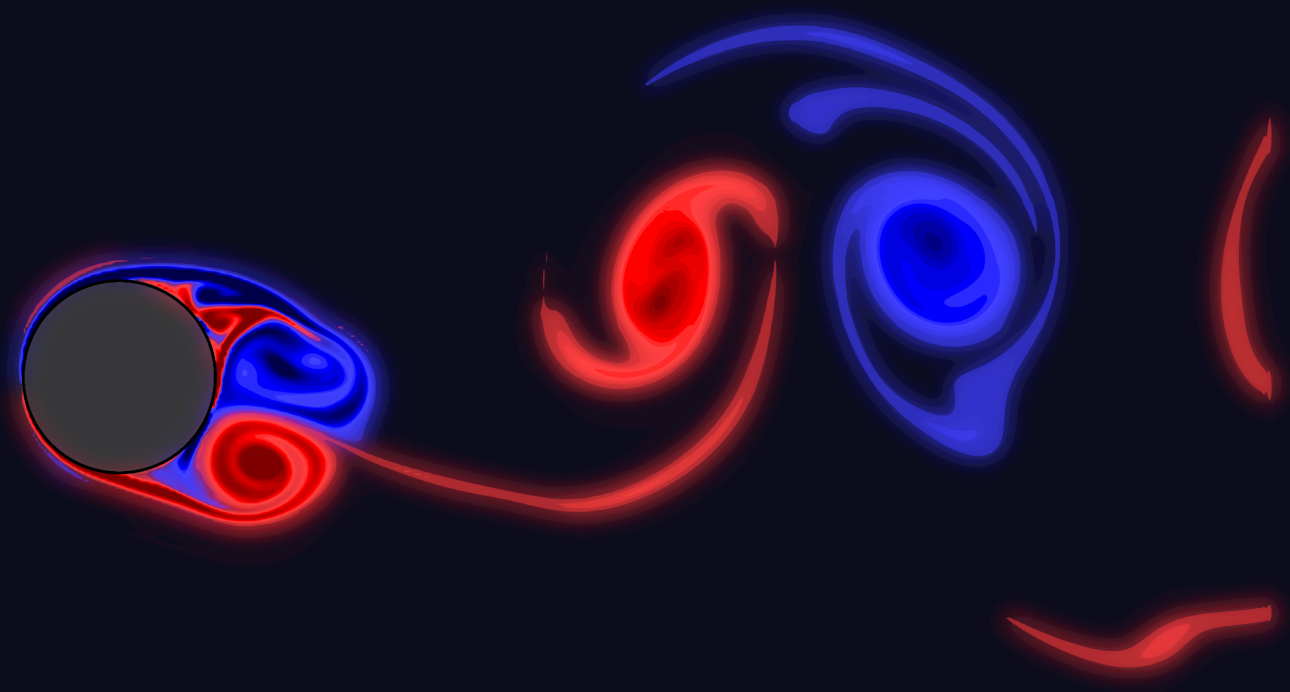
# Accelerating Incompressible CFD with Latent-Space Neural ODEs

## A hybrid autoencoder–Neural ODE framework with out-of-distribution-gated fallback, validated on unsteady cylinder flow

Msc Thesis

Matthias Claeys

Delft University of Technology



TU Delft

# Accelerating Incompressible CFD with Latent-Space Neural ODEs

## A hybrid autoencoder–Neural ODE framework with out-of-distribution-gated fallback, validated on unsteady cylinder flow

by

Matthias Claeys

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday July 6, 2026 at 11:00 AM.


| | | |
|---|---|---|
| Student number: | 5241936 | |
| Project duration: | May 15, 2025 – July 6, 2026 | |
| Thesis committee: | Dr. B. Font | TU Delft, supervisor |
| | Dr. J. Poblador Ibanez, | TU Delft |


Throughout this thesis, the author used Large Language Models (LLMs) in a supporting capacity. These models were primarily used to summarise literature, code suggestions and verification, and to refine the clarity and structure of the written text. They additionally served as a sparring partner for testing ideas and reasoning through approaches. All final decisions, interpretations, and conclusions in this work are solely those of the author



TU Delft

# Preface

This thesis was written for the completion of the Master of Science programme in Marine Technology at the Delft University of Technology, specifically the Ship Hydromechanics track. It marks the end of a long journey fuelled by curiosity in fluid dynamics and artificial intelligence (and caffeine). I have always known I wanted to be close to water, from dreaming of being an oceanographer like Jacques Cousteau as a kid, to wanting to be a surfboard shaper when I was a teen — and being lightly discouraged by my parents — and studying marine technology at TU Delft, there was always a common denominator: water.

The combination of the challenging theory behind computational fluid dynamics and the rapidly evolving field of artificial intelligence scratched an itch inside me and motivated me to keep learning and eventually pursue this thesis topic. Although the road to completion wasn't always easy, from all the long days, nights and weekends spent in the library overanalysing specific functions in my code so that it would not make my pc crash, to the sudden onset of confusion when a package update broke my whole framework for a week. This thesis has taught me valuable lessons, be they technical, theoretical, or deeply personal, and it will have contributed significantly to my further development as a person.

Curiosity and caffeine alone wouldn't have put me where I am today; I was greatly supported by the people around me. I would therefore like to thank my supervisor, Dr Bernat Font, who provided me with valuable guidance on both my thesis and my academic development, for which I am extremely grateful.

I also want to thank my fellow master's students, who motivated me to be in the library at 8 am for the past 2 years and with whom I shared stress, frustration, happiness, and a lot of beers at Bouwpub. So thank you, Tine, Frederick and Maxime. I also want to thank my friends who have been with me from my bachelor years, who I got to share a lot of courses with and were always a valuable resource for laughs and answers to assignment questions, so thank you, Bram and Rajan, I wouldn't have survived the *maritieme gang* for 6 years without you. I also want to specifically thank Evan, with whom I got to share weekly thesis meetings, and more than enough frustration about shared technical issues. Lastly, I want to thank Moeder Delftsche, who provided me with tons of friends and experiences, and made me feel right at home from my first days in Delft.

And of course, a very big thank you to my parents, who have endured me from when I was a nagging little brat to the grown man that I am now. I will be forever grateful for the opportunities and support they have given me, since I wouldn't be the person I am today without them. They have been, and will continue to be, my leading example for what perseverance can deliver in life.

*Matthias Claeys*
*Delft, June 2026*

# Summary

The computational cost of scale-resolving CFD is dominated, in most mesh-based incompressible solvers, by the time integration of the discretised Navier–Stokes equations and the pressure projection that enforces incompressibility at every step. This thesis investigates how far a learned latent-space surrogate can replace that step in an unsteady, vortex-shedding flow without degrading the physical fidelity of the simulation it accelerates.

The proposed framework couples a convolutional autoencoder, a latent neural ordinary differential equation (NODE), and a runtime out-of-distribution (OOD) gate to the differentiable incompressible solver WaterLily.jl. The autoencoder compresses each instantaneous velocity field, augmented with the boundary-data-immersion geometry field, into a 16-dimensional latent state, and is trained with a physics-informed objective that adds divergence and vorticity penalties to the $L_1$ reconstruction loss, so the decoded fields stay near-divergence-free and retain the shear-layer vorticity that sets the near-body loading. The NODE learns the latent dynamics and is fitted with multiple shooting, partitioning each trajectory into short segments to suppress the gradient blow-up that destabilises single-shooting training over the long, chaotic rollouts spanning several shedding cycles. At inference, a $k$-nearest-neighbour (KNN) monitor measures the distance between each predicted latent vector and the encoded training set; once a rollout drifts beyond a calibrated threshold, the prediction is rejected, and control returns to WaterLily, which re-stabilises the flow before the loop resumes. Repeated flags trigger retraining of the NODE on the newly gathered trajectories.

The framework is developed on two-dimensional flow past a circular cylinder at $Re = 2500$ and evaluated on $256^2$ and $512^2$ grids. In its in-distribution regime, the hybrid loop preserves both the engineering- and statistics-level observables: on the finer grid, the time-averaged drag is recovered to within $0.64\%$ over the accelerated windows, and the mean-velocity and Reynolds-stress fields correlate spatially with the reference at $\rho \geq 0.997$, with the fluctuating lift the only quantity to carry a visible rollout error (its r.m.s. amplitude decaying to a roughly $7\%$ error inside the hybrid regions). Once trained, a single latent rollout is about $85\times$ cheaper per convective time unit (CTU) than the reference time integration, giving an inference-loop speedup of roughly $1.48\times$ at the base resolution and climbing above $2\times$ on the refined $512^2$ grid, since the memory-bound solver cost scales with grid size while the rollout cost is fixed by the latent dimension.

The decisive limitation is economic rather than physical. Charging the one-off network training to the comparison collapses the effective speedup of a from-scratch run to roughly $0.07\times$ at the base point, about an order of magnitude slower than simply running WaterLily, because training consumes the large majority of the wall-clock budget and is not repaid within a single simulation window. The parameter studies expose one recurring trade-off behind this: narrowing the latent bottleneck, shortening the autoencoder schedule, or loosening the multiple-shooting group each simplifies the dynamics the NODE must learn and lengthens the average in-distribution rollout, but coarsens the recovered second-order statistics, so acceleration and turbulence-statistics fidelity are pulled against one another, with the OOD gate setting the operating point on that curve. The training cost is not irreducible: it amortises across reused pipelines, and trimming the schedule alone lifts the effective speedup to about $0.62$ on the finer grid. Pushed out of distribution to $Re = 5000$, the autoencoder smooths the held-out near-wall fluctuations and the instantaneous loading becomes unreliable (hybrid-region force errors approaching $9\%$) even as the time-averaged fields stay faithful, locating the autoencoder's reconstruction ceiling as the binding accuracy constraint. The work thus establishes that latent NODE rollouts can absorb part of the conventional time integration without sacrificing the integrated loading or the wake statistics, and identifies training cost, not the inference loop, as the bottleneck any practical deployment must overcome.

# Contents

# List of Figures

# List of Tables

# Nomenclature

## Abbreviations

| Abbreviation | Definition |
|---|---|
| AD | Automatic Differentiation |
| AE | Autoencoder |
| AI | Artificial Intelligence |
| BC | Boundary Condition |
| BDIM | Boundary Data Immersion Method |
| CC | Correlation Coefficient |
| CFD | Computational Fluid Dynamics |
| CFL | Courant–Friedrichs–Lewy (condition) |
| CNN | Convolutional Neural Network |
| CPU | Central Processing Unit |
| CTU | Convective Time Unit |
| DL | Deep Learning |
| DNS | Direct Numerical Simulation |
| GPU | Graphics Processing Unit |
| HPC | High-Performance Computing |
| KD-tree | $k$-Dimensional tree |
| KNN | $k$-Nearest Neighbours |
| LES | Large-Eddy Simulation |
| LSTM | Long Short-Term Memory |
| MAE | Mean Absolute Error |
| ML | Machine Learning |
| MLP | Multilayer Perceptron |
| MSE | Mean Squared Error |
| NN | Neural Network |
| NODE | Neural Ordinary Differential Equation |
| NSE | Navier–Stokes Equations |
| ODE | Ordinary Differential Equation |
| OFAT | One-Factor-at-a-Time |
| OOD | Out-of-Distribution |
| PDE | Partial Differential Equation |
| PINN | Physics-Informed Neural Network |
| POD | Proper Orthogonal Decomposition |
| PPE | Pressure-Poisson Equation |
| RANS | Reynolds-Averaged Navier–Stokes |
| RMS | Root Mean Square |
| ROM | Reduced-Order Model |
| RST | Reynolds-Stress Tensor |

# Symbols

All variables presented are non-dimensional, unless stated otherwise.

| Symbol | Definition | Unit |
|---|---|---|
| $C_{\text{base}}$ | Base channel count of the autoencoder | [–] |
| $C_{\text{in}}$ | Number of input channels | [–] |
| $C_D$ | Drag coefficient | [–] |
| $C_L$ | Lift coefficient | [–] |
| $C_L^{\text{rms}}$ | Root-mean-square lift coefficient | [–] |
| $d$ | Signed-distance function to the body surface | [–] |
| $d_k$ | KNN score (mean distance to the $k$ nearest neighbours) | [–] |
| $D$ | Cylinder diameter | [–] |
| $E$ | Number of autoencoder training epochs | [–] |
| $f_\phi$ | Neural ODE; learned latent dynamics network | [–] |
| $k$ | Number of nearest neighbours (OOD monitor) | [–] |
| $L$ | Characteristic length scale ($= D$) | [–] |
| $L_1$ | $L_1$ (mean-absolute) reconstruction loss | [–] |
| $L_x$, $L_y$ | Domain width and height | [–] |
| $\mathcal{L}$ | Loss / objective function | [–] |
| $\mathcal{L}_{\text{data}}$, $\mathcal{L}_{\text{physics}}$ | Data and physics terms of a physics-informed loss | [–] |
| $\mathcal{L}_{\text{div}}$, $\mathcal{L}_{\text{curl}}$ | Divergence and vorticity (curl) loss terms | [–] |
| $\mathcal{L}_{\text{ms}}$ | Multiple-shooting loss | [–] |
| $n$ | Number of spatial dimensions | [–] |
| $n_{\text{conv}}$ | Number of convolutional blocks | [–] |
| $n_{\text{dense}}$ | Number of dense bottleneck layers | [–] |
| $n_s$ | Multiple-shooting group size | [–] |
| $n_{\text{switch}}$ | Prediction cadence (CFD steps between attempts) | [–] |
| $\bar{n}_{\text{roll}}$ | Average rollout length | [–] |
| $N_s$ | Number of shooting segments | [–] |
| $N_u$ | Number of velocity components | [–] |
| $N_x$, $N_y$ | Number of grid points in $x$, $y$ | [–] |
| $N_z$ | Latent-space dimension | [–] |
| $N_\Omega$ | Number of fluid cells in the domain | [–] |
| $p$ | Pressure | [–] |
| $\mathcal{P}$ | Trained pipeline (autoencoder, NODE, KNN monitor) | [–] |
| $q$ | Threshold quantile of the OOD monitor | [–] |
| $Re$ | Reynolds number | [–] |
| $\mathcal{S}$, $\mathcal{S}_{\text{ref}}$ | Hybrid and reference simulation | [–] |
| $S_{\text{eff}}$ | Effective speedup (including training cost) | [–] |
| $S_{\text{loop}}$ | Inference-loop speedup | [–] |
| $t$ | Time | [s] |
| $t^*$ | Non-dimensional (convective) time | [–] |
| $T_{\text{wall}}$ | Wall-clock time | [s] |
| $T_{\text{wall}}^{\text{ref}}$ | Reference-solver wall time | [s] |
| $T_{\text{wall}}^{\text{h, tot}}$ | Total hybrid wall time (excluding training) | [s] |
| $T_{\text{wall}}^{\text{h, training}}$ | Network-training wall time | [s] |
| $u_i$, $\mathbf{u}$ | Velocity component / field | [m/s] |
| $\hat{\mathbf{u}}$ | Reconstructed (decoded) velocity field | [m/s] |
| $U$ | Free-stream / characteristic velocity | [m/s] |
| $V_i$ | Prescribed body velocity | [m/s] |
| $\mathbf{x}$ | Spatial coordinate (position vector) | [–] |
| $\mathbf{x}_{\text{in}}$ | Augmented network input $[\mathbf{u}\,;\,\boldsymbol{\mu}_0]$ | [–] |
| $x_c$, $y_c$ | Cylinder-centre coordinates | [–] |
| $\mathbf{z}$ | Latent state vector | [–] |
| $\tilde{\mathbf{z}}$ | Predicted latent state vector | [–] |

| Symbol | Definition | Unit |
|---|---|---|
| $\boldsymbol{\mathcal{T}}_z$ | Set of encoded training latent trajectories | [–] |
| $\tilde{\boldsymbol{\mathcal{T}}}_z$ | Rolled-out (predicted) latent trajectory set | [–] |
| $\varepsilon$ | Mean absolute error (field metric) | [–] |
| $\epsilon$ | Boundary-immersion thickness | [–] |
| $\eta$ | Learning rate | [–] |
| $\theta$ | Autoencoder (encoder/decoder) weights | [–] |
| $\lambda$ | Weight decay | [–] |
| $\lambda_c$ | Continuity weight (multiple shooting) | [–] |
| $\lambda_{c,r}$ | Retraining continuity weight | [–] |
| $\lambda_d$, $\lambda_p$ | Data and physics weights (general physics-informed loss) | [–] |
| $\lambda_{\text{div}}$ | Divergence loss weight | [–] |
| $\lambda_{\text{curl}}$ | Curl (vorticity) loss weight | [–] |
| $\boldsymbol{\mu}_0$, $\boldsymbol{\mu}_0^\epsilon$ | Zeroth kernel moment; body-geometry field | [–] |
| $\nu$ | Kinematic viscosity | [$m^2$/s] |
| $\rho$ | Spatial correlation coefficient | [–] |
| $\tau$ | OOD threshold on the KNN score | [–] |
| $\varphi_\theta$ | Encoder | [–] |
| $\phi$ | Neural ODE weights | [–] |
| $\psi_\theta$ | Decoder | [–] |
| $\omega$ | Vorticity | [1/s] |
| $\Delta t^*$ | Non-dimensional latent integration step | [–] |
| $\Omega$ | Fluid domain | [–] |

## Operators and conventions

| Notation | Meaning |
|---|---|
| $\langle\,\cdot\,\rangle$ | Time average |
| $(\,\cdot\,)'$ | Fluctuating component (deviation from the time average) |
| $\hat{(\,\cdot\,)}$ | Reconstructed / decoded quantity |
| $\tilde{(\,\cdot\,)}$ | Predicted (rolled-out) quantity |

# 1 Introduction

## 1.1. Background and Motivation

Turbulent flows are found in a wide range of natural and engineering systems, from large-scale atmospheric flows and ocean currents to industrial processes and aerodynamic applications. Over the past decades, Computational Fluid Dynamics (CFD) has proven itself to be an effective and powerful means of simulating unsteady fluid flow, both laminar and turbulent. It has proven its use across various industries, where it has played a central role in the analysis, design, and certification of aerospace vehicles, environmental systems, industrial products, and even healthcare developments. Advances in numerical methods and high-performance computing (HPC) have ensured steady advances in reliability and accuracy of CFD, making it a critical element in modern engineering workflows [85].

### 1.1.1. Need for high-fidelity yet efficient simulation

Even with the technological advancements of the last few decades, the computational cost of accurate CFD simulations remains a bottleneck. This is especially the case if one aims to resolve unsteady, turbulent flows in or around complex geometries. These types of simulations require a mesh of such high quality that even the smallest and most complex flow phenomena can be captured accurately [84]. This presents two problems, each of which requires substantial time to evaluate: one is fabricating the mesh through which we want to apply our CFD algorithms, and the other is considering the CFD algorithms and the subsequent turbulence models. In practice, companies navigate a spectrum of

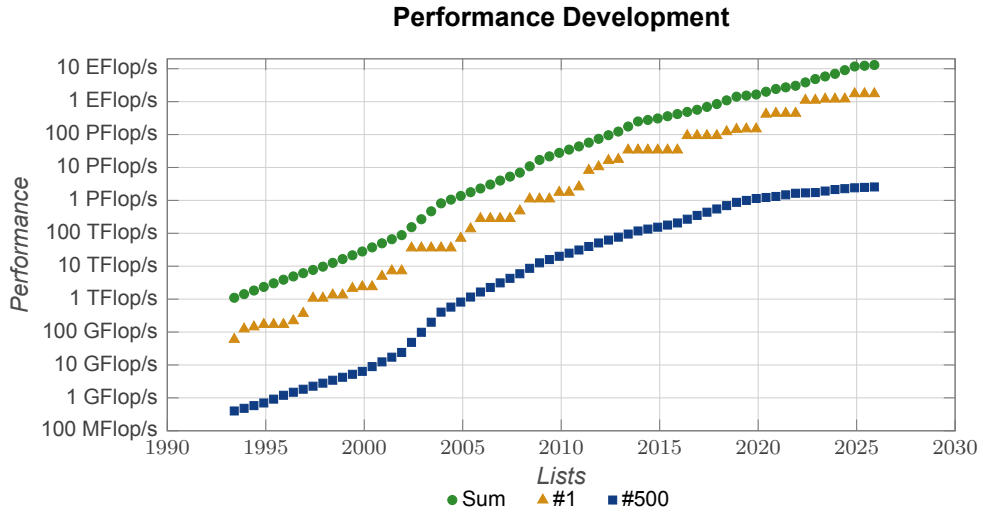



**Figure 1.1:** Performance trends of the TOP500 supercomputers (1993–2026), showing total performance (Sum), top system (#1), and 500th system (#500). Performance is in FLOP/s on a logarithmic scale. Adapted from TOP500 Project [92]

methods that trade computational efficiency for accuracy, ranging from Reynolds Averaged Navier–Stokes (RANS) to Large-Eddy Simulation (LES), hybrid RANS–LES approaches, and ultimately Direct Numerical Simulation (DNS) to gain a competitive edge in the highly competitive aeronautical markets [84]. This indicates the need for frameworks that can decrease the computational costs of high-fidelity CFD simulations.

Compounding this is the slowing of hardware-driven performance gains. Ever since 1965, following Gordon Moore's observation [54], the number of transistors in a single integrated circuit has doubled approximately every two years, while cost, power, and area have remained fixed. However, as transistors reach the atomic scale and fabrication costs continue to rise, the classical technological driver underpinning Moore's Law for 50 years is failing and is anticipated to have flattened by 2025 [78], as shown in Figure 1.1. This 'death march' of Moore's Law [52] opens the door for new hard- and software architectures, as well as creative CFD workflows.

Besides the recent flattening of Moore's Law, researchers have been passionately innovating for several decades and finding novel ways to reduce the computational cost of CFD simulations. One way to accelerate CFD simulations is to employ advances in multicore computer architecture developed over the last few decades [33]. These novel methods enable the use of multicore and manycore CPU (Central Processing Unit) architectures [30], as well as GPUs (Graphics Processing Units) to accelerate CFD solvers [101]. To achieve desirable acceleration results, these hardware methods have been paired with advanced numerical schemes that employ mesh-based, mesh-free, or hybrid methods to achieve significant simulation speed-ups [33].

Together with advances in computing architecture, the rapid progress in Machine Learning (ML) and Artificial Intelligence (AI) offers promising new strategies for accelerating CFD simulations. Fluid mechanics has always been a field where an enormous amount of data must be handled. This can be derived from either experimental results or large-scale numerical simulations [10]. Together with these vast amounts of data and the ability to handle them, ML offers a promising outlook for accelerating modern unsteady CFD simulations.

It is in this landscape that the subject of this thesis finds itself. Bounded by the computational bottleneck created by accurate CFD simulations, and driven by technological advancements in AI technology. This thesis aims to develop a new approach to reducing the computational costs of unsteady flow simulations by leveraging modern AI frameworks. More specifically, this work will focus on using neural ordinary differential equations to accelerate the time integration step. Since this is, more often than not, the most costly part in mesh-based solving algorithms. In light of this context, this thesis aims to answer the following central research question and subquestions:

Research Question

**How effectively can neural ordinary differential equations accelerate the time integration step in CFD solvers for unsteady simulations without compromising accuracy?**

1. How well does the neural ODE acceleration framework preserve the flow's statistical observables when transferred across Reynolds numbers and grid resolution?
2. How sensitive is the solution accuracy of neural ODEs to network architecture and training data quality?
3. How can the quality of a neural ODE rollout be monitored at runtime to determine when its predictions can be trusted?

# 2

# Literature Review

## 2.1. Scope and Goal for literature review

This chapter provides an overview of the current knowledge and available literature on ML-accelerated CFD simulations. As a start, Section 2.2 explains the most frequently implemented methods in which turbulent simulations can be resolved. Next Section 2.3 will introduce how current advances in machine learning and artificial intelligence are being incorporated in the current CFD landscape.

## 2.2. Turbulent flows: simulation and modelling

Fluid dynamics is governed by a set of coupled partial differential equations (PDE's) called the Navier-Stokes Equations (NSE). These equations are based on the assumptions of conservation of mass and momentum. These equations are shown in 2.1 in *Einstein notation* [59]. Note that the equations seen in 2.1 display the governing equations of a fluid of which we have assumed a constant density, which means that density $\rho$ is a material property and not something that needs to be solved for.

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{2.1a}$$

$$\underbrace{\frac{\partial u_i}{\partial t}}_{\text{Local acceleration}} + \underbrace{u_j \frac{\partial u_i}{\partial x_j}}_{\text{Convective term}} = \underbrace{-\frac{1}{\rho}\frac{\partial p}{\partial x_i}}_{\text{Pressure gradient}} + \underbrace{\nu \frac{\partial^2 u_i}{\partial x_j \partial x_j}}_{\text{Viscous diffusion}} + \underbrace{g_i}_{\text{Body forces}} \tag{2.1b}$$

Where $u$ is the 3-dimensional velocity vector, $x$ is the position vector, $p$ is the pressure in the fluid, $\nu$ is the kinematic viscosity of the fluid, $g$ is the gravitational constant, $t$ is time and finally the subscripts $i$ and $j$ indicate spatial directions.

These equations have proven to be a tough fluid dynamics challenge and have yet to be solved analytically. Due to their non-linearity and strong coupling, they display complex behaviour. This behaviour mainly arises from the non-linear convective term in the momentum equation. This fact is particularly pronounced for high Reynolds number flows, where small-scale turbulent structures dominate [59]. In current mathematics, solutions to the most fundamental questions regarding the NSE remain elusive. Questions such as: do solutions exist, and are they unique [17]? It is due to this complexity and lack of understanding that the Clay Mathematics Institute has listed the NSE as one of the 7 Millennium Problems. Because no analytical solution exists, engineers and researchers must discretise, model, and make assumptions about various parts of the NSE to obtain numerical solutions to these equations.

### 2.2.1. DNS

One way to evaluate the NSE is through a Direct Numerical Simulation (DNS). A DNS aims to capture all length and timescales of a turbulent flow directly, without modelling and without further assumptions.

Furthermore, DNS must accurately resolve the temporal resolution of the turbulent flow it aims to simulate. Following Kolmogorov's microscale hypothesis [37] and the two-point correlation function [65], the smallest and largest turbulent length scales can be obtained, respectively. Using these length scales, the mesh spacing and mesh dimensions can be determined; this way, all relevant length scales can be represented in the turbulent flow field without resorting to modelling. These length scales are strongly correlated to the Reynolds numbers, which is the ratio of inertial forces and viscous forces.

$$Re = \frac{\text{Inertial forces}}{\text{Viscous forces}} = \frac{ul}{\nu} \tag{2.2}$$

Where $u$ is the flow velocity and $l$ is the characteristic length. At large Reynolds numbers, turbulent flows exhibit a wide range of scales, where the smallest turbulent length scales become progressively smaller as $Re$ increases. This means that for a high $Re$ DNS, a very fine mesh would need to be generated.

In fact, when following Kolmogorov in assuming the smallest scales of turbulence are universal and isotropic, the total number of grid points (degrees of freedom) required for a DSN scales with $Re^{9/4}$ [18]. Factoring in the necessary change in timestep to maintain a constant CFL number, the total computational effort for DNS would be equal to $\mathcal{O}(Re^3)$ [18]. This extensive computational requirement makes DNS highly accurate but prohibitively expensive for general practical use. However, it remains indispensable for fundamental turbulence research due to its ability to provide complete knowledge of the flow without empirical approximations.

### 2.2.2. RANS

One way to reduce the computational cost of numerical simulations of the NSE was stipulated by Reynolds [68] in 1895. In this work, he proposed decomposing the instantaneous velocity into two distinct parts: the time-averaged part and the fluctuating part.

$$u_i = \langle u_i \rangle + u_i' \tag{2.3}$$

In Equation 2.3, the instantaneous velocity $u_i$ is decomposed into the time-averaged part $\langle u_i \rangle$ and the fluctuating component $u_i'$. This decomposing into time-averaged and fluctuating parts is called *'Reynolds decomposition'*. Substituting Equation 2.3 into Equation 2.1 results in the incompressible Reynolds Averaged Navier Stokes (RANS) equations [55]. Because the time averaging in the Reynolds decomposition is much larger than the largest timescale of the turbulent fluctuations, the evolution of the mean flow component can be determined [69]. However, this also means that none of the turbulent fluctuations is resolved; instead, turbulent scales are modelled using a closure model, which must provide a suitable representation of their effects on the mean flow.

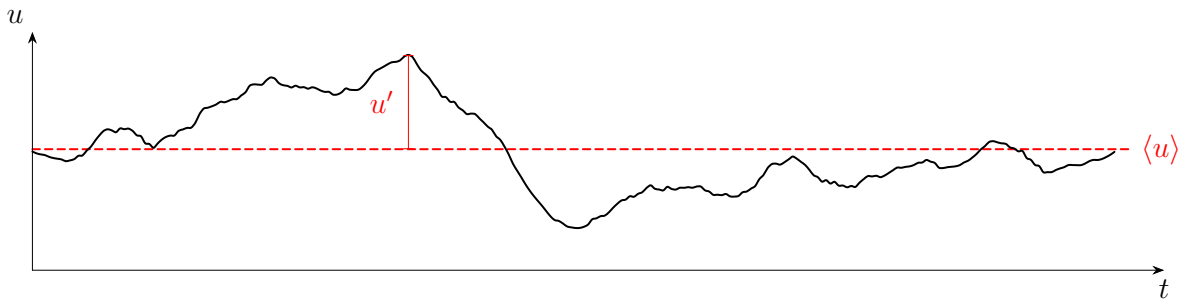



**Figure 2.1:** Schematic of the Reynolds decomposition for incompressible flow: the instantaneous velocity $u(t)$ (solid line) is split into its time-averaged mean $\langle u \rangle$ (dashed red line) and the time-dependent fluctuation $u'(t) = u(t) - \langle u \rangle$

After applying the substitution, Equation 2.1 can then be solved to obtain the time-averaged solution of the NSE seen in 2.4. These incompressible RANS equations are similar to the conventional NSE seen in Equation 2.1, the only difference being the addition of the divergence of the averaged products of

the Reynolds stress tensor: $\frac{\partial \langle u_i' u_j' \rangle}{\partial x_j}$. To ensure accurate results, the effect of the Reynolds Stress Tensor (RST) needs to be modelled by turbulence models. Because the time-dependent fluctuations are not resolved, RANS simulations can be executed on grids with larger mesh spacing, which inherently reduces the number of required operations and thus the computational cost.

$$\frac{\partial \langle u_i \rangle}{\partial x_i} = 0 \tag{2.4a}$$

$$\frac{\partial \langle u_i \rangle}{\partial t} + \langle u_j \rangle \frac{\partial \langle u_i \rangle}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \langle p \rangle}{\partial x_i} + \nu \frac{\partial^2 \langle u_i \rangle}{\partial x_j \partial x_j} + g_i - \frac{\partial \langle u_i' u_j' \rangle}{\partial x_j} \tag{2.4b}$$

To achieve accurate simulations, the effect of the RST needs to be modelled. The most used models are those of Launder and Spalding [39], Wilcox [100], Spalart and Allmaras [87], Launder, Reece, and Rodi [40] and Speziale, Sarkar, and Gatski [88].

