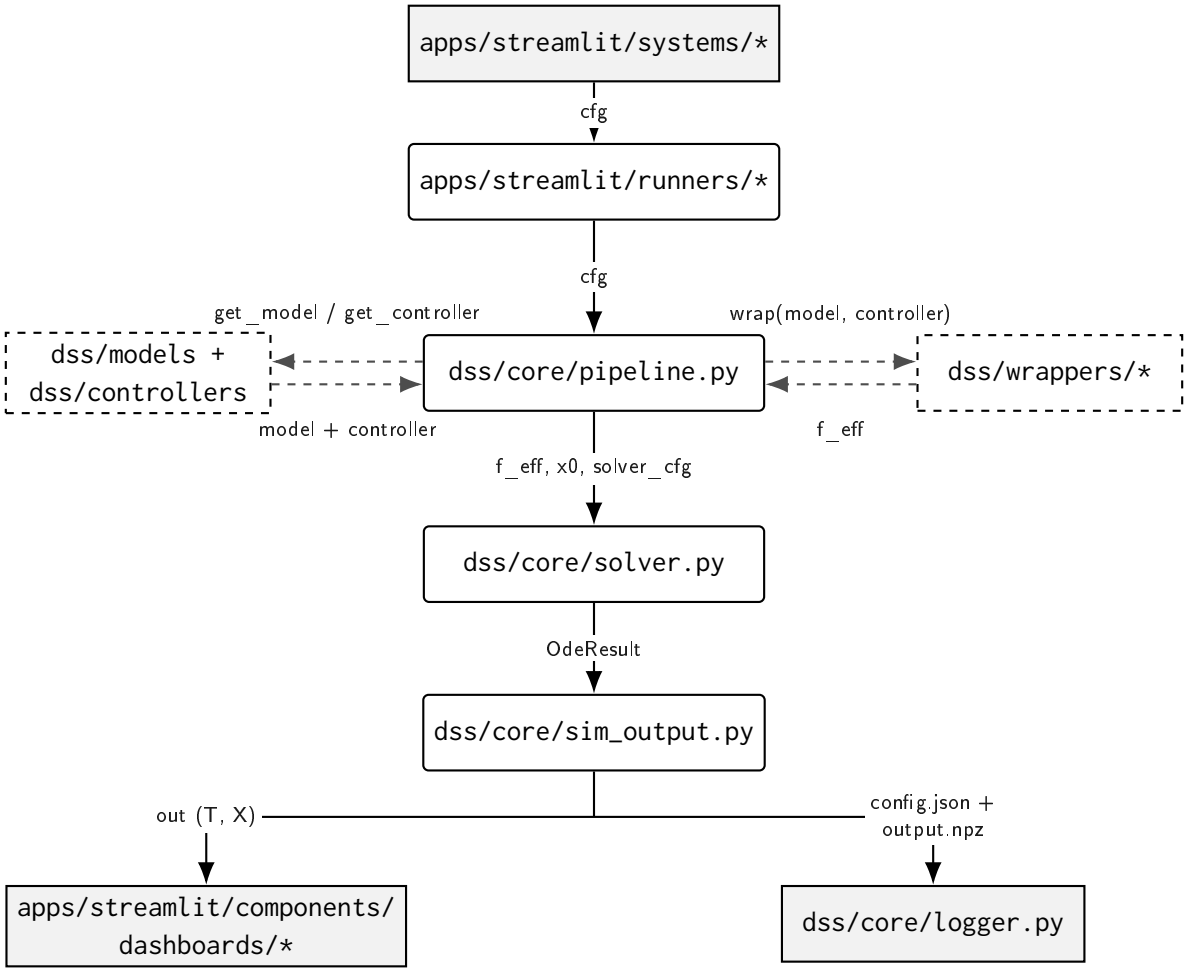


Figure 5.2 System implementation workflow

Table 5.1 Mapping from architecture layers to concrete source-code modules

Layer	Implementation modules
Input	`apps/streamlit/systems/*`
Simulation (orchestration)	`apps/streamlit/runners/*`, `dss/core/pipeline.py`
Model	`dss/models/*`
Controller (optional)	`dss/controllers/*`
Wrapper (optional)	`dss/wrappers/*`
Solver	`dss/core/solver.py`
Visualization	`apps/streamlit/components/dashboards/*`
Logging	`dss/core/logger.py`

Figure 5.1 highlights only the directories that participate in the execution path described in this chapter. The package `dss/` contains the reusable simulation core: model implementations, optional controllers, wrappers that compose closed-loop behaviour, and the solver/pipeline code that exposes a solver-facing dynamics function. The Streamlit application under `apps/streamlit/` is responsible only for building a configuration `cfg`, dispatching execution via runners, and rendering the returned results. Offline experiments used in Chapter 6 are implemented as command-line tools in `tools/`, which reuse the same core components without depending on the GUI.

This separation enforces a one-way dependency boundary: the Streamlit layer depends on `dss/`, while `dss/` contains no Streamlit-specific code. This prevents GUI concerns from leaking into numerical integration and supports reproducible offline execution.

5.3 Implementation workflow and data flow

Before detailing the module-level execution path, it is useful to relate Figure 5.2 to the layered architecture introduced in Chapter 3. The Streamlit GUI implements the Input and Visualization roles, the pipeline acts as the Simulation coordinator, registries provide Model/Controller selection, optional wrappers provide composition when requested by `cfg`, and the solver integrates the resulting solver-facing dynamics. The mapping is summarised in Table 5.1.

Figure 5.2 summarises the single execution path of the simulator at the level of concrete source-code modules. A run starts in `apps/streamlit/systems/*`, where the GUI converts user-selected settings into a configuration dictionary `cfg`. The configuration is forwarded by `apps/streamlit/runners/*` to `dss/core/pipeline.py`, which orchestrates construction of the solver-facing problem.

The dashed side connections indicate auxiliary steps performed by the pipeline. It resolves the requested components via the registries in `dss/models` and (optionally) `dss/controllers`. If the configuration requires additional composition, the pipeline uses `dss/wrappers/*` to construct an effective right-hand-side function $f_{\text{eff}}(t, x)$; otherwise,

the model dynamics are passed directly. The numerical integration is executed by dss/core/solver.py through SciPy solve_ivp, which returns an OdeResult. Finally, dss/core/sim_output.py normalises the solver output into a standard object out (at minimum T and X), which is consumed by the Streamlit dashboards and may be persisted by dss/core/logger.py for reproducibility.

5.4 Core library (dss/)

The package dss/ implements the reusable simulation core shown in Figure 5.1 and used in the workflow of Figure 5.2. It contains solver-facing contracts, registries for component selection, optional composition wrappers, and the pipeline/solver/output boundary that allows both the GUI and offline tools to execute simulations through the same core entry points.

Contracts (dss/core/contracts.py)

All numerical integration is performed by calling a single dynamics function compatible with the general state-space form introduced in Chapter 2. The minimal requirement is the DynamicalSystem protocol exposing a callable dynamics(t, state, inputs=None). In addition, selected systems provide optional capabilities used for diagnostics and visualisation: state labels, planar positions for 2D animation (HasPositions), and energy diagnostics (HasEnergy). These optional interfaces are queried by the GUI layer but do not change the solver API.

The resulting solver-facing contracts are summarised in Listing 5.1. The pipeline assumes only DynamicalSystem: the solver calls dynamics(t, state, inputs) to evaluate the right-hand side $\dot{x} = f(t, x, u)$. Optional protocols such as HasPositions and HasEnergy enable additional GUI features (2D animation and energy-drift diagnostics) and are detected at runtime via @runtime_checkable Protocols, avoiding a rigid inheritance hierarchy for models.

Models (dss/models/*)

Each mathematical model described in the previous chapters is implemented as a Python class under dss/models/. Model creation is routed through a registry exposed in dss/models/__init__.py. The pipeline calls the factory interface (e.g., get_model(name, mode, **params)) to obtain a solver-facing object. The factories provide a stable construction interface for the pipeline: they validate and filter configuration keys, apply default values, and return a solver-facing object exposing dynamics(t,x,inputs) and (where implemented) optional helper methods for diagnostics and visualisation.

```python
@runtime_checkable
class DynamicalSystem(Protocol):
    def dynamics(self, t: float, state: np.ndarray, inputs=None) ->
        np.ndarray: ...

@runtime_checkable
class HasPositions(Protocol):
    def positions(self, state: np.ndarray): ...

@runtime_checkable
class HasEnergy(Protocol):
    def energy_check(self, state: np.ndarray) -> np.ndarray: ...
```

Listing 5.1 Solver-facing contracts (excerpt)

Controllers (dss/controllers/*)

Controllers are implemented as optional components under dss/controllers/. They are registered under string identifiers in dss/controllers/__init__.py. The public function get_controller(name) returns the controller requested by the configuration. Controller usage is optional and is employed only in scenarios that require additional composition (as reflected by the dashed dependency arrows in Figure 5.2).

Wrappers (dss/wrappers/*)

Wrappers are used only when a simulation requires composition beyond a standalone model. A wrapper remains solver-facing (it exposes a dynamics(t,x,inputs) method) but internally combines multiple components (e.g., model and controller) and constructs an effective right-hand-side function $f_{\text{eff}}(t, x)$. When no wrapper is requested, the pipeline passes the model dynamics directly to the solver. This conditional composition is captured in Figure 5.2: the pipeline interacts with dss/wrappers/* only if required by cfg.

Run configuration (cfg)

Execution is configuration-driven: both the GUI layer and offline tools describe a run using a dictionary cfg. Table 5.2 summarises the canonical keys expected by the pipeline. Controller and wrapper keys are optional and are used only when additional composition is requested.

Pipeline (dss/core/pipeline.py)

Table 5.2 defines the canonical keys consumed by run_config(cfg). The orchestration logic is implemented in dss/core/pipeline.py. The entry point run_config(cfg) performs: (i) system construction by resolving the requested components via registries

Table 5.2 Canonical configuration keys used by the simulation pipeline

Key	Type	Meaning
`model.name`	string	Model identifier resolved via the model registry.
`model.mode`	string	Optional variant selector (default: `default`).
`model.params`	dict	Model parameters (passed to the factory/constructor).
`initial_state`	array-like	Initial state vector $x(t_0)$.
`solver.t0`, `solver.t1`	float	Simulation interval bounds.
`solver.dt`	float	Desired output sampling spacing (used to build `t_eval`).
`solver.method`	string	`solve_ivp` method (e.g., RK45, DOP853, Radau).
`solver.rtol`, `solver.atol`	float	Integration tolerances.
`controller.name`	string	Optional controller identifier (when used).
`controller.params`	dict	Optional controller parameters.
`wrapper.name`	string	Optional wrapper identifier (when used).

in `dss/models` and (optionally) `dss/controllers`, (ii) optional wrapper composition to form an effective dynamics function, (iii) solver execution via the solver wrapper, and (iv) normalisation of solver results into the standard output object described below. In this design, the pipeline is the only component that needs to know how configuration keys map to concrete classes and how the execution pieces are composed.