### 2.2.3. LES

Another widely implemented turbulent flow framework was initially proposed by Smagorinski [82] to stabilise under-resolved weather simulations. This proposition was later applied to engineering-related flows by [20] and subsequently by Schumann [76]. In this work, the proposition was made only to resolve the large-scale turbulent structures of the flow, while modelling the effect of the small-scale structures. Because only the large-scale eddies are being resolved in this simulation, this type of simulation is called a Large Eddy Simulation (LES). The governing equations of LES are determined by applying a low-pass filter operation to Equation 2.1, which results in the governing equations seen in Equation 2.5. After the filtering operation has been applied, a sort of 'smooth' version of the velocity fluctuations is achieved, which is displayed in Figure 2.2.

$$\frac{\partial \overline{u}_i}{\partial x_i} = 0 \tag{2.5a}$$

$$\frac{\partial \overline{u}_i}{\partial t} + \overline{u}_j \frac{\partial \overline{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_i} + \nu \frac{\partial^2 \overline{u}_i}{\partial x_j \partial x_j} + g_i - \frac{\partial \tau_{ij}}{\partial x_j} \tag{2.5b}$$

Where $\overline{u}_i$ is the filtered velocity. The filter operation is applied through a filter kernel $G$, where the filter cut-off width is determined to be proportional to the grid size. As seen in Equation 2.5b, a new term arises $\frac{\partial \tau_{ij}}{\partial x_j}$, which contains the effects of the turbulent scales that the filter operation has cut off. Because most energy is dissipated from a flow in the most minor scales [59], this 'Sub Grid Scale' (SGS) tensor needs to be modelled in such a way that it artificially drains enough energy from resolved scales. Many different kinds of SGS models have been proposed, most of which utilise an 'Eddy viscosity' to equate the amount of energy drained to the strain the fluid experiences. The most used eddy viscosity models are ones like the Smagorinsky model (Smagorinski [82]), the Scale Similarity Model (Bardina, Ferziger, and Reynolds [5]), the Dynamic Smagorinsky Model (Germano et al. [24], Lilly [43]), the Wall-Adapting Local Eddy-viscosity model (Nicoud and Ducros [58]) and the Vreman model (Vreman [96]).

Because large-scale structures are directly resolved, LES is more accurate than the RANS approach for the same flow case, as these structures contain most of the turbulent kinetic energy and are responsible for most of the momentum transfer and mixing. Because LES accuracy depends on the filter width, which in turn is related to the grid spacing, grid refinement will yield more accurate LES results. At the same time, refinement will also significantly increase the computational cost of the simulation. That is why, for high-accuracy flow cases, acceleration frameworks are essential to keep computation costs within usable limits.

Since the concept of LES was proposed in 1963, it has evolved significantly and gained wider acceptance, particularly in academic and research settings. Despite the increase in computational power over the past few decades [78], the cost of performing LES remains higher than that of comparable

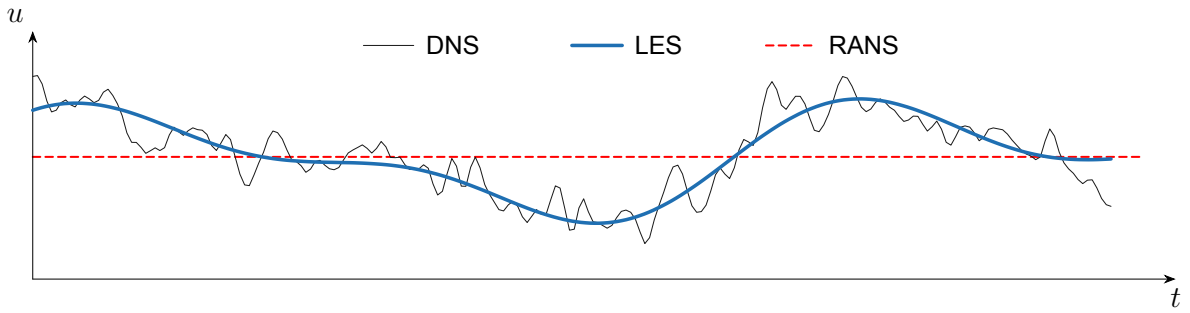



**Figure 2.2:** Schematic of the velocity signal $u(t)$ at a fixed point as captured by the three turbulence-modelling paradigms. DNS resolves the full range of scales; LES resolves the large, energy-containing scales and models the sub-filter motions; RANS retains only the time-averaged mean.

RANS simulations. This higher cost is due to the need for LES to explicitly resolve the temporal evolution of larger turbulent eddies, which requires finer grid resolutions and more time steps, in contrast to RANS, where the equations are time-averaged and all turbulent fluctuations are modelled.

However, recent academic advancements in LES frameworks, as highlighted by Weymouth and Font [98], suggest that LES can be performed at relatively low computational cost, even with limited methodological knowledge. This perspective aligns with the assertion made by Piomelli [64], who stated that "LES will be used in more complex configurations and will yield reliable results for engineering analysis."

Despite the high computational cost of LES, advancements in modelling and algorithm design will continue to reduce the cost of high-fidelity simulations. That is why, contrary to Zhiyin [102], we expect LES to be more widely adopted in general engineering applications. Even more so with the acceleration frameworks discussed Section 2.3.

## 2.3. Machine/Deep learning Enhanced CFD

The integration of machine learning (ML) into CFD offers promising opportunities to accelerate turbulent flow simulations. While traditional performance-enhancing methods can deliver significant gains, they are constrained by their reliance on numerical algorithms. This is where data-driven methods can complement traditional methods by leveraging the massive amounts of high-fidelity simulation data available.

It is worth noting that recent years have seen an exponential growth in the number of studies exploring ML-enhanced CFD. Countless methods incorporating ML into the CFD landscape have been developed, ranging from broadly adopted to more niche applications. Comprehensive reviews, such as those by Vinuesa and Brunton [95] and Wang et al. [97], demonstrate the rapid advancements in the field. Accordingly, this literature review covers relevant ML frameworks that align with the central research question posed in chapter 1, prioritising methods aimed at accelerating scale-resolved CFD simulations rather than providing a comprehensive survey of the field. Adjacent applications are therefore only addressed where they directly inform the proposed approach, as it is beyond the scope of this work to cover all relevant contributions in detail.

Before highlighting the current landscape of ML-enhanced CFD simulations, Section 2.3.1 and Section 2.3.2 will briefly handle ML and NN terminology before Section 2.3.3 will introduce key neural network architectures which are frequently used in the CFD environment. Where Section 2.3.6, Section 2.3.5, and 2.3.4 will showcase how ML can be leveraged to accelerate CFD simulations by enhancing solver components, employing data-driven ROMS and creating physics-informed surrogate models.

### 2.3.1. Introduction to Machine Learning

Machine Learning is a branch of artificial intelligence that aims to develop algorithms capable of learning patterns and making predictions from data without being explicitly programmed. ML models can essentially be expressed as a function $\boldsymbol{y}(\boldsymbol{x})$ which takes an input vector $\boldsymbol{x}$ and transforms it to an out-

put vector $\boldsymbol{y}$. The precise form of $\boldsymbol{y}(\boldsymbol{x})$ is determined during the *training phase* based on the training data [7]. The training of the model requires a dataset that is usually split into training, validation and test sets. The training set is used to fit the model parameters, while the validation set can be used to determine hyperparameters, such as the learning rate and network architecture. The test set evaluates the model's generalisation to unseen data [11].

Depending on the target variable, ML objectives are typically categorised into two categories: regression (continuous outputs such as velocity or pressure predictions) and classifications (discrete categories, e.g., image recognition and Optical Character Recognition).

A central challenge in ML is ensuring that models generalise beyond the training data, meaning that they can reliably generate correct outputs for unseen data that was not in their training set. Overfitting occurs when a model has been trained to achieve an extremely high accuracy on the training data, resulting in poor predictions on unseen test data. While underfitting reflects a model that is too simple to capture the underlying dynamics of the training data, resulting in poor performance even on the training data [11]. To address these pitfalls, regularisation strategies, careful dataset design, and physically informed constraints can be employed to reduce model complexity and penalise unrealistic predictions [2, 74].

ML algorithms are broadly grouped into three categories: supervised learning, where models learn from labeled data (most common in CFD, like mapping flow fields to forces); unsupervised learning, which identifies patterns or reduced representations from unlabelled data (e.g., clustering coherent flow structures or dimensionality reduction); and reinforcement learning, where an agent interacts with an environment to optimize a reward.

A detailed treatment of machine learning theory is beyond the scope of this work. Still, it can be found in standard references such as Bishop [7], Burkov [11], and Daniel [19], which provide the broader theoretical background underlying the CFD-oriented applications discussed further down in this work.

### 2.3.2. Introduction to Neural Networks

Neural networks (NNs) are a class of ML models inspired by the biological nervous system, designed to approximate complex nonlinear functions. They consist of interconnected computational units, or neurons, which receive inputs, process them through an activation function and produce an output. A general architecture of a NN can be seen in Figure 2.3. The use of non-linear activation functions enables NNs to act as universal function approximators [32], capable of representing highly complex mappings between inputs and outputs.

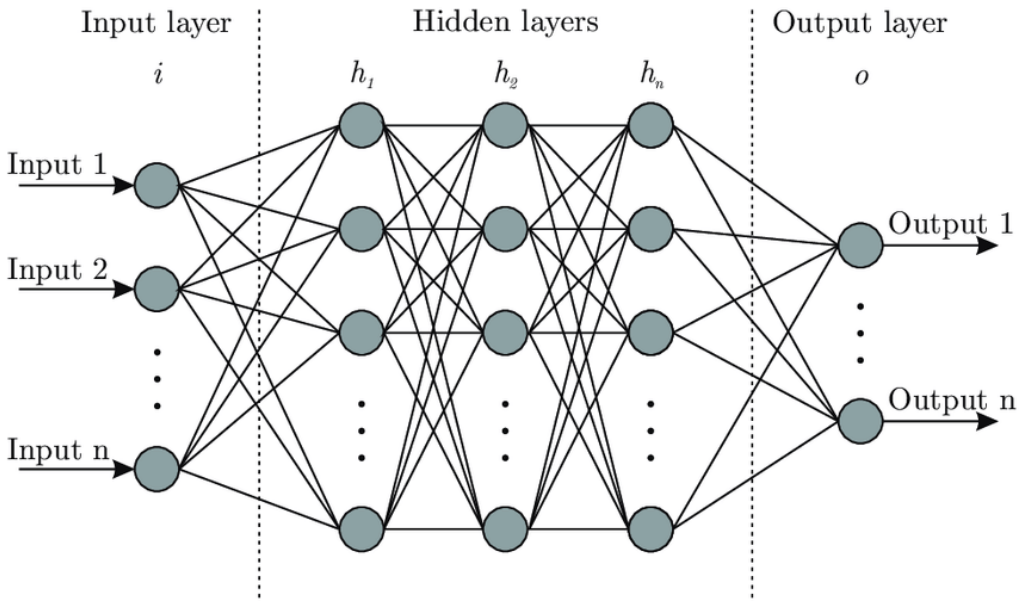



**Figure 2.3:** General architecture of a Feed Forward Neural network, adapted from [57]

Training a neural network involves minimising a loss function that quantifies the difference between

predictions and actual values across the dataset. This optimisation is typically performed via gradient descent and its variants, with gradients computed efficiently using backpropagation [2]. To achieve a desirable accuracy, hyperparameters such as learning rate, network depth, and activation function can be optimised by monitoring the network's performance. After training is completed, a set of weights and biases is created that defines how the network transforms inputs into outputs.

The architecture of an NN can significantly affect its performance on different tasks. The simplest form of NNs are feed-forward neural networks (FNNs), as seen in Figure 2.3, where information flows from input to output in a single pass.

### 2.3.3. Common Neural Network Architectures Employed in CFD

**Convolutional Neural Networks**

Convolutional Neural Networks (CNNs) are a frequently used class of neural networks specifically designed to process grid-structured input data that exhibit strong spatial dependencies in local regions of the grid [2]. That is why CNNs are often used for image recognition, since adjacent spatial locations in an image often have similar pixel colour values. CNNs are characterised by convolution operations, which involve taking a dot product between a grid-structured set of weights (known as a filter or kernel) and the input data to extract features.

Because of the strong capabilities of CNNs to extract features from image snapshots, they present themselves as good candidates to be implemented for CFD predictions. Since CFD flow fields can be represented as a structured grid, CNNs offer an outstanding possibility to learn patterns such as vortices or boundary layers effectively. This can be helpful for tasks such as flow prediction, surrogate modelling, or acceleration [42, 29].

**Neural Ordinary Differential Equations**

Neural Ordinary Differential Equations (NODEs) are a type of deep neural network model introduced by Chen et al. [14] that differ substantially from conventional neural networks. They can be seen as the continuous-time counterpart of recurrent architectures, where an RNN advances a hidden state through a discrete sequence of steps [2], a NODE replaces that discrete update with a continuous one. More specifically, this continuous formulation is the limit of a Residual Neural Network (ResNet [31]), in which the latent state is represented by a stack of layers, as shown in Figure 2.4. NODEs thus learn the mapping from sequential data by using a continuous representation of the latent space.

Where the stacking of the latent state in ResNets can be seen as an Euler discretisation of a continuous transformation, shown in Equation 2.6:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{2.6}$$

Where $h_{t+1}$ is the newly determined latent state, which is constructed of a summation of the current latent state $h_t$ through the addition of a function of the network parameters $\theta_t$ and the value of the current prediction $f(h_t, \theta_t)$, when the added layers become large and the steps taken become smaller, this progression of the latent state can be represented by a differential Equation 2.7.

$$dh(t)/dt = f(h(t), t, \theta) \tag{2.7}$$

A conventional ODE solver can then be used to solve this equation. This formulation provides adaptive computation capabilities, as ODE solvers can dynamically adjust step sizes based on the solution's complexity, enabling an optimal trade-off between speed and accuracy. This essentially allows NODEs to define a smooth flow through the latent space, rather than the discrete series of transformations that ResNets produce. Figure 2.5 shows a graphical representation of the latent space vector field.

NODEs also provide a memory-efficient way to train deep models using the adjoint sensitivity method, which propagates through the ODE by solving a second augmented ODE backwards in time. Because the neural network can be viewed as a universal approximator, and the ODE solver can evaluate the latent layers at any time point and timestep, the resulting model provides smooth predictions without

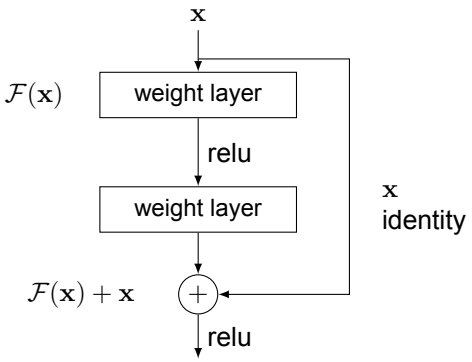



**Figure 2.4:** ResNet layer stacking, adapted from [31]

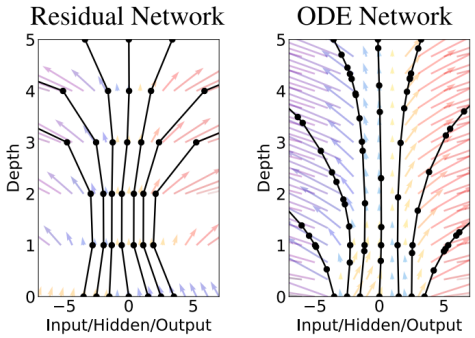



**Figure 2.5:** Left: Discrete evolution of ResNet predictions, Right: continuous evolution of predictions by NODE. Adapted from [14]

being constrained to discrete time steps or layers. This behaviour makes NODEs particularly well-suited for predicting systems where continuous-time behaviour is essential, even when the supplied training data is irregularly sampled in time.

In the context of CFD time-series prediction, NODEs can offer key advantages. Their adaptive behaviour can enable variable-time-step integration, which can be optimal for accelerating potential CFD simulations. However, since NODEs involve solving ODEs and fluid systems are often high-dimensional, solving these ODEs can be computationally intensive. That is why, in recent literature, such as the work of Shankar et al. [79], NODEs have been implemented within data-driven reduced-order models, where the dynamics of the latent space are predicted using NODEs. This aligns with the objective of this thesis, where instead of predicting the time integration of the NSE, the authors have implemented NODEs in the latent space of a ROM, which is used to predict the entirety of the latent dynamics, which can then be used as a surrogate model for a specific flow case.

Another recent advancement in Neural ODE research has been the implementation of a physics-informed loss term. This embeds the governing equations of the system that the model aims to predict directly into the training process. Sholokhov et al. [80] introduced a collocation-based physics-informed loss function that enforces PDE constraints at sampled points, improving extrapolation, stability, and performance in situations where there is not a lot of training data available. Additionally, Ghanem et al. [25] proposed an optimisation framework that incorporates physics-informed loss terms, enabling the accurate recovery of hidden states when only partial measurements are available as training data. These works demonstrate that embedding physical laws into the NODE loss function allows for a more robust model in scenarios with sparse, noisy, or incomplete data. This approach could be particularly valuable in CFD, where enforcing conservation laws and physical consistency can lead to better flow-field predictions with fewer training data.

### 2.3.4. Machine Learning Enhanced Solver Components

One way to leverage ML to accelerate scale-resolved CFD simulations is to embed machine learning directly into specific components of CFD solvers. Rather than learning the whole flow evolution, these approaches aim to improve the efficiency and accuracy of key solver stages. Calculations such as the Pressure-Poisson equation, RANS turbulence closure models, sub-grid scale models, or even learning-based discretisation are all possibilities that have been presented in recent literature.

#### Pressure-Poisson prediction

An essential component of incompressible CFD solvers is the pressure-velocity coupling algorithm, which ensures that the computed velocity field remains divergence-free. This coupling is typically enforced in operator splitting methods, where the velocity field is first advected, and the resulting velocity field $\mathbf{u}^*$ does not satisfy the continuity equation 2.1a. This is then corrected using the Pressure-Poisson Equation (PPE) 2.8 [95].

$$\frac{\Delta t}{\rho}\nabla^2 p = -\nabla \cdot \mathbf{u}^* \tag{2.8}$$

Where $\Delta t$ is the simulation time step, $\rho$ the fluid density and $p$ the pressure field, finding the solution to this equation is typically the most computationally expensive for the numerical solver. This is why exploring alternative strategies for this computation could yield promising acceleration.

Chen, Jin, and Li [15] employed a multi-scale Graph Neural Network (GNN) to either entirely or partially replace the traditional iterative PPE solver, achieving solution speeds nearly three times faster at $Re = 1000$ and almost twice as fast at $Re = 5000$. Ajuria et al. [3] developed a hybrid strategy using CNNs to predict the pressure field alongside a traditional Jacobi solver, reaching speed-ups of up to 320% while maintaining the predetermined accuracy.

More recently, Sousa, Afonso, and Rodrigues [83] proposed a versatile method applicable to various geometries, combining Multilayer Perceptron (MLP) networks with Principal Component Analysis (PCA) to achieve errors of around 3% at low computational cost, though the surrogate models struggled to extrapolate to significantly different flow regimes. Chillón et al. [16] accelerated an existing algebraic multigrid (AMG) solver by using a GNN to predict the coefficients of a sparse pseudo-inverse Jacobi smoother. Because the non-linearity is confined to the smoother construction, the method preserves the solver's linearity and convergence guarantees while delivering wall-clock speed-ups of 4–37% across structured and unstructured meshes, generalising to grids far larger than those seen in training.

**Turbulence and sub-grid scale modelling**
In the context of RANS simulations, modelling the Reynolds stress tensor (RST) is essential for providing closure. Traditional closure models, such as eddy viscosity models, rely on empirical formulations which may not generalise well across different flow conditions. To address this, ML frameworks offer a data-driven route to learning more accurate representations of the RST from high-fidelity data. The most notable example is Ling, Kurzawski, and Templeton [44], who embedded Galilean invariance into a custom Deep NN architecture to force physically consistent predictions, enabling it to outperform traditional eddy viscosity models [95]. Building on this, Cai et al. [12] proposed an improved Tensor Basis Neural Network (TBNN) that, combined with transfer learning, achieved substantial accuracy gains in predicting the Reynolds stress anisotropy tensor even with limited training data.

A closely related closure problem arises in LES, where the effect of the unresolved sub-grid scales on the resolved scales must be modelled. Conventional SGS models rely on case-specific constants, such as scale similarity [5], or eddy viscosity [43, 24], which may not be applicable across all flow regimes. ML has been used to develop more adaptive SGS models in two broad ways: supplementing the unresolved energy on the coarse mesh through supervised learning, and using reinforcement learning to stabilise the coarse simulation [95]. For the supervised approach, Beck, Flad, and Munz [6] employed CNNs to learn the mapping from coarse-grid quantities to the closure terms without a priori assumptions, while Sirignano and MacArt [81] optimised their closure model a posteriori by solving the LES equations and their adjoints directly, outperforming dynamic Smagorinsky and AMD [71] models even on coarse meshes and generalising to out-of-sample bluff body shapes. As an example of the second approach, Kurz, Offenhäuser, and Beck [38] formulated SGS modelling as a reinforcement learning task, with a CNN-based agent adapting the eddy viscosity in space and time based solely on the local flow state.

Despite these advances, ML-based turbulence and SGS modelling is still not ready for widespread engineering use. The models often perform well on the flows they are trained on but struggle to generalise to new or complex cases. The NASA Symposium on Turbulence Modelling [73], together with a self-released paper by Spalart [86], highlighted that fundamental issues in the RANS framework remain, and that machine learning alone will not resolve them. Combined with the limited (though increasing) availability of high-fidelity data and a lack of robustness in industrial settings, this makes it difficult for engineers to fully trust these models, and the field will require greater transparency and standardisation before reliable real-world deployment is possible.

### 2.3.5. Data-Driven Reduced Order Models

Data-driven reduced-order models have been utilised to accelerate CFD simulations by approximating high-fidelity systems in a lower-dimensional form that retains most of the relevant flow phenomena.

Data-driven ROMs extract dominant patterns from simulation or experimental snapshots without requiring explicit knowledge of the appropriate governing equations. Unlike physics-based ROMs, which typically use linear projections (such as POD) to solve the NSE on a basis with a reduced number of DOFs, data-driven ROMs rely on snapshots of the full-dimensional state vector to produce projection operations through a training process. This utilises the previously explained capabilities of neural networks to approximate a nonlinear function. These methods can achieve significant speedups after being successfully trained on supplied CFD snapshot data, or they can be used in a transfer learning fashion to accelerate CFD simulations, as will be explained in further detail.

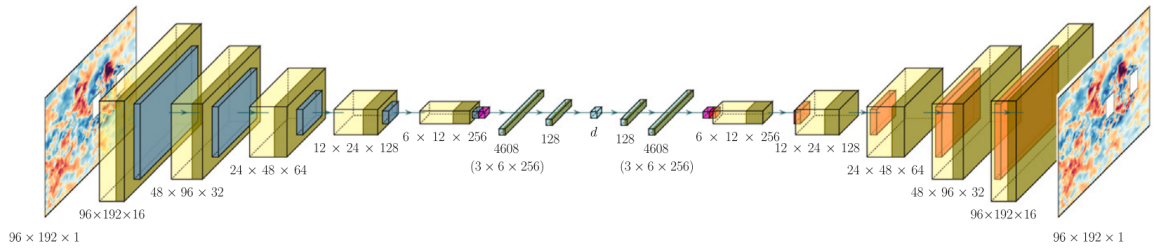



**Figure 2.6:** Schematic view of a CNN autoencoder extracting non-linear modes from a CFD snapshot, source [23]

Many data-driven ROMs are built on autoencoder architectures, which share the encoder–decoder structure. The encoder compresses each high-dimensional snapshot into a low-dimensional latent vector, and the decoder reconstructs the field from this latent representation while minimising the reconstruction error. The non-linear activation functions within the network allow the autoencoder to capture the complex non-linear behaviour of turbulent structures; without them, it would collapse to a linear map spanning the same subspace as POD. This offers a key advantage over methods like POD, which are limited to linear representations [62]. For grid-structured velocity fields, autoencoders typically employ convolutional layers, which exploit spatial locality and weight sharing to efficiently extract local flow features. Figure 2.6 shows an overview of how an autoencoder reduces and reconstructs a single CFD snapshot to and from a latent space.

What distinguishes data-driven ROMs from plain autoencoders is the requirement that the latent space also capture and predict the system's temporal dynamics. This means that, in addition to learning a compact representation of the input data, the latent variables must be predicted to accurately reproduce the simulation's future state [95]. This can be achieved either by selecting latent-space dimensions that enable accurate prediction of the future flow field, or by utilising regression architectures better suited to time-series prediction, such as RNNs, LSTMs, or neural ODEs.

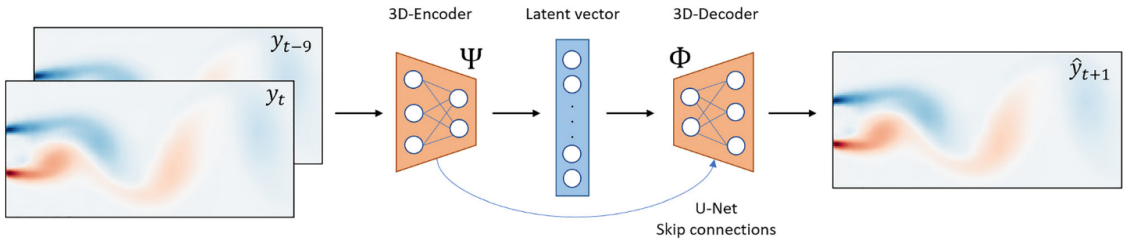



**Figure 2.7:** Example architecture of deep convolutional ROM, source [62]

Some notable examples of deep learning-based ROMs include the work of Pant et al. [62], where the authors leveraged 3D Autoencoder and 3D U-Net-based architectures to create non-linear projections from reduced-order states to fluid flow snapshots, as depicted in Figure 2.7. The authors employ a looped prediction strategy, in which the last 10 snapshots of a simulation are used to train the network to predict the next timestep, $\hat{y}_{t+1}$. This prediction is then appended to the last nine snapshots and used as input to predict $\hat{y}_{t+2}$. This process is repeated recursively for 20 timesteps. This looping strategy reduced the simulation speed by 2 orders of magnitude [62]. Similarly, CFDnet [60] accelerates RANS simulations by leveraging its CNN ROM to predict fluid properties, while still utilising the CFD solver to

ensure that domain-specific convergence constraints are met. This allowed the authors to deliver highly accurate predictions while still achieving acceleration rates from 1.9 up to 7.7 times the benchmark simulations.

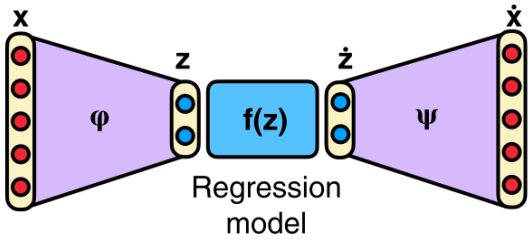



**Figure 2.8:** Example architecture of autoencoder with regression in the latent space, adapted from [95]

As previously mentioned, the dynamics in the latent space can also be modelled by utilising a regressor in the latent space of the autoencoder, which would make the architecture of the data-driven ROM look like Figure 2.8. A notable example of such an architecture is the work by Shankar et al. [79], which demonstrates how NODEs are utilised to model system dynamics in the latent space. By explicitly decoupling the reduction and the latent system dynamics, the presented framework achieved highly accurate reconstructions of flow dynamics over long integral timescales. Instead of using NODEs for the dynamical modelling in the latent space, LSTMs can also be used, as they present good capabilities for time series predictions, as indicated in the work by Mohan et al. [53]. In this work, however, a NODE is favoured, as it represents the latent dynamics as a continuous-time vector field rather than the discrete recurrence employed by an LSTM. As it lends itself better to irregularly sampled time series [14, 72]. Additionally, the continuous formulation permits adaptive-step integration and lends itself to training strategies that a recurrent LSTM does not naturally support, both of which are revisited in this thesis's methodology.

### 2.3.6. Physics Informed Surrogate Models

Physics-informed surrogate models demonstrate strong potential in accelerating computationally intensive simulations. A surrogate model is a broad term for a computationally efficient approximation of a complex system. A ROM, which was handled previously, is a specific type of surrogate that reduces the system's dimensionality by applying a projection to the high-dimensional dynamics in a lower-dimensional space. Meaning that all ROMS are essentially surrogate models, not all surrogates are ROMS.

Purely data-driven surrogate models, such as DeepONet [47], require large amounts of data to achieve reliable predictive performance. In real-world scenarios, where data is either sparse or prohibitively expensive to collect, models that can generalise effectively on smaller datasets are necessary [21]. That is why physics-informed surrogate models embed fundamental physical laws and constraints directly into their architecture or training process. This integration aims to create efficient approximations that are not only accurate but also physically consistent, even when dealing with limited training data [67].

The integration of physics into these models is achieved mainly through two key mechanisms: physics-informed loss functions and specialised network architecture. Physics-informed loss functions incorporate the residuals of the governing partial differential equations, such as conservation of mass 2.1a and momentum 2.1b, into the loss function during training. This way, the model can be regularised during training, penalising larger prediction errors that violate the relevant physical laws. Essentially, an artificial bias is added to the network, which promotes data efficiency since the projections are inherently directed to adhere to physical laws. Equation 2.9 shows a general form of a physics-informed loss function, where $\mathcal{L}_{\text{data}}$ is a standard loss function like MSE or MAE, and $\mathcal{L}_{\text{physics}}$ can be derived from the physical system the network has to learn. In the case of a CFD simulation, this latter part can be built from the continuity and momentum equations. These two loss functions can then be combined using a weighted sum, with $\lambda_d$ and $\lambda_p$ serving as weights.

$$\mathcal{L} = \lambda_d \mathcal{L}_{\text{data}} + \lambda_p \mathcal{L}_{\text{physics}} \tag{2.9}$$

A prominent example is Physics-Informed Neural Networks (PINNs), first proposed by Raissi, Perdikaris, and Karniadakis [67], which use deep learning to solve PDEs, exploiting automatic differentiation during backpropagation to compute partial derivatives [95]. PINNs have been utilised in various CFD scenarios, including hydrodynamic surrogate models [21], high-speed flow approximations [50], and more general NS cases [22, 89], where they have demonstrated excellent results in predicting flow parameters with limited training data. However, in the context of accelerating fluid simulations, PINNs have a disadvantage: their training speed limits them from achieving significant speed-ups in a CFD workflow [35, 28]. On the other hand, PINNs have been successfully used as closure models in RANS and LES frameworks; their workings will be explained later in this literature review.

Alternatively to PINNs, the Finite Volume Method Network FVMN [34] is a physics-informed surrogate model which specifically exemplifies a physics-informed surrogate model tailored for CFD, integrating principles of the finite volume method (FVM) directly into its design. Unlike previously explained methods that only include information from a single grid point, FVMN employs a tiered input system that incorporates physical quantities from a central grid point and its neighbouring grid points into the neural network's input matrix. This allows the network to imitate the mass and heat transport among neighbouring cells, which is considered the most crucial process in the FVMN. Additionally, instead of directly predicting the future state, FVMN employs a derivative-based output system, which predicts the change (derivative) of physical quantities over a given time step.