Solver (`dss/core/solver.py`)

Numerical integration uses a wrapper around `scipy.integrate.solve_ivp`. The solver wrapper constructs the requested output sampling grid from `(t0,t1)` and the desired output spacing `dt` (via `t_eval`). This `dt` controls *output sampling*; the integrator still uses adaptive internal step sizes determined by the selected method and tolerances (`rtol`, `atol`). The solver returns SciPy's `OdeResult`, which is then converted to the standard output format.

Output and logging (`dss/core/sim_output.py`, `dss/core/logger.py`)

All results are normalised to a standard output object `out` with mandatory keys `T` (time vector) and `X` (state trajectory). Optional arrays (e.g., derived signals) may be added without breaking consumers, because visualisation depends primarily on `T` and `X` plus optional system capabilities (positions, energy, labels). For reproducibility, the logger module provides the ability to persist run artefacts under `logs/`, typically storing the configuration (`config.json`) and trajectory arrays (`output.npz`) for later inspection and to support the offline experiments reported in Chapter 6.

5.5 Graphical interface (apps/streamlit/)

The Streamlit application provides an interactive front-end for the execution path defined in Figure 5.2. Its responsibility is limited to (i) collecting user inputs, (ii) translating them into the canonical configuration dictionary `cfg` (Table 5.2) via per-model runners, and (iii) rendering the returned standard output `out`. Numerical integration and model logic remain in the core library `dss/` (Section 5.4).

Entry point and routing (streamlit_app.py, apps/streamlit/registry.py)

The GUI entry point is `streamlit_app.py`. It sets the page layout, injects the local CSS from `apps/streamlit/assets/style.css`, and presents a top-level model selector. Model pages are registered in `apps/streamlit/registry.py` through a dictionary of lazy factories (`SYSTEM_FACTORIES`), which prevents constructing all page specifications at startup. After the user selects a model, the application constructs a corresponding `SystemSpec` and delegates rendering to the generic page renderer `render_system` implemented in `apps/streamlit/layout.py`.

System views and input controls (apps/streamlit/systems/*)

Each system view under `apps/streamlit/systems/` exposes a `get_spec()` function that returns a `SystemSpec` structure. A `SystemSpec` defines three core elements: (i) a `controls(prefix)` function that renders widgets and returns a dictionary of values, (ii) a `run(controls)` function that triggers simulation and returns `(cfg, out)`, and (iii) a dashboard constructor (typically `make_dashboard(cfg, out, ui)`).

Widget keys are namespaced by a per-system `prefix` (the spec identifier). This ensures that multiple systems can share common widget names without collisions in `st.session_state`. To avoid running the simulation on every widget change (a Streamlit default), the views group most inputs inside an `st.form`. The form returns a `run_clicked` flag; only when the user explicitly clicks *Run* does the application execute the runner and update the stored results.

To keep the views consistent across models, common UI fragments are implemented in `apps/streamlit/components/ui_sections.py` (e.g., time horizon `t0,t1,dt`, solver tolerances and method, animation/performance limits, and logging settings). Many views also provide preset selectors that populate typical parameter sets for fast exploration and reduce the amount of manual tuning required for demonstrations.

GUI overview (example run)

Figure 5.3 shows the Streamlit interface during an example run (Lorenz system). The left panel groups user inputs that are translated into the canonical configuration dictionary `cfg` (Table 5.2): model selection and presets, system parameters, initial state, simulation horizon, and numerical solver settings. After the user triggers execution via

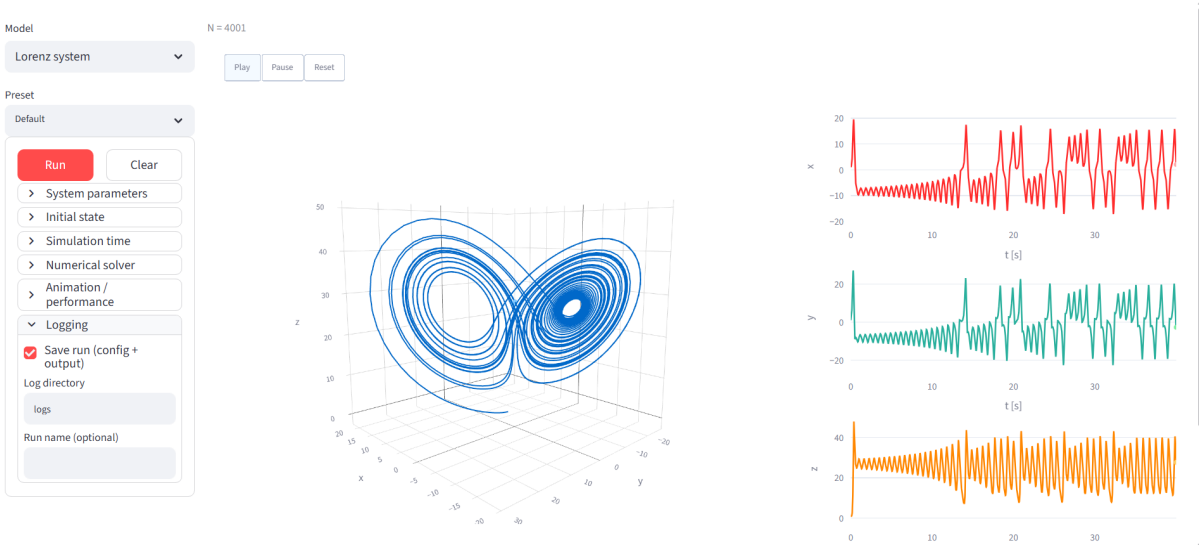


Figure 5.3 Streamlit GUI example (Lorenz system): control panel and visualisation outputs

Run, the core pipeline is executed and returns the standard output out (at minimum `T` and `X`). The right panel visualises out as interactive plots, including time trajectories and a 3D phase portrait. The logging panel illustrates the optional persistence of `config.json` and `output.npz` under `logs/` for reproducibility.

Runners and configuration construction (`apps/streamlit/runners/*`)

The execution glue between the GUI and the core pipeline is implemented as per-model runners under `apps/streamlit/runners/`. Each runner translates view-level values into the canonical structured configuration dictionary `cfg` (Table 5.2), including: model identifier and parameters (`cfg["model"]`), initial state (`cfg["initial_state"]`), and solver settings (`cfg["solver"]`).

The common helper `_common.py` (located in `apps/streamlit/runners/`) provides the convenience wrapper `run_from_cfg(cfg)` with additional keyword arguments. It calls the core entry point `run_config(cfg)` (implemented in `dss/core/pipeline.py`) and returns the normalised configuration `cfg2` together with the standard output out. Optional persistence is controlled by the GUI logging settings: when enabled, the runner creates a `SimulationLogger` and requests saving `config.json` and `output.npz` under `logs/`.

Dashboards and visualisation (`apps/streamlit/components/dashboards/*`)

Visualisation is implemented using Plotly. For each model, a dedicated dashboard module under `apps/streamlit/components/dashboards/` consumes out and produces a figure containing the required plots (time trajectories, phase portraits, and auxiliary diagnostics). The GUI reads only the standard fields (`T`, `X`) and optional UI values

(e.g., plot limits, trail options). Any model-specific quantities needed for plotting are computed within the dashboard modules from the returned trajectories.

GUI performance safeguards and reproducibility

The GUI includes safeguards to keep interaction responsive even when simulations produce dense trajectories. Two mechanisms are applied: (i) *plot downsampling*, where dashboards reduce the number of rendered points to a user-defined maximum (`max_plot_pts`); and (ii) *animation limiting*, where the number of rendered frames is bounded (`max_frames`) and the playback rate is controlled by an explicit animation FPS setting. These limits affect only visualisation and do not change the underlying solver accuracy, since numerical integration is performed on the full solver output grid requested by `cfg`. Finally, the optional logging controls allow the user to persist `cfg` and results for later inspection and for reproducible offline experiments in Chapter 6.

5.6 Offline experiments (`tools/`)

In addition to the Streamlit application, the repository provides command-line scripts under `tools/` used to generate the tables and figures reported in Chapter 6. These scripts reuse the same core implementation (`dss/`): they instantiate the same model classes, call the same numerical integration backend, and store results in reproducible files.

The utility `tools/ch6_generate_62.py` generates the artefacts used in Section 6.2. The utility `tools/ch6_perf_baseline_uniform.py` produces the baseline benchmarks reported in Section 6.4. Together, these two scripts support the workflow of Chapter 6.

In both cases, the scripts follow the same pattern. First, they define a scenario (model, parameters, initial state, horizon and sampling). Next, they run integration through the core pipeline in `dss/core/pipeline.py` and the solver wrapper in `dss/core/solver.py`. Finally, they export artefacts (e.g., `.png`, `.csv`, `.tex`) into a target directory under `figures/`.

5.7 Limitations

This section summarises limitations of the implementation and the single execution path (Figure 5.2). Quantitative validation and performance measurements are reported in Chapter 6.

The simulator is intentionally scoped to the set of systems implemented and discussed in the thesis; it does not aim to provide the full coverage or extensibility expected from general-purpose simulation environments. In particular, only the models, scenarios and configuration options exposed through the registries and runners are supported.

Auxiliary capabilities used for diagnostics and visualisation are not uniformly available across all models. Interfaces such as planar positions for animation (`HasPositions`),

energy checks (HasEnergy) and model-specific state labels are implemented only where meaningful. For this reason, the numerical core relies exclusively on the mandatory solver-facing dynamics function and the standard output fields T and X, while dashboards opportunistically use optional capabilities when present.

Composition beyond a standalone model is configuration-dependent. Controllers and wrappers are applied only when requested by cfg and supported by the available components, and therefore not every model exposes the same closed-loop or wrapper-based execution options.

Finally, reproducibility is ensured at the level of simulator inputs and outputs by logging the run configuration and trajectories (e.g., config.json and output.npz). Exact reproduction across machines also depends on the external software environment (package versions) and hardware context; these are not captured automatically unless recorded separately (e.g., in docs/).

Chapter 6

Experiments and validation

This chapter evaluates the *Dynamic System Simulator* from both a numerical and an educational perspective. The aim is to verify that the simulator produces trajectories consistent with the mathematical models introduced in Chapter 4, that the numerical integration behaves in a controlled way, and that the overall workflow remains suitable for student experiments carried out through the software architecture of Chapter 3 and the graphical interface of Chapter 5.

All experiments are executed programmatically using the same solver and logging infrastructure as in the rest of the project. Short Python scripts construct model instances using the configuration mechanisms of Chapter 5, run simulations with the solver settings introduced there and record the resulting state trajectories and derived quantities through the logger layer described in Chapter 3. The figures and numerical summaries presented in this chapter are based directly on these simulations, so that the reported behaviour can be reproduced by rerunning the corresponding experiments with the same configuration.

6.1 Evaluation goals and methodology

The evaluation in this chapter serves four closely related purposes. The first purpose is to check that each model exhibits the qualitative dynamics expected from its mathematical description: small oscillations for the ideal pendulum, self-sustained oscillations for the Van der Pol circuit, sensitive dependence on initial conditions for chaotic systems and successful inversion and stabilisation for the cart–pole. The second purpose is to quantify numerical accuracy on problems where analytical results or simple invariants are available, for example the small-angle period and total mechanical energy of a simple pendulum. The third purpose is to measure the computational cost of typical simulations and to examine how runtime depends on the solver tolerances chosen in Chapter 5. The fourth purpose is to demonstrate that the simulator can be used in an educational setting, both through stand-alone scripts and through the Streamlit-based graphical user interface.

The experimental setup reflects the layered structure of the simulator. For each scenario a dedicated script selects a model and, when needed, a controller, assigns

numerical parameters and initial conditions, and specifies the simulation horizon and sampling density. The configuration is passed to the solver layer, which integrates the state–space equations and returns the time histories of the state and of selected output variables. The logger evaluates additional diagnostic quantities such as energies, reference periods or distances between trajectories, and these signals form the basis for the plots and tables in this chapter.

The remainder of the chapter is organised as follows. Section 6.2 discusses the functional behaviour of the main models and presents the corresponding time histories and phase portraits. Section 6.3 analyses numerical convergence using tolerance sweeps. Section 6.4 contains a performance study based on measured runtimes for different systems and tolerance settings. Finally, Section 6.5 documents practical limitations of the graphical user interface in typical interactive scenarios.

6.2 Functional behaviour of the models

This section provides qualitative validation of the implemented models by inspecting representative trajectories, phase portraits and derived signals. The examples are intended to confirm that the simulator reproduces the characteristic behaviours discussed in Chapter 4: damped oscillations for the pendulum, complex coupled motion for the double pendulum, electromechanical transients for the DC motor, emergence of a limit cycle in the Van der Pol oscillator and chaotic motion for the Lorenz system. (Results for the cart–pole benchmark are discussed in the educational case studies and GUI sections, where the control task is central.)

Single pendulum (damped)

Figure 6.1 summarises a representative damped pendulum run, including time histories, the phase portrait and the energy diagnostic. The angular displacement and velocity exhibit decaying oscillations, the phase portrait spirals toward the stable equilibrium, and the total mechanical energy decreases over time due to dissipation.

Double pendulum

Figure 6.2 summarises a representative double pendulum run. The coupled angles and angular velocities exhibit irregular, strongly nonlinear behaviour, and the planar phase portraits illustrate the complex state evolution in each joint coordinate.

Inverted pendulum on a cart

Figure 6.3 shows the open-loop response of the ideal cart–pole model. Without feedback, the pendulum does not stabilise around the upright position and the state evolves periodically within the wrapped-angle representation.

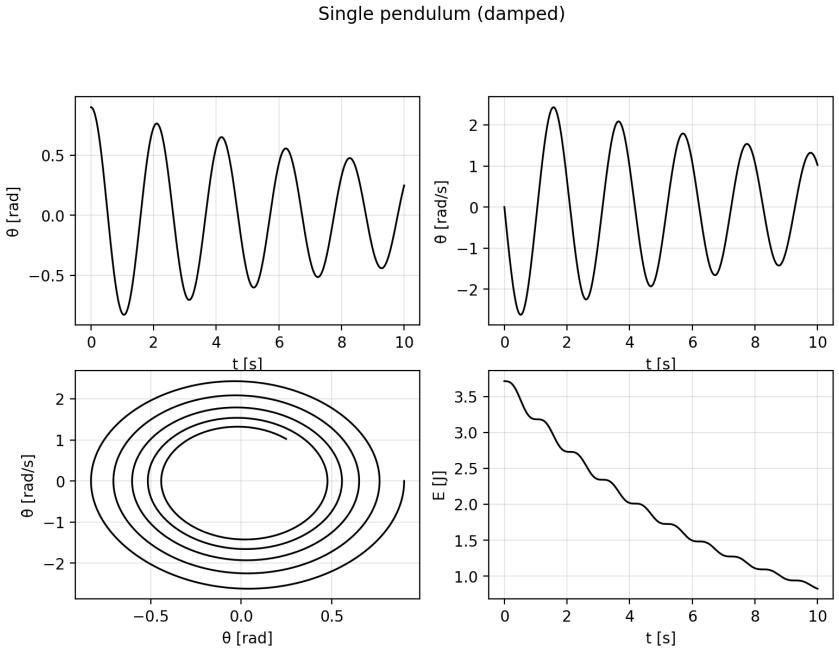


Figure 6.1 Single pendulum: $\theta, \dot{\theta}$, $(\theta, \dot{\theta})$, $E(t)$

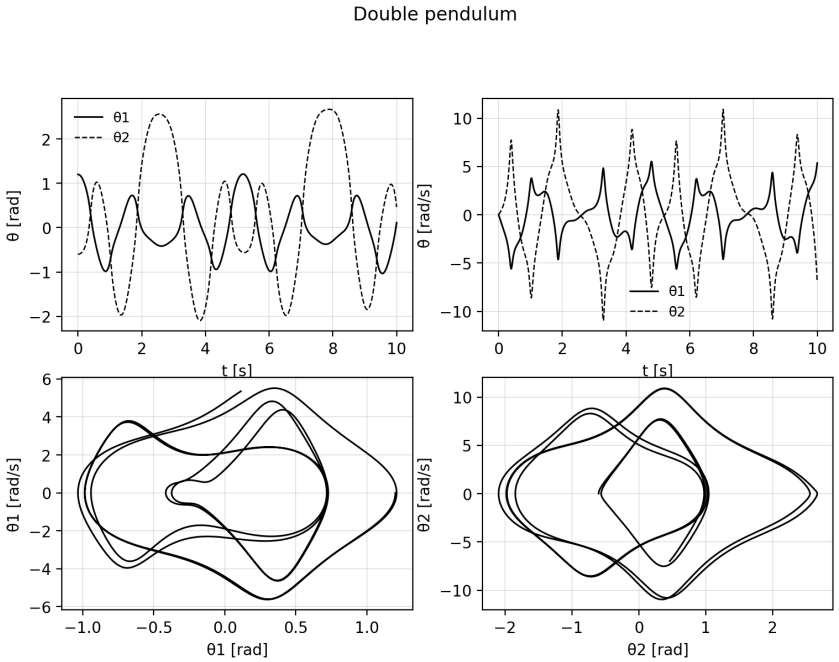


Figure 6.2 Double pendulum: $\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2$ and phase portraits

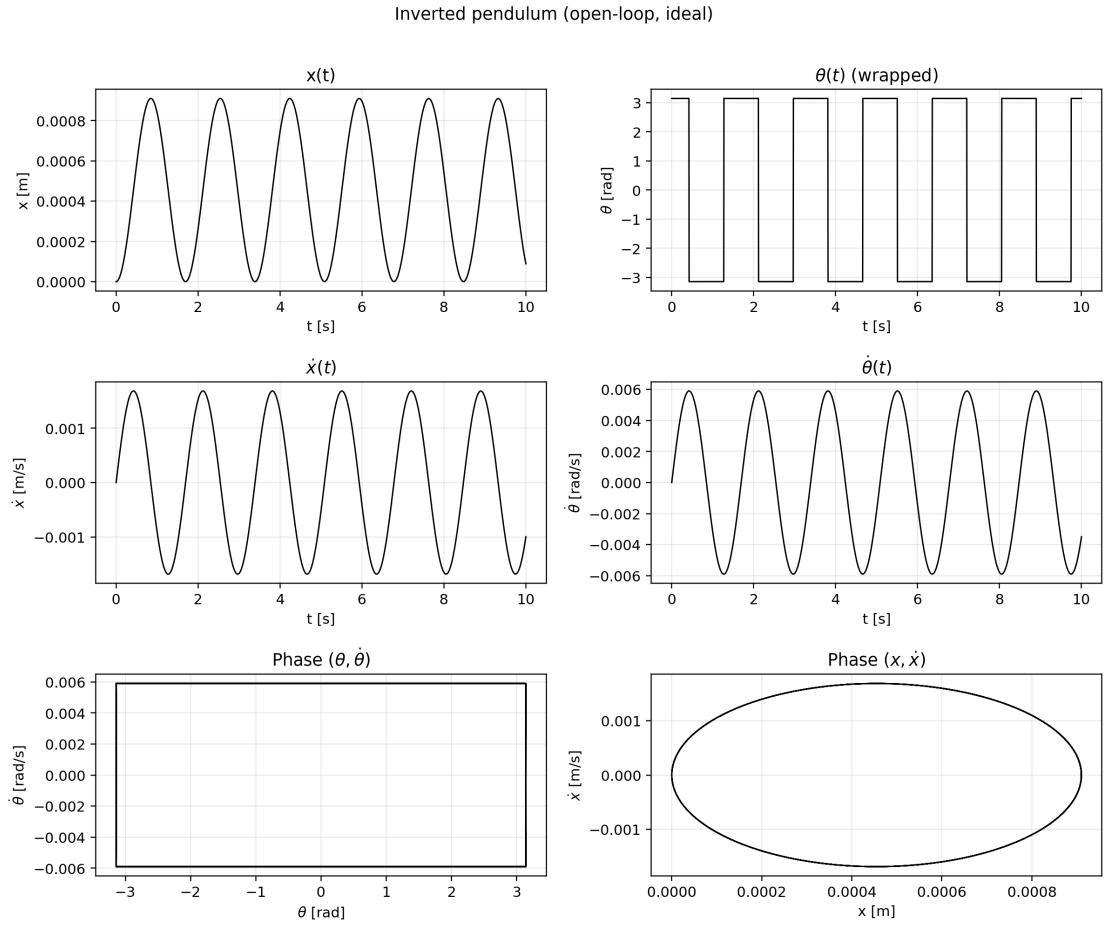


Figure 6.3 Cart–pole (open loop): $x, \theta, \dot{x}, \dot{\theta}$ and phase portraits