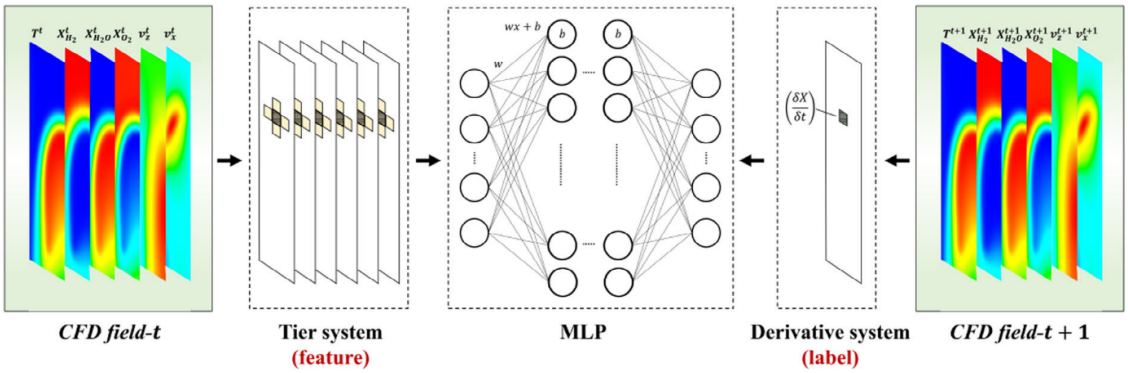



**Figure 2.9:** Overview of the FVMN architecture, showing the tier-derivative input/output system built around a conventional FFN (or MLP), adapted from [34]

In the original work Jeon, Lee, and Kim [34], significant improvements in accuracy were showcased compared to general models without the tier/derivative system implemented, where the authors managed to achieve a relative error with respect to CFD data of not more than 0.05%, compared to 1.12% for the conventional model.

### 2.3.7. Hybrid CFD-ML acceleration schemes

The previous surrogate and reduced-order approaches aim to replace the CFD solver outright, which makes them vulnerable to error accumulation over long predictions. This is where hybrid CFD-ML acceleration schemes instead keep the solver in the loop and couple it to a learned model, in which the two operate in tandem, with the solver safeguarding physical consistency. These schemes broadly fall into two categories: those in which the solver and a surrogate alternate, with a monitored error signal handing control back to the solver when a tolerance is exceeded, and those in which the learned model corrects the solver at every timestep. An example of the former category is discussed first.

The FVMN has shown great results in tandem with CFD, but the system's linearly increasing gradient error becomes apparent when used for longer-term transient predictions. The authors proposed an ML-CFD cross-coupling framework. The Residual-based Physics-Informed Transfer learning strategy (RePIT) [35] is a hybrid framework in which both a CFD solver and the FVMN alternately predict time-series fluid flow. RePIT continuously monitors the residuals of the governing equations, NSE 2.1, which are part of the physics-informed loss function employed by the FVMN in the case of fluid flows. Figure 2.10 shows a schematic overview of the transfer learning framework. It shows that when the

residuals of the loss function ($\varepsilon$) exceed a predefined tolerance ($\varepsilon_{th}$), the CFD solver will use the latest prediction and continue solving the governing equations to reduce the residuals back to a desired value. Afterwards, these newly computed CFD snapshots are used to update the network parameters, enabling new predictions.

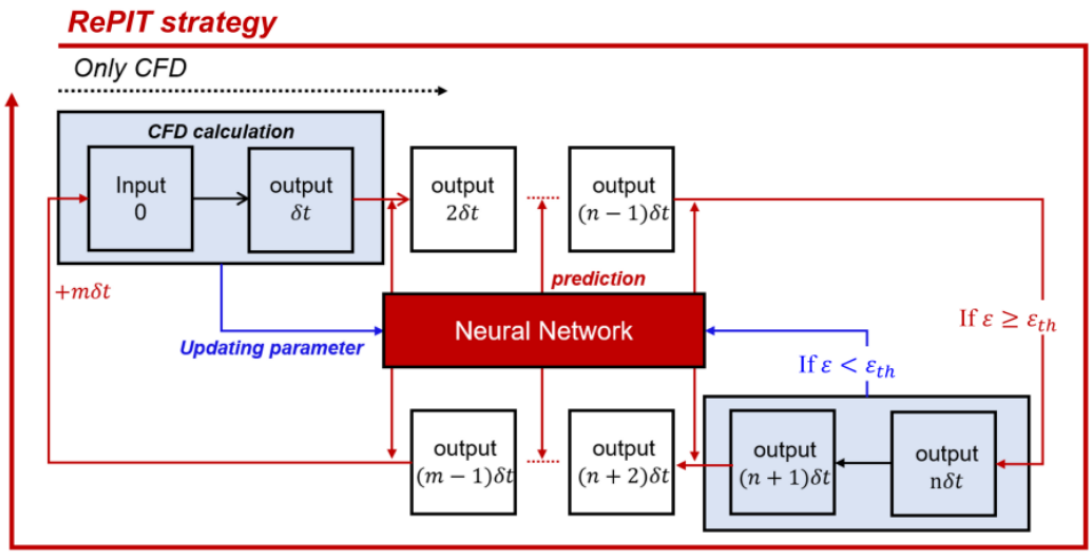



**Figure 2.10:** Schematic overview of the RePIT strategy showing the transfer learning process, adapted from [35]

This framework leads to significant computational speed-ups without compromising accuracy. The authors implemented this method in a natural convection CFD case, achieving a 1.9 times acceleration, including network parameter updates. This demonstrates the FVMN's strong capabilities, as its tier-based input system requires minimal training data. RePIT shows strong potential because it is universally adaptable to a wide range of flow cases and boundary conditions.

A different class of hybrid schemes embeds the learned model inside the solver loop, correcting the discretised operator at every timestep rather than alternating with the solver. Kochkov et al. [36] develop a learned interpolation and correction operators within a differentiable solver, reaching accuracy similar to a conventional solver run at $8$–$10\times$ finer resolution and yielding $40$–$80\times$ speed-ups that remain stable over long rollouts and generalise to Reynolds numbers outside the training range. Um et al. [94] extends this idea to the training itself, running the differentiable solver forward during training so that the network learns to undo the error the solver accumulates at each step, thereby keeping the corrections stable over several hundred consecutive steps.

A related framework couples a CNN to the iterative solver to accelerate its convergence while leaving the solver to enforce physical consistency. Building on CFDnet [61], SURFNet [61] trains primarily on low-resolution data and transfers it to a handful of high-resolution cases, feeding the predictions back into the solver to achieve roughly $2\times$ acceleration that is independent of grid size.

### 2.3.8. Challenges and Future Directions

Machine learning has presented interesting solution paradigms for accelerating CFD simulations across various fronts. The most promising direction is the development of hybrid ML-CFD methods. While traditional CFD solvers and ML models alternate between computing and predicting time series data, methods like RePIT [35] and CFDNet [60] demonstrate that, when training data and loss functions are controlled, error accumulation can be mitigated, yielding significant speedups. Their online operation also enables the transferability of hybrid methods to a range of flow cases, making them more applicable to real-world CFD workflows.

The ML-enhanced CFD field is also showing promising advances through more sophisticated architectures and loss functions. Frameworks such as PINNs [67] and PNODEs [80] can enforce physical

constraints, such as incompressibility, and learn from observational data to provide accurate predictions of complex flow fields. It can therefore be argued that embedding physical laws into neural network architectures is an essential feature of future acceleration frameworks.

Despite these advancements, key challenges and opportunities remain in the field. One of these is the interpretability of the ML models. Due to their often complex nature, ML models tend to be black box models. This opacity hinders their performance compared with physics-based models, making it difficult to understand why a model makes a particular prediction [95, 26]. This is a critical issue for generalisation, as models trained solely on data may struggle with unseen geometries and flow conditions, or when extrapolating beyond the training data's probability distribution. While advancements in hardware architecture [4, 13] and automatic data generation continue to aid in providing the high-quality datasets needed for these models, the ultimate goal is to enable ML to not only accelerate simulations but also advance discoveries of new physical mechanisms and revisit empirical laws in fluid mechanics through more transparent and quantifiable models.

While efforts continue for automatic data generation and automated experimentation pipelines to provide the large, high-quality datasets needed, the ultimate goal is to enable ML to not only accelerate simulations but also to assist in discovering new physical mechanisms and revisiting existing empirical laws in fluid mechanics, by yielding more transparent and certifiable models [10, 95].

Despite the numerous advancements made in applying machine learning to CFD, the current literature still reveals several limitations. Many studies have employed NODEs only within standalone reduced-order models [70, 79], rather than integrating them directly into the CFD solver to accelerate the core time-integration step. Notably, their ability to serve as continuous time integrators has not been fully leveraged in scale-resolved simulations. That is why this work proposes a method inspired by the RePIT framework, replacing its full-order FVMN with a neural ODE that acts in a learned latent space. The RePIT framework showed that long-term, stable acceleration can be achieved by alternating between CFD and machine-learning predictions, guided by residual monitoring of the governing equations. The implementation of NODEs into a RePIT-like strategy can thus leverage its alternating solver–surrogate scheme and transfer-learning updates, complemented by NODEs' continuous-time formulation. This strategy aims to directly address the time-integration bottleneck in CFD solvers while keeping the roll-out reliable: instead of RePIT's governing-equation residual, an integration monitor flags when the latent state leaves the training distribution and hands control back to the solver. By pursuing this integration, the framework seeks to provide concrete answers to the research questions introduced earlier: whether NODEs can deliver solver-level acceleration without compromising accuracy, and how runtime integration monitoring can make such integration feasible.

# 3

# Methodology

This chapter describes the full methodology used to develop and evaluate the machine-learning-based acceleration framework for accelerating CFD simulations. The approach combines a convolutional autoencoder (AE), a Neural Ordinary Differential Equation (NODE) and a KNN integration monitor to learn and evolve low-dimensional representations of chaotic flow fields. The entire framework is developed and validated on a two-dimensional test case: a chaotic flow past a circular cylinder. This restriction sets the dimensionality of each component described below, while the constituent methods themselves are independent of spatial dimension. This chapter will begin by highlighting the simulation setup used to develop the framework, explaining the numerical solver, boundary conditions, and simulation dimensions. Following this, the AE, NODE, and KNN integration monitor will be explained, along with the design choices behind them.

## 3.1. Framework Overview

The framework developed in this thesis replaces the time integration of the discretised Navier–Stokes equations (3.1) with that of a low-dimensional latent state of said equations. Profiling of the solver loop of the WaterLily simulation environment showed that the pressure projection and convection–diffusion routines are the most costly steps in the loop, and both scale with the total number of degrees of freedom $N_x N_y N_u$ [98]. Across the three test cases reported (a Taylor–Green vortex, a sphere, and a moving cylinder), the pressure-Poisson projection accounts for roughly half of the per-step cost ($46$–$57\%$) and the convection–diffusion routine for a further $22$–$31\%$, so the two routines together dominate the time-stepping loop. Compressing the flow state to a latent vector of dimension $N_z \ll N_x N_y N_u$ and integrating it in time with a learned ODE offers a direct route to acceleration, provided that the latent state captures the relevant dynamics and that the surrogate model is used during regimes where it can be trusted. Figure 3.1 shows the ML pipeline, referred to as $\mathcal{P}$, that is developed in this thesis.

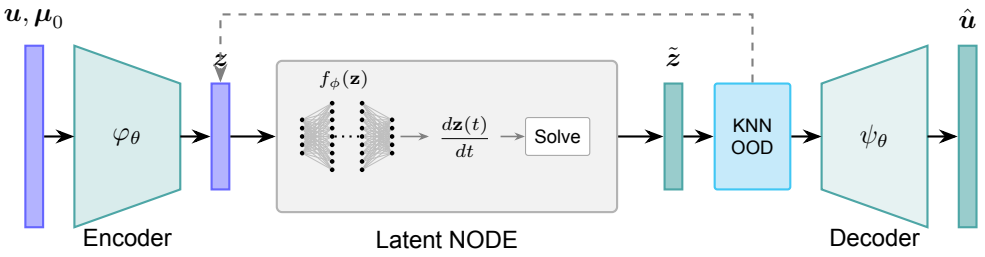



**Figure 3.1:** ML pipeline $\mathcal{P}$ of the framework developed in this thesis. The augmented flow state $(\mathbf{u}, \boldsymbol{\mu}_0^{\epsilon})$ is encoded into a latent vector $\mathbf{z}$, advanced in time by the Neural ODE $f_\phi$, monitored by the KNN out-of-distribution detector, and decoded back to a velocity field $\hat{\mathbf{u}}$ that is re-inserted into the WaterLily solver.

The autoencoder provides the non-linear compression onto the latent space. The encoder $\varphi_\theta$ compresses the augmented input $\mathbf{x}_{\text{in}} = [\mathbf{u}\,;\,\boldsymbol{\mu}_0]$, formed by concatenating the velocity field $\mathbf{u}$ with the zeroth

kernel moment $\boldsymbol{\mu}_0$ along the channel dimension, into a latent vector $\mathbf{z} \in \mathbb{R}^{N_z}$; the physical meaning of both fields and the rationale for this augmentation are discussed in Section 3.3.2. The decoder $\psi_\theta$ then reconstructs the velocity field $\hat{\mathbf{u}}$ from the latent space back to full-dimensional physical space.

The middle component of $\mathcal{P}$ in Figure 3.1, the neural ODE, provides the temporal dynamics of the latent state on the latent manifold. A small feed-forward network $f_\phi$ is used to learn the underlying rate of change of the latent dynamics, and can then be used to find the future latent $\tilde{\mathbf{z}}$ state by numerically integrating the learned time derivative from a known initial condition. The neural ODE is trained on encoded ground-truth trajectories; these trajectories are then partitioned into shorter segments and integrated independently as their own initial-value problem. The resulting pieces are then fitted to the data while penalising any mismatch at the segment boundaries. This strategy, called multiple shooting, stabilises learning over the longer rollouts required to span the relevant vortex-shedding cycles.

After the neural ODE, the KNN integration monitor provides a runtime out-of-distribution signal. It is fitted to the latent states of the full-order snapshots used during AE and NODE training, after both networks have converged, and returns the mean distance to its $k$ nearest neighbours in the training cluster. During inference, this distance can be used to monitor the neural ODE's rollout accuracy and to ensure the decoder receives a valid latent vector for reconstruction. The KNN monitor is treated as a non-trainable component and never enters either network's loss

After neural ODE integration and KNN monitor flagging, the time-integrated latent vector is reconstructed in physical space using the decoder and is then ready to be injected back into the WaterLily simulation.

The remainder of this chapter treats each component of Figure 3.1 in turn, together with the design choices made for the cylinder test case considered in this work. Section 3.4 describes the autoencoder architecture, its physics-informed loss and training procedure; Section 3.5 the Neural ODE and its multiple-shooting training strategy; and Section 3.6 the KNN integration monitor and the fallback strategy it triggers. Section 3.7 then describes how these components are coupled to the WaterLily solver during inference. Figure 3.1 serves as a reference throughout the chapter, allowing the reader to position each component in the global framework as a whole.

## 3.2. WaterLily setup

WaterLily is used as the foundational numerical framework for this work. WaterLily is an open-source, incompressible viscous flow solver written entirely in the Julia programming language Weymouth and Font [98]. It is completely backend-agnostic, meaning it supports serial and multithreaded CPU operations, as well as GPUs from different vendors. Furthermore, because WaterLily is implemented purely in Julia, the solver is fully differentiable via automatic differentiation. Because of WaterLily's backend-agnostic, differentiable architecture, it is well-positioned to integrate with optimisation and machine learning frameworks.

All simulations in this work are two-dimensional ($n = 2$), so the velocity field reduces to $\mathbf{u} = (u_1, u_2)$ on a planar Cartesian grid. Although the BDIM formulation and the WaterLily solver hold for arbitrary $n$, the cylinder test case is restricted to 2D to keep the offline data generation and autoencoder training tractable.

### 3.2.1. Numerical Solver

The incompressible NSE 2.1 are discretised in non-dimensional form in a finite-volume manner on a uniform Cartesian grid with staggered velocity-pressure variable placement. WaterLily employs the Boundary Data Immersion Method (BDIM) to handle immersed bodies on the uniform Cartesian grid [49]. Rather than generating a body-conforming mesh, BDIM smoothly couples the governing equations of the fluid and solid domains into a single equation that can be applied over the entire computational domain. To summarise this approach, the incompressible NSE, as introduced in Equation 2.1b, is defined over the fluid domain $\Omega_f$

$$\dot{\mathcal{F}}: \quad \frac{\partial u_i}{\partial t} = -\frac{\partial p}{\partial x_i} - u_j \frac{\partial u_i}{\partial x_j} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \qquad \forall\, i, j \in 1 \dots n \tag{3.1}$$

while a prescribed body velocity is enforced over the solid domain $\Omega_b$:

$$\mathcal{B}: \quad u_i = V_i \qquad \forall\, i \in 1 \dots n \tag{3.2}$$

where $u_i$ are the velocity components, $p$ is the pressure scaled by the fluid density, $\nu$ is the kinematic viscosity, $V_i$ is the body velocity and $n$ denotes the dimensions of the simulation. BDIM combines these two sets of equations by applying a convolution with a smoothing kernel $\epsilon$, to produce a combined meta-equation:

$$u_{\epsilon,i} = \mu_0^\epsilon\, f_i + (1 - \mu_0^\epsilon)\, b_i + \mu_1^\epsilon \frac{\partial}{\partial x_n}(f_i - b_i) \tag{3.3}$$

Here, $f_i$ and $b_i$ represent the time-integrated fluid and body equations, respectively, and $x_n$ denotes the surface normal direction. The key quantities to note are $\mu_0^\epsilon$ and $\mu_1^\epsilon$, which are the zeroth and the first order moments of the one-dimensional immersion kernel. These are both functions of the signed distance $d$ from a given point in the computational domain $\Omega$ to the nearest body surface. The signed distance is defined as positive in the fluid ($d > 0$) and negative inside the body ($d < 0$). In particular, $\mu_0^\epsilon$ acts as a smooth interpolation function: it equals 1 in the fluid region (for $d \geq \epsilon$), 0 inside the body (for $d \leq \epsilon$), and creates a smooth transition on the boundary region of the body $|d| < \epsilon$. The first moment $\mu_1^\epsilon$ is only nonzero in this boundary region and was put in place to improve accuracy in the presence of large discontinuities in velocity gradients at the boundary, ensuring second-order convergence of the method [49].

Time integration is performed using an explicit predictor-corrector scheme, with incompressibility enforced via a pressure-projection step at each substep. Incompressibility is enforced by solving the Pressure-Poisson equation with geometric multigrid to accelerate convergence of the solver. The governing time step is automatically adjusted to maintain a stable Courant-Friedrichs-Lewy (CFL) number. No explicit subgrid-scale model is employed; instead, WaterLily operates as an implicit Large Eddy Simulation (iLES) solver, where the numerical dissipation from the flux-limited convection scheme acts as the effective SGS model [51].

The smoothed velocity field $u_{\epsilon,i}$ is stored as a multi-dimensional array $\mathbf{u}$ of dimensions $(N_x+2) \times (N_y+2) \times n$, where $N_x$ and $N_y$ are the number of interior grid points in each spatial direction, the additional two cells per dimension are concatenated on the extremities of the computational domain and are used as ghost cells to impose boundary conditions.

For the ML framework developed in this work, two fields from the WaterLily simulation are of central importance: the velocity field $\mathbf{u}$ and the zeroth kernel moment $\boldsymbol{\mu_0}$. Where $\mathbf{u}$ represents the full flow state that the AE compresses into a low-dimensional latent representation, which will then be integrated in time by the Neural ODE. The zeroth order kernel moment $\boldsymbol{\mu_0}$ encodes the geometry of the immersed body at each grid point: it is unity in $\Omega_f$, zero in $\Omega_b$ and varies smoothly over the boundary. Because $\boldsymbol{\mu_0}$ is defined on the velocity faces, one geometry-moment component exists for each velocity component, so both fields share the same dimensions $N_x \times N_y \times N_u$. The purpose of passing $\boldsymbol{\mu_0}$ as an additional input to the model is to enable the model to learn the no-slip boundary condition and maintain physical consistency near the immersed surface. This way, the network receives explicit information about the body at a given flow state $\mathbf{u}$. In practice, both fields are extracted from the simulation at each saved timestep as a pair ($\mathbf{u}$, $\boldsymbol{\mu_0}$) and stored together for training and inference.

### 3.2.2. Physical Setup

As mentioned before, WaterLily solves the incompressible NSE on a uniform Cartesian grid with a unit cell spacing $h = 1$. The simulation operates in a non-dimensional fashion, in which all geometric dimensions are non-dimensionalised using two reference scales: the characteristic length $L$, expressed as a number of grid cells, and the characteristic velocity $U$, derived from the imposed inflow velocity. These input parameters, together with the Reynolds number, can then be used to define the kinematic viscosity: $\nu = {}^{UL}/_{Re}$, and the convective time units: $t^* = {}^{tU}/_{L}$. Next to these parameters, all spatial quantities in the solver are expressed in grid cells, and all velocities relative to $U$. Which means that the numerical solver carries no inherent physical units.

The test case used to validate the model described in Section 3.1 consists of a two-dimensional flow past a stationary circular cylinder, which induces vortex shedding in its wake. The simulation consists of a square domain with the cylinder placed in the upstream quarter, where the cylinder geometry is defined by the signed distance function and enforced via the BDIM. The simulation parameters are summarised in Figure 3.1 and the domain is illustrated in Figure 3.2.

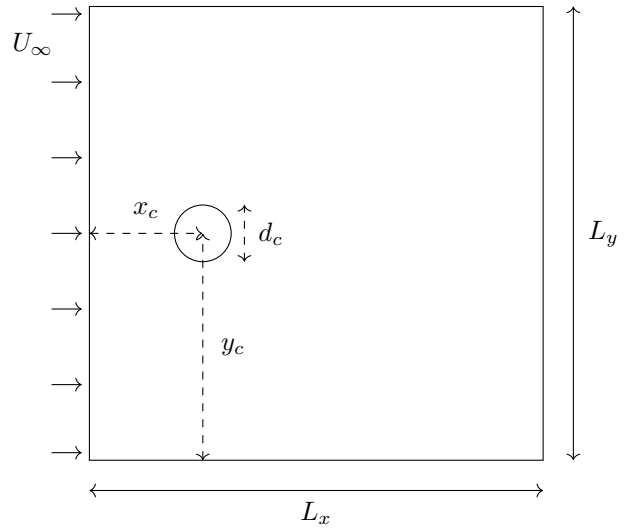



**Figure 3.2:** Domain layout.

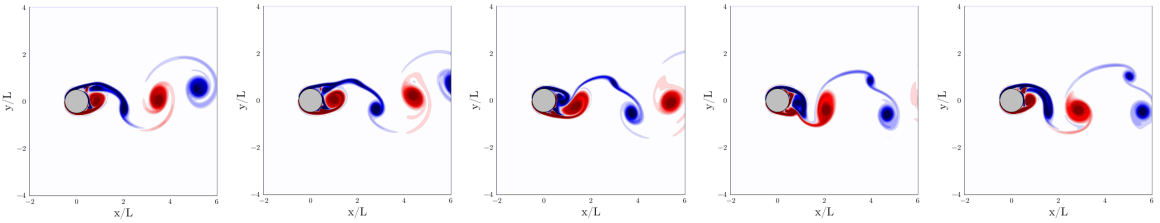
| Parameter | Symbol | Value |
|---|---|---|
| Domain width | $L_x$ | 256 |
| Domain height | $L_y$ | 256 |
| Cylinder center x | $x_c$ | $L_x/4$ |
| Cylinder center y | $y_c$ | $L_y/2$ |
| Cylinder diameter | $d_c$ | $L_x/8$ |
| Reynolds Number | $Re$ | $2500$ |
| Free stream velocity | $U_\infty$ | $(1, 0)$ |

**Table 3.1:** Simulation parameters.

A uniform free-stream velocity is imposed at the inflow, which acts as the characteristic velocity such that: $U = U_\infty$ and the cylinder diameter serves as the characteristic length scale, such that $L = d_c$. The domain uses the Biot-Savart boundary conditions [99], which replace the regular slip boundary conditions by a Biot-Savart integral over the vorticity inside the domain to determine the velocity at any point along the domain boundary. This allows the domain to be kept compact and square without introducing significant blockage effects.

Since the primary objective of this work is to develop a framework that can capture, reconstruct and accelerate the underlying flow dynamics of any unsteady CFD simulation, rather than a specific experiment or flow case, no effort is made to match the simulation parameters to any real physical system. The simulation parameters were set to infer a desired shedding behaviour, which can be used to test the capabilities of the developed framework.

A Reynolds number of $Re = 2500$ is selected as the base case for developing the framework. All snapshots used to train the ML components of $\mathcal{P}$ are drawn from this simulation, and the accelerated framework is validated against it in its in-distribution regime; the generalisation of the trained $\mathcal{P}$ to various Reynolds numbers is performed in Section 4.4. At this Reynolds number, the flow exhibits well-defined periodic vortex shedding, characterised by the alternating release of counter-rotating vortices that form the well-recognisable Kármán vortex street. Additionally, the flow produces various ranges of temporal and spatial scales beyond the primary shedding frequency, which can be seen in the secondary vortex structures in the wake snapshots in Figure 3.3.




(a) $t_n^*$ (b) $t_{n+\Delta t}^*$ (c) $t_{n+2\Delta t}^*$ (d) $t_{n+3\Delta t}^*$ (e) $t_{n+4\Delta t}^*$

**Figure 3.3:** Vorticity field $\omega L/U$ of five consecutive snapshots. Indicating positive (red) and negative (blue) vorticity. Showing the primary (large) and secondary (small) vortex structures in the wake

This makes $Re = 2500$ an interesting test case for the surrogate model: the dominant shedding period provides a clear, learnable periodic structure, while the secondary flow features introduce complexity and nonlinearity. This challenges the data-driven approach and motivates the inclusion of physics-informed regularisation and transfer learning capabilities in the framework. Where a lower Reynolds number would produce various length and time scales in the wake, preventing the model from properly testing its ability to capture multi-scale dynamics. The presence of these additional structures can be seen when plotting the force coefficients over time in Figure 3.4.

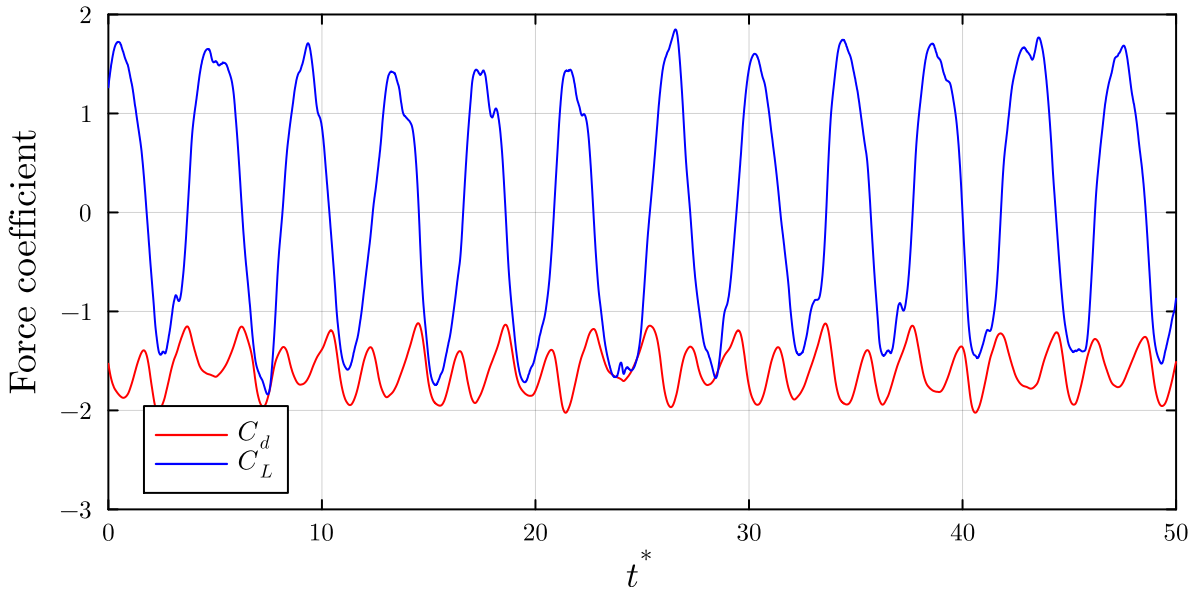



**Figure 3.4:** lift ($C_L$) and drag coefficient ($C_D$) for 2D flow past the circular cylinder at $\mathrm{Re} = 2500$, differences in amplitude and higher-frequency oscillations indicate the presence of the secondary vortex structures in the wake

While higher Reynolds numbers would introduce a broader spectrum of flow scales and further challenge the surrogate model, $Re = 2500$ represents a practical upper bound given the computational constraints of this work. Increasing the Reynolds number requires finer spatial resolution to properly resolve all relevant structures, which would rapidly increase the memory footprint and simulation wall time. The full flow-field snapshots used to train the ML framework already consume a significant portion of the available memory; a further increase in resolution would exceed the practical hardware limitations of the systems used. The chosen Reynolds number, therefore, strikes a balance between flow complexity and computational practicality.

## 3.3. Data Preprocessing

Before being passed to the autoencoder, the raw velocity snapshots produced by WaterLily need to be preprocessed to ensure proper inference and generalisability: Removal of ghost cells, assembly of the

multi-channel input tensor, and per-channel normalisation. These steps need to be applied consistently across training, validation, and inference.

### 3.3.1. Ghost Cell Removal

WaterLily employs ghost cells at the domain boundary to enforce the Biot-Savart boundary conditions [99]. These ghost cells carry values that are not physically meaningful for the flow-state observations. Since we want to learn the underlying dynamics of the flow rather than the boundary conditions, we remove ghost cells from the computational domain prior to training. Removing the ghost cells also has the additional benefit of keeping the number of interior grid points a power of 2, which will prove useful in the parametric construction of the autoencoder.

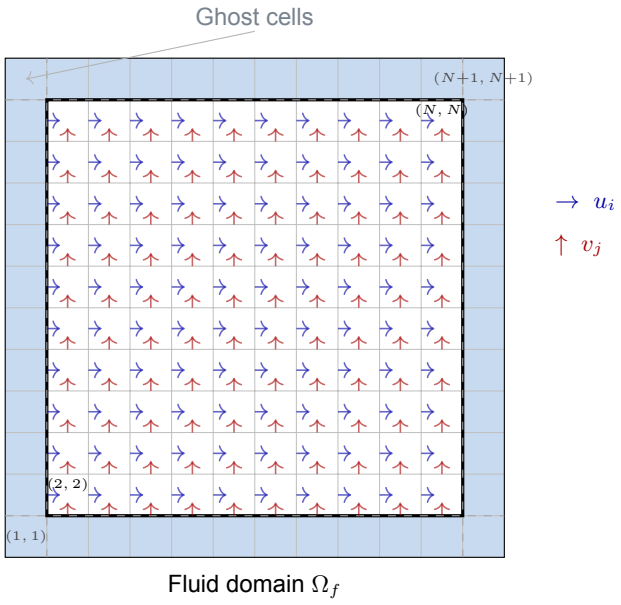



**Figure 3.5:** Illustration of the WaterLily grid showing the location of ghost cells

The ghost cells are removed by slicing the raw arrays from index $2$ to $N$ in each spatial dimension, yielding interior fields of size $N_x \times N_y$ for both the velocity field $\mathbf{u} \in \mathbb{R}^{N_x \times N_y \times N_u}$ and the zeroth kernel moment $\boldsymbol{\mu_0} \in \mathbb{R}^{N_x \times N_y \times N_u}$, where $N_x$ and $N_y$ are the number of grid cells, and $N_u$ is the number of velocity components.