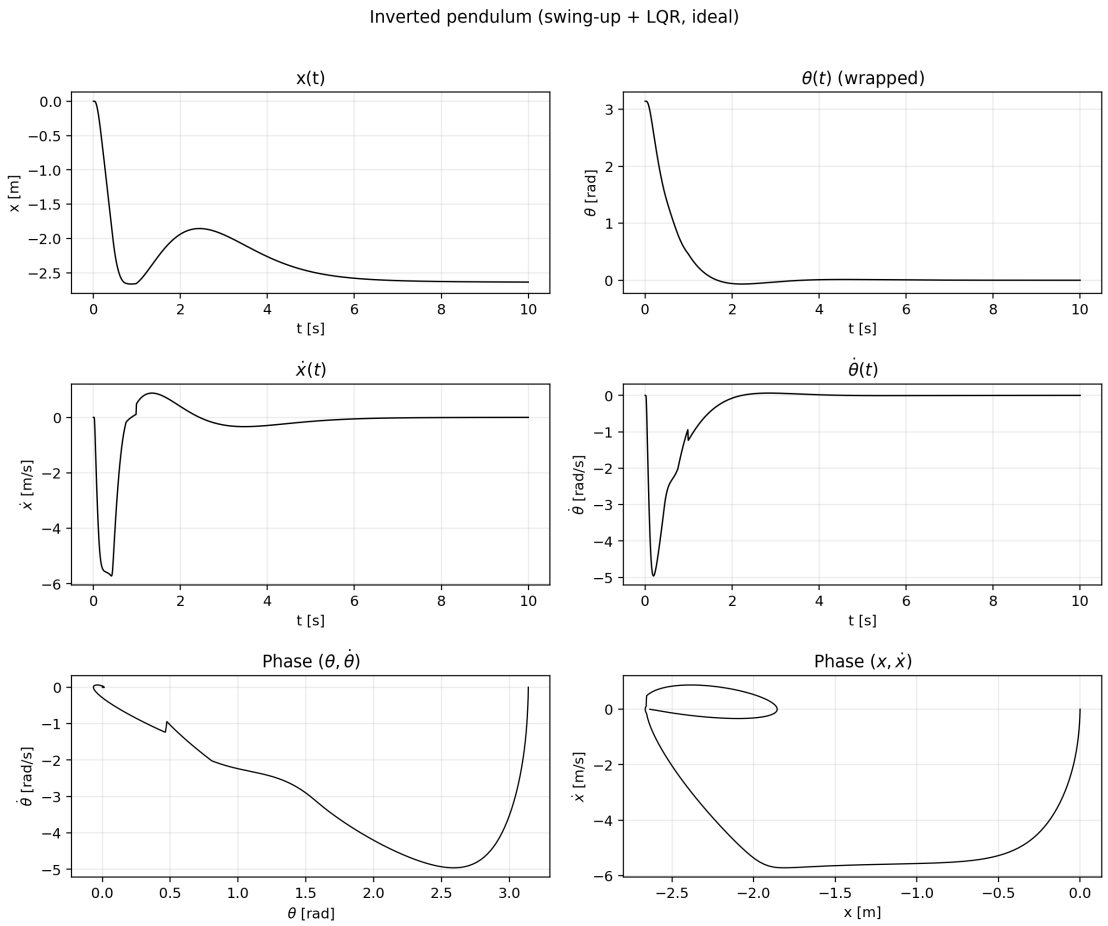


Figure 6.4 Cart–pole (swing-up + LQR): $x, \theta, \dot{x}, \dot{\theta}$ and phase portraits

Figure 6.4 presents the closed-loop case using the combined swing-up and LQR stabilisation strategy. The controller drives the pendulum toward the upright equilibrium and damps both the angular and cart motion, as reflected by the decay toward the origin in the phase portraits.

DC motor

Figure 6.5 presents the step response of the DC motor model. A constant input voltage produces a rapid current transient followed by acceleration of the rotor; the angular position grows as the integral of the speed.

Van der Pol oscillator

Figures 6.6 illustrate the Van der Pol oscillator for increasing nonlinearity parameter μ. For small μ the oscillations are close to sinusoidal; as μ increases, the dynamics transitions toward relaxation oscillations with slow drifts and fast jumps. In each case, the phase portrait approaches a stable limit cycle.

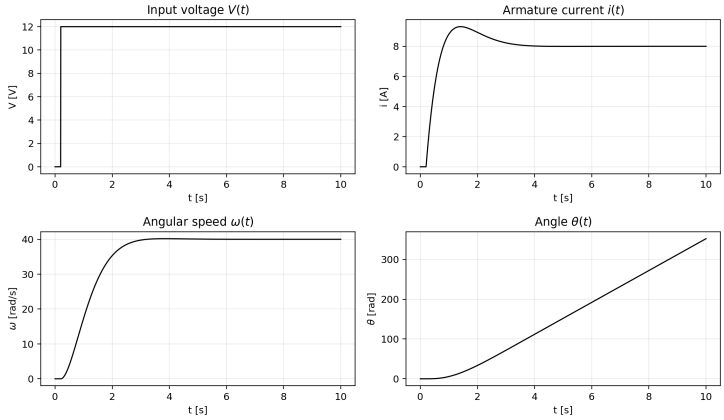


Figure 6.5 DC motor: V, i, ω, θ step response

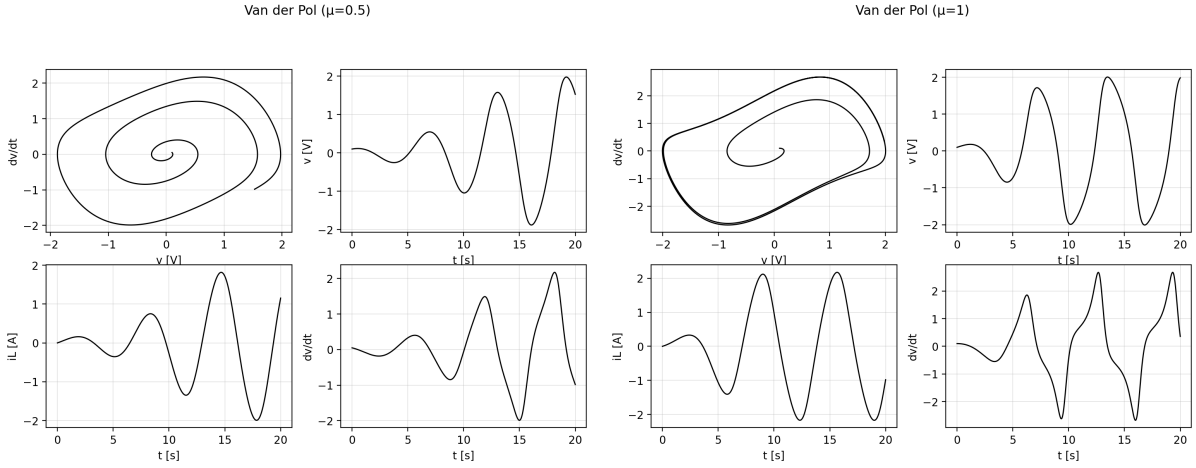


Figure 6.6 Van der Pol: $(v, \dot{v})$ and time histories for $\mu = 0.5$ and $\mu = 1.0$

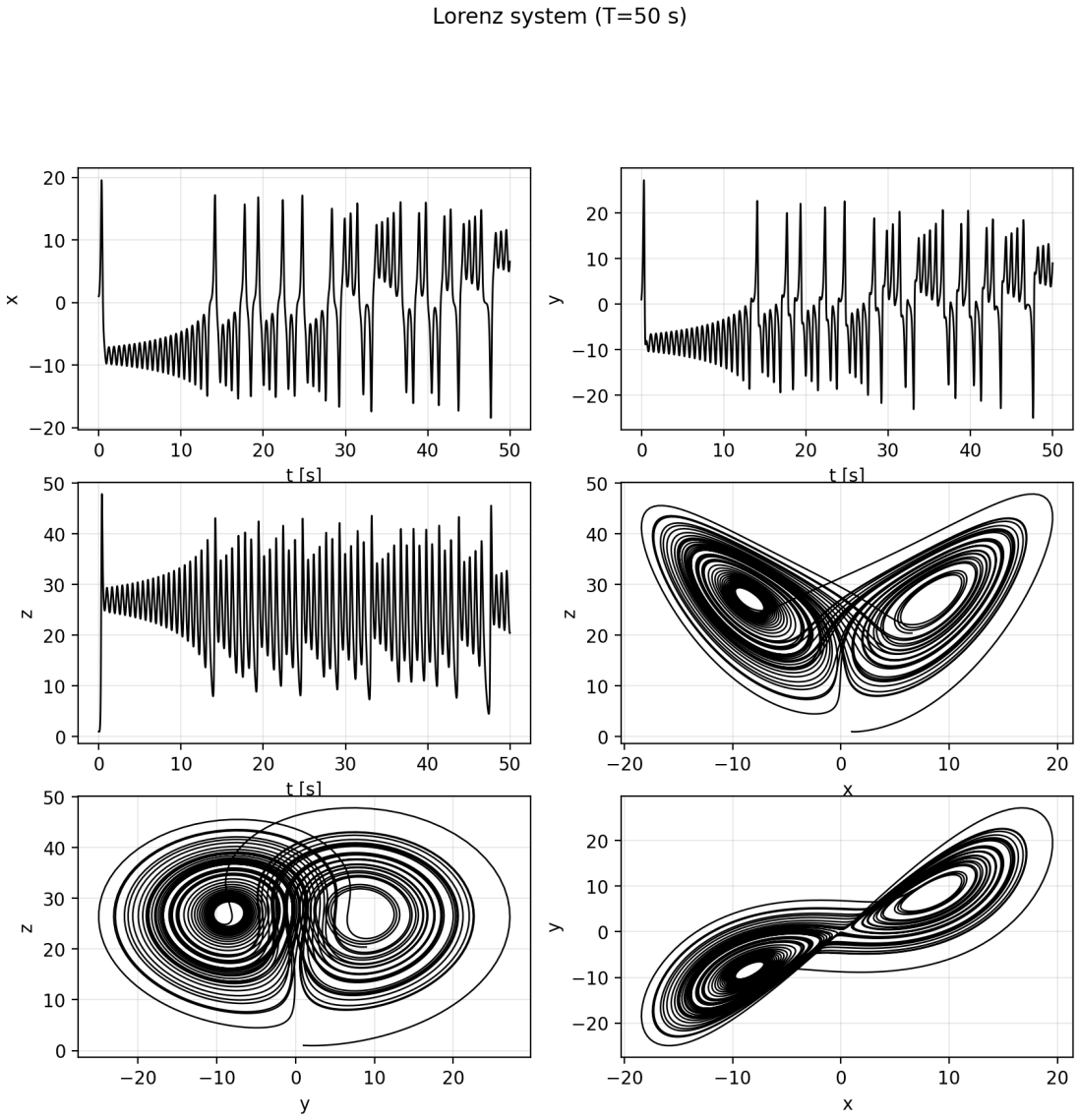


Figure 6.7 Lorenz: x, y, z time histories and planar projections

Lorenz system

Figure 6.7 shows a longer Lorenz simulation horizon to capture the characteristic attractor geometry. The time histories remain bounded but aperiodic, and the phase-plane projections reveal the familiar butterfly-shaped structure.

6.3 Numerical convergence study

This section assesses numerical convergence of the simulator integration pipeline by comparing solutions obtained under progressively tighter error tolerances. Two representative systems are used: (i) a damped single pendulum (smooth, low-dimensional dynamics) and (ii) the Lorenz system (chaotic, sensitive dynamics). The goal is to verify that tightening tolerances yields trajectories that converge to a high-accuracy *proxy*

reference on a fixed sampling grid, and to justify a practical default tolerance range for subsequent experiments.

Protocol

For each model, the same scenario (parameters and initial conditions) is integrated using multiple solver methods and tolerance pairs $(\texttt{rtol}, \texttt{atol})$. All runs are evaluated on an identical uniform time grid

$$t_k = k\Delta t, \qquad k = 0, \ldots, N-1, \qquad \Delta t = \frac{T}{N-1},$$

implemented via `t_eval`. This ensures point-wise comparability across methods and tolerances without requiring post-hoc interpolation.

The pendulum is simulated over $T_p = 10\,\mathrm{s}$ with initial state $(\theta(0), \dot{\theta}(0))^\top = (0.8, 0.0)^\top$ (using the same parameters as in Sec. 6.2). The Lorenz system uses the classical parameter set $(\sigma, \rho, \beta) = (10, 28, 8/3)$ and initial state $(x(0), y(0), z(0))^\top = (1, 1, 1)^\top$. Because the Lorenz system exhibits chaotic divergence of nearby trajectories, convergence is assessed on a short horizon $[0, T_\ell]$ with $T_\ell = 5\,\mathrm{s}$ to separate numerical error from intrinsic exponential separation.

The following integration methods are compared: `RK45` (baseline explicit Runge–Kutta), `DOP853` (high-order explicit Runge–Kutta), and `Radau` (implicit method). A high-accuracy *proxy reference* trajectory is computed using `DOP853` with very tight tolerances $(\texttt{rtol}, \texttt{atol}) = (10^{-12}, 10^{-15})$ on the same t_k grid.

A logarithmic tolerance grid is used:

$$\texttt{rtol} \in \{10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\}, \qquad \texttt{atol} = 10^{-3}\,\texttt{rtol}.$$

This range intentionally spans regimes from clearly under-resolved to effectively converged trajectories.

Error metrics

Let $x_{\mathrm{tol}}(t_k)$ denote the state computed with a given tolerance pair and $x_{\mathrm{ref}}(t_k)$ the proxy reference state. The point-wise residual is defined as

$$e(t_k) = \left\| x_{\mathrm{tol}}(t_k) - x_{\mathrm{ref}}(t_k) \right\|_2 ,$$

where $\|\cdot\|_2$ denotes the Euclidean (ℓ_2) norm over the full state vector, yielding a single scalar error value at each sample time t_k. Two aggregated metrics are reported:

$$e_{\max} = \max_k e(t_k), \qquad e_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_k e(t_k)^2}.$$

For the Lorenz system, these metrics are computed only on $t \in [0, T_\ell]$ to avoid conflating numerical error with chaos-driven separation at longer horizons.

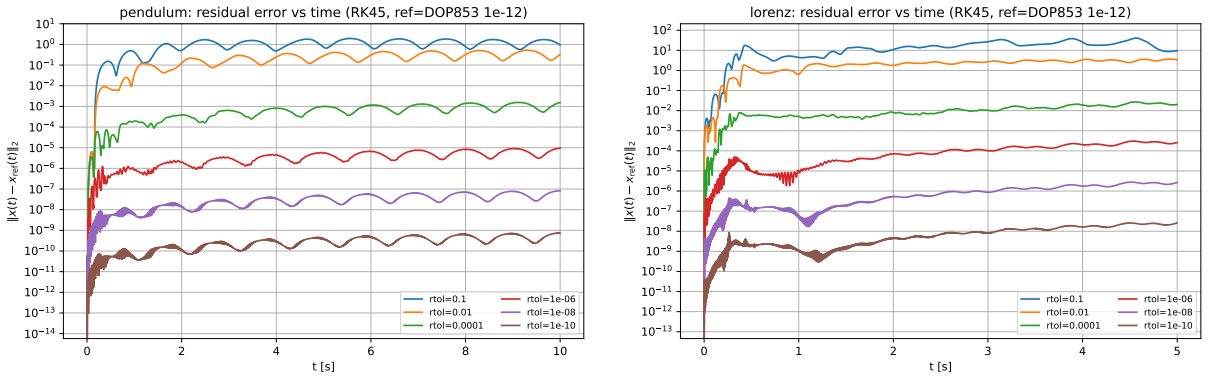
Summary convergence results (error metrics and solver statistics) for the pendulum and Lorenz benchmarks are reported in Tables 6.1 and 6.2.

Table 6.1 Convergence summary for the single pendulum (errors relative to the reference trajectory).

Model	Method	rtol	atol	$e\_\max$	e_RMS	runtime [s]	nfev
pendulum	DOP853	1×10^{-10}	1×10^{-13}	3.593×10^{-10}	8.438×10^{-11}	4.082×10^{-2}	1958
pendulum	DOP853	1×10^{-8}	1×10^{-11}	4.590×10^{-8}	1.019×10^{-8}	2.386×10^{-2}	1160
pendulum	DOP853	1×10^{-6}	1×10^{-9}	1.259×10^{-5}	2.626×10^{-6}	1.516×10^{-2}	665
pendulum	DOP853	1×10^{-4}	1×10^{-7}	7.279×10^{-4}	1.373×10^{-4}	1.291×10^{-2}	470
pendulum	DOP853	1×10^{-2}	1×10^{-5}	3.878×10^{-2}	1.018×10^{-2}	6.742×10^{-3}	263
pendulum	DOP853	1×10^{-1}	1×10^{-4}	2.168×10^{-1}	6.699×10^{-2}	5.824×10^{-3}	221
pendulum	RK45	1×10^{-10}	1×10^{-13}	7.559×10^{-10}	3.517×10^{-10}	2.040×10^{-1}	5906
pendulum	RK45	1×10^{-8}	1×10^{-11}	8.333×10^{-8}	3.895×10^{-8}	6.282×10^{-2}	2402
pendulum	RK45	1×10^{-6}	1×10^{-9}	9.498×10^{-6}	4.538×10^{-6}	2.417×10^{-2}	1118
pendulum	RK45	1×10^{-4}	1×10^{-7}	1.554×10^{-3}	7.813×10^{-4}	1.053×10^{-2}	440
pendulum	RK45	1×10^{-2}	1×10^{-5}	5.226×10^{-1}	2.755×10^{-1}	5.328×10^{-3}	146
pendulum	RK45	1×10^{-1}	1×10^{-4}	1.942	1.192	3.806×10^{-3}	122
pendulum	Radau	1×10^{-10}	1×10^{-13}	5.909×10^{-11}	1.118×10^{-11}	1.383	22 988
pendulum	Radau	1×10^{-8}	1×10^{-11}	6.085×10^{-9}	1.088×10^{-9}	4.494×10^{-1}	7412
pendulum	Radau	1×10^{-6}	1×10^{-9}	7.335×10^{-7}	1.204×10^{-7}	1.541×10^{-1}	2287
pendulum	Radau	1×10^{-4}	1×10^{-7}	6.183×10^{-5}	1.567×10^{-5}	3.795×10^{-2}	763
pendulum	Radau	1×10^{-2}	1×10^{-5}	9.109×10^{-3}	3.869×10^{-3}	1.838×10^{-2}	263
pendulum	Radau	1×10^{-1}	1×10^{-4}	1.137×10^{-1}	6.026×10^{-2}	1.479×10^{-2}	210

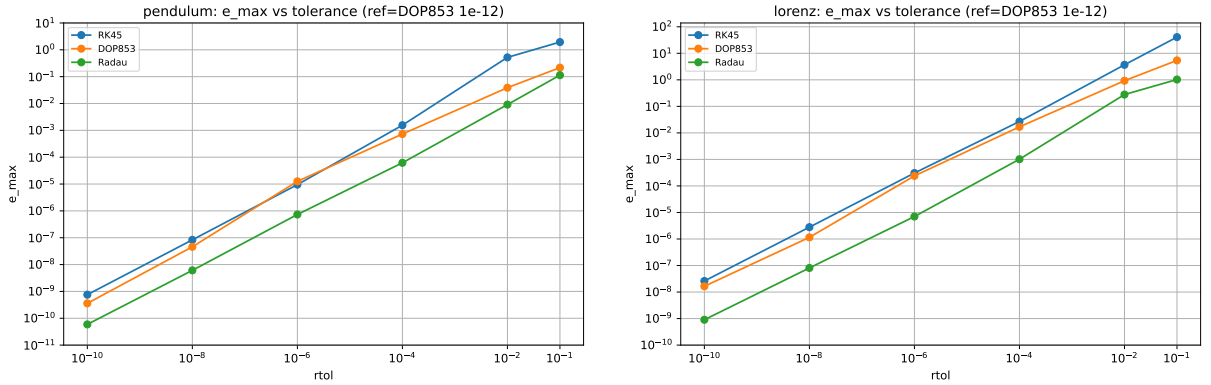
Table 6.2 Convergence summary for the Lorenz system on $t \in [0, 5\,\mathrm{s}]$ (errors relative to the reference trajectory).