### 3.3.2. Geometry-Augmented Input

The model is provided with explicit geometric information about the simulation domain by concatenating the zeroth kernel moment $\boldsymbol{\mu_0}$ to the velocity field $\mathbf{u}$ along the channel dimension, forming the input tensor

$$\mathbf{x}_{\text{in}} = [\mathbf{u}\,;\,\boldsymbol{\mu_0}] \in \mathbb{R}^{N_x \times N_y \times C_{\text{in}}}. \tag{3.4}$$

For our 2D simulation, each training sample therefore consists of a 4-channel input tensor and a 2-channel reconstruction target. The reconstruction target is the velocity field $\mathbf{u}$, which contains the $x$- and $y$-components of the velocity.

Since $\boldsymbol{\mu_0}$ is unity in the fluid and zero inside the solid, its inclusion provides the encoder with explicit geometric context, allowing it to produce latent representations that are informed by the simulation geometry. The decoder reconstructs only the 2-channel velocity field $\hat{\mathbf{u}}$; predicting $\boldsymbol{\mu_0}$ is unnecessary since, in the absence of fluid-structure interaction, the body velocity is fully determined by Equation 3.2 and therefore known at any time $t^*$.

Although the body is stationary in the cases considered in this work, with $\boldsymbol{\mu_0}$ identical across all snapshots, retaining it as an explicit input channel is a deliberate design choice intended to extend the

framework to moving-body problems. For a translating or deforming geometry with a dynamics SDF, the kernel moment becomes time-dependent, $\boldsymbol{\mu_0} = \boldsymbol{\mu_0}(t^*)$, and the input channel then passes the instantaneous body location directly to the encoder, rather than requiring it to be inferred from the velocity field alone. Providing this geometric information explicitly is expected to ease training in such cases, as the encoder doesn't need to recover the moving boundary location from velocity components alone. This extension was not implemented or evaluated in the present work, where $\boldsymbol{\mu_0}$ provides only static geometric context and is left as a direction for future work.

### 3.3.3. Normalization

The two velocity channels do not share a common scale: at $Re = 2500$ the streamwise component $u$ fluctuates around the freestream value $U$, whereas $v$ oscillates about zero. Passing these raw fields to the encoder would allow the larger-magnitude streamwise channel to dominate the training loss, biasing the latent representation towards the mean flow at the expense of dynamically relevant fluctuations. To assess both components equally during training, each velocity component is standardised with a per-component z-score transform. For each velocity component $i$:

$$\tilde{u}_i = \frac{u_i - \mu_i}{\sigma_i + \epsilon}, \qquad \mu_i = \langle u_i \rangle, \quad \sigma_i = \sqrt{\langle (u_i - \mu_i)^2 \rangle}, \tag{3.5}$$

where $\langle \cdot \rangle$ averages over all spatial locations and training snapshots, and $\epsilon = 10^{-8}$ guards against division by zero. Centring the streamwise channel removes the constant freestream offset, so the encoder allocates its latent capacity to flow fluctuations about the mean rather than to the uniform background flow.

The per velocity component mean, and standard deviation $(\mu_i, \sigma_i)$ are computed once on the training set and stored with the trained weights. The same frozen values are reused for validation and inference; recomputing them per batch or per split would leak information across the train–test boundary and break the consistency between training and inference. The geometric mask channels $\boldsymbol{\mu_0}$ are left untouched, as they already take values in $[0, 1]$ and their fluid/solid meaning would be destroyed by standardisation.

This way, the encoder and decoder operate entirely in a standardised space. The decoder's raw output is interpreted in this same standardised space, and the inverse of Equation 3.5, $\hat{u}_i = \tilde{u}_i\, \sigma_i + \mu_i$, recovers the physical reconstruction $\hat{\mathbf{u}}$. This inverse transform is applied before the reconstruction and physics-informed loss terms of Section 3.4.2 are evaluated, so that the divergence and vorticity penalties act on dimensional velocity fields.

## 3.4. Autoencoder

To harness the Neural ODE's predictive capabilities, we need a way to reduce the dimensionality of our dynamical system. This is the role the autoencoder encompasses: it provides a way to produce a compact, non-linear representation of the instantaneous flow state on which the latent dynamics can subsequently be learned. A convolutional autoencoder is well-suited to this task, as its filters are designed to extract local spatial features from grid-structured fields, and the use of non-linear activation functions allows it to represent flow states on a curved, low-dimensional manifold.

The autoencoder learns an efficient reduced-order representation of the instantaneous full-order flow state. The AE consists of two parts, namely an encoder and a decoder. The encoder $\varphi_\theta : \mathbb{R}^{N_x \times N_y \times C_{in}} \to \mathbb{R}^{N_z}$, maps the instantaneous full-order augmented input to a latent state vector $\mathbf{z} = \varphi_\theta(\mathbf{x}_{\text{in}})$, with small dimensions such that $N_z \ll N_x N_y C_{\text{in}}$. The decoder $\psi_\theta : \mathbb{R}^{N_z} \to \mathbb{R}^{N_x \times N_y \times N_u}$ then maps this latent state vector back to physical space $\hat{\mathbf{u}} = \psi_\theta(\mathbf{z})$ with the following goal.

$$\hat{\mathbf{u}} \simeq \mathbf{u}, \quad \text{where} \quad \hat{\mathbf{u}} = \psi_\theta(\mathbf{z}), \quad \mathbf{z} = \varphi_\theta(\mathbf{x}_{\text{in}}) \tag{3.6}$$

### 3.4.1. Architecture

The encoder and decoder are constructed parametrically from a small set of hyperparameters: the number of convolutional blocks $n_{\text{conv}}$, the number of dense layers $n_{\text{dense}}$, the base channel count $C_{\text{base}}$,

the convolution kernel size $k$, the pooling kernel size $p$, and the latent dimension $N_z$, all of which are summarised in Table 3.2.

| Symbol | Description |
|---|---|
| $n_{\text{conv}}$ | Number of convolutional blocks per encoder/decoder |
| $n_{\text{dense}}$ | Number of fully-connected layers per encoder/decoder |
| $C_{\text{base}}$ | Base channel count (first encoder block) |
| $k$ | Convolution kernel size |
| $p$ | Pooling/upsampling kernel size |
| $s$ | Convolution stride |
| $N_z$ | Latent dimension |

**Table 3.2:** Governing hyperparameters for parametric autoencoder construction

An illustrative overview of the autoencoder architecture with these hyperparameters is shown in Figure 3.6, which shows how the encoder, dense bottleneck, and decoder connect to form the full compression-decompression evolution. The following subsections will discuss the architecture of these components individually.

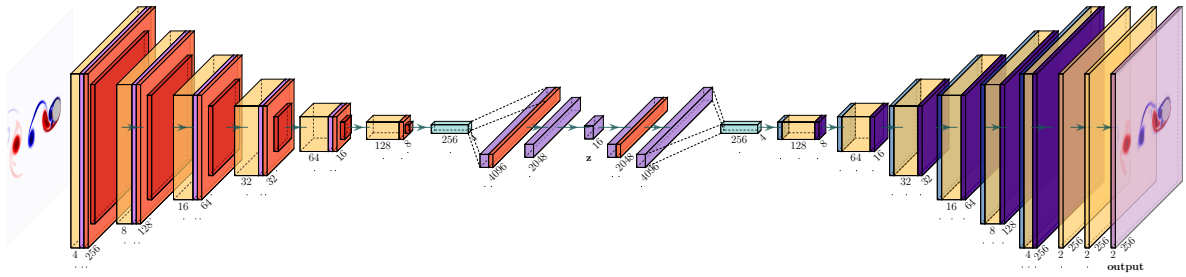



**Figure 3.6:** Schematic view of the CNN-AE architecture with $n_{\text{conv}} = 6$, $n_{\text{dense}} = 2$, $C_{\text{base}} = 8$, $k = 3$, $p = 2$, $s = 1$, $N_z = 16$ and $C_{\text{in}} = 4$. The colour coding for each layer is: 2D-convolution: , Batch normalization: , ReLU activation: , max pooling: , fully-connected layer: , GELU activation: , compressed feature map: , upsampling: .

### Encoder

The encoder consists of $n_{\text{conv}}$ stacked compression blocks, followed by a sequence of $n_{\text{dense}}$ fully connected layers that project the flattened feature maps onto the latent vector $\mathbf{z} \in \mathbb{R}^{N_z}$. Each compression block applies, in order, a 2D convolution with kernel size $k$ and stride $s$, a batch-normalisation layer, a ReLU activation, and a $p \times p$ max-pooling operation. Same padding is used in every convolution so that spatial resolution is reduced only by the pooling step. The convolution extracts local spatial features from the instantaneous flow state, and batch normalisation stabilises the input distribution of the layer across a mini-batch. Batch normalisation stabilises training by smoothing the loss landscape, scaling and shifting the layer inputs using trainable parameters to preserve model expressivity [75], thereby allowing larger learning rates and reducing sensitivity to weight initialisation. ReLU introduces the non-linearity to represent the flow states in a low-dimensional latent space.

The encoder achieves compression by doubling the channel count after each block and halving the spatial resolution via max-pooling, as shown mathematically in Equation 3.7 & 3.8. This means that at each convolutional step, the number of feature maps is doubled to extract more information from the unsteady flow snapshot, while at each downsampling step, the dimension is reduced to allow the next layer to combine the identified flow features into a new feature map. This allows the extraction of larger and more complex features for progressively deeper convolutional networks from simple non-linear combinations of the previous ones [23].

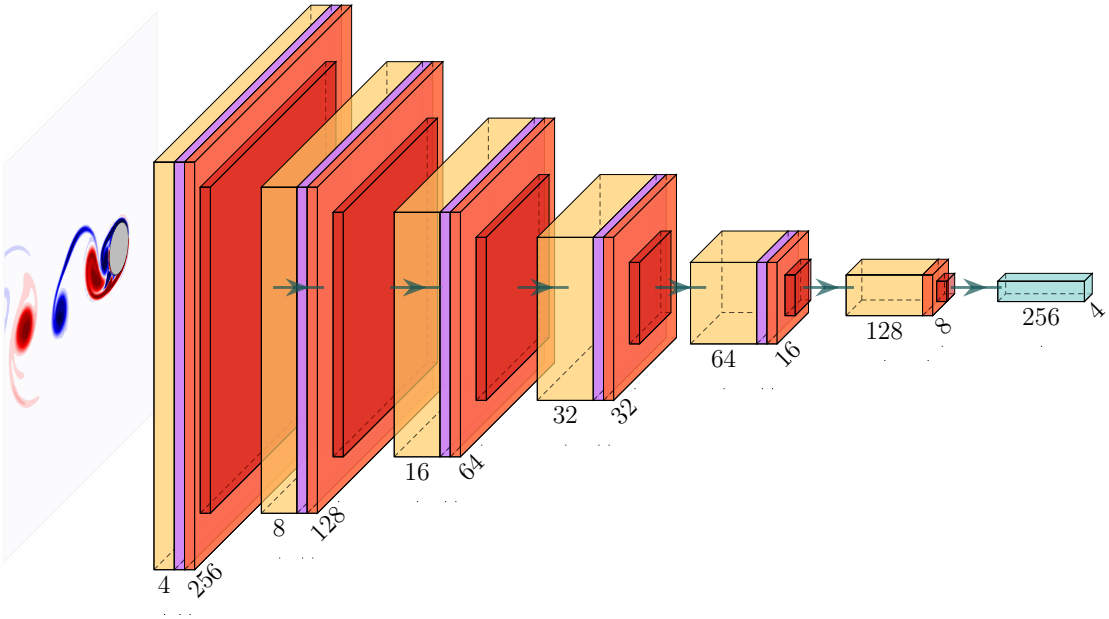



**Figure 3.7:** Example encoder architecture with $n_{\text{conv}} = 6$, $p = 2$, $k = 3$, $C_{\text{in}} = 4$ and $C_{\text{base}} = 8$. The colour coding for each layer is: 2D-convolution: , Batch normalization: , ReLU activation: , max pooling: & compressed feature map:

$$C_i = C_{\text{base}}\, 2^{\,i-1}, \qquad i = 1, \ldots, n_{\text{conv}}, \tag{3.7}$$

$$H_i = N_x/p^{\,i}, \qquad W_i = N_y/p^{\,i}, \qquad i = 1, \ldots, n_{\text{conv}}. \tag{3.8}$$

These deeper feature maps, therefore, encode larger-scale flow structures, such as the periodic wake pattern, while the shallow maps retain local features such as shear layers and boundary-layer gradients. An example schematic of an encoder is shown in Figure 3.7, illustrating how spatial size is reduced while the number of channels increases.

Batch normalisation is only applied to the interior convolutional blocks of the encoder and is deliberately omitted from the first and last blocks of the encoder. Since the first block operates directly on the augmented input data $\mathbf{x}_{\text{in}} = [\mathbf{u}\,;\,\boldsymbol{\mu}_0]$, whose channels have already been brought to a controlled scale by the per-channel normalisation, as explained in Section 3.3.3. If the input were re-normalised across the batch, knowing that the body moment $\boldsymbol{\mu}_0$ consists only of values between 1 and 0, the augmented input would no longer be physically meaningful. The last convolution block produces the final feature map that is flattened and passed to the dense layers. Normalising this feature map would couple the latent coordinates to batch-level statistics, thereby distorting the latent dynamics on which the neural ODE must be trained.

**Dense layers**

After the final compression block, the feature map of dimensions $H_{n_{\text{conv}}} \times W_{n_{\text{conv}}} \times C_{n_{\text{conv}}}$ is flattened and passed through a sequence of $n_{\text{dense}}$ fully connected layers. Each successive layer halves the number of nodes, with the last layer projecting onto the $N_z$-dimensional latent vector $\mathbf{z} = \varphi_\theta(\mathbf{x}_{\text{in}})$ shown in Figure 3.8.

The dense bottleneck performs the bulk of the dimensionality reduction: the convolutional blocks reorganise the input into a compact representation with a large number of channels, and the dense layers then learn a non-linear projection from this representation to the low-dimensional space of the latent state. No activation function is applied to the final dense layer, so that the latent vector $\mathbf{z}$ is unconstrained in $\mathbb{R}^{N_z}$ rather than being constrained by an activation function. This ensures that the Neural ODE operates on the same unconstrained domain as the encoder.

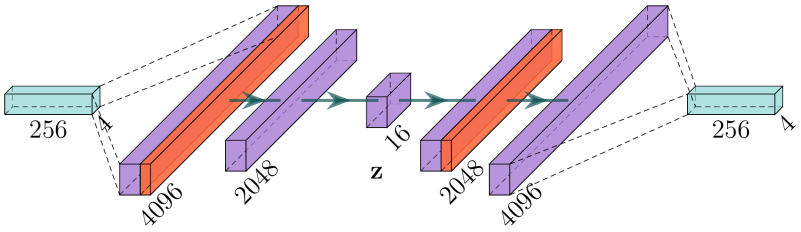



**Figure 3.8:** Example of the dense compression/decompression with $n_{\text{dense}} = 2$ and a latent size of 16. The colour coding for each layer is: ReLU activation: , fully-connected layer: & compressed feature map:

The latter half of the dense bottleneck mirrors the first half, projecting $\mathbf{z}$ back to the same dimensions as the last dense layer of the dense encoder. After this, it doubles the number of nodes at each successive layer until the original flattened feature map size $H_{n_{\text{conv}}} \cdot W_{n_{\text{conv}}} \cdot C_{n_{\text{conv}}}$ is recovered, with ReLU activations between layers. The resulting vector is then reshaped into a feature map of dimensions $H_{n_{\text{conv}}} \times W_{n_{\text{conv}}} \times C_{n_{\text{conv}}}$, serving as the starting point for the decompression blocks.

**Decoder**

The decoder $\psi_\theta$ reverses the operation done by the encoder, taking the feature map at the end of the dense bottleneck and passing it through $n_{\text{conv}}$ decompression blocks, each of which consists of a bilinear upsampling by a factor of $p$, a 2D convolution with kernel size $k$ and stride $s$, and a GELU activation.

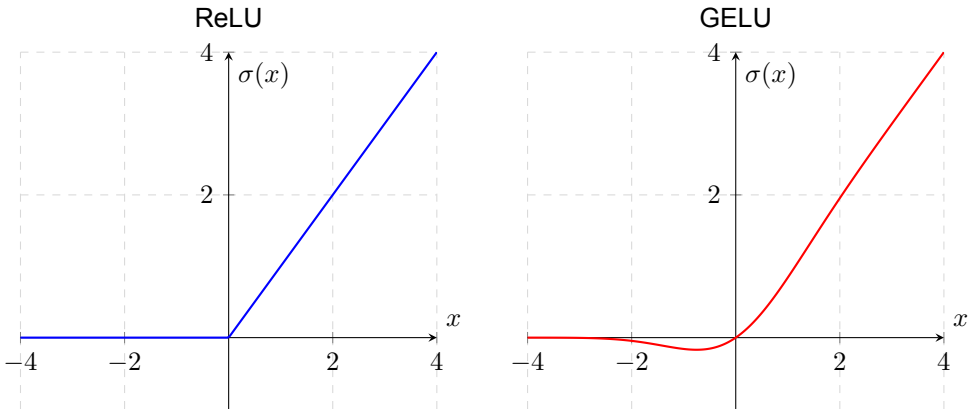



**Figure 3.9:** Comparison of the ReLU and GELU activation functions. ReLU zeroes out all negative inputs, while GELU smoothly gates negative inputs near zero and asymptotes to the identity for large positive inputs.

The upsampling step reverses the spatial compression introduced by the encoder's pooling operation, aiming to restore the latent vector to its full physical dimensions. The subsequent convolution learns a smooth refinement of the upsampled feature map. In contrast to the encoder's ReLU activation function, the decoder uses GELU activation because the reconstructed velocity field contains both positive and negative values, and ReLU's hard zero clipping would restrict the reconstruction of the negative-velocity regions of the wake. GELU, by contrast, allows the decompression blocks to produce negative values, so that periodic flow features can be properly reconstructed. Figure 3.9 shows a graphical comparison of these two activation functions.

$$C_i = C_{\text{base}}\, 2^{\,n_{\text{conv}}-i}, \qquad i = 1, \ldots, n_{\text{conv}}, \tag{3.9}$$

$$H_i = N_x \cdot p^{\,i-n_{\text{conv}}}, \qquad W_i = N_y \cdot p^{\,i-n_{\text{conv}}}, \qquad i = 1, \ldots, n_{\text{conv}}. \tag{3.10}$$

In the decoder, the decompression blocks halve the channel count while doubling the spatial dimension in a manner symmetrical to the encoder's compression path, as shown in Equation 3.9 & 3.10. The final decompression block is followed by two additional convolutional operations, seen on the right

in Figure 3.10. The first of these applies a $k \times k$ convolution that maps the last decoder channel count, which is the same as the input channel count in the encoder $C_{\text{in}}$, to $N_u$, so that they match the dimensions of the target velocity field. The second layer applies another $k \times k$ convolution with $N_u \to N_u$ channels and serves as a learned smoothing layer that suppresses checkerboarding artefacts produced by repeated bilinear upsampling. Both of these last layers have linear activation functions, so the reconstructed velocity field is a linear combination of the last feature map.

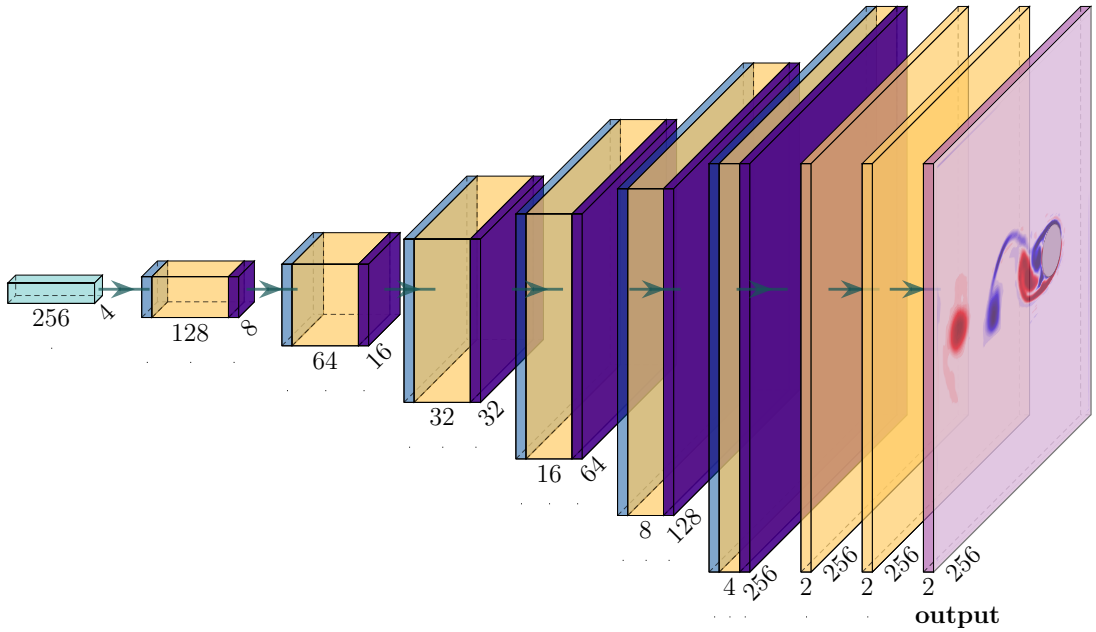



**Figure 3.10:** Example decoder architecture with $n_{\text{conv}} = 6$, $p = 2$, $k = 3$, $C_{\text{in}} = 4$ and $C_{\text{base}} = 8$.
The colour coding for each layer is: upsampling: , 2D-convolution: , GELU activation: & compressed feature map:

Batch normalisation is not included in the decoder, because the goal of the decoder is to reconstruct a specific physical field with known statistics (like divergence, kinetic energy or mean velocity), and it thus should not regularise the intermediate feature maps as it would interfere with the correct reconstruction of the learned flow states.

### 3.4.2. Loss Function

Following the physics-informed surrogate models introduced in Section 2.3.5, the autoencoder in this work is trained by minimising a composite loss function (3.11) that combines a reconstruction accuracy term with two physics-based regularisers derived from the physical and numerical properties of the simulation we are aiming to enhance.

$$\mathcal{L}_{\text{AE}} = \mathcal{L}_{\text{rec}} + \lambda_{\text{div}} \mathcal{L}_{\text{div}} + \lambda_{\text{curl}} \mathcal{L}_{\text{curl}} \tag{3.11}$$

The reconstruction loss $\mathcal{L}_{\text{rec}}$ measures the pointwise accuracy of the encoded and decoded velocity field against the reference field $\mathbf{u}$ produced by WaterLily.

$$\mathcal{L}_{\text{rec}} = \left\langle \left\| \mathbf{u} - \psi_\theta(\varphi_\theta(\mathbf{u})) \right\| \right\rangle \tag{3.12}$$

Where $\|\cdot\|$ denotes a general norm like L1 or L2 over all spatial locations and velocity components, and $\langle\cdot\rangle$ denotes an average over the available data during training. The reconstruction target is the $N_u$ dimensional velocity field $\mathbf{u}$; the kernel moment $\boldsymbol{\mu}_0$ is excluded from the target since it is a static

geometric field fully determined by the signed-distance function and carries no temporal dynamics that the autoencoder needs to learn.

The divergence loss $\mathcal{L}_{\text{div}}$ regularises the autoencoder to respect the incompressibility constraint 2.1a in the reconstructed velocity field, and is computed as the mean squared divergence over the interior domain:

$$\mathcal{L}_{\text{div}} = \left\langle \left\| \frac{\partial \hat{u}_i}{\partial x_i} \right\|_2^2 \right\rangle \tag{3.13}$$

Here, divergence is discretised using the same second-order central differences scheme as WaterLily's internal finite-volume scheme. Since WaterLily is an incompressible solver, we know that the divergence of the reconstructed velocity field should be equal to zero. By penalising the autoencoder for violating mass conservation, we know that the compressed latent trajectories, on which the neural ODE is trained, will contain minimal residual information that could lead to mass conservation violations. This is especially important during later inference rollouts, as these latent divergence residuals could accumulate, leading to reconstructed flow states that massively violate mass conservation.

The vorticity loss $\mathcal{L}_{\text{curl}}$ penalises differences between the vorticity of the reference and reconstructed velocity fields:

$$\mathcal{L}_{\text{curl}} = \left\langle \|\omega - \hat{\omega}\|_2^2 \right\rangle, \qquad \omega = \frac{\partial u_j}{\partial x_i} - \frac{\partial u_i}{\partial x_j} \tag{3.14}$$

Where $\hat{\omega}$ denotes the vorticity of the reconstructed flow snapshot $\hat{\mathbf{u}}$. The inclusion of the vorticity loss serves 2 purposes. First, the vorticity $\omega$ is constructed from the off-diagonal components of the velocity gradient tensor, $\partial u_i/\partial x_j$ and $\partial u_j/\partial x_i$, which are important for shear-related phenomena in the flow. Without imposing an explicit penalty on these off-diagonal derivatives, the autoencoder can struggle to accurately reconstruct the sharp gradients that define the shed vortices and the boundary-layer separation near the cylinder surface. Secondly, the Biot-Savart boundary conditions rely on the fluid's vorticity to determine the velocity at the domain boundary. By training the autoencoder to accurately reconstruct the vorticity of a flow snapshot, the decoded velocity fields are more likely to produce boundary-consistent ghost-cell values when the Biot-Savart conditions are re-applied during the hybrid solver workflow described in Section 3.7.3.

The weights $\lambda_{\text{div}}$ and $\lambda_{\text{curl}}$ are tunable hyperparameters that control the relative importance of each physics constraint during training. Setting them too low could allow the autoencoder to minimise the reconstruction loss without respecting the flow's physics. Setting them too high biases the optimiser toward the physics terms, potentially degrading pointwise accuracy. The choice of these weights thus involves a balance: they should be large enough to regularise the latent trajectories and produce physically relevant reconstructions, but not so large that they dominate the loss landscape and prevent the reconstruction term from driving the network to learn finer-scale flow features.

### 3.4.3. Training Setup

Since the framework in this thesis is designed to run alongside a running WaterLily simulation, the autoencoder is trained on a range of snapshots from that simulation. Where the gathered snapshots are split temporally first, according to the preset run time of the simulation $t_{\text{run}}$, then all snapshots with $t^* < t^*_{\text{train}}$ form the training and validation pool. A temporal split was chosen over a random split because the latter would have allowed snapshots that are separated by less than a fraction of a shedding period to be assigned to different splits; given the strong periodicity of the flow, such near-duplicate pairs would cause the validation and test errors to collapse onto the training error and limit any proper analysis of the loss of generalisation beyond the training horizon. From the training and validation pool, $N_{\text{train}}$ snapshots are sampled at uniform temporal spacing, of which an additional $20\%$ are held out as a validation set for epoch-level monitoring. The remaining snapshots are drawn exclusively from $t^*_{\text{train}} \leq t^* < t_{\text{run}}$, and form a held-out test set used to assess generalisation beyond the training horizon. The absolute number of snapshots in the training can be kept modest because the relevant metric of dataset quality is not the number of snapshots, but rather the number of distinct phases of the shedding cycle

covered, the training window in Figure 3.11 shows how $t^*_{\text{train}}$ was set to include 4 main shedding cycles to ensure that sufficient temporal and spatial scales are present in the training data.

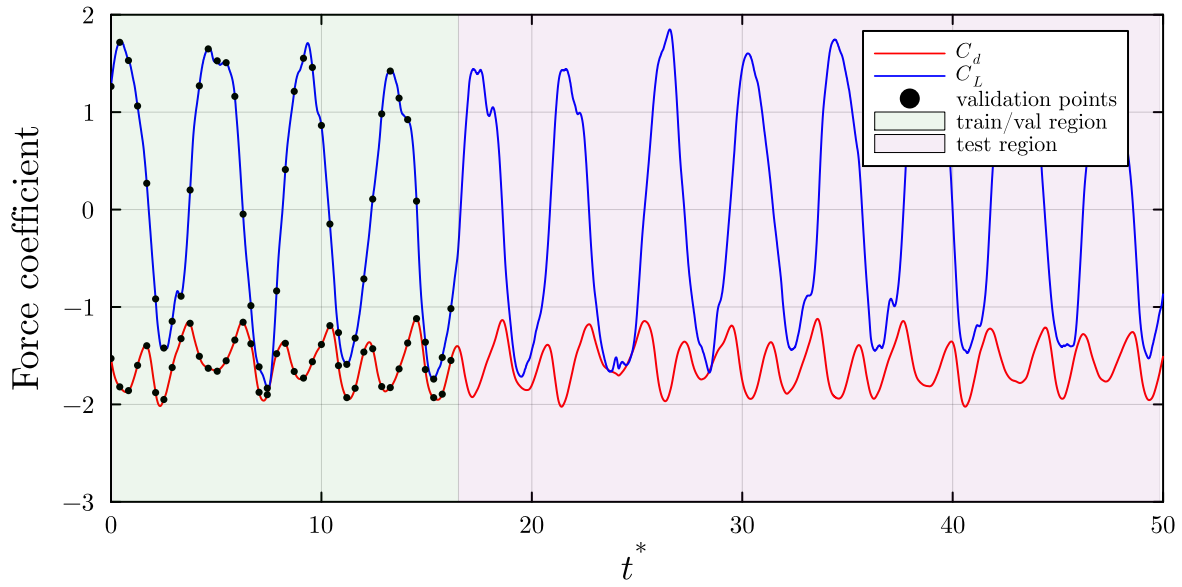



**Figure 3.11:** Train/validation and test split of the autoencoder over the simulation window. Drag ($C_d$) and lift ($C_l$) coefficients are shown over $t^*$, with the green band ($t^* \leq 17.5$) marking the train/validation region and the pink band the held-out test region; dots denote validation points. The split reserves the later shedding cycles for testing, so that reconstruction accuracy is assessed on flow states the autoencoder has not seen during training.

During training, the network weights $\theta$ are optimised using AdamW [46]. Important to note is that the encoder and decoder are optimised at the same time and thus share the same network weights, only after training is finished are the encoder and decoder treated as separate transformations. The learning rate is set according to the training regime, be it initial training or transfer learning training, with a decoupled weight decay of $\lambda = 9 \cdot 10^{-4}$, and gradients are computed via reverse-mode automatic differentiation using `Zygote.jl`. Mini-batches of $16$ snapshots are used to ensure smooth convergence of the loss function, and to save memory on the GPU. All tensors are stored in single precision (`Float32`) and training is performed on a single NVIDIA GPU on the DelftBlue HPC cluster [1].