Model	Method	rtol	atol	$e\_\max$	e_RMS	runtime [s]	nfev
lorenz	DOP853	1×10^{-10}	1×10^{-13}	1.650×10^{-8}	2.099×10^{-9}	8.830×10^{-2}	2873
lorenz	DOP853	1×10^{-8}	1×10^{-11}	1.152×10^{-6}	1.626×10^{-7}	3.122×10^{-2}	1802
lorenz	DOP853	1×10^{-6}	1×10^{-9}	2.388×10^{-4}	2.971×10^{-5}	1.761×10^{-2}	1049
lorenz	DOP853	1×10^{-4}	1×10^{-7}	1.684×10^{-2}	9.553×10^{-3}	1.538×10^{-2}	662
lorenz	DOP853	1×10^{-2}	1×10^{-5}	9.233×10^{-1}	3.230×10^{-1}	1.362×10^{-2}	353
lorenz	DOP853	1×10^{-1}	1×10^{-4}	5.402	2.082	1.332×10^{-2}	341
lorenz	RK45	1×10^{-10}	1×10^{-13}	2.609×10^{-8}	1.181×10^{-8}	1.576×10^{-1}	6194
lorenz	RK45	1×10^{-8}	1×10^{-11}	2.791×10^{-6}	1.255×10^{-6}	7.072×10^{-2}	2492
lorenz	RK45	1×10^{-6}	1×10^{-9}	3.063×10^{-4}	1.379×10^{-4}	3.627×10^{-2}	1010
lorenz	RK45	1×10^{-4}	1×10^{-7}	2.685×10^{-2}	1.258×10^{-2}	1.274×10^{-2}	488
lorenz	RK45	1×10^{-2}	1×10^{-5}	3.683	2.399	9.996×10^{-3}	242
lorenz	RK45	1×10^{-1}	1×10^{-4}	4.080×10^{1}	1.838×10^{1}	9.006×10^{-3}	320
lorenz	Radau	1×10^{-10}	1×10^{-13}	9.028×10^{-10}	1.892×10^{-10}	1.456	27 767
lorenz	Radau	1×10^{-8}	1×10^{-11}	8.098×10^{-8}	1.877×10^{-8}	4.716×10^{-1}	9050
lorenz	Radau	1×10^{-6}	1×10^{-9}	7.026×10^{-6}	1.919×10^{-6}	1.457×10^{-1}	2921
lorenz	Radau	1×10^{-4}	1×10^{-7}	1.017×10^{-3}	4.152×10^{-4}	5.197×10^{-2}	1014
lorenz	Radau	1×10^{-2}	1×10^{-5}	2.783×10^{-1}	1.532×10^{-1}	2.238×10^{-2}	431
lorenz	Radau	1×10^{-1}	1×10^{-4}	1.028	5.212×10^{-1}	2.545×10^{-2}	422



(a) Pendulum (b) Lorenz

Figure 6.8 Residual error $e(t)$ relative to the proxy reference for RK45 (logarithmic y-axis).



(a) Pendulum (b) Lorenz

Figure 6.9 Maximum residual e_{max} versus tolerance `rtol` (log–log axes)

Results and discussion

Figure 6.8 shows the time evolution of the residual $e(t)$ for RK45 across the tolerance grid. The pendulum exhibits a gradual and largely monotonic reduction in residual level as tolerances are tightened, indicating consistent behaviour of the solver wrapper on smooth dynamics. For Lorenz, residuals increase more rapidly in time, which is expected for chaotic dynamics; therefore the evaluation is restricted to $t \in [0, 5\,s]$.

Figure 6.9 summarizes the maximum residual e_{max} as a function of `rtol` for all compared methods (log–log axes). A near-monotonic decrease of e_{max} with tighter tolerances indicates consistent numerical behaviour and supports using tolerance settings as an effective accuracy knob. Deviations from monotonicity (if present at the tightest settings) are typically attributable to floating-point limits and solver-internal step selection heuristics rather than systematic wrapper errors.

The full numerical summary (including e_{max}, e_{RMS}, and solver cost statistics such as runtime and number of function evaluations `nfev`) is reported in Tables 6.1 and 6.2.

Across both problems, the tabulated results confirm the expected accuracy–cost trade-off: tightening tolerances reduces $e_{\max}$ and e_{RMS} but increases runtime and nfev. For the explicit methods, DOP853 consistently reaches lower errors than RK45 at comparable (or lower) cost at tighter tolerances, while Radau achieves the smallest residuals at a given tolerance but with a clear runtime overhead due to implicit integration. A practical compromise visible in both tables is that moving from $\mathtt{rtol} = 10^{-4}$ to 10^{-6} yields a substantial error reduction with only a modest runtime increase, whereas gains beyond roughly 10^{-8} are progressively smaller relative to the added solver work. For both systems, tighter tolerances increase nfev for all methods, and the relative ordering between methods largely matches the observed runtime curves. This confirms that, for these benchmarks, runtime is dominated by the amount of solver work (internal stage/step evaluations) rather than by fixed overhead.

6.4 Computational performance

This section evaluates the computational cost of simulations generated by the *Dynamic System Simulator*. The goal is twofold: (i) to provide a baseline benchmark across the implemented models under a single default solver configuration, and (ii) to characterize the accuracy–cost trade-off by measuring how runtime and solver work scale with tolerance settings.

Measurement protocol

All timings are measured as wall-clock runtime of a single call to the solver wrapper (integration only; no figure rendering). To reduce noise, each benchmark is repeated multiple times and the median runtime is reported. In addition to runtime, the solver-provided statistic nfev (number of right-hand-side evaluations) is recorded as a hardware-independent proxy of computational effort.

For experiments that use a uniform output grid, the numerical solution is evaluated on a fixed time vector

$$t_k = k\Delta t, \qquad k = 0, \ldots, N-1, \qquad \Delta t = \frac{T}{N-1},$$

passed via t_eval. Here N denotes the number of reported output samples (the length of t_eval), not the number of internal adaptive solver steps. In the scripts used in this chapter, the grid is defined through the sampling rate fps as $N = T \cdot \mathrm{fps} + 1$.

For the baseline comparison, scenarios are generated under a single shared configuration via tools/ch6_perf_baseline_uniform.py: DOP853, $(\mathtt{rtol}, \mathtt{atol}) = (10^{-4}, 10^{-7})$, horizon $T = 10\,\mathrm{s}$, and a uniform output grid with $N = 2001$ samples ($\Delta t = 0.005\,\mathrm{s}$). Each timing is repeated $R = 5$ times and the median is reported.

Table 6.3 Baseline computational cost across implemented models under a uniform benchmark (median over repeated runs)

Model	n	T	N	Method	rtol	atol	runtime [s]	nfev
dc_motor	2	10	2001	DOP853	1e-04	1e-07	6.808e-03	650
pendulum	2	10	2001	DOP853	1e-04	1e-07	6.669e-03	485
vanderpol	2	10	2001	DOP853	1e-04	1e-07	4.614e-03	359
lorenz	3	10	2001	DOP853	1e-04	1e-07	1.494e-02	1295
double_pendulum	4	10	2001	DOP853	1e-04	1e-07	3.604e-02	1754
inverted_open	4	10	2001	DOP853	1e-04	1e-07	8.797e-03	431
inverted_closed	4	10	2001	DOP853	1e-04	1e-07	7.569e+00	132800

Baseline benchmark across implemented models

As a baseline, each model is simulated on the same horizon ($T = 10\,s$) under the same uniform output grid (reported as N in the table), using the solver configuration DOP853 with $(\texttt{rtol}, \texttt{atol}) = (10^{-4}, 10^{-7})$. Table 6.3 reports the median runtime and the corresponding nfev for each model.

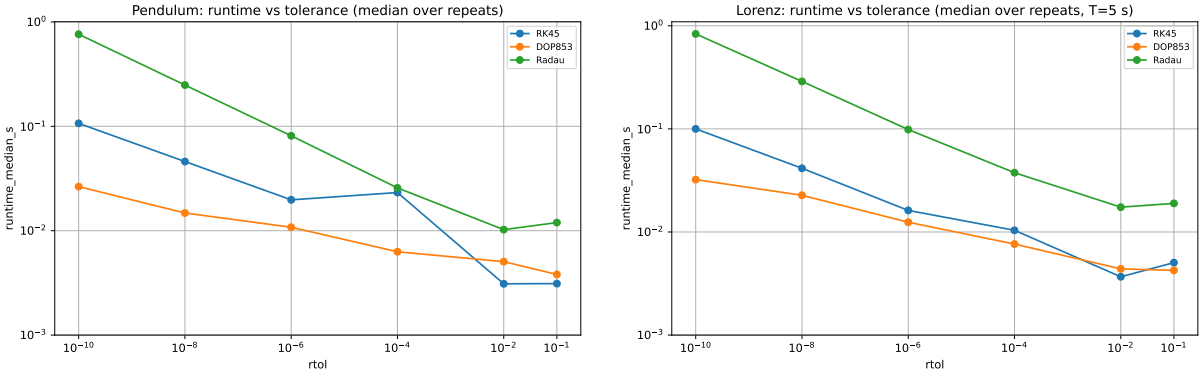
For the open-loop dynamical cores (pendulum, Van der Pol, DC motor, Lorenz, double pendulum, and the open-loop cart–pole), the median runtime remains in the millisecond range, with the double pendulum being the most expensive among the open-loop benchmarks due to its coupled four-state dynamics. The inverted_open (cart–pole without feedback) run is comparable to other four-state models.

The inverted_closed case exhibits a pronounced cost spike (seconds rather than milliseconds). This is consistent with the very large increase in nfev, which indicates that the adaptive solver is forced to take substantially more internal stages/steps. In the simulator, the closed-loop cart–pole integrates the plant together with a nonlinear control law evaluated inside the right-hand side at every solver call. Two effects can jointly explain the observed behaviour: (i) *non-smooth closed-loop dynamics* (e.g., switching between swing-up and stabilisation, saturation, or discontinuous logic) which tends to reduce admissible step sizes for explicit adaptive Runge–Kutta methods, and (ii) *implementation overhead inside the dynamics callback*, if controller-related quantities are recomputed too frequently (e.g., repeated gain construction, repeated coordinate transforms, or extra diagnostics evaluated per RHS call). In practice, this means that while the plant itself is not dramatically harder than other four-state models, the *closed-loop evaluation strategy* dominates the runtime in this scenario.