## 3.5. Neural ODE

After having trained the autoencoder, each instantaneous flow field can be compressed to the low-dimensional latent space: $\mathbf{z} = \varphi_\theta(\mathbf{x}_{\text{in}}) \in \mathbb{R}^{N_z}$. Compressing each available snapshot in the training set in time-ordered fashion at a fixed discrete time interval $\Delta t$ (i.e., $\mathcal{T}_{z,i} = \varphi_\theta(\mathcal{T}_{u,i}$, where $\mathcal{T}_{u,i}$ is a trajectory of full-order states) produces a sequence of latent vectors that serves as the ground truth on which the neural ODE can be trained:

$$\mathcal{T}_{z,i} = [\mathbf{z}(t_{0,i}),\, \mathbf{z}(t_{0,i} + \Delta t),\, \mathbf{z}(t_{0,i} + 2\Delta t),\, \ldots,\, \mathbf{z}(t_{0,i} + t_{\text{train},i})], \quad i = 1, \ldots, N_T, \quad t_{\text{train},i} = n_{t,i}\Delta t \tag{3.15}$$

Where $N_T$ represents the total number of ground truth trajectories of the same system, $n_{t,i}$ and $t_{0,i}$ denote the training trajectory length and the start time of a specific trajectory $i$, respectively, and $t_{\text{train},i} = n_{t,i}\,\Delta t$ is the corresponding trajectory duration in physical time. For brevity we denote the full collection of $N_T$ training trajectories by $\boldsymbol{\mathcal{T}}_u \equiv \{\mathcal{T}_{u,i}\}_{i=1}^{N_T}$ and its latent image by $\boldsymbol{\mathcal{T}}_z = \varphi_\theta(\boldsymbol{\mathcal{T}}_u) \equiv \{\mathcal{T}_{z,i}\}_{i=1}^{N_T}$. An example of these resulting latent trajectories is shown in Figure 3.12 for four representative components. The latent trajectories typically consist of one or more dominant modes, such as $z_1$ and $z_4$, which exhibit a clear period behaviour that corresponds to vortex shedding frequency visible in the force history of Figure 3.4. Less dominant modes such as $z_4$ and $z_{12}$ carry lower amplitudes and contain more complex

frequency content that correspond to secondary vortex structures and irregular frequencies visible in Figure 3.3 and 3.4 respectively.

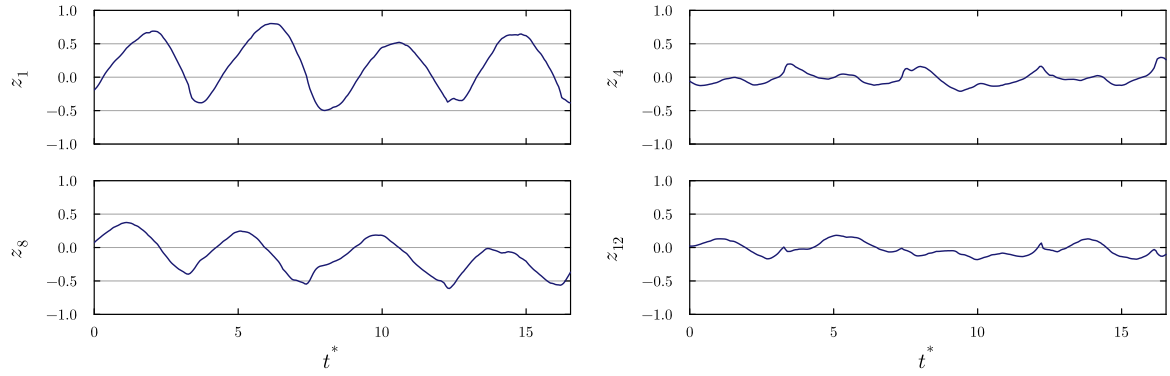



**Figure 3.12:** Representative latent components of the encoded training data over 4 shedding periods at $\mathrm{Re} = 2500$. The quasi-periodic structure reflects the vortex shedding cycle captured by the autoencoder.

The neural ODE is tasked with learning the dynamics of this trajectory so it can integrate it forward in time. As introduced in 2.3.3, the Neural ODE parametrises the time derivative of the latent state with a neural network, so that future latent states are obtained by integrating 3.16 from a known initial condition ($\mathbf{z}(t_{0,i})$) with a standard ODE solver

$$\frac{d\mathbf{z}(t)}{dt} = f_\phi(\mathbf{z}(t)), \qquad \mathbf{z}(t_0) = \varphi_\theta(\mathbf{x}_{\text{in}}(t_0)). \tag{3.16}$$

In Equation 3.16 $f_\phi$ is a neural network with trainable parameters $\phi$. Given an initial condition $\mathbf{z}(t_0)$, a conventional time integrator can be used to find the latent state $\tilde{\mathbf{z}}(t)$ at any desired time $t$. This means that, for the neural ODE defined above, a predicted latent trajectory can be produced that is analogous to the ground-truth trajectory in Equation 3.15 and starts at the same initial condition [56].

$$\tilde{\mathcal{T}}_{z,i} = [\tilde{\mathbf{z}}(t_{0,i}),\, \tilde{\mathbf{z}}(t_{0,i}+\Delta t),\, \tilde{\mathbf{z}}(t_{0,i}+2\Delta t),\, \ldots,\, \tilde{\mathbf{z}}((t_{0,i}+t_{\text{train},i})], \quad i = 1, \ldots, N_T, \quad t_{\text{train},i} = n_{t,i}\Delta t \tag{3.17}$$

Because $N_z \ll N_x N_y N_u$, the time integration operates on a low-dimensional ODE rather than the original discretised Navier-Stokes system, making both neural ODE training and inference computationally inexpensive. The following subsections highlight the architecture of $f_\phi$, the multiple-shooting training strategy used to stabilise learning over long trajectories, and the validation procedure applied to the resulting latent dynamics model.

### 3.5.1. Architecture

The right-hand side of Equation 3.16 is parametrised by the fully-connected neural network $f_\theta$. This MLP is deliberately kept compact, since the autoencoder has already distilled the flow states into a low-dimensional representation; a small network suffices to learn the latent dynamics. This has the added benefit of having a low computational cost during both training and time integration. A schematic of the network architecture is shown in Figure 3.13.

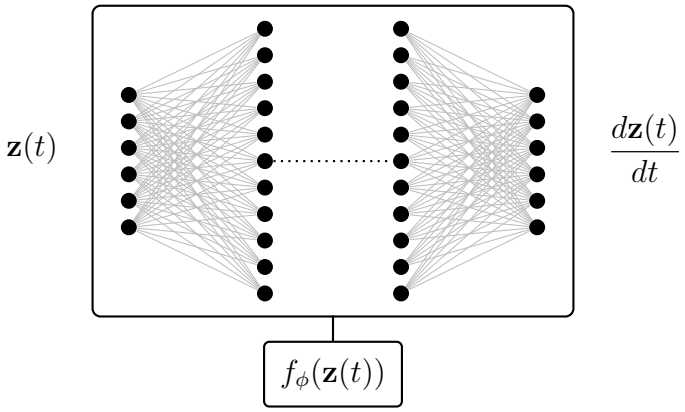



**Figure 3.13:** Architecture of the Neural ODE right-hand side $f_\phi(\mathbf{z})$

Given an initial condition $\mathbf{z}(t_0) = \varphi_\theta(\mathbf{x}_{\text{in}}(t_0))$, future latent states are obtained by numerically integrating Equation 3.16 with a common numerical ODE solver like the Tsitouras 5th-order Runge–Kutta method (`Tsit5`).

### 3.5.2. Training with Multiple Shooting

The parameters $\phi$ of the feed-forward network $f_\phi$ in the neural ODE are typically optimised by minimising an objective function, like the mean squared error, between the ground truth trajectories $\mathcal{T}_{z,i}$ and the predicted trajectories $\tilde{\mathcal{T}}_{z,i}$ during the training stage using a standard optimiser like `AdamW`. This objective function would then be given by:

$$\mathcal{L}_{\text{NODE}} = \left\langle \frac{1}{n_t} \sum_{j=1}^{n_t} \|\mathbf{z}(t_0 + j\Delta t) - \tilde{\mathbf{z}}(t_0 + j\Delta t)\|_2^2 \right\rangle \tag{3.18}$$

Where $\langle\cdot\rangle$ denotes the average over all available training trajectories $N_T$ and the inner sum accumulates the per-snapshot reconstruction error. Training the neural ODE by directly minimising Equation 3.18 requires integrating the full latent trajectory from a single initial condition $\tilde{\mathbf{z}}(t_{0,i})$ and backpropagating through the entire rollout; this is conceptually the simplest approach. It fails reliably over longer rollouts required to capture a sufficient number of shedding cycles in the regime considered here.

This approach, called single shooting, fails because the optimiser must simultaneously select parameters $\phi$ that fit the predicted trajectory to every snapshot along its full length [93]. For oscillatory problems or long rollouts, this is a difficult optimisation problem with many local minima. Additionally, neural networks exhibit a spectral bias during gradient-descent training, such that low-frequency components of the target function are learned more easily than high-frequency ones [66].

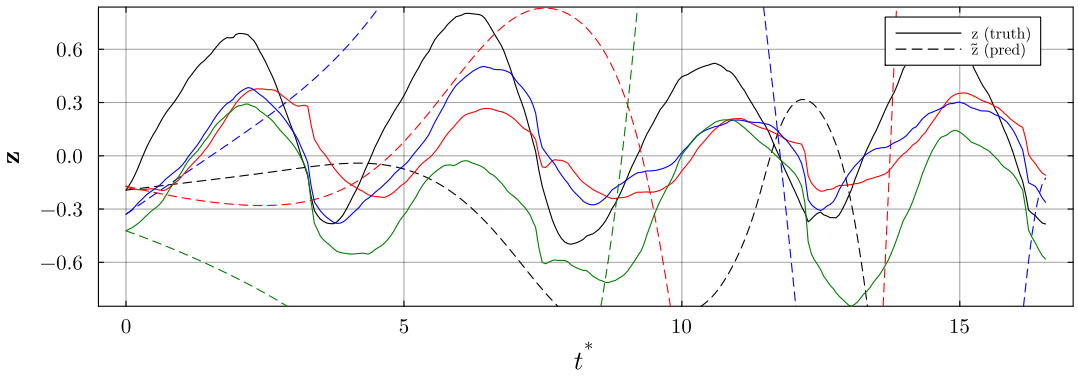



**Figure 3.14:** Single-shooting training of the neural ODE on a representative latent trajectory at $\mathrm{Re} = 2500$, displaying 4 latent components. Illustrating the failure mode single-shooting suffers from during long rollouts

Figure 3.14 shows how the learned latent trajectory gets stuck learning the complex, multi-frequency latent dynamics when operating in a single-shooting fashion.

To avoid this single-shooting failure mode, the neural ODE is trained using *multiple shooting*, a method originally developed for boundary-value problems in numerical analysis [8] and later adapted to neural ODEs by Turan and Jaschke [93]. The central idea is to replace the single, long integration required by single-shooting with several short, independent integrations whose endpoints are constrained to overlap. A specific trajectory $\mathcal{T}_{z,i}$, spanning $[t_0, t_{\text{train}}]$, is partitioned into $N_s + 1$ shorter intervals by introducing a grid of shooting points $t_0 = \tau_0 < \tau_1 < \cdots < \tau_{N_s} = t_{\text{train}}$, where each shooting group exists of $n_s$ samples of $\mathcal{T}_{z,i}$. At each interval, the neural ODE is integrated as an independent initial-value problem, with its initial condition treated as an additional optimisation variable. Figure 3.15 shows how the trajectory $\tilde{\mathcal{T}}_{z,i}$ is therefore allowed to be discontinuous across interval boundaries during training, and the resulting shooting gaps.

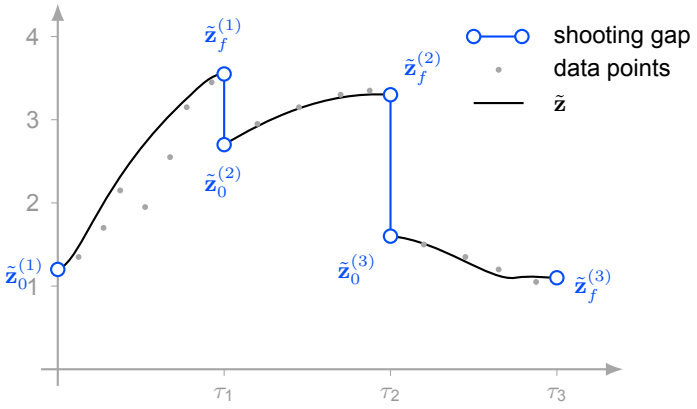



**Figure 3.15:** Multiple shooting schematic, showing the shooting gaps that need to be minimised during training, adapted from Turan and Jaschke [93]

The learned trajectory becomes meaningful only if the shooting gaps between the beginning and end points of each segment are minimised to zero, meaning that the following constraint needs to be satisfied:

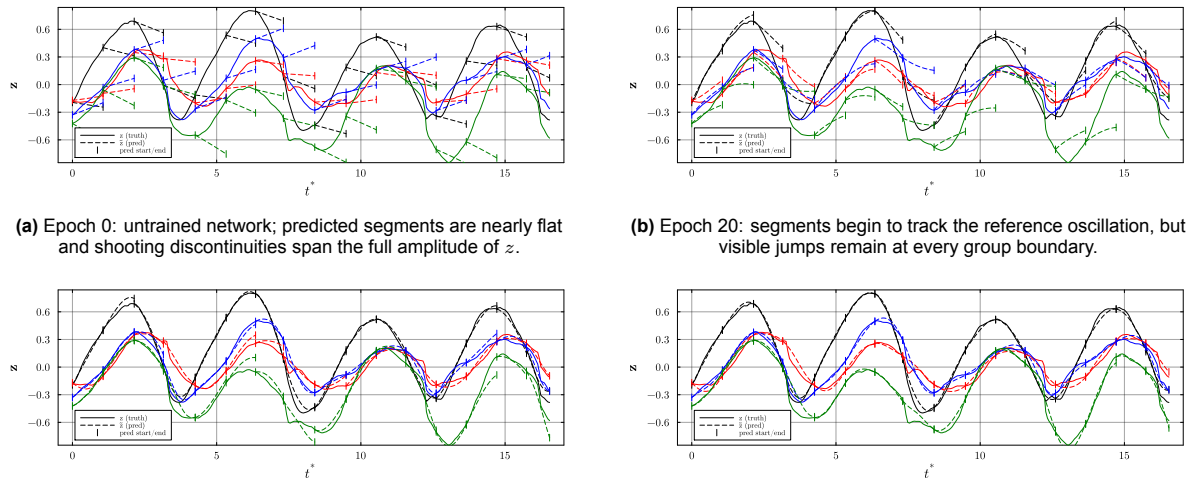
$$\tilde{\mathbf{z}}_f^{(s)} - \tilde{\mathbf{z}}_0^{(s+1)} = 0, \qquad s = 1, \ldots, N_{s,i}, \tag{3.19}$$

Supervised learning of neural networks is typically performed by optimising an unconstrained problem, which is related to the constrained optimisation problem [93]. A common approach is to define a proxy cost function, $\mathcal{L}_{\text{ms}}$, which has the constraints as penalty terms:

$$\mathcal{L}_{\text{ms}} = \left\langle \sum_{s=1}^{N_s+1} \sum_{j=1}^{n_s} \left\| \mathbf{z}_j^{(s)} - \tilde{\mathbf{z}}_j^{(s)} \right\|_2^2 + \lambda_c \sum_{s=1}^{N_s} \left\| \tilde{\mathbf{z}}_f^{(s)} - \tilde{\mathbf{z}}_0^{(s+1)} \right\|_2^2 \right\rangle, \tag{3.20}$$

The first term is a cost function similar to 3.18, but instead of computing the loss for each sample in the trajectory, it is a per-segment fit summed over all segments and their $n_s$ latent vectors. The second term is the *continuity penalty*, weighted by $\lambda_c$, that drives the shooting gaps in Equation 3.19 toward zero. The continuity weight $\lambda_c$ controls the trade-off between segment-wise accuracy and global trajectory coherence. It is typically set to a large value to ensure continuity, but remains a hyperparameter that needs to be tuned, as a too-large $\lambda_c$ can cause numerical issues, while a too-small $\lambda_c$ can lead to constraint violation.

In practice, the implementation of Turan and Jaschke [93] that is available in `DiffEqFlux.jl` was used in this work, in which the shooting initial conditions $\tilde{\mathbf{z}}_0^{(s)}$ are not free variables but are pinned to the ground-truth latents $\mathbf{z}_0^{(s)}$ at every iteration. The continuity penalty in (3.20) therefore acts between the integrated endpoint $\tilde{\mathbf{z}}_f^{(s)}$ of segment $s$ and the data initial condition $\mathbf{z}_0^{(s+1)}$ of segment $s{+}1$. Figure 3.16 shows how the segments are fitted to the ground truth latent trajectories.



**(a)** Epoch 0: untrained network; predicted segments are nearly flat and shooting discontinuities span the full amplitude of $z$.

**(b)** Epoch 20: segments begin to track the reference oscillation, but visible jumps remain at every group boundary.

**(c)** Epoch 100: shooting discontinuities begin to close, although further refinement is still needed.

**(d)** Epoch 250: converged trajectory; predicted and reference curves overlap to within line width across the full horizon.

**Figure 3.16:** Evolution of the multiple shooting latent trajectory during training. The reference trajectory $z_{\mathrm{ref}}$ (solid) is compared to the predicted trajectory $\tilde{z}_{\mathrm{pred}}$ (dashed) at four stages of training.

## 3.6. Integration Monitoring

Like other autoregressive surrogates, neural ODEs tend to struggle with the effects of accumulating error during inference outside of the training timespan [45]. This means that the trained latent dynamics $f_\phi$ cannot be trusted indefinitely. Over a long enough rollout, the predicted state $\tilde{\mathbf{z}}$ will drift away from the ground-truth trajectory $\mathbf{z}$, and no amount of additional training removes this entirely. Because we want to integrate in the latent space for as long as possible to achieve the maximum speedup, there needs to be a way for the surrogate model to check whether its current prediction can still be trusted, and a fallback mechanism for when it cannot. Since $\mathcal{P}$ is trained in situ, we can't compare the rollout prediction accuracy to a saved ground truth, so we need to compare a new prediction to available data.

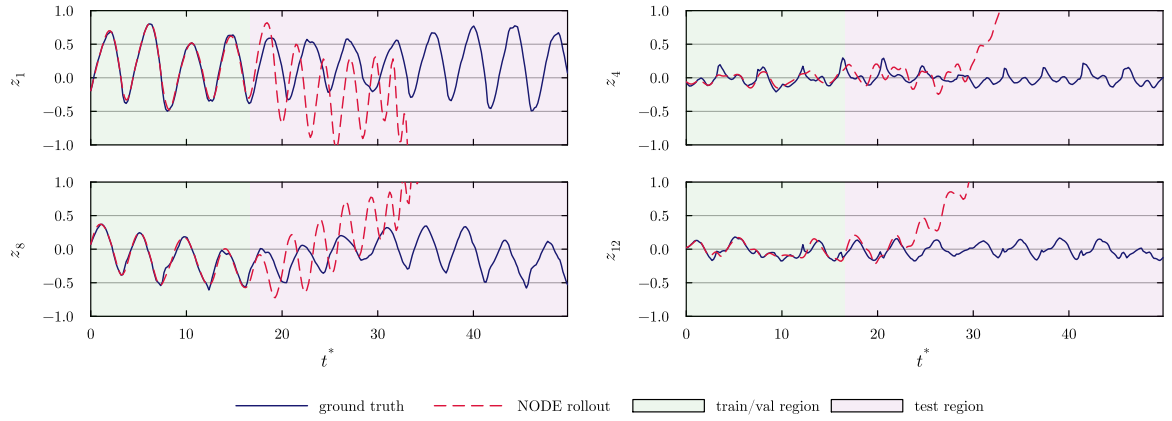



**Figure 3.17:** Latent dynamics of the trained Neural ODE over the full simulation window, shown for four representative components $z_i$. Encoded ground truth (solid) is compared against a single NODE rollout (dashed) integrated from $t^* = 0$. Showing how the rollout holds phase and amplitude inside the train/validation region (green) but drifts in the test region (pink), motivating a runtime monitor on the rollout quality.

This means that statistics of the training data need to be used to monitor the rollout prediction accuracy. One approach is to monitor the residuals of the predicted flow field, as in the RePIT framework of Jeon et al. [35], where the residual of the governing equations on the reconstructed field determines whether the network or the CFD solver advances to the next step. This does have a significant drawback in our study: each latent state vector must be projected back into physical space to compute the residuals of the governing equations, a costly operation precisely what this framework aims to mitigate.

A different class of approaches monitors the latent state directly through a statistical model of the training data. This is similar to the problem that Out-of-Distribution (OOD) detection aims to solve, where various methods have been developed to detect whether a sample lies outside the training range of an ML model [41, 91, 77]. However, as Sun et al. [90] argues, most of these methods make strong assumptions on the underlying distribution of the feature space.

Taking into account the fact that the framework in this thesis aims to be geometry agnostic, and that the training of the encoder is $\varphi_\theta$ is a nonlinear stochastic process which cannot be assumed to produce the same encoder weights $\theta$ every time; this means that the compression operation yields a different latent geometry for every training run; any assumed distribution would have to be re-fitted for each new geometry or retraining, and its shape is not known in advance. A non-parametric approach is therefore needed, one that uses the training samples directly without imposing a distribution on them. For this reason, this work uses a $k$-nearest-neighbour (KNN) score, following the deep-feature detector of Sun et al. [90]. The score measures the distance between a predicted latent vector and its K nearest neighbours in the set of encoded training snapshots, and provides information about the shape of the underlying distribution. The construction of this scoring metric and its use are described in the following parts of this section.

### 3.6.1. KNN-Based OOD Scoring

The $k$-nearest-neighbour score is computed in two stages. First, after training, the encoded training snapshots $\mathcal{T}_z$ are L2-normalised and stored in a KD-tree, which allows efficient computation of the KNN distance; secondly, the KD tree is used to query the distance to the k nearest neighbour of the newly predicted latent vector $\tilde{\mathbf{z}}$, shown in the pseudocode below in algorithm 1, following the method proposed by Sun et al. [90].

**Algorithm 1:** OOD Detection with Deep Nearest Neighbours in the Latent Space

**Input:** latent training trajectories $\mathcal{T}_z$, neighbour count $k$, threshold $\tau$, query latent state $\tilde{\mathbf{z}}$
**Output:** OOD decision $\mathbf{1}\{\, d_k(\tilde{\mathbf{z}}^{(n)}) \geq \tau \,\}$

`// Offline: reference set construction`

$\ell_2$-normalise each latent snapshot in $\mathcal{T}_z \equiv \{\mathcal{T}_{z,i}\}_{i=1}^{N_T}$ with $i = 1, \ldots, N_T$, $n = 0, \ldots, n_{t,i}$:

$\mathbf{z}_{i,n}^{(n)} = \mathbf{z}_{i,n} / \|\mathbf{z}_{i,n}\|_2$

Build a KD-tree $\mathcal{K}$ over $\mathcal{Z}_{\text{ref}} = \{\mathbf{z}_{i,n}^{(n)}\}$

`// Online: inference-time scoring`

$\ell_2$-normalise the query latent state: $\tilde{\mathbf{z}}^{(n)} \leftarrow \tilde{\mathbf{z}} / \|\tilde{\mathbf{z}}\|_2$

Query $\mathcal{K}$ for the $k$ nearest neighbours of $\tilde{\mathbf{z}}^{(n)}$

Compute the distance to the $k$-th neighbour: $d_k(\tilde{\mathbf{z}}^{(n)}) \leftarrow \|\tilde{\mathbf{z}}^{(n)} - \mathbf{z}_{(k)}^{(n)}\|_2$

**return** $\mathbf{1}\{\, d_k(\tilde{\mathbf{z}}^{(n)}) \geq \tau \,\}$

The $\ell_2$-normalisation in the KNN OOD algorithm ensures a uniform rescaling of the latent vectors such that modes with dominant amplitudes, like $z_1$ and $z_8$ visible in Figure 3.12, would not dominate the OOD score. Without normalisation, the Euclidean distance would be dominated by the large-amplitude modes, and the OOD score would be insensitive to drift in the components that encode the secondary vortical structures. Geometrically, dividing each latent vector by its norm projects the training data onto the unit hypersphere, the set of points at unit distance from the origin in the $N_z$-dimensional latent space, so the nearest-neighbour distances compare the directions of the latent states along normalised lengths rather than their overall magnitudes.

The OOD detection effectively returns the distance $k$-th nearest training neighbour $d_k(\tilde{\mathbf{z}}^{(n)})$, and still needs to be compared to a preset threshold $\tau$ to be able to determine whether the neural ODE integration is still similar to the data it was trained on. This threshold $\tau$ is calibrated from said training distribution. When constructing the KD tree, the score $d_k$ is evaluated for every training snapshot against its own neighbours (excluding itself, because this is a trivial self-match at distance zero), producing a distribution of in-distribution scores. The threshold can then be set as the $q$-quantile of this distribution, with $q = 0.99$ as the default. If required, the user can set $q \geq 1$, in which case the implementation interprets the value as a safety margin above the worst training score, $\tau = q \cdot \max_i d_k(\mathbf{z}_i^{(n)})$, which is

occasionally useful when calibration on a small or noisy training set would otherwise produce an overly tight threshold.

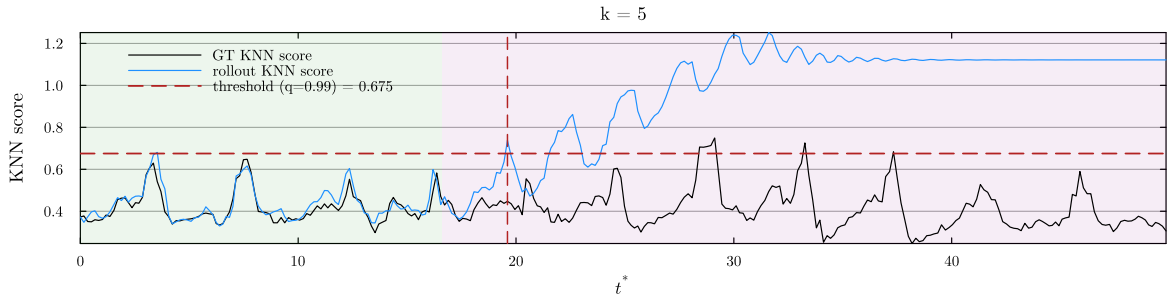



**Figure 3.18:** KNN out-of-distribution score along $t^*$ for the ground-truth latent trajectory (black) and the NODE rollout (blue), with $k = 5$. The rollout score tracks the ground truth inside the training window and crosses the threshold $\tau$ (dashed, set at the $q = 0.99$ quantile) near $t^* \approx 20$, beyond which the latent state is flagged as out-of-distribution. The shedding frequency is visible in both signals.

Figure 3.18 shows the KNN score of the ground-truth latent trajectory compared to that of the neural ODE rollout. The shedding frequency is again visible in both signals. The rollout score (blue) tracks the ground truth closely while inside the training window, but begins to diverge shortly after $t^* \approx 18$, crosses $\tau$ near $t^* \approx 20$, and remains well above it for the remainder of the rollout. This confirms that the neural ODE captures the underlying latent dynamics accurately enough to extrapolate beyond the training horizon for a limited time, but that its predictions must be monitored to ensure the reconstructed full-order state remains physically consistent.

### 3.6.2. Fallback Strategy

The KNN score from Section 3.6.1 is used to decide, at every prediction attempt, whether the latent rollout produced by $f_\phi$ can still be trusted or whether the conventional WaterLily solver should resume time integration. Whenever $d_k(\tilde{\mathbf{z}}^{(n)}) \geq \tau$, the predicted latent state is discarded and the CFD solver advances the flow field $\mathbf{u}$ from its last accepted state to stabilise the hybrid simulation again. The KNN-OOD is also used to assess whether the newly encoded data from the encoder are similar to the samples in the latent trajectories used during training, thereby serving as a monitor of autoencoder accuracy. A pseudocode example of the implemented fallback strategy is displayed in algorithm 2 below.

It is important to note that when the OOD flag is triggered, the latent neural ODE is *not* retrained immediately. A single OOD event does not imply that $f_\phi$ cannot represent the latent dynamics of the current latent state anymore; it mainly implies that the current latent rollout has accumulated too much error and, as a result, drifted into a region of the latent space where the rollout can no longer be certified against the training distribution. The conventional solver therefore takes over for $n_{\text{switch}}$ steps, advancing $\mathbf{u}$ through the Navier–Stokes solver to stabilise the simulation back to a recognisable regime that the encoder can then compress again. Retraining of the autoencoder and neural ODE is triggered when the integration monitor flags the new latent state a predetermined number of times. At that point, the current simulation state has already drifted too far from the training regime, meaning the assumption that the existing training set covers the active flow regime is no longer defensible. The in situ pipeline then updates its reference data and retrains the ML pipeline entirely.

## 3.7. Integrated Acceleration Workflow

Having introduced and motivated the three trained components of the framework in sections 3.4–3.6, we are now ready to link these components into a single inference loop that runs alongside the WaterLily solver to accelerate it via cheaper latent-space updates. The autoencoder, neural ODE, and integration monitor remain the same as in their training definitions and can now be applied within a hybrid runtime that, at every step, decides whether the next state is advanced by WaterLily or by the latent dynamics $f_\phi$.