Runtime scaling with tolerance

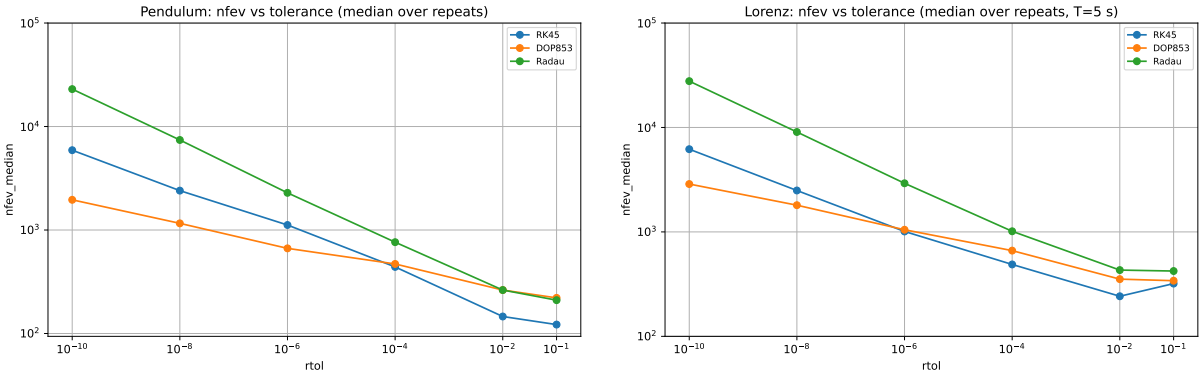
To study how computational cost scales with accuracy requirements, a tolerance sweep is performed for two representative systems: the single pendulum and the Lorenz system. The sweep compares RK45, DOP853, and Radau over a logarithmic tolerance grid

$$\texttt{rtol} \in \{10^{-1}, 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\}, \qquad \texttt{atol} = 10^{-3} \cdot \texttt{rtol}.$$



(a) Pendulum (b) Lorenz (T = 5 s)

Figure 6.10 Median runtime versus solver tolerance `rtol` (log–log axes) for selected methods. The Lorenz benchmark uses a shorter horizon to keep comparisons meaningful for chaotic dynamics



(a) Pendulum (b) Lorenz (T = 5 s)

Figure 6.11 Median number of right-hand-side evaluations `nfev` versus solver tolerance `rtol` (log–log axes)

For the Lorenz system, the sweep is evaluated on a shorter simulation horizon (T = 5 s) to match the convergence window used in Sec. 6.3 and to keep comparisons consistent.

Figure 6.10 shows the median runtime as a function of `rtol`. As expected, tighter tolerances generally increase runtime, although the slope depends on the method. The solver work proxy `nfev` is reported in the corresponding summary tables and provides a more hardware-independent indicator of computational effort than wall-clock time.

Table 6.3 (uniform baseline) and the tolerance-sweep figures (Figures 6.10– 6.11) are obtained under different scenario settings (e.g., horizon and output sampling density), so their *absolute* runtimes should not be compared directly; the intended comparison is the *within-sweep* scaling with `rtol` and the corresponding solver work proxy `nfev`.

Practical recommendations

The uniform baseline benchmark (Table 6.3) confirms that most implemented models can be simulated within milliseconds under a shared configuration, which is suitable for interactive educational use. The closed-loop inverted pendulum case is a notable exception: its cost is dominated by closed-loop evaluation and solver step-size reduction, so improving its performance is primarily a software/control-implementation task rather than a solver choice alone.

The tolerance sweep indicates diminishing runtime gains when loosening tolerances beyond approximately $\texttt{rtol} \approx 10^{-4}$ for representative problems: increasing `rtol` improves runtime only modestly once overheads and coarse step-size regimes dominate. For numerical studies requiring visibly improved agreement with a reference trajectory (Sec. 6.3), tolerances can be tightened at the cost of higher `nfev` and longer runtimes.

6.5 Practical limitations of the GUI

Perceived GUI responsiveness depends on two coupled factors: (i) numerical integration time and (ii) the cost of transferring and rendering the resulting trajectories in the browser. For computationally demanding scenarios (notably the inverted pendulum task), solver runtime can dominate the end-to-end update time. For long horizons or dense sampling, the dashboard payload (number of points, number of animation frames and Plotly figure complexity) becomes a second bottleneck because the data must be serialized, transferred to the client, and rendered interactively. The current version prioritises correctness and reproducibility over a fully polished UX; the limitations below are therefore documented explicitly and indicate directions for future work.

The main practical limitations are as follows. For stiff or complex dynamics, or for models with expensive right-hand-side evaluations (e.g., the cart–pole), solver runtime can dominate the end-to-end update time; in practice, reducing the output density (larger extttdt / lower animation FPS) and relaxing tolerances (extttrtol, extttatol) improves responsiveness at the cost of visual smoothness and/or accuracy.

Dense trajectories and long animations also increase the size of Plotly figures that must be transferred to the browser. Large payloads may hit Streamlit WebSocket message-size limits and can slow down rendering even when the solver is fast. The current implementation relies on user-configurable limits and does not yet include automatic payload-aware adaptation.

When many control groups are expanded, the sidebar becomes long and requires scrolling, which can reduce the visible visualisation area during tuning. A more compact layout (e.g., sticky panels and compact controls) is a clear direction for future work. Finally, because Streamlit executes runs synchronously, the UI cannot provide background execution, cancellation, or queuing while a run is computing. The GUI is therefore best suited for single-run exploration rather than batch work; it focuses on configuring and inspecting one run at a time, while systematic sweeps and side-by-side comparisons are performed using the offline scripts and logged outputs.

Chapter 7

Conclusions

7.1 Recap of the thesis

This thesis developed an open and educational simulator for continuous-time dynamical systems, designed to support transparent experimentation with parameters, initial conditions and solver settings. The work combined a layered system design (Chapter 3), mathematically grounded state-space modelling (Chapter 2), a concrete software implementation with a GUI front-end (Chapter 5), and an experimental validation and benchmarking campaign (Chapter 6) to assess correctness, numerical behaviour and practical usability.

7.2 Key outcomes

From the architectural perspective, the thesis delivered a coherent execution structure that separates responsibilities while keeping the overall workflow auditable. The simulator enforces a clear boundary between the reusable numerical core and the user-facing layers, so that the same execution path can be triggered by the GUI or by offline scripts, and the resulting outputs can be visualised and optionally persisted for reproducibility.

From the implementation perspective, the simulator provides a uniform solver-facing interface for heterogeneous systems. Mechanical, electrical and abstract benchmark models are handled consistently as state-space systems providing a right-hand side compatible with `solve_ivp`. Optional components (such as controllers and composition logic) are integrated through the same pipeline rather than through model-specific code paths, which improves maintainability and supports incremental extension without rewriting the solver or visualisation code.

From the validation perspective, the experimental results confirm that the implemented models reproduce characteristic qualitative behaviours and that the numerical pipeline exhibits expected convergence trends as tolerances are tightened. Baseline runtime measurements further indicate that interactive exploration is feasible for most scenarios under practical default settings, while also identifying cases where end-to-end

responsiveness is limited primarily by integration cost and/or by the cost of transferring and rendering dense visualisations in the GUI.

7.3 Limitations

The simulator prioritises clarity, extensibility and didactic usefulness over maximum efficiency and full generality. The numerical backend relies on general-purpose ODE solvers, which is robust for the target systems but does not exploit specialised methods (e.g., structure-preserving integration or advanced event handling). The model library is intentionally limited to a representative set suitable for a thesis-scale project, and broader coverage would require additional abstractions and a larger validation campaign. Finally, the Streamlit-based GUI is functional but not fully polished: long control panels, synchronous execution and large visualisation payloads can reduce usability for dense trajectories, long animations and computationally demanding scenarios, and systematic multi-run comparison is better handled through offline scripts.

7.4 Future work

Future work can strengthen the project along several axes while preserving the current architecture. On the numerical side, the solver layer can be extended with richer event/discontinuity handling and additional methods targeted at specific system classes. On the modelling and control side, the library can be expanded with additional benchmark systems, controller families and analysis utilities to support a wider range of laboratory exercises. On the GUI side, usability and performance can be improved through a more compact layout, payload-aware plotting/animation strategies, and a dedicated workspace for comparing multiple runs; longer-running simulations could benefit from job-style execution and cancellation. Finally, reproducibility can be strengthened by recording environment metadata (package versions) and by packaging the project together with curated example configurations and teaching exercises, enabling future classroom-oriented evaluation of learning outcomes.

Bibliography

[HLW06] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer, 2nd edition, 2006.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. Array programming with numpy. *Nature*, 585:357–362, 2020.

[HNW93] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 1993.

[Hun07] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[Kha02] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, 3rd edition, 2002.

[Mod25] Modelica Association. Modelica language specification. https://specification.modelica.org/, 2025. Accessed: 2025-12-21.

[NR14] James W. Nilsson and Susan A. Riedel. *Electric Circuits*. Pearson, 10 edition, 2014.

[Plo25] Plotly. Plotly python documentation. https://plotly.com/python/, 2025. Accessed: 2025-12-19.

[Pyt25] Python Software Foundation. Python documentation. https://docs.python.org/3/, 2025. Accessed: 2025-12-19.

[SK16] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, 2nd edition, 2016.

[Str25] Streamlit. Streamlit documentation. https://docs.streamlit.io/, 2025. Accessed: 2025-12-19.

[The25a] The MathWorks, Inc. Matlab documentation. https://www.mathworks.com/help/matlab/index.html, 2025. Accessed: 2025-12-19.

[The25b] The MathWorks, Inc. Simulink documentation. https://www.mathworks.com/help/simulink/index.html, 2025. Accessed: 2025-12-19.

[TM13] Krzysztof Tchoń and R. Muszyński. *Analytic Mechanics*. Wrocław University of Technology, 2013.

[V+20] Pauli Virtanen et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.

[Wik25a] Wikipedia contributors. DC motor. https://en.wikipedia.org/wiki/DC_motor, 2025. Accessed: 2025-12-19.

[Wik25b] Wikipedia contributors. Double pendulum. https://en.wikipedia.org/wiki/Double_pendulum, 2025. Accessed: 2025-12-19.

[Wik25c] Wikipedia contributors. Energy-shaping control. https://en.wikipedia.org/wiki/Energy-shaping_control, 2025. Accessed: 2025-12-19.

[Wik25d] Wikipedia contributors. Inverted pendulum. https://en.wikipedia.org/wiki/Inverted_pendulum, 2025. Accessed: 2025-12-19.