**Algorithm 2:** KNN-OOD-gated latent rollout (`predict_flex`)

**Input:** trained pipeline $\mathcal{P} = (\varphi_\theta, \psi_\theta, f_\phi, \mathcal{K}, \tau)$, current flow state $(\mathbf{u}, \mu_0)$ at time $t_0^*$, prediction step $\Delta t^*$

**Output:** decoded flow field $\hat{\mathbf{u}}$, integration horizon $n_{\text{integr}}$, retrain flag $r$

`// encode current CFD state`

$\mathbf{z}_0 \leftarrow \varphi_\theta(\mathbf{u}, \mu_0)$

$\mathbf{z}_0^{(n)} \leftarrow \mathbf{z}_0 / \|\mathbf{z}_0\|_2$

**if** $d_k(\mathbf{z}_0^{(n)}; \mathcal{K}) \geq \tau$ **then**

  **return** $\varnothing$, $0$, true `// encoded state already OOD; defer to CFD`

**end**

$r \leftarrow$ false, $n_{\text{integr}} \leftarrow 1$, $t_n^* \leftarrow t_0^* + \Delta t^*$;

`// flexible-horizon rollout with per-step OOD check`

$\tilde{\mathbf{z}}_{\text{old}} \leftarrow \text{NODEPredict}\big(f_\phi,\ \mathbf{z}_0,\ [t_0, t_n]\big)$

**while** *true* **do**

  $\tilde{\mathbf{z}}_{\text{new}} \leftarrow \text{NODEPredict}\big(f_\phi,\ \mathbf{z}_0,\ [t_0, t_n + \Delta t^*]\big)$

  $\tilde{\mathbf{z}}_{\text{new}}^{(n)} \leftarrow \tilde{\mathbf{z}}_{\text{new}} / \|\tilde{\mathbf{z}}_{\text{new}}\|_2$

  **if** $d_k(\tilde{\mathbf{z}}_{\text{new}}^{(n)}; \mathcal{K}) \geq \tau$ **then**

    $r \leftarrow$ true

    **break** `// cut rollout, hand control back to CFD`

  **end**

  $n_{\text{integr}} \leftarrow n_{\text{integr}} + 1$

  $t_n^* \leftarrow t_n^* + \Delta t^*$

  $\tilde{\mathbf{z}}_{\text{old}} \leftarrow \tilde{\mathbf{z}}_{\text{new}}$

**end**

`// decode last accepted latent state`

$\hat{\mathbf{u}} \leftarrow \psi_\theta(\tilde{\mathbf{z}}_{\text{old}}, \mu_0)$

**return** $\hat{\mathbf{u}}$, $n_{\text{integr}}$, $r$

### 3.7.1. Execution modes

A single hybrid run consists of 4 distinct phases. The simulation begins with the initial data-collection phase, in which WaterLily runs as usual from $t_0^*$ to $t_{\text{train}}^*$. During the initial data-collection phase, relevant snapshots are sampled to populate the training data for the autoencoder and the neural ODE. Following this, the *training* phase begins at $t_{\text{train}}^*$, during which $\varphi_\theta$, $\psi_\theta$, $f_\phi$ and the KNN reference set $\mathcal{K}$ are constructed. Once training is finished, the trained components are handed off to the simulation environment, and the *hybrid loop* phase begins, in which CFD time-stepping is alternated with the latent neural ODE. If the KNN OOD monitor has repeatedly flagged a drift outside of the training distribution during this phase, the simulation enters the *retraining* phase, where $\mathcal{P}$ quits rolling out in the latent space, WaterLily resumes for a short time, and more snapshots are sampled to use as new data to retrain both networks on the augmented dataset before the hybrid loop is rejoined.

These four phases specify the logical order in which each computation proceeds, but not how that order is managed across hardware. Two execution models can be distinguished. In *serial* operation, the one actually implemented and measured in this work, all four phases run sequentially on a single lane, so each phase's cost is added to a single total wall time. In the *parallel* model, the intended deployment is for $\mathcal{P}$ to train in parallel with WaterLily and hybrid rollouts.

**Serial execution**

In the serial execution method, all four phases execute sequentially in a single process on a single execution lane, as depicted in Figure 3.19. Wall-time runs strictly left to right, and each phase completes before the next begins. WaterLily advances $\mathcal{S}$ through the initial data collection phase; it is then paused while $\mathcal{P}$ is trained; the hybrid loop subsequently alternates conventional WaterLily steps with NODE rollouts; and when the flag count reaches $n_{\text{max,flags}}$, the hybrid loop halts so that WaterLily can resume for a short time to harvest the update trajectory, after which both networks are refitted on the augmented dataset with the solver and rollouts idle. Only once retraining completes does the loop resume. This

is precisely the sequence given in algorithm 3, and the recorded timings are correspondingly additive, with no overlap between the solver and the pipeline.

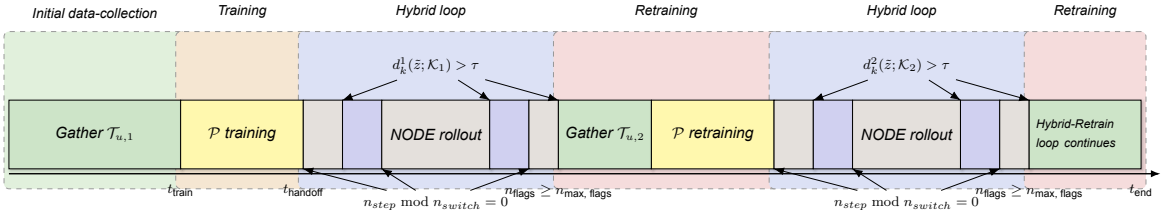



**Figure 3.19:** serial execution of the integrated acceleration workflow. The same phases run end-to-end in a single lane, with wall time progressing from left to right. Training and retraining pause WaterLily, so each phase completes before the next begins; the hybrid loop alternates conventional solver steps with NODE rollouts, and a rollout is flagged once $d_K^2(\tilde{\mathbf{z}}, \mathcal{K}_1) > \tau$.

The downside of the serial operation is the loss of the resource isolation that motivates the parallel model: with a single lane, training, CFD, and latent inference no longer occupy separate resources but compete for the same one in sequence, so every second spent training or retraining is charged directly to the run's wall-time rather than hidden behind continued solver execution. The training of the autoencoder and neural ODE, being the most expensive phase, therefore dominates the wall-time of a serial run. Beyond this, repeatedly switching the single lane among training, conventional time-stepping, and latent inference is not free: each transition incurs its own overhead, and these context switches accumulate over a run, further inflating the total wall time.

Running on a single lane keeps the implementation straightforward: there are no concurrent processes to coordinate, no in-memory exchange of weights and state between a training lane and a solver lane, and no need to reconcile a model that is still fitting against a flow that has already moved on. The trade-off is that the phases no longer overlap, so the costs of training, retraining, and switching between training, inference, and CFD are all counted towards the total wall time. The parallel model of Figure 3.7.1 remains the intended deployment; the implementation and wall-time implications of the serial choice made here are examined in Section 4.2.2

### Parallel execution

The intended deployment, illustrated in Figure 3.20, is a parallel model in which WaterLily and the training of $\mathcal{P} = (\varphi_\theta, \psi_\theta, f_\phi, \mathcal{K}, \tau)$ advance on separate lanes. The solver never pauses: it keeps integrating $\mathcal{S}$ and harvesting snapshots while the networks are fitted and, later, refitted concurrently, with the updated pipeline injected at $t^*_{\text{handoff}}$. Because the two lanes run side by side, the training cost overlaps the solver rather than adding to the wall time.

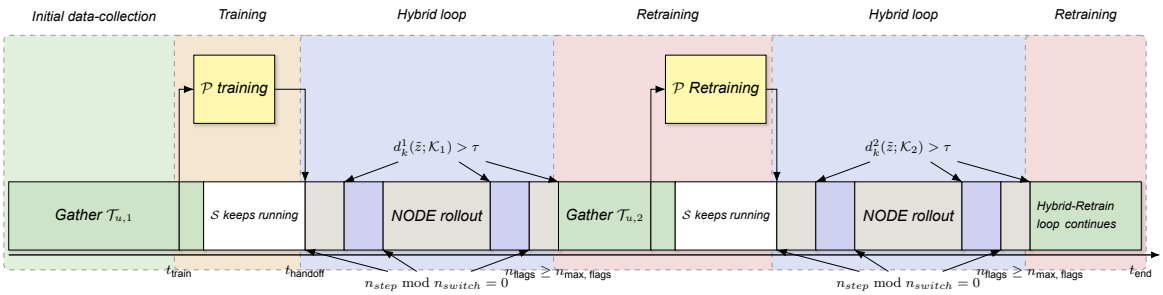



**Figure 3.20:** Idealised parallel execution. After an initial data-collection phase, WaterLily advances the simulation $\mathcal{S}$ in its own lane, while the pipeline $\mathcal{P}$ is trained and, later, retrained concurrently on a separate lane. Because the two lanes run concurrently, the training cost overlaps with the solver's execution rather than adding to the total wall time.

This execution model allows the resources for WaterLily and $\mathcal{P}$ training to be isolated from each other; this way, each computational lane can be fit for the workload it actually runs. The WaterLily lane carries only the conventional time-stepping on and the cheap latent rollouts on CPU, while the training lane can be allocated its own resources and tuned to fit the autoencoder and neural ODE more optimally

than would be tolerable if those cycles were charged against the live solver. Realising this requires concurrent computation, an in-memory mechanism for exchanging weights and state rather than disk I/O, and explicit handling of the fact that the model under training necessarily lags the live flow. The differentiable, backend-agnostic design of WaterLily makes such a coupling feasible in principle, but its implementation is beyond the scope of the presented work.

### 3.7.2. Overview of Hybrid Loop

The Hybrid loop controls how WaterLily is interwoven with the neural ODE rollouts. Once the ML pipeline $\mathcal{P} = (\varphi_\theta, \psi_\theta, f_\phi, \mathcal{K}, \tau)$ has been trained it is ready to be handed to the hybrid loop. Here, the simulation $\mathcal{S}$ is advanced from $t^*_{\text{handoff}}$ to $t^*_{\text{accel, end}}$, deciding at every step whether the next state is produced by the conventional solver or by the latent dynamics $f_\phi$. The full procedure is given in algorithm 3; the rest of this subsection walks through its behaviour and motivates two cadence hyperparameters $n_{\text{switch}}$ and $n_{\text{max, flags}}$.

**Algorithm 3:** Integrated acceleration workflow (`run_hybrid`)

**Input:** trained pipeline $\mathcal{P} = (\varphi_\theta, \psi_\theta, f_\phi, \mathcal{K}, \tau)$; WaterLily simulation $\mathcal{S}$;
hyperparameters: $n_{\text{switch}}$, $\Delta t^*$, $t^*_{\text{accel,end}}$, $n_{\text{max, flags}}$,$t^*_{\text{update}}$

**Output:** accelerated hybrid trajectory $\hat{\mathcal{T}}_u$ of $\mathcal{S}$ on $[t^*_{\text{train}},\, t^*_{\text{accel,end}}]$

```
// outer loop: alternates CFD time-stepping with periodic ML attempts
n_flag ← 0
n_step ← 0                                   // step counter within the current hybrid window
while t*(S) < t*_accel,end do
    if n_step mod n_switch = 0 then
        (û, n_integr, r) ← predict_flex(P, S, Δt*)        // surrogate rollout (Algorithm 2)
        if û ≠ ∅ and n_integr ≥ 1 then
            S ← inject(S, û, n_integr · Δt*)                          // see Section 3.7.3
        else
            S ← sim_step!(S)                    // nothing inserted; advance one CFD step
        end
        if r = true then
            n_flag ← n_flag + 1
        end
    else
        S ← sim_step!(S)                                  // conventional CFD time-step
    end
    n_step ← n_step + 1
    if n_flag ≥ n_max, flags then
        // adaptive retraining branch (Section 3.7.4)
        S, T_u,update ← run_cutoff(S, t*_update)        // pure CFD for t*_update for new trajectory
        T_u ← T_u ∪ T_u,update                                        // extend training set
        P ← retrain(P, T_u)                                  // retrain AE, NODE; refit K
        n_flag ← 0,  n_step ← 0
    end
end
return S, T̂_u
```

The loop starts with a surrogate rollout, which invokes `predict_flex` (Algorithm 2). The KNN-rolled-out return values are then used to advance the simulation in time and space. The decoded flow field $\hat{\mathbf{u}}$ and integration horizon $n_{\text{integr}}$ together indicate a successful rollout: $\hat{\mathbf{u}}$ is injected into $\mathcal{S}$ via the coupling routine described in Section 3.7.3, advancing simulation time by $n_{\text{integr}} \cdot \Delta t^*$ in a single step. When the KNN monitor has flagged the initial encoded value as OOD, an empty rollout ($n_{\text{integr}} = 0$) will be returned; in this case, the outer loop falls back to a single CFD step so as not to lose simulation progress.

After a rollout has been completed, the hybrid loop stabilises $\mathcal{S}$ by advancing by a single call to `sim_step!`,

the same routine that produced the initial data. After either a prediction rollout or a WaterLily step, the step count is increased to enable monitoring of the ML–CFD switching.

Following $n_{\text{max, flags}}$ rollouts, the loop hands control back to WaterLily for a select period $t^*_{\text{update}}$ to collect a fresh trajectory $\mathcal{T}_{u,\text{update}}$, appends it to the training trajectories $\boldsymbol{\mathcal{T}}_u$, and refits the pipeline. Both counters are then reset, and the hybrid loop resumes. The mechanics of retraining the pipeline are laid out in Section 3.7.4

The parameter $n_{\text{switch}}$ ensures that the latent rollout is attempted only after enough CFD steps to make the prediction worthwhile, given the overhead. Each call to `predict_flex` has a fixed cost from the encode–rollout–decode cycle, and the time gained from injecting $n_{\text{integr}} \cdot \Delta t^*$ into the simulation must be large enough to offset that cost. A low $n_{\text{switch}}$ wastes calls on intervals over which the latent state has barely changed, while a high $n_{\text{switch}}$ leaves potential acceleration unused.

The cumulative flag count $n_{\text{max, flags}}$ is used to distinguish when either the autoencoder needs to be retrained on new data, or to track how far removed the neural ODE is from its last seen simulation time $t^*_{\textbf{train}}$. As discussed in Section 3.6.1, the rollout drift is generally governed by the rollout distance, meaning that a rollout drift away from the ground truth does not mean that the latent dynamics $f_\phi$ are not applicable anymore in the current time regime. Requiring $n_{\text{max, flags}}$ flags before retraining acts as a buffer, which ensures that all relevant performance can be extracted from the trained parameters $\phi$ before retraining is needed.

### 3.7.3. WaterLily Coupling

The injection routine on line 7 of algorithm 3 is responsible for coupling the predicted velocity field to the simulation $\mathcal{S}$ and applying the correct boundary condition to the ghost cells of the computational domain. It ensures that $\hat{\mathbf{u}}$ does not destabilise the simulation by re-establishing a physically self-consistent state before conventional CFD time-stepping continues. Because the prediction $\hat{\mathbf{u}}$ can not be guaranteed to be completely divergence-free after the rollout, and more importantly, is generated without any knowledge of the pressure field that should accompany it through the projection step. Simply replacing the $u_{\mathcal{S}} \leftarrow \hat{\mathbf{u}}$ (algorithm 3, line 7) would result in the simulation having a pressure field that lags $n_{\text{integr}} \cdot \Delta t^*$ behind the velocity field, polluting both the velocity update and the next Poisson residual [49]. We therefore reconstruct $p$ *before* resuming time integration, in two stages, as summarised in algorithm 4.

**Algorithm 4:** Velocity-field injection (`inject`)

**Input:** simulation $\mathcal{S}$; surrogate prediction $\hat{\mathbf{u}}$; rollout span $n_{\text{integr}} \cdot \Delta t^*$
**Output:** updated simulation $\mathcal{S}$, ready to resume CFD time-stepping

```
// overwrite simulation state with surrogate prediction
1 u_S ← û,   p_S ← 0
2 Δt_S ← Δt_S ∪ {n_integr · Δt*}
// stage 1 -- recover pressure consistent with u_S
3 (u_S, p_S) ← biot_project(u_S)                          // enforce ∇·u_S = 0
// stage 2 -- Heun step to restore physical (u_S, p_S)
4 (u_S, p_S) ← biot_mom_step(u_S, p_S)     // NSE update; BDIM re-imposes near-body physics
5 return S
```

**First Projection**

The first stage (algorithm 4, line 4) is a single Biot–Savart projection applied using the injected velocity field. Following [99], the partitioned Poisson–Biot–Savart system is evaluated to make the injected velocity field actually divergence-free, and to calculate the matching pressure field to achieve the divergence-free condition. After this step, the interior velocity satisfies $u_{i,i} = 0$ and a pressure field $p_{\mathcal{S}}$ has been recovered that is coupled with $\hat{\mathbf{u}}$, removing the lag introduced by the rollout.

**Second Predictor-Corrector**

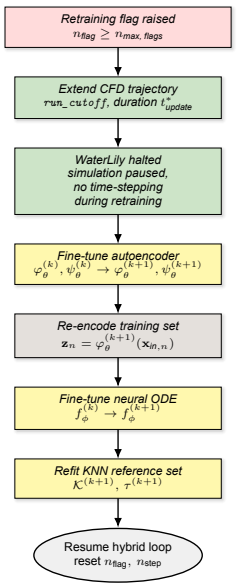
The second stage (algorithm 4, line 5) is a single Heun predictor–corrector step using the Biot–Savart projection on both substeps, identical to the scheme `mom_step!` that applies during ordinary time integration. This step is needed because the pressure recovered by the first Biot–Savart Projection is

coupled to $\hat{\mathbf{u}}$, in the sense that it makes the predicted field divergence-free, but $\hat{\mathbf{u}}$ itself is not a correct Navier–Stokes state; particularly due to the fact that the decoder struggles with reproducing sharp gradients near the body boundary, which do not match the kernel-smoothed momentum balance that BDIM enforces near the immersed boundary [49]. When the Poisson solver is asked to make such a field divergence-free, the resulting pressure contains pressure values that absorb these inconsistencies into a non-physical pressure impulse. If we were to sample the pressure force on the body at this stage, the resulting forces would produce a sudden peak in $C_d$ and $C_l$, which could offset the stability of the simulation $\mathcal{S}$.

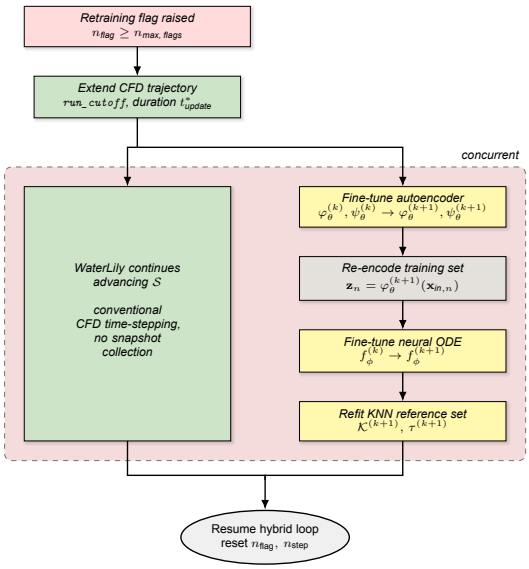
The second step resolves this discrepancy by letting the discretised Navier–Stokes operators act on the state once. This operation combines the convection-diffusion update with the BDIM correction, such that the correct physics are applied near the body region $\mathcal{B}$ through the $\mu_0^\epsilon, \mu_1^\epsilon$ moments of the immersed-boundary kernel, the projection step in the `mom_step` then recovers a pressure that is consistent with the correct physics rather than with the predicted reconstruction. One predictor-corrector step has been found sufficient in practice; it was observed that the force-coefficient peaks that appear after stage 1 are completely mitigated by the second step, and WaterLily can proceed without issue. The computational cost of the additional step of these 3 Poisson solves is negligible compared to the rollout horizon $n_{\text{integr}} \cdot \Delta t^*$ that the neural ODE has just produced.

### 3.7.4. Adaptive Retraining

When the hybrid loop has flagged for $\mathcal{P}$ to be retrained after $n_{\text{max, flags}}$ rollouts, the hybrid simulation stops switching between CFD and latent rollouts and enters the retraining stage. We are now situated in line 18 from algorithm 3; following the pseudocode, the retrain routine will begin gathering new data so that the network weights $\theta$ and $\phi$ of the autoencoder $(\varphi_\theta, \psi_\theta)$ and neural ODE $(f_\phi)$ can be updated, after which the hybrid loop is rejoined. As with the execution models in Section 3.7.1 and Figure 3.7.1, this update can be mapped to computational resources in two ways, as shown in Figure 3.21. In the serial scheme (Figure 3.21a), WaterLily is halted for the full refit, so that the entire retraining span is charged directly to the run's wall-time before the hybrid loop can resume. In the parallel scheme (Figure 3.21b), the refit is instead performed concurrently with the running simulation $\mathcal{S}$, which continues advancing the flow using conventional CFD time-stepping.




**(a)** Serial retraining: the same snapshot harvest is followed by a sequential refit of the pipeline with WaterLily paused, so the cost is charged directly to wall-time and the loop resumes only once retraining completes.




**(b)** Parallel retraining: the CFD trajectory is extended by $t^*_{\text{update}}$ to harvest fresh snapshots, after which the pipeline is refitted concurrently while WaterLily keeps advancing $\mathcal{S}$

**Figure 3.21:** Adaptive retraining under the two scheduling strategies.

The retraining phase is structured around a single design principle: rather than discarding the trained pipeline $\mathcal{P}^{(r)} = (\varphi_\theta^{(r)}, \psi_\theta^{(r)}, f_\phi^{(r)}, \mathcal{K}^{(r)}, \tau^{(r)})$ and learning a new one from scratch, the existing networks

are fine-tuned on the augmented training set $\boldsymbol{\mathcal{T}}_u^{(r+1)} = \boldsymbol{\mathcal{T}}_u^{(r)} \cup \mathcal{T}_{u,\text{update}}$. This allows the existing training data to be augmented with $\mathcal{T}_{u,\text{update}}$ from the same simulation, which shares the dominant statistical structure of the already-encoded data. This means that the existing weights $\theta^{(r)}, \phi^{(r)}$ have already been fitted to similar flow dynamics and do not need a high number of optimisation steps to update themselves to learn the newly seen flow dynamics. The following paragraphs will detail the exact meaning of the retrain flagging and the operations needed for a single retraining routine.

**Retrain flagging**

At this stage, it is important to note what the retrain flag actually signifies. As mentioned in Section 3.6, the KNN monitor tracks the proximity of the rolled-out latent state $\tilde{\mathbf{z}}$ to the training snapshots in $\boldsymbol{\mathcal{T}}_z$; this means that an OOD flag indicates that the neural ODE is being asked to extrapolate into the region of latent space for which empirical evidence of the validity of $f_\phi$ still exists. It doesn't directly indicate that the rolled-out trajectory is inaccurate, but given the periodic nature of the flow, the assumption can be made that a drift away from the common distribution can indicate that $f_\phi$ has learned a certain flow behaviour that was not explicitly seen in the training trajectory $\boldsymbol{\mathcal{T}}_z$, but is present in the complete simulation trajectory. This means that the retraining criterion has a certain uncertainty aspect to it, where it cannot know if the drift that it has repeatedly flagged is caused by accumulated error by the neural ODE, or if the repeated drift is caused by an unseen, but correct, flow prediction.

The retraining criterion thus implements a deliberate conservative policy: rather than wait for a visible divergence in the reconstructed flow field, by which point unphysical state has already been injected into $\mathcal{S}$ via the coupling routine of Section 3.7.3, the loop quits the latent rollout-CFD switching as soon as the neural ODE is repeatedly asked to operate outside the data that can support the new prediction for it. $n_{\text{max, flags}}$ controls the strength of this policy, where a small value will trigger retraining at the first flagged drift, accepting the cost of refitting in exchange for accurate coverage; a large value will tolerate more rollouts between retraining, accepting a greater risk of inaccurate rollouts in exchange for better acceleration performance.

**Data Extension**

Following the flowchart in Figure 3.21a and line 19 in algorithm 3, after the CFD data has been extended using the same snapshot sampling cadence as in the initial data-collection phase, the new trajectory can be appended to the existing training trajectory:

$$\boldsymbol{\mathcal{T}}_u^{(r+1)} = \boldsymbol{\mathcal{T}}_u^{(r)} \cup \mathcal{T}_{u,\text{update}} \tag{3.21}$$

Where the new trajectory $\mathcal{T}_{u,\text{update}}$ has its own distinctive trajectory length $n_{t,update}$, but has a matching $\Delta t$ as the initial trajectory to ensure that the additional trajectory can be evaluated in a similar fashion to the initial trajectory.

**Autoencoder Retraining**

As mentioned before, after having augmented $\boldsymbol{\mathcal{T}_u}$ with new CFD snapshots, the weights $\theta$ of the autoencoder can be updated such that they are able to produce trustworthy latent trajectories on which the neural ODE can be trained. Essentially, the autoencoder retraining is done in the same way as the preliminary training of Section 3.4.3, the main difference being that the weights are not initialised from a random state but inherited from the converged parent model $\theta^{(k)}$.

During retraining, the augmented snapshot set is again partitioned temporally into a training and validation pool and a held-out test pool, following the same convention as the initial training: the $20\%$ validation split monitors the loss at epoch level, while the test pool, drawn from the most recent snapshots, gauges generalisation beyond the training horizon. The partition of the augmented set is shown in Figure 3.22. The newly harvested trajectory $\mathcal{T}_{u,\text{update}}$ appears as the second training chunk near $t^* \approx 29$ and is temporally disjoint from the original window $t^* \in [0, 20]$. The empty patch in the middle of the forces indicates the range over which the neural ODE was rolled out to generate latent trajectories; these trajectories are not sampled as training data for retraining the models.

The autoencoder retraining inherits the methodology of the preliminary training, including the loss 3.11 and its physics-informed components. The only quantities that are altered during retraining are the

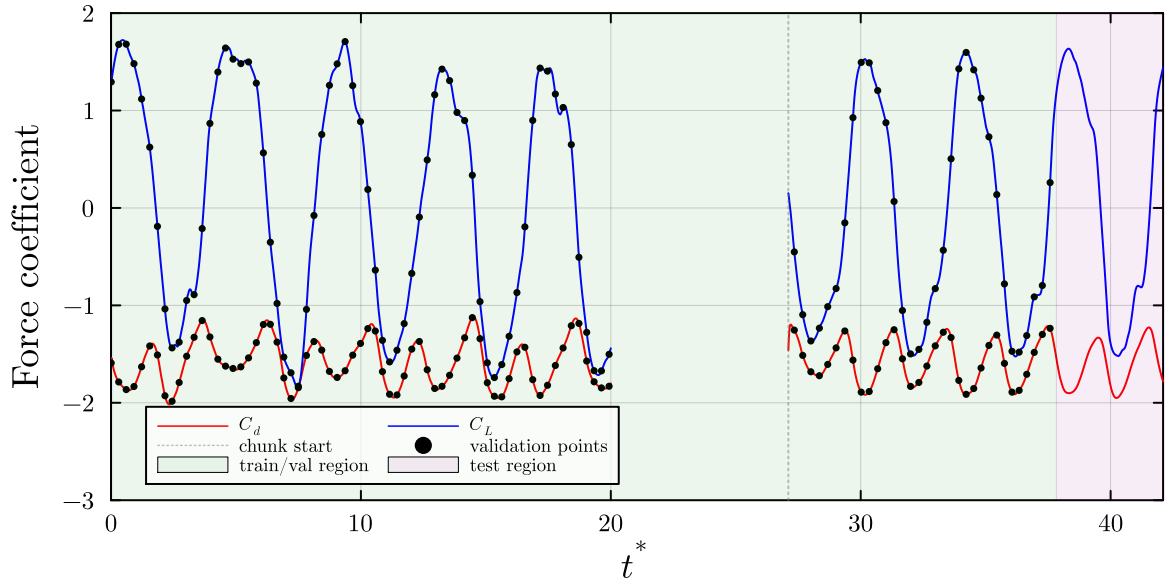
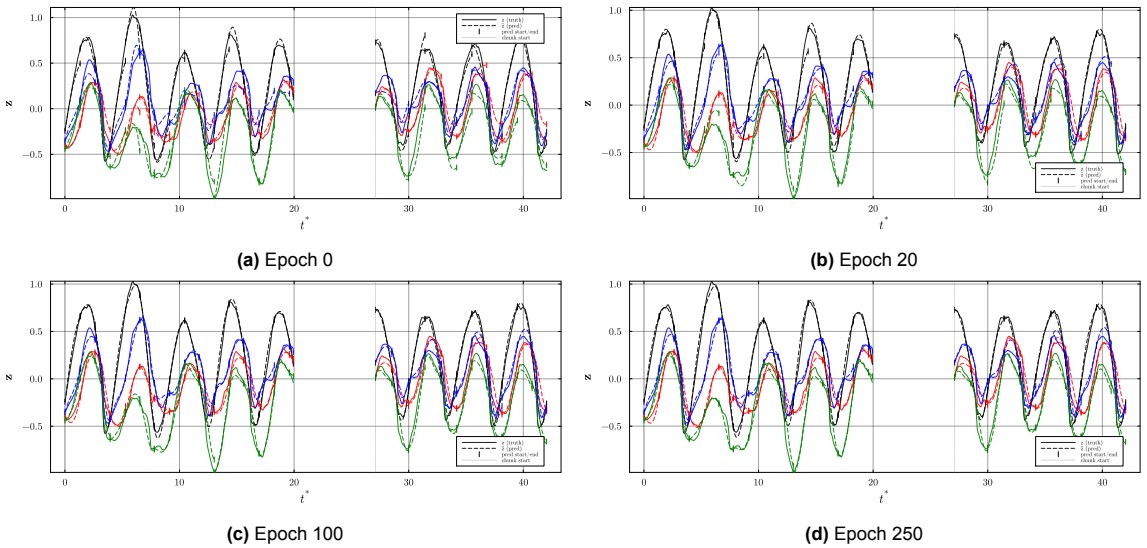



**Figure 3.22:** Train/validation (green) and held-out test (pink) split of the augmented set after retraining. Drag ($C_d$) and lift ($C_l$) coefficients are shown over $t^*$; The set spans two disjoint chunks, with the vertical break marking the second harvested chunk; dots denote validation points. The latest cycles are reserved for testing.

learning rate $\eta$ and the number of training epochs. The learning rate is lowered relative to the initial training phase because the inherited weights $\theta^{(r)}$ already produce accurate flow reconstructions; retaining the original, higher rate risks perturbing them away from the region of the loss landscape they have already converged to, rather than refining them.

### Neural ODE retraining

Before the neural ODE can be retrained, the augmented trajectory $\boldsymbol{\mathcal{T}}_u^{(r+1)}$ needs to be re-encoded with the updated autoencoder. Since the encoder weights have changed, $\theta^{(r+1)} \neq \theta^{(r)}$, the latent trajectory used for the previous fit no longer match the space the encoder now produces, so a new set $\boldsymbol{\mathcal{T}}_z^{(r+1)} = \varphi_\theta^{(r+1)}(\boldsymbol{\mathcal{T}}_u^{(r+1)})$ is generated.



**(a)** Epoch 0 **(b)** Epoch 20

**(c)** Epoch 100 **(d)** Epoch 250

**Figure 3.23:** Evolution of the multiple-shooting trajectories during neural ODE retraining, shown for four latent components. The reference $z_{\mathrm{ref}}$ (solid) is compared to the prediction $\tilde{z}_{\mathrm{pred}}$ (dashed) at four training stages, matching the full-order trajectories in Figure 3.22

The neural ODE is then retrained from the checkpoint $f_\phi^{(r)}$ with the same multiple-shooting loss (3.20) used before. The difference is that the set now holds $N_T + 1$ trajectories instead of $N_T$, so the average $\langle\cdot\rangle$ in (3.20) runs over the original trajectories and the newly added one together. A visual representation of the new fitted trajectories is displayed in Figure 3.23.

Contrasting the initial multiple shooting training in Figure 3.16, where the network starts from a random state, and the epoch-0 prediction is nearly flat. Here, the inherited network $f_\phi^{(r)}$ already tracks the first chunk closely at epoch 0, since that part of the latent dynamics was learned during the previous fit. Looking at the second trajectory, the figure also shows that $f_\phi$ already fits the unseen trajectory fairly well, since the initial shooting gaps are not very large. This means the training schedule can be shortened because the network begins near a good solution.

Following the logic in Figure 3.21a, after the neural ODE has been trained, the KNN integration monitor can be reconfigured using the new augmented trajectory $\boldsymbol{\mathcal{T}_u}$ in the same fashion as was displayed in 1. After having finished this, the updated pipeline can be reassembled using the updated autoencoder ($\varphi_\theta^{(r+1)}$, $\psi_\theta^{(r+1)}$) and latent dynamics $f_\phi^{(r+1)}$.

# 4

# Results

The preceding chapter assembled the hybrid pipeline: the convolutional autoencoder, the latent neural ODE, the out-of-distribution monitor, and the WaterLily coupling. This chapter describes how the pipeline behaves when it runs. Every result is measured against a reference simulation advanced purely by WaterLily from the same initial condition, so that the comparison isolates the effect of the latent-space acceleration alone.

The chapter is organised into four parts. Section 4.1 defines the experimental conventions: how the hybrid and reference simulations are kept co-located in time, how the workflow is mapped onto the available hardware, and the accuracy and acceleration metrics against which all subsequent results are reported. Section 4.2 characterises the base operating point in full, reporting the integrated loading, the time-averaged flow and Reynolds stresses, the latent dynamics, and the wall-time breakdown from which the two acceleration metrics follow. Section 4.3 then isolates the hyperparameters that most strongly govern the speed–fidelity trade-off, varying each in turn about the baseline to justify the adopted values and to expose the trade-offs they control. Finally, Section 4.4 steps away from the base point, lowering the Reynolds number to a cleaner shedding regime and raising it on a refined grid to assess how acceleration and physical fidelity respond to the flow regime and grid resolution.

## 4.1. Experimental Setup and Conventions

Every hybrid run reported in this chapter is accompanied by a reference simulation $\mathcal{S}_{\text{ref}}$ that is advanced purely by WaterLily. The reference simulation is instantiated using the exact same initial condition of the hybrid simulation $u_0$ at $t^* = 0$. This initial condition is a saved flow state taken from a previously developed simulation, so the wake has already reached the fully developed, periodic vortex-shedding regime, and the trajectories are free of transient start-up effects. This means that $\mathcal{S}$ and $\mathcal{S}_{\text{ref}}$ are exact copies of each other in terms of grid, immersed geometry and time-stepping parameters, and are integrated with the same solver during each conventional CFD step. The reference simulation is never re-synchronised to the hybrid state: once the two share an initial condition, they evolve as two independent trajectories, with $\mathcal{S}_{\text{ref}}$ representing the outcome the conventional solver would have produced if there weren't any latent-space acceleration. This allows us to quantify the effects of the pipeline integrations, since the two simulations differ only in whether the latent dynamics $f_\phi$ are used to advance the simulation.

All runs reported in this chapter use the serial execution mode, explained in Section 3.7.1: the data-collection, training, hybrid-loop and retraining phases execute sequentially within a single allocation, so each phase's wall time adds directly to the total. The parallel model of Figure 3.7.1 is the intended deployment but is not implemented here and left as a setup for future work; the speed-up metrics below (Section 4.1.1) are defined on this additive, non-overlapping basis.

The simulation clocks are kept aligned by monitoring the mismatch in $t^*$ across the simulations. After each update of $\mathcal{S}$, be it a numerical CFD step or a latent rollout, $\mathcal{S}_{\text{ref}}$ is advanced by repeated solver steps, governed by the CFL condition [98], until its convective time $t^*(\mathcal{S}_{\text{ref}})$ has caught up to that of the

hybrid simulation, so the two are co-located in $t^*$ to within a single time step at every checkpoint.

While both simulations advance in time, relevant mean flow components (time-averaged velocity, pressure, and velocity fluctuations) are stored in the WaterLily `MeanFlow` object. The mean flow components of the hybrid and reference simulations are sampled at fixed intervals. It is important to note that during the latent rollout (algorithm 2), each new latent vector $\tilde{\mathbf{z}}$ that corresponds to the saving interval for the mean flow components is also saved, batched and decoded together with the final rollout prediction that has not been flagged as OOD by the KNN monitor, this assures that the mean flow components of the hybrid simulation are updated using the exact same amount of snapshots at the same $t^*$.

### 4.1.1. Comparison metrics

Because the cylinder wake at $Re = 2500$ is chaotic, the hybrid and reference trajectories can quickly separate in time when given a slight perturbation in the velocity field. Because of this, comparing $\mathcal{S}$ and $\mathcal{S}_{\text{ref}}$ using a pointwise field error is not a meaningful measure of accuracy, and is therefore not used. The simulations are instead compared through statistical and integral quantities, which are the physically reproducible observables of the flow.

**Computational cost**

The primary motivation of the hybrid scheme is acceleration, so the first metric to compare is wall time. Both simulations integrate the identical convective-time window $0 \mapsto t^*_{\text{end}}$ on the same hardware, so the ratios below isolate the algorithmic gain rather than a platform difference. Two speed-up metrics are reported:

$$S_{\text{loop}} = \frac{T^{\text{ref}}_{\text{wall}}}{T^{\text{h, cfd}}_{\text{wall}} + T^{\text{h, rollout}}_{\text{wall}}} = \frac{T^{\text{ref}}_{\text{wall}}}{T^{\text{h, tot}}_{\text{wall}}}, \qquad S_{\text{eff}} = \frac{T^{\text{ref}}_{\text{wall}}}{T^{\text{h, tot}}_{\text{wall}} + T^{\text{h, training}}_{\text{wall}}} \tag{4.1}$$

Where $S_{\text{loop}}$ is the speedup achieved by the hybrid loop (algorithm 3); the denominator splits the loop costs into conventional CFD steps that stabilise the field between rollouts, $T^{\text{h, cfd}}_{\text{wall}}$, and the latent rollouts of $\mathcal{P}$, $T^{\text{h, rollout}}_{\text{wall}}$. It reflects the cost of hybrid ML-CFD timestepping after the ML models have been trained. $S_{\text{eff}}$ additionally charges the training costs $T^{\text{h, training}}_{\text{wall}}$ of the networks in $\mathcal{P}$, and is able to quantify the speed-up of the complete integrated workflow of Figure 3.19.

**Integrated forces**

Pressure force coefficients are used to compare performance in an engineering context. It allows the verification of the hybrid loop using the integrated loading on the cylinder, independent of where in the wake the two solutions agree. The time-mean drag $\langle C_D \rangle$ and the root-mean-square lift $C_L^{\text{rms}}$ are compared against the reference, as shown in Figure 3.4, through their absolute errors

$$e_{C_D} = \left|\langle C_D \rangle - \langle C_D \rangle_{\text{ref}}\right|, \qquad e_{C_L} = \left|C_L^{\text{rms}} - C_{L,\text{ref}}^{\text{rms}}\right|, \tag{4.2}$$

together with their relative counterparts

$$e^{\text{rel}}_{C_D} = \frac{\left|\langle C_D \rangle - \langle C_D \rangle_{\text{ref}}\right|}{C_{D,\text{ref}}}, \qquad e^{\text{rel}}_{C_L} = \frac{\left|C_L^{\text{rms}} - C_{L,\text{ref}}^{\text{rms}}\right|}{C_{L,\text{ref}}^{\text{rms}}}, \tag{4.3}$$

The mean drag coefficient allows the assessment of the physically meaningful streamwise loading, whereas the mean lift coefficient vanishes due to symmetry, so the fluctuation amplitude $C_L^{\text{rms}}$ is compared. The comparison of the relative errors allows comparison between different hybrid simulations. These integral force metrics confirm loading but tell nothing about the errors in the wake of the simulation, which motivates the comparison metrics of the field quantities below.

**Mean flow and Reynolds stresses**

Each time-averaged field is compared against its reference cell-by-cell over the fluid domain $\Omega$ through two complementary metrics, namely the mean absolute error (MAE):

$$\varepsilon_q = \frac{1}{N_\Omega} \sum_{\mathbf{x}\in\Omega} \left| q(\mathbf{x}) - q_{\text{ref}}(\mathbf{x}) \right|, \qquad q \in \left\{ \langle u \rangle,\, \langle v \rangle,\, \langle u'u' \rangle,\, \langle v'v' \rangle,\, \langle u'v' \rangle \right\}, \tag{4.4}$$

with $N_\Omega$ the number of fluid cells, which quantifies the pointwise error in magnitude between $\mathcal{S}$ and $\mathcal{S}_{\text{ref}}$. The spatial Pearson correlation coefficient measures whether the two fields share the same spatial structure independent of amplitude:

$$\rho_q = \frac{\sum_{\mathbf{x}\in\Omega} \left(q - \bar{q}\right)\left(q_{\text{ref}} - \bar{q}_{\text{ref}}\right)}{\sqrt{\sum_{\mathbf{x}\in\Omega} \left(q - \bar{q}\right)^2}\, \sqrt{\sum_{\mathbf{x}\in\Omega} \left(q_{\text{ref}} - \bar{q}_{\text{ref}}\right)^2}}, \qquad \bar{q} = \frac{1}{N_\Omega} \sum_{\mathbf{x}\in\Omega} q(\mathbf{x}), \tag{4.5}$$

The two are complementary: $\varepsilon_q$ captures the amplitude bias of $\mathcal{P}$ directly, whereas $\rho_q$ stays near unity as long as the wake topology is preserved, even when the fluctuation magnitudes are damped.

All of the metrics above are expressed over the full simulation time window (unless mentioned otherwise), being $0 \mapsto t^*_{\text{end}}$, which includes all CFD sections in the initial data-gathering, hybrid, training and retraining phases of the acceleration workflow.

### 4.1.2. Execution environment and hardware mapping

All reported hybrid simulations were carried out on a single `gpu-a100` node of the DelftBlue HPC cluster [1]; the node hardware and the requested SLURM allocation are summarised in Table 4.1. A single job requested 12 of the node's 64 CPU cores and one of its four A100 GPUs; the remaining cores and GPUs are unused.

| | `gpu-a100` node | Requested per run |
|---|---|---|
| CPU | 2× Intel Xeon Gold 6448Y, 64 cores @ 2.1 GHz | 12 cores |
| GPU | 4× NVIDIA A100 80 GB | 1× A100 80 GB |
| Memory | 512 GB | 48 GB |

**Table 4.1:** DelftBlue `gpu-a100` node hardware and the per-run SLURM allocation used throughout this chapter.

However, not all phases highlighted in Figure 3.19 are run on the same device. Only the autoencoder training $(\varphi_\theta, \psi_\theta)$ is placed on the A100: because this training procedure is dominated by dense, batched convolutions over the $256 \times 256$ velocity fields, all of which are operations that efficiently map onto the GPU architecture. Every other component ($\mathcal{S}_{\text{ref}}$, initial data-collection, retraining data-collection, stabilisation CFD steps, the neural ODE training $f_\phi$, and the latent rollout) runs on the 12 CPU cores. Both $\mathcal{S}$ and $\mathcal{S}_{\text{ref}}$ are advanced on WaterLily's multithreaded-CPU backend, with Julia launched on the 12 available threads so that the solver's kernel loops get parallelised across the 12 allocated cores.

This split in devices is deliberate. The neural ODE is a small network whose training and inference costs are dominated by the sequential ODE solve and reverse-mode differentiation; as a result, neither its training nor its inference benefits from running on GPU hardware in this case, so placing $f_\phi$ on the A100 would yield little performance benefit. Furthermore, the CFD domain being relatively small at $256^2$ ($\sim$0.07 M DOF) also means that it does not benefit from the advantages of WaterLily's GPU backend, since that advantage only materialises once the GPU memory is substantially filled [98]. With neither phase able to use the GPU optimally, mapping them onto the GPU means repeatedly transferring small arrays across the CPU–GPU boundary to feed kernel operations that are too cheap to recoup the transfer overhead. Because the transfer cost outweighs any benefit the GPU could provide in these phases, every component except autoencoder training is run on the CPU. Keeping all timed phases on the same device also has a methodological advantage: the reported $S_{\text{loop}}$ becomes a CPU-to-CPU comparison on identical hardware, isolating the algorithmic gain of the hybrid loop rather than conflating it with differences between devices.

# 4.2. Hybrid Acceleration of the Cylinder Flow

## 4.2.1. Baseline Configuration

As a starting point for presenting the acceleration results, a base pipeline $\mathcal{P}$ is set up against which all subsequent experiments are compared. The hyperparameters for each $\mathcal{P}$ component ($\varphi_\theta,\ \psi_\theta,\ f_\phi,\ \mathcal{K}$) are listed and discussed in the paragraphs below.

**Autoencoder**

Following the autoencoder design choices highlighted in Section 3.4, a baseline autoencoder is constructed. The autoencoder is trained to minimise the physics-informed loss function (3.11), using the $L_1$ reconstruction term with the divergence and curl regularisers. The baseline architecture was chosen via a grid search that swept over latent dimensions $N_z$, physics-informed weights $\lambda_{\text{div}},\ \lambda_{\text{curl}}$, convolutional blocks $n_{\text{conv}}$ and dense bottleneck layers $n_{\text{dense}}$. The exact scoring mechanism and all other evaluated models are discussed in Section A.2.

| Hyperparameter | Symbol | Value |
|---|---|---|
| *Architecture* | | |
| Latent dimension | $N_z$ | $16$ |
| Convolutional blocks | $n_{\text{conv}}$ | $5$ |
| Dense bottleneck layers | $n_{\text{dense}}$ | $1$ |
| Base channel count | $C_{\text{base}}$ | $8$ |
| Convolution kernel | $k$ | $3$ |
| Pooling kernel | $p$ | $2$ |
| Convolution stride | $s$ | $1$ |
| *Training* | | |
| Reconstruction loss | – | $L_1$ |
| Divergence weight | $\lambda_{\text{div}}$ | $100$ |
| Curl weight | $\lambda_{\text{curl}}$ | $100$ |
| Weight decay | $\lambda$ | $9 \times 10^{-4}$ |
| Learning rate | $\eta$ | $1 \times 10^{-3}$ |
| Retrain Learning rate | $\eta_r$ | $2 \times 10^{-4}$ |
| Optimiser | – | AdamW |
| Batch size | – | $16$ |
| Epochs | – | $500$ |
| Retrain Epochs | – | $100$ |
| Trajectory Downsample | – | $300$ |
| Train - Validation split | – | $0.2$ |
| Initial Test split | – | $0.25$ |

**Table 4.2:** Baseline autoencoder configuration.

An interesting thing to note is the choice of $n_{\text{dense}}$. During hyperparameter optimisation, the resulting score in Table A.1 consistently favoured a single dense bottleneck layer ($n_{\text{dense}} = 1$) over deeper alternatives. This reduces the dense stack in the encoder to a linear compression from the last feature map $H_{n_{\text{conv}}} \cdot W_{n_{\text{conv}}} \cdot C_{n_{\text{conv}}}$ to the latent space, meaning the encoder gains nothing from the additional ReLU nonlinearity in the intermediate dense layers for this compression case. The reason is that the additional ReLU activations rectify the negative values of the latent dynamics to zero, as shown in Figure 3.9. These negatively signed latent dynamics are essentially lost and cannot be recovered by the encoder. This loss of latent information directly degrades reconstruction quality, which is why deeper dense stacks score worse in the sweep.

The choice for $n_{\text{conv}}$ shows that convolution up to a small spatial dimension (like in the example architecture in Figure 3.7) is also adversarial to autoencoder performance, showing how a feature map with small spatial dimensions can not contain enough information to accurately be compressed to a latent vector $\mathbf{z}$. Which is why the autoencoder tuning scheme preferred five $n_{\text{conv}}$ to the deeper options

**Neural ODE**

The latent dynamics $\frac{\partial \mathbf{z}}{\partial t} = f_\phi(\mathbf{z})$ are parametrised by a single-hidden-layer multilayer perceptron and trained with the multiple-shooting loss (3.20). The hidden width is set by `dense_mult` $\times N_z$, and the shooting hyperparameters correspond to the group size $n_s$ and continuity weight $\lambda_c$ as introduced in Section 3.5.2.

| Hyperparameter | Symbol | Value |
|---|---|---|
| *Network* | | |
| Latent dimension | $N_z$ | $16$ |
| Hidden-width multiplier | `dense_mult` | $2$ |
| Hidden layers | – | $1$ |
| Activation | – | `tanhshrink` |
| *Integration* | | |
| ODE solver | – | `Tsit5` |
| Relative tolerance | – | $1 \times 10^{-3}$ |
| Absolute tolerance | – | $1 \times 10^{-5}$ |
| *Training* | | |
| Optimiser | – | AdamW |
| Learning rate | $\eta$ | $1 \times 10^{-2}$ |
| Iterations | – | $300$ |
| Retrain Iterations | – | $150$ |
| Group size | $n_s$ | $15$ |
| Continuity weight | $\lambda_c$ | $200$ |
| Retrain Continuity Weight | $\lambda_{c,r}$ | $400$ |
| Training samples | – | $300$ |

**Table 4.3:** Baseline neural ODE configuration.

As discussed in Section 3.5.1, the network is kept deliberately small With $N_z = 16$ and `dense_mult` $= 2$ it maps the latent state from $16 \to 32 \to 16$, which is sufficient to capture the smooth, quasi-periodic latent dynamics of the shedding cycle while keeping the rollout cost well below that of a conventional solver step.

The neural ODE uses `tanhshrink` as activation function, $x \mapsto x - \tanh x$. This choice was made empirically during the earlier single-shooting experiments of Section 3.5.2, where, among activation functions tried, it produced the most stable fits over the horizons on which single shooting was able to converge to a solution. Under the multiple-shooting scheme adopted here, the fit is decomposed into $N_s + 1$ independent segment integrations (Equation 3.20), each treated as a short single-shooting operation. Because training stability over long single-shot runs is no longer a relevant metric in the multiple-shooting regime, the precise form of the activation function was found to be non-decisive for this flow case. Because of this, the choice of `tanhshrink` is retained here without re-tuning.

The group size $n_s$ and continuity weight $\lambda_c$ are set empirically at $15$ and $200$; their influence on rollout stability and accuracy, together with the empirical justification for this choice, are examined in Section 4.2.7.

**Hybrid Solver and OOD Monitor**

The hybrid solver alternates between WaterLily time integration and latent rollouts, gated by the KNN out-of-distribution monitor of Section 3.6.1 using the fallback strategy of Section 3.6.2.

| Parameter | Symbol | Value |
|---|---|---|
| *OOD monitor* | | |
| Nearest neighbours | $k$ | $5$ |
| Threshold quantile | $q$ | $0.999$ |
| *Hybrid loop* | | |
| initial data-gathering window | $t^*_{\text{handoff}}$ | $20$ |
| Training window | $t^*_{\text{train}}$ | $17.5$ |
| Update window | $t^*_{\text{update}}$ | $10$ |
| Hybrid horizon | $t^*_{\text{end}}$ | $100$ |
| Prediction cadence | $n_{\text{switch}}$ | $150$ |
| Integration step | $\Delta t^*_{\text{pred}}$ | $0.011$ |
| Max amount of OOD flags | $n_{\text{max, flags}}$ | $3$ flags |

**Table 4.4:** Baseline hybrid-solver and OOD-monitor configuration.

The values of $t^*_{\text{handoff}}$, $t^*_{\text{train}}$ and $t^*_{\text{update}}$ were empirically chosen to display the learning progress of the ML pipeline, where $t^*_{\text{train}} = 17.5$ corresponds to roughly 4 shedding periods for the discussed flow case, as visible in Figure 3.4. The difference of $2.5$ with $t^*_{\text{handoff}} = 20$ is subsequently used as test data for AE training to perform a preliminary assessment of AE performance when compressing flow snapshots outside its initial training data.

The cadence of the hybrid loop is controlled by $n_{\text{switch}}$ and $\Delta t_{\text{pred}}$, where $n_{\text{switch}}$ is the number of CFD steps between rollout attempts in the hybrid loop (algorithm 3). It controls the stabilisation time between rollout attempts and, in effect, also the achievable speedup of the hybrid loop; each block of $n_{\text{switch}}$ CFD steps is the cost of allowing the injected field to relax back to a stable state before the next rollout attempt. The second variable, $\Delta t_{\text{pred}}$, sets the physical time advanced per latent step, so a successful rollout of $n_{\text{integr}}$ steps injects $n_{\text{integr}} \cdot \Delta t_{\text{pred}}$ of simulation time at once. A larger $\Delta t_{\text{pred}}$ allows for a bigger gain per step, but advances the latent state further, raising potential drift and the chance of crossing the KNN threshold (Section 3.6.1). The value for $\Delta t^*_{\text{pred}}$ was determined empirically, as it is the mean $\Delta t^*$ of the reference simulation.

### 4.2.2. Wall-Time Performance

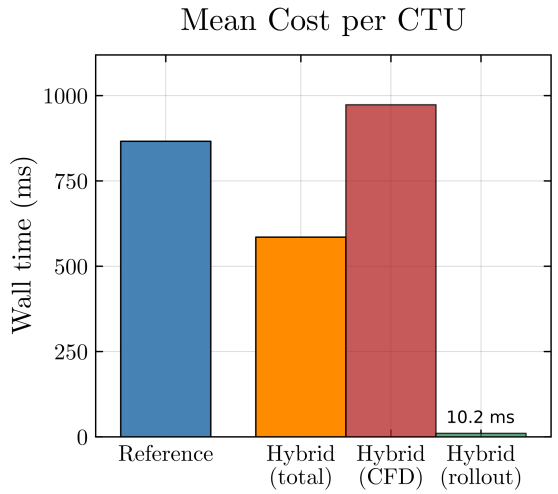
Having established the parameters of the baseline hybrid simulation in Section 4.2.1, the computational performance of the hybrid simulation can be assessed by means of wall time. Figure 4.1 shows the acceleration performance of the hybrid loop compared to the reference simulation in terms of cost per CTU and total wall-time cost.

Figure 4.1 highlights how cheap a single rollout of $\mathcal{P}$ is compared to regular CFD integration. Using $\mathcal{P}$ to predict a single CTU is $84.6$ times cheaper than calculating the same time window using regular CFD timestepping. This bar chart, however, does not include wall time for the training processes of $\varphi_\theta$, $\psi_\theta$, and $f_\phi$.

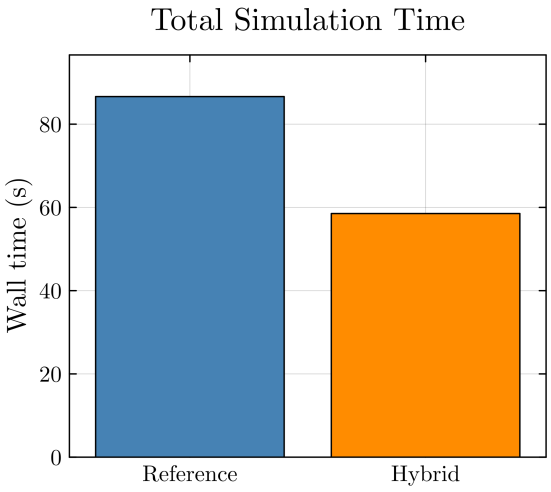
When comparing the total wall time over the same simulation window (Figure 4.1b), the hybrid loop is able to integrate $\mathcal{S}$ to $t^*_{\text{end}}$ in $58.85$ seconds compared to $\mathcal{S}_{\text{ref}}$ integrating the same time window in $86.63$ seconds. Which means the speedup achieved by the hybrid loop can be expressed as:

$$S_{\text{loop}} = \frac{T^{\text{ref}}_{\text{wall}}}{T^{\text{h, tot}}_{\text{wall}}} = \frac{86.63}{58.85} = 1.47 \tag{4.6}$$

This value for $S_{\text{loop}}$ is far below what the per-CTU rollout cost in Figure 4.1a suggests, because the loop still spends most of its time in regular CFD: every rollout is followed by $n_{\text{switch}} = 150$ CFD steps to stabilise the field, and these steps are slightly more expensive than in the reference run due to the extra encoding and monitoring carried out at each step. The cheap rollout sets the upper bound for the acceleration that the hybrid loop is able to achieve, but the number of rollouts attempted (governed by $n_{\text{flag, max}}$) and the number of CFD steps between rollouts decide how close the hybrid loop is able to get to this upper bound.




**(a)** Mean wall-time cost per CTU. The hybrid simulation is resolved into its conventional CFD and latent rollout contributions.




**(b)** Total wall time of the hybrid and reference simulation over the same convective-time window.

**Figure 4.1:** Comparison of reference simulation to the hybrid loop in terms of (a) cost per CTU and (b) actual wall time cost

This per-CTU advantage is also specific to the base-case resolution. The WaterLily solver is memory-bound, so its cost per CTU grows with grid size [98], whereas the rollout cost is fixed by the latent dimension $N_z = 16$ and is independent of the grid. The ceiling on $S_{\text{loop}}$ therefore rises at finer resolutions: the present $256^2$ grid is a comparatively cheap case in which to demonstrate the gain, and the relative advantage of the latent rollout would be larger on a grid where CFD is more expensive.

| **Phase** | **Component** | **Time (s)** | **% of total** |
|---|---|---|---|
| initial data-gathering | – | 21.16 | 1.80 |
| Train | AE | 422.00 | 35.81 |
| | NODE | 260.00 | 22.06 |
| Hybrid section 1 | CFD + rollout | 1.17 | 0.10 |
| | CFD + rollout | 1.19 | 0.10 |
| | CFD + rollout | 1.17 | 0.10 |
| Retrain 1 | Gather snapshots | 9.58 | 0.81 |
| | AE | 66.00 | 5.60 |
| | NODE | 132.00 | 11.20 |
| Hybrid section 2 | CFD + rollout | 1.18 | 0.10 |
| | CFD + rollout | 1.40 | 0.12 |
| | CFD + rollout | 1.21 | 0.10 |
| Retrain 2 | Gather snapshots | 9.22 | 0.78 |
| | AE | 66.00 | 5.60 |
| | NODE | 174.00 | 14.76 |
| Hybrid section 3 | CFD + rollout | 1.16 | 0.10 |
| | CFD + rollout | 1.17 | 0.10 |
| | CFD + rollout | 1.16 | 0.10 |
| Retrain 3 | Gather snapshots | 7.79 | 0.66 |
| **Total** | | **1,178.56** | **100.00** |

**Table 4.5:** Wall-clock breakdown of the hybrid simulation pipeline (base run).

To assess the actual speedup of the complete acceleration workflow, as described in Section 3.7, the wall times of the training procedures of the ML components in $\mathcal{P}$ need to be accounted for. Table 4.5

shows the wall time for the full pipeline, including network training. The initial training dominates: the autoencoder takes $422$ s ($35.81\%$), and the neural ODE $260$ s ($22.06\%$), so building the networks accounts for almost two-thirds of the total time before a single rollout. The three retraining cycles add a further $\sim 460$ s, while the actual CFD-plus-rollout work contributes less than $1\%$ of the total of $1176.56$ s.

Taking training into account, the effective speedup of the full workflow is

$$S_{\text{eff}} = \frac{T_{\text{wall}}^{\text{ref}}}{T_{\text{wall}}^{\text{h, tot}} + T_{\text{wall}}^{\text{h, training}}} = \frac{86.63}{58.56 + 1120} \approx 0.073, \tag{4.7}$$

This means that for a single hybrid run, the complete workflow is roughly an order of magnitude slower than simply running the reference over the selected window $0 \mapsto t^*_{\text{end}}$. One of the reasons to put the training times of the autoencoder and the neural ODE in perspective: if training of these models were to happen in parallel, $t^*_{\text{handoff}}$ would be pushed back by $1018$ CTU, which would put $t^*_{\text{handoff}} = 1038$ CTU. It would be only at this simulation time that the effective cost of $\mathcal{S}$ could be reduced by performing rollout predictions.

This means that, for a single hybrid run, the complete workflow is roughly an order of magnitude slower than simply running the reference over the selected window $0 \mapsto t^*_{\text{end}}$. The reason is purely the fixed training cost, over this window the hybrid loop saves $86.63 - 58.85 = 27.78$ s, so a single run would need to integrate for roughly $T_{\text{wall}}^{\text{h, training}}/(\text{saving per CTU}) \approx 5000$ CTU before the training cost can be repaid, against the $100$ CTU that were actually integrated. This cost can be recovered only if a trained $\mathcal{P}$ is reused across $\sim 50$ comparable runs.

If, as proposed in Section 3.7, training happens parallel to the running simulation $\mathcal{S}$, the handoff time $t^*_{\text{handoff}}$ would be pushed back significantly. Since the training of the autoencoder and the neural ODE together takes $882$ s, which is equivalent to $\sim 1018$ CTU of reference integrations. The base window of $100$ CTU is far too short to hide the training behind CFD: the reference run would have finished long before the networks would have finished training.

### 4.2.3. Force Coefficients

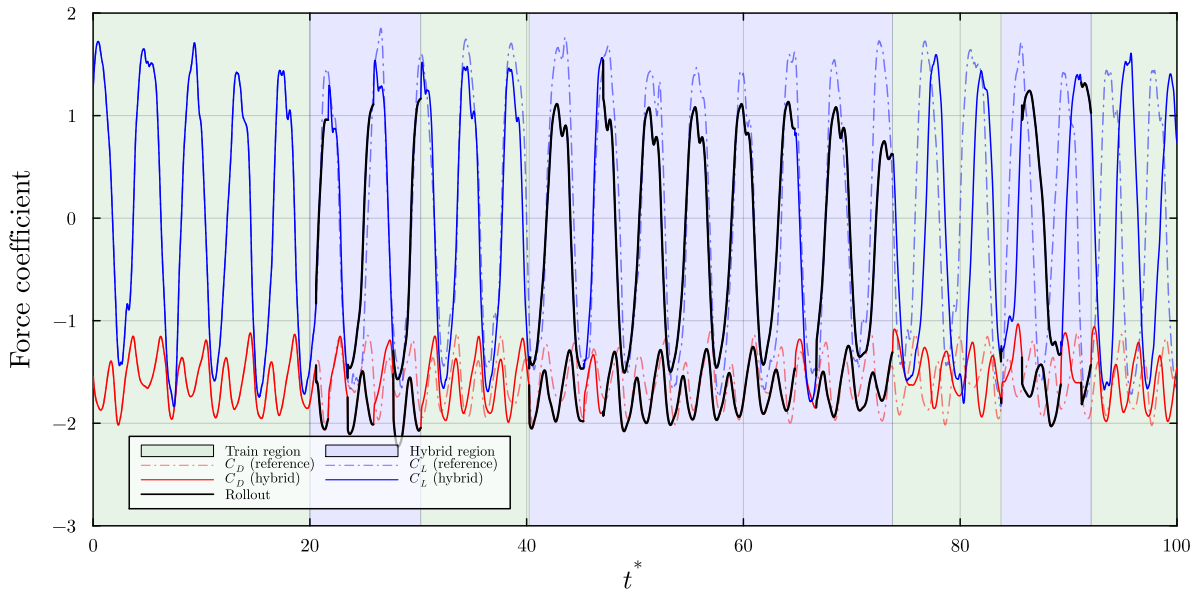



**Figure 4.2:** Drag and lift coefficients over time for the reference (dash–dot) and hybrid (solid) simulations, with NODE rollout predictions overlaid (black). Green bands mark pure-CFD intervals (initial data-gathering and retraining); blue bands mark the hybrid windows in which latent rollouts are attempted. Retraining is triggered at $t^* \approx 30.2$, $t^*$, $t^* \approx 73.76$ and $t^* \approx 92.07$

Reading Figure 4.2 chronologically, following the time axis as in the serial-integrated workflow diagram Figure 3.19, the 4 stages of the acceleration workflow are also evident in Figure 4.2. In the initial data-gathering phase, the hybrid and reference simulations start from the same initial state at $t^* = 0$.