[Wik25e] Wikipedia contributors. Linear–quadratic regulator. https://en.wikipedia.org/wiki/Linear%E2%80%93quadratic_regulator, 2025. Accessed: 2025-12-19.

[Wik25f] Wikipedia contributors. Lorenz system. https://en.wikipedia.org/wiki/Lorenz_system, 2025. Accessed: 2025-12-19.

[Wik25g] Wikipedia contributors. Pendulum. https://en.wikipedia.org/wiki/Pendulum, 2025. Accessed: 2025-12-19.

[Wik25h] Wikipedia contributors. Van der pol oscillator. https://en.wikipedia.org/wiki/Van_der_Pol_oscillator, 2025. Accessed: 2025-12-19.

[Wol25a] Wolfram Research, Inc. Wolfram language & system documentation center (mathematica). https://reference.wolfram.com/language/, 2025. Accessed: 2025-12-19.

[Wol25b] Wolfram Research, Inc. Wolfram system modeler documentation center. https://reference.wolfram.com/system-modeler/, 2025. Accessed: 2025-12-19.

List of Figures

Appendix A

Dynamic System Simulator (DSS) — User Guide

A.1 Purpose and scope

Dynamic System Simulator (DSS) is a Python-based project for simulating and visualising classic continuous-time dynamical systems (mechanical, electrical, and electromechanical) using a reusable numerical core and an interactive Streamlit graphical interface. The simulator supports two main workflows: (i) an interactive GUI for exploratory experiments and qualitative inspection, and (ii) offline experiment scripts for repeatable runs that generate figures/tables for evaluation.

A.2 Where to download

The project is hosted as a public Git repository:

`https://github.com/Kacperencki/DynamicSystemSimulator`

Recommended method (clone the repository):

```
git clone https://github.com/Kacperencki/DynamicSystemSimulator.git
cd DynamicSystemSimulator
```

A.3 System and hardware requirements

Operating system

Windows 10/11, Linux, or macOS.

Hardware (practical)

- CPU: any modern 2+ core processor

- RAM: $\geqslant$ 8 GB recommended for dense sampling, long horizons, or high FPS animations
- GPU: not required

Software

- Python 3.10+ (recommended)
- `pip` (Python package installer)
- (optional) Git (to clone the repository)

A.4 Installation

After cloning the repository, run the following commands from the repository root.

1. Create and activate a virtual environment:

```
python -m venv .venv

# Windows:
.venv\Scripts\activate

# macOS/Linux:
source .venv/bin/activate
```

2. Install all dependencies:

```
python -m pip install --upgrade pip
python -m pip install -r requirements.txt
```

3. Verify the installation:

```
python -c "import dss; print('DSS ready')"
```

The `requirements.txt` file installs all packages needed to run the simulator: `numpy`, `scipy`, `pandas`, `matplotlib`, `plotly`, and `streamlit`.

A.5 Running the simulator

Interactive mode (Streamlit GUI)

From the project root run:

```
streamlit run streamlit_app.py
```

Streamlit prints a local URL (typically `http://localhost:8501`) which opens the GUI in a web browser.

Offline mode (repeatable experiments in `tools/`)

Offline experiments used to generate thesis artifacts are stored in `tools/`. Example runs (paths and arguments may be adjusted as needed):

```
python tools/ch6_generate_62_white_blacklines_with_inverted_v2.py
python tools/ch6_perf_baseline_uniform.py --out figures/chapter_05/section6.4 \
  --method DOP853 --rtol 1e-4 --atol 1e-7 --T 10 --fps 200 --repeats 5
```

Outputs are typically written into the thesis figures directory (`figures/chapter_05/...`).

A.6 Repository layout (orientation)

The repository is organised into the following top-level components:

- `dss/` — core simulation library (models, controllers, wrappers, solvers, logging)
- `apps/streamlit/` — Streamlit GUI application (systems registry, UI components, dashboards/runners)
- `tools/` — offline experiments and artifact generation scripts (e.g., Chapter 6 plots/tables)
- `docs/` and `mkdocs.yml` — project documentation (MkDocs)
- `tests/` — unit tests for models and controllers
- `streamlit_app.py` — main entrypoint to launch the GUI

A.7 GUI workflow (how to run one simulation)

A typical interactive run consists of the following steps:

1. Select a `Model` (e.g., Single pendulum, Lorenz, DC motor, Inverted pendulum).
2. Select a `Preset` (optional) to load a standard parameter set.
3. Select a `Mode` (if available), e.g. ideal vs damped, open-loop vs closed-loop.
4. Expand the left-panel sections to configure:
   - `Physical parameters` (model constants),
   - `Initial state` $x(t_0)$,
   - `Simulation time` (t_0, t_1, output sampling Δt),
   - `Numerical solver` (method, `rtol`, `atol`),
   - `Animation / performance` (animation FPS, maximum number of frames),

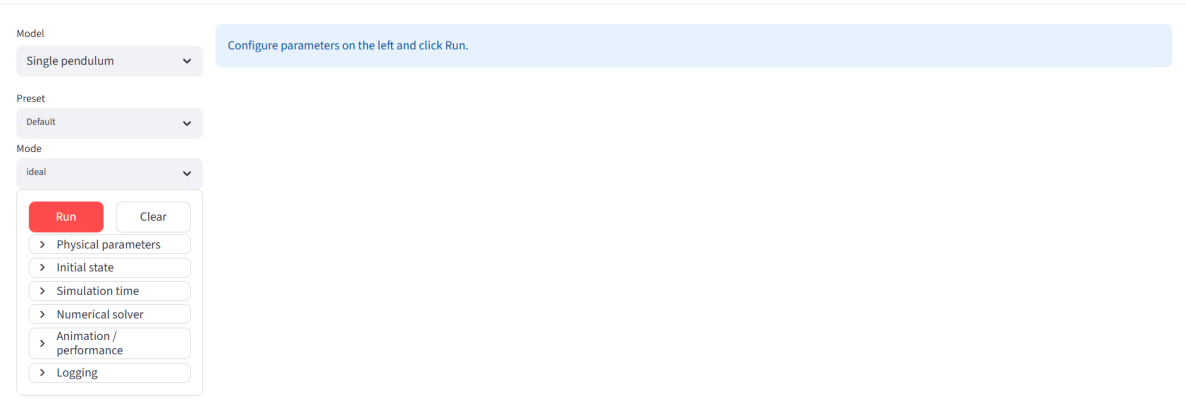


Figure A.1 GUI start view: select model/preset/mode and press Run

- (optional) Logging (store configuration and outputs).

5. Press Run to execute the simulation and update the results panel.
6. If an animation is available, use Play / Pause / Reset to inspect motion.

Sampling density vs solver accuracy

Two distinct settings influence what is displayed and how long the run takes:

- **Output sampling density** (e.g., Δt) determines how many points are produced for plotting/animation.
- **Integrator accuracy** is controlled by rtol/atol and the solver's adaptive step size.

High sampling density can slow down the GUI due to rendering and data transfer, even if numerical accuracy does not improve.

A.8 Screenshots (GUI usage example)

The three screenshots shown in Figures A.1–A.3 provide a minimal usage example for the Single pendulum model.

A.9 Outputs and reproducibility (logging)

DSS can store the configuration and numerical outputs of a run to support reproducibility. In the GUI, export/logging options (if enabled for the selected system) typically store:

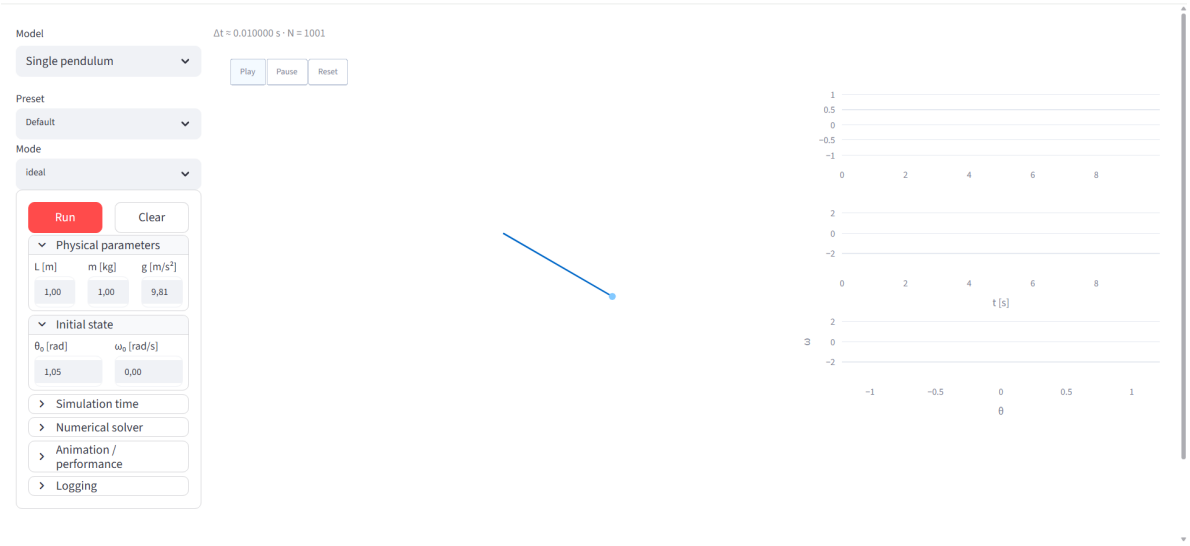


Figure A.2 Expanded control panel: parameters, initial state, time settings, solver, and logging

Figure A.3 Results view: animation controls and plots after a successful run

- a configuration snapshot (parameters, initial state, solver settings),
- numerical arrays (time vector and simulated trajectory),
- optional metadata (solver statistics and run identifiers).

A.10 Documentation

Additional project documentation is provided in `docs/`. If MkDocs is installed, a local documentation site can be started via:

```
mkdocs serve
```

A.11 Troubleshooting

`ModuleNotFoundError` / missing packages

Ensure the virtual environment is active, then reinstall:

```
python -m pip install --upgrade pip
python -m pip install -r requirements.txt
```

Streamlit does not start

Try launching Streamlit via the Python module:

```
python -m streamlit run streamlit_app.py
```

GUI is slow

Reduce the amount of data rendered:

- increase Δt (lower sampling density),
- lower animation FPS and/or maximum number of frames,
- shorten the simulation horizon $T = t_1 - t_0$,
- start with moderate tolerances (e.g., `rtol=1e-4`, `atol=1e-6`) and tighten only if needed.