From there, both simulations are advanced in time using WaterLily, and instantaneous flow snapshots are saved to disk as training data for $\mathcal{P}$. Having reached $t^*_{\text{train}}$, the components of $\mathcal{P}$ can be trained; exact loss dynamics of the ML components are reported in Section 4.2.6 and 4.2.7. After the training procedures, the hybrid loop (algorithm 3) is engaged at $t^*_{\text{handoff}} = 20$ and the first rollouts are attempted.

The black lines in Figure 4.2 trace the pressure force coefficients on the cylinder wall that $\mathcal{P}$ has rolled out and injected back into the simulation. The rollouts track the reference $C_L$ closely at first, but a small phase and amplitude error accumulates as each latent trajectory is integrated further from its initial point; this is the drift that the KNN monitor is designed to catch, and each rollout is cut off just before drifting outside of the known latent distribution range. An example of this drift is visible at the end of the second hybrid window (around $t^* \approx 74$), the hybrid $C_L$ trace has accumulated a visible phase offset relative to the reference, the two signals are no longer in step, and there is a significant amplitude mismatch. Because of the long rollout and the chaotic nature of the wake, a certain amount of divergence is expected and is in itself not an error; however, it places $\mathcal{S}$ on a different point of the shedding cycle than the reference, so the two simulations are no longer pointwise comparable and only their statistical observables remain meaningful.

To demonstrate that the hybrid simulation accurately reflects the characteristic shedding frequency, the shedding period $T^*$ is measured for each cycle. The period is read directly off the $C_L$ trace of Figure 4.2 as the time between successive maxima, and Figure 4.3 shows the resulting distributions over the complete simulation window. The mean shedding period for the reference and the hybrid simulations is very closely aligned.

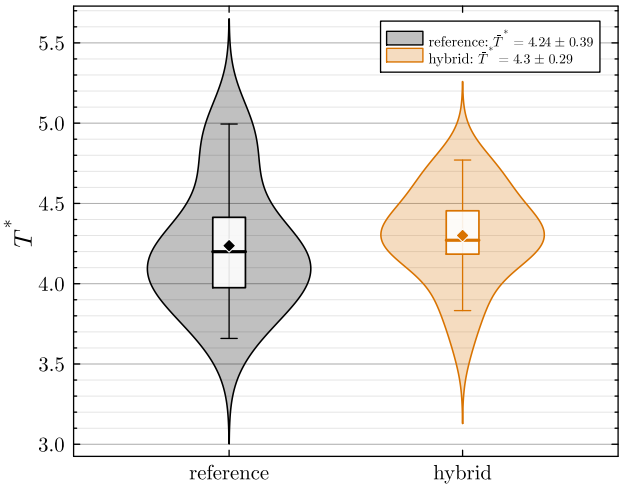



**Figure 4.3:** Distribution of the shedding period $T^*$ across the full simulation window for $\mathcal{S}_{\text{ref}}$ and $\mathcal{S}$, read from successive $C_L$ maxima. Diamonds mark the means; the two distributions overlap within the cycle-to-cycle variance.

| | $\bar{T}^*$ | $St$ | $\sigma_{St}$ |
|---|---|---|---|
| Reference | 4.237 | 0.236 | 0.0045 |
| Hybrid | 4.301 | 0.233 | 0.0032 |

**Table 4.6:** Mean shedding period $\bar{T}^*$ and Strouhal number $St = 1/\bar{T}^*$ for the reference and hybrid simulations, measured peak-to-peak from the $C_L$ trace. $\sigma_{St}$ is the standard error on the mean.

The mean periods are $\bar{T}_{\text{ref}} = 4.24$ and $\bar{T}_{\text{hyb}} = 4.30$, giving $St_{\text{ref}} = 0.236$ and $St_{\text{hyb}} = 0.233$, a difference of about $1.5\%$ that sits well within the cycle-to-cycle spread of either run. The corresponding Strouhal numbers are shown in Figure 4.6. The difference between the reference and hybrid, $\Delta St = 0.0035$, is smaller than the combined standard error on the two means, $\sqrt{\sigma^2_{St,\text{ref}} + \sigma^2_{St,\text{hyb}}} \approx 0.0055$, showing how there is no meaningful difference between the shedding frequencies of the two simulations.

The violin plots also show that the hybrid has a smaller variance in the shedding period than the reference. This same behaviour is visible in the second hybrid region of Figure 4.2, where each rolled-out shedding cycle closely matches the others in both amplitude and period, indicating that $\mathcal{P}$ has learned a sort of mean shedding behaviour. The hybrid therefore sheds at the same Strouhal number as the reference, which is consistent with the leading latent components of $f_\phi$ oscillating at the shedding frequency (Section 4.2.7); $\mathcal{P}$ has learned the dominant shedding rhythm correctly rather than a shifted one.

But, as mentioned before, around $t^* \approx 74$ the hybrid $C_L$ trace has accumulated a visible phase offset relative to the reference. Figure 4.3 shows that the shedding period of $\mathcal{S}$ is very closely aligned to the reference, but it doesn't display where this offset kicks in and what causes it. To view where $\mathcal{S}$ begins to drift, of course, the instantaneous phase of each lift signal is tracked over time using an instantaneous Hilbert transform. The Hilbert transform turns the real, oscillating $C_L$ into a complex signal whose angle is a continuously growing phase $\varphi(t)$; the difference $\Delta\varphi = \varphi_{\text{hyb}} - \varphi_{\text{ref}}$, expressed in cycles, can then be assessed to view how much, and where the simulations begin to drift away from each other.

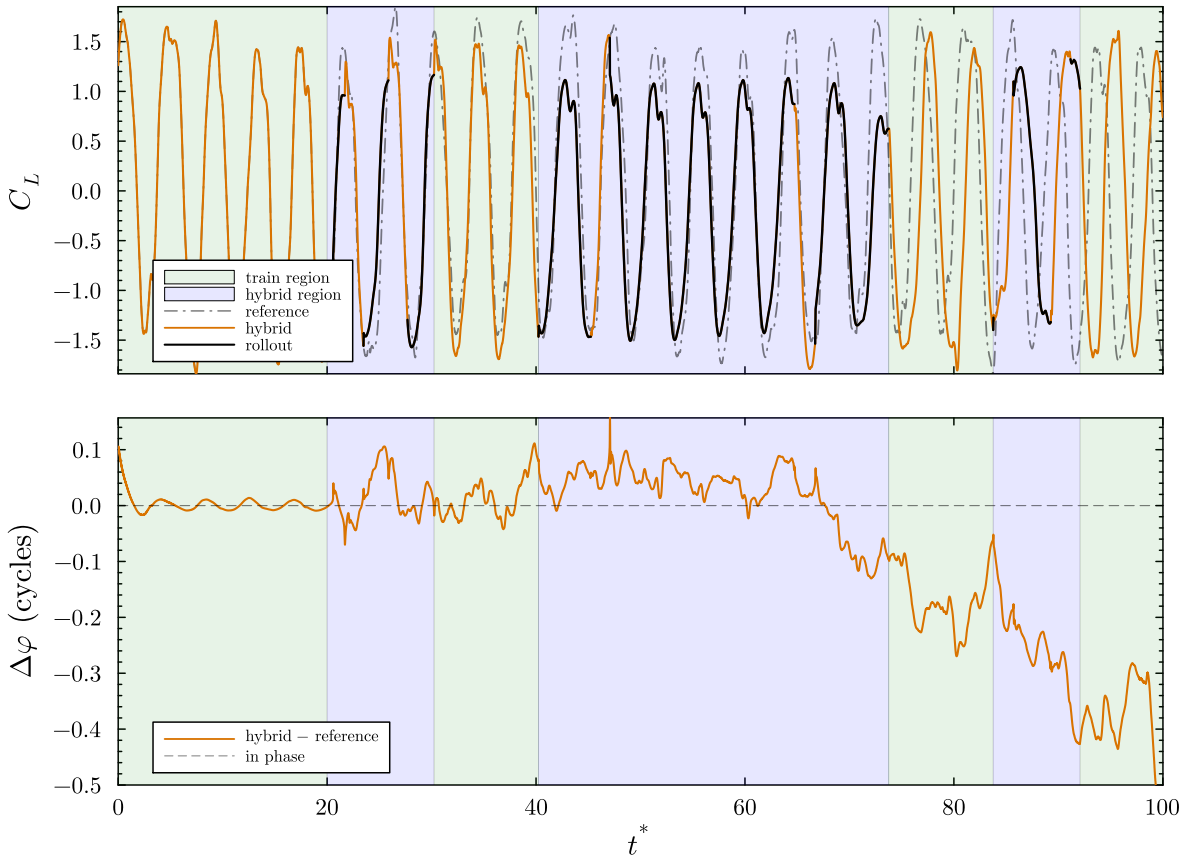



**Figure 4.4:** Running phase difference $\Delta\varphi = \varphi_{\text{hyb}} - \varphi_{\text{ref}}$ of the two lift traces, in cycles. Phase-locked through both hybrid windows, drifting to about $-0.4$ cycle after the final rollout of the second window.

Figure 4.4 shows that the two simulations stay decently in phase, within roughly $\pm 0.1$ of a cycle, throughout the first and second rollout regions. Towards the end of the second hybrid region, however, the dissipative behaviour of $\mathcal{P}$ erodes the amplitude of the rolled-out cycles (rollouts 4–6) and $\mathcal{S}$ no longer tracks the reference, so $\Delta\varphi$ starts to drift away from zero. During the second retrain, the phase difference remains roughly constant, but once the last hybrid region begins, it is pushed further, reaching about $-0.4$ cycles by the end of the run. This renewed drift indicates that the freshly harvested snapshots do not yet represent the current state of $\mathcal{S}$ well enough; $f_\phi$ and $\mathcal{K}$ remain weakly fit to this regime, so each rollout nudges the simulation further off course rather than back towards the reference.

The per-rollout behaviour of the hybrid loop is summarised in Table 4.7, listing the number of steps each latent rollout advanced the simulation before the KNN monitor returned control, along with the physical time gained and the score at hand-off.

Table 4.7 shows interesting patterns. Firstly, every completed rollout is cut off the moment its KNN score crosses the threshold (rollout 1 at $0.328$ against $0.326$, for example), verifying that the adaptive scheme of algorithm 2 stops each trajectory at the edge of the known latent distribution.

Secondly, the length of each rollout varies over the hybrid regions. This variation in rollout length

| # | NODE model | Steps | $\Delta t^*$ | $t^*_{\text{r, end}}$ | KNN score | Threshold |
|---|---|---|---|---|---|---|
| 1 | initial | 101 | 1.105 | 21.687 | 0.328 | 0.326 |
| 2 | initial | 217 | 2.373 | 25.869 | 0.329 | 0.326 |
| 3 | initial | 231 | 2.526 | 30.233 | 0.331 | 0.326 |
| 4 | 1st retrain | 457 | 4.998 | 45.247 | 0.384 | 0.383 |
| 5 | 1st retrain | 1615 | 17.664 | 64.740 | 0.387 | 0.383 |
| 6 | 1st retrain | 642 | 7.022 | 73.757 | 0.384 | 0.383 |
| 7 | 2nd retrain | 1 | 0.011 | 83.789 | 0.473 | 0.413 |
| 8 | 2nd retrain | 326 | 3.566 | 89.284 | 0.417 | 0.413 |
| 9 | 2nd retrain | 82 | 0.897 | 92.075 | 0.414 | 0.413 |

**Table 4.7:** Per-rollout summary of the KNN-gated NODE inference. Rollouts 1–3 use the initial NODE; rollouts 4–6 use the retrained NODE (raised KNN threshold). Rollouts 4 and 6 were rejected at the encoder gate after a single step.

reflects how well the current state of $\mathcal{S}$ is represented in the encoded training trajectories (produced by the trained encoder $\varphi_\theta$) on which $f_\phi$ and the KNN monitor $\mathcal{K}$ have been trained and fit. Since both components draw their logic from the same set of training trajectories $\boldsymbol{\mathcal{T}}_z$, $f_\phi$ can only have learned the dynamics that those trajectories contain, and $\mathcal{K}$ scores any new state by its distance to those same points. If the current state of $\mathcal{S}$ is in a regime that has not been represented by a majority of the states in $\boldsymbol{\mathcal{T}}_z$, it can either serve as a bad initial state for the NODE integration, or it can be in a regime that is not covered by the latent dynamics $f_\phi$. Either possibility will trigger a quick flag from the KNN monitor, resulting in a shorter rollout length.

The rollout length is therefore a direct reflection of the overlap between the active flow regime and the distribution of latent states in the training trajectories. This is visible in both Figure 4.2 and Table 4.7, where in the first hybrid region, the neural ODE produces rollouts of mostly similar lengths, showing that it has learned and fit the dynamics of the initial data-gathering CFD steps to a certain confidence, but to produce longer rollouts, more data is needed. In rollouts $4-6$, $f_\phi$ has been updated to produce longer rollouts that stay inside the regime that can be represented by $\boldsymbol{\mathcal{T}}_u$. These longer rollouts also show a gradual decay in oscillation amplitude over their length, indicating that $\mathcal{P}$ has learned a slightly dissipative version of the shedding cycle.

This dissipative behaviour also has a downstream effect. Each of the rollouts $4$–$6$ inject a slightly lower-amplitude field into $\mathcal{S}$ through the coupling routine of Section 3.7.3; this compounded error accumulates and pushes the simulation into a state that $\boldsymbol{\mathcal{T}}_z$ no longer represents. The subsequent retrain phase attempts to close this gap by augmenting new CFD snapshots to the existing training data $\boldsymbol{\mathcal{T}}_u^{(r+1)}$ as a minority of the snapshots. This means that the bulk of the multiple-shooting loss remains dominated by the older, well-represented dynamics, and the converged weights $\phi^{(r+1)}$ fit the new state only weakly. As a result, both $f_\phi$ and $\mathcal{K}$ remain weakly fit to this regime, so the encoded initial state is a poor starting point for the integration and an immediate KNN trigger. That is why 2 out of 3 rollouts in the last hybrid section are cut short almost as soon as they begin (Figure 4.2), after which the framework resumes data collection to update the network weights.

The effect of this dissipative behaviour of $\mathcal{P}$ is reflected in Table 4.8, where the mean $C_D$ gets conserved compared to the reference simulation, whereas the RMS value of the $C_L$ does appear to show the effects of the dissipative behaviour at the end of the second rollout.

The mean drag is recovered to within $e_{\text{rel}} \approx 0.17\%$ in both the full domain and the hybrid regions, since $\langle C_D \rangle$ depends only on the first moment of the signal and carries no information about the oscillation amplitude. Where $C_L^{\text{rms}}$, being peak-weighted, is sensitive to amplitude mismatches and shows a lower value for the hybrid simulation during both the whole simulation window, as in the hybrid sections. Over the full computational window, the relative error is $3.48\%$, but when restricting the averaging window to the hybrid regions, it rises to $11.36\%$. The full-domain figure is diluted by the pure-CFD intervals, which track the reference and contribute their undamped lift-coefficient amplitudes to the global RMS. The hybrid-region $C_L^{\text{rms}}$ error is therefore the more faithful measure of error, confirming that the amplitude decay in rollouts $4$–$6$ translates into a measurable loss of lift fluctuation.

| Region | Quantity | Solution | | Error | |
|---|---|---|---|---|---|
| | | Reference | Hybrid | $e$ | $e_{\mathrm{rel}}$ [%] |
| Full domain | $\langle C_D \rangle$ | $-1.588$ | $-1.586$ | $0.003$ | $0.168$ |
| | $C_L^{\mathrm{rms}}$ | $1.185$ | $1.144$ | $0.041$ | $3.479$ |
| Hybrid regions | $\langle C_D \rangle$ | $-1.582$ | $-1.582$ | $0.003$ | $0.169$ |
| | $C_L^{\mathrm{rms}}$ | $1.184$ | $1.050$ | $0.135$ | $11.361$ |

**Table 4.8:** Time-mean drag, and RMS lift for the reference and hybrid simulations, with absolute error $e$ and relative error $e_{\mathrm{rel}}$. The full-domain row averages over the whole window $0 \mapsto t^*_{\mathrm{end}}$; the hybrid-region row restricts the averaging to the intervals in which latent rollouts were active.

### 4.2.4. Mean Flow and Reynolds-Stress Statistics

The force coefficients of Section 4.2.3 certify that the hybrid loop reproduces the *integrated* loading on the body, but they collapse the entire flow onto two scalar time series and therefore say nothing about *where* in the domain the hybrid and reference solutions agree or diverge. Because of the chaotic nature of the cylinder wake at the current Reynolds number, a small perturbation can easily decorrelate hybrid and the reference trajectories. To assess the performance of the accelerated simulation, Figure 4.5 shows the time-averaged velocity components ($\langle u \rangle, \langle v \rangle$) of the hybrid simulation (right columns) compared to the reference (middle column) and their absolute difference (left column).

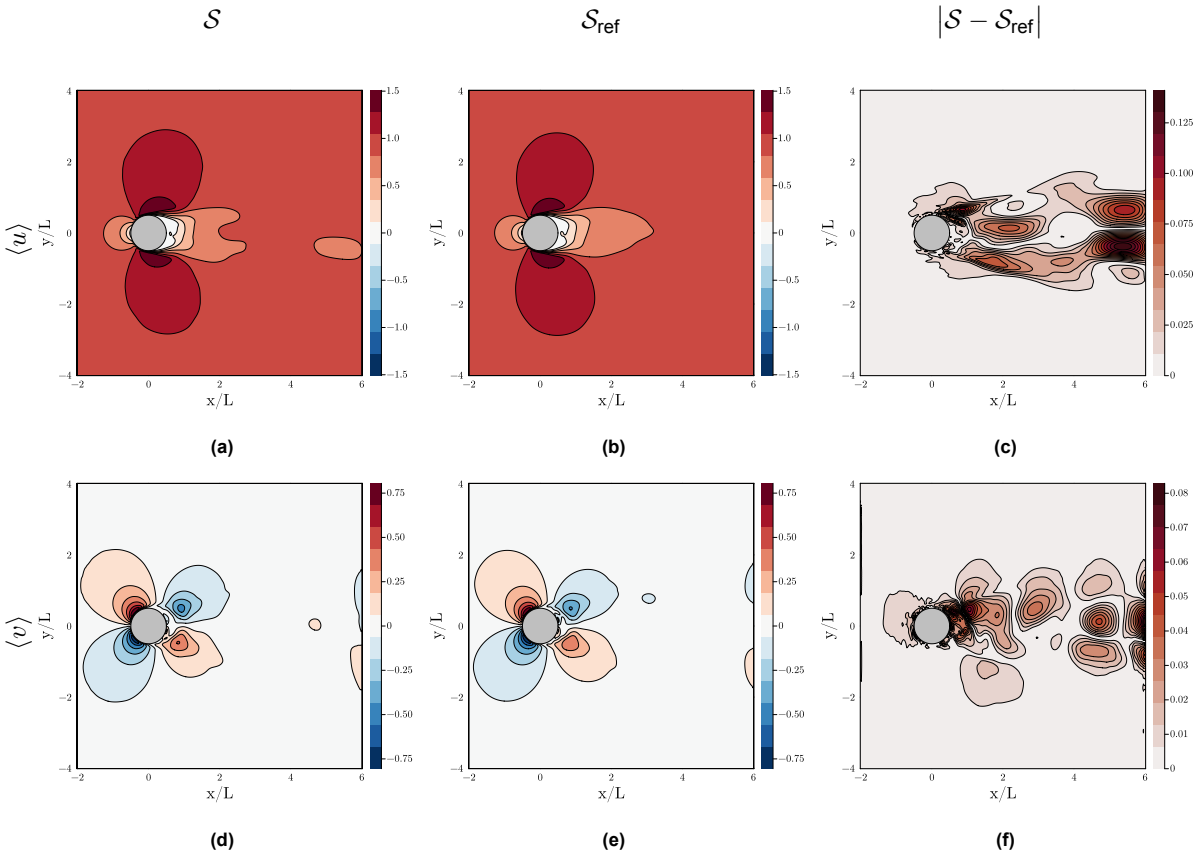



**Figure 4.5:** Mean velocity components: $\langle u \rangle$ (top), $\langle v \rangle$ (bottom). Columns: $\mathcal{S}$, $\mathcal{S}_{\mathrm{ref}}$, absolute difference. Lengths scaled by $L$, velocities by $U$.

The time-averaged velocity components are computed by sampling instantaneous flow snapshots of $\mathcal{S}$ and $\mathcal{S}_{\mathrm{ref}}$ at the exact same $t^*$ using a matching sampling interval. Even if the time-dependent properties of $\mathcal{S}$ are different to those of $\mathcal{S}_{\mathrm{ref}}$, the time-averaged flow fields should produce the same statistics.

Figure 4.5a and Figure 4.5b show that both $\langle u \rangle$ have converged in time, showing smooth contours and

gradients around the cylinder walls. The only visual difference is in the wake of the cylinder, where the hybrid simulation is less smooth than the reference. Figure 4.5d and Figure 4.5e also seem to show similar behaviour, where the contours of the mean spanwise flow of the reference are a bit less smooth compared to that of the reference simulation, indicating that the hybrid simulation lacks a small amount of averaging data.

The absolute-difference fields (Figure 4.5c & Figure 4.5f) show that the discrepancy between the two simulations is negligible over the body and the upstream region and is confined almost entirely to the wake, $x/L \gtrsim 1$. The error peaks at approximately $0.13\,U$ for $\langle u \rangle$ and $0.08\,U$ for $\langle v \rangle$ mostly in the positive spanwise area just behind the cylinder. This matches what could be seen in Figure 4.2, where the rollouts produced lift forces with a lesser amplitude in the positive region of the cylinder wall. This is exactly the region where the most difference between the mean flow components exists.

**Reynolds Stress Components**

Having computed the time-averaged velocity components, the time-averaged fluctuations of the flow can also be assessed. Using Equation 2.3, fluctuating velocity components are extracted and studied to assess the effect of the rollouts on the fluctuating field of $\mathcal{S}$. This allows the three independent components of the Reynolds-stress tensor (RST) to be compared against the reference. Where the normal stresses $\langle u'u' \rangle$ and $\langle v'v' \rangle$ display the energy of the streamwise and transverse fluctuations, and the shear stress $\langle u'v' \rangle$ indicates the momentum transport across the wake.

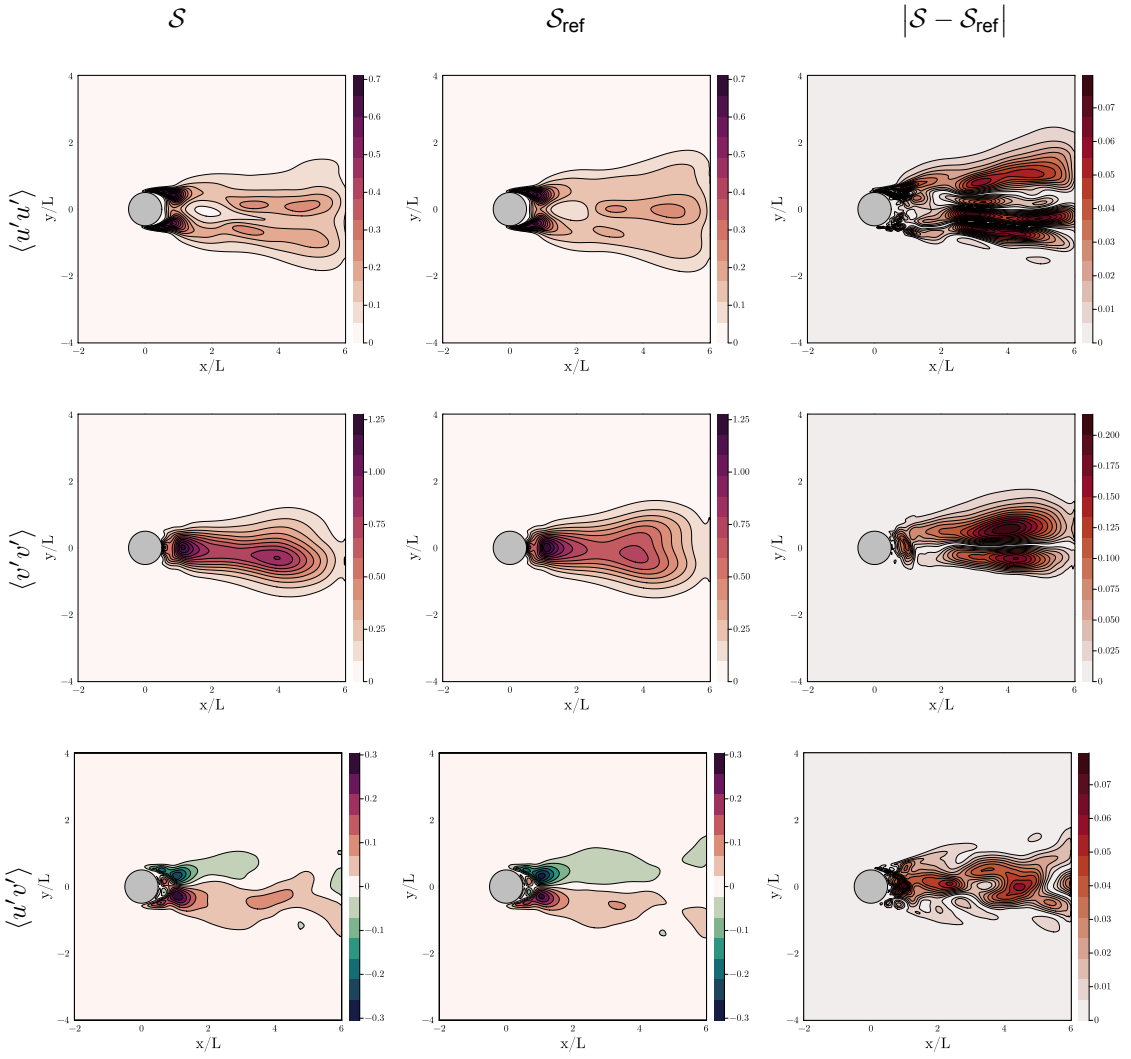



**Figure 4.6:** Reynolds-stress components, top to bottom: $\langle u'u' \rangle$, $\langle v'v' \rangle$, $\langle u'v' \rangle$. Columns: hybrid $\mathcal{S}$, reference $\mathcal{S}_{\text{ref}}$, and absolute difference $\left|\mathcal{S} - \mathcal{S}_{\text{ref}}\right|$. Lengths scaled by $L$, velocities by $U$.

Just as the RST components reported in the work by Parnaudeau et al. [63], the hybrid and the reference simulation both produce the expected wake structure: a twin-lobed $\langle u'u' \rangle$ from the separating shear layers, a centreline-peaked $\langle v'v' \rangle$ in the region $1$–$2\,L$ downstream with the largest magnitude of the three, and an antisymmetric $\langle u'v' \rangle$ that vanishes on $y/L = 0$. The hybrid simulation closely recovers the normal stresses $\langle u'u' \rangle$ and $\langle v'v' \rangle$, matching both the peak locations and the lobe topology. The shear stress shows the biggest difference; the positive lobe is substantially weaker than in the reference, so the hybrid only partially preserves the antisymmetry over this simulation window.

The rightmost column of Figure 4.6 shows the discrepancies, especially in the wake $x/L \gtrsim 1$, with peak errors of around $0.08$, $0.2$ and $0.08$ ($\sim 11,\ 16\ \&\ 26\%$ of the reference peaks); $\langle u'v' \rangle$ carries the largest relative error, as was also visible due to the fact that the positive half of its "butterfly" shape missing compared to the nicely symmetric contour plot of the reference shear component. Crucially, this error sits in the same near-body wake region as the mean-flow discrepancy (Figure 4.5c) and the rollout $4$–$6$ lift decay (Section 4.2.3), which means that forces, mean flow, and Reynolds stresses indicate the same error for this hybrid case. As these figures have shown, however, the bias of $\mathcal{P}$ toward producing lower-amplitude fluctuations is an error in magnitude rather than a corruption of the dynamics.

| | Mean flow | | Reynolds stresses | | |
|---|---|---|---|---|---|
| | $\langle u \rangle$ | $\langle v \rangle$ | $\langle u'u' \rangle$ | $\langle v'v' \rangle$ | $\langle u'v' \rangle$ |
| $\varepsilon_q$ | 0.009 | 0.005 | 0.006 | 0.014 | 0.004 |
| $\rho_q$ | 0.9932 | 0.9928 | 0.9789 | 0.9791 | 0.9275 |

**Table 4.9:** Quantitative comparison between the hybrid and reference fields: mean absolute error (MAE) and spatial Pearson correlation coefficient, computed over the wake region.

While the contour plots in Figure 4.6 convey the spatial agreement qualitatively, Table 4.9 quantifies it for both the mean flow and the three RST components over the wake region. The mean flow is recovered almost exactly, with $\rho_q \approx 0.993$ for both $\langle u \rangle$ and $\langle v \rangle$. The normal stresses follow closely behind at $\rho_q \approx 0.979$, confirming that the hybrid field reproduces the wake topology correctly, just as was seen in the contours. The shear stress is the weakest component, dropping to $\rho_{u'v'} = 0.9275$; this lower value can be explained by the partially missing positive lobe; its small $\varepsilon_q = 0.004$ reflects the low magnitude of $\langle u'v' \rangle$ rather than close agreement.

### 4.2.5. Spectral Fidelity of the Wake

In addition to the time-averaged statistics examined above, a physically faithful hybrid simulation must also reproduce the distribution of turbulent energy across temporal scales. This is assessed by sampling the transverse velocity $v$ at a fixed wake location $(x/L, y/L) = (2, 0.5)$ in both $\mathcal{S}$ and $\mathcal{S}_{\text{ref}}$. To separate the error introduced by the latent dynamics from that introduced by the autoencoder, a third signal is added: every snapshot of $\mathcal{S}_{\text{ref}}$ is passed through the trained autoencoder, $\hat{\mathbf{u}} = \psi_\theta(\varphi_\theta(\mathbf{u}))$, and the transverse velocity is extracted at the same downstream location. This per-frame reconstruction, labelled *AE*, contains the reconstruction error of the decoder $\psi_\theta$ alone, with no contribution from the learned dynamics $f_\phi$, and therefore acts as a control that isolates the decoder's spectral signature.

The same three signals are shown in the time domain in Figure 4.7, together with the neural ODE rollout. The AE reconstruction tracks the reference closely but is slightly dissipative, smoothing the sharpest peaks and troughs of the velocity signal. The hybrid velocity signal, similar to the force coefficients in Figure 4.2, reproduces the shedding cycle but dampens its high-frequency oscillations relative to the reference and, in places, relaxes to a mean velocity signal rather than the full fluctuations, most visibly in the second hybrid region.

These signals are compared in the frequency domain via their one-sided power spectra $\mathrm{PS}(v)$, estimated using Welch's method with five segments of the simulation window. Each segment, therefore, spans approximately five shedding periods. The resulting spectra are shown in Figure 4.8 against the non-dimensional frequency $fL/U$.

In the low-frequency part of the spectrum, all three signals produce the same frequencies. The primary shedding frequency is clearly present at $fL/U \approx 0.2$ in every case, as was already visible in the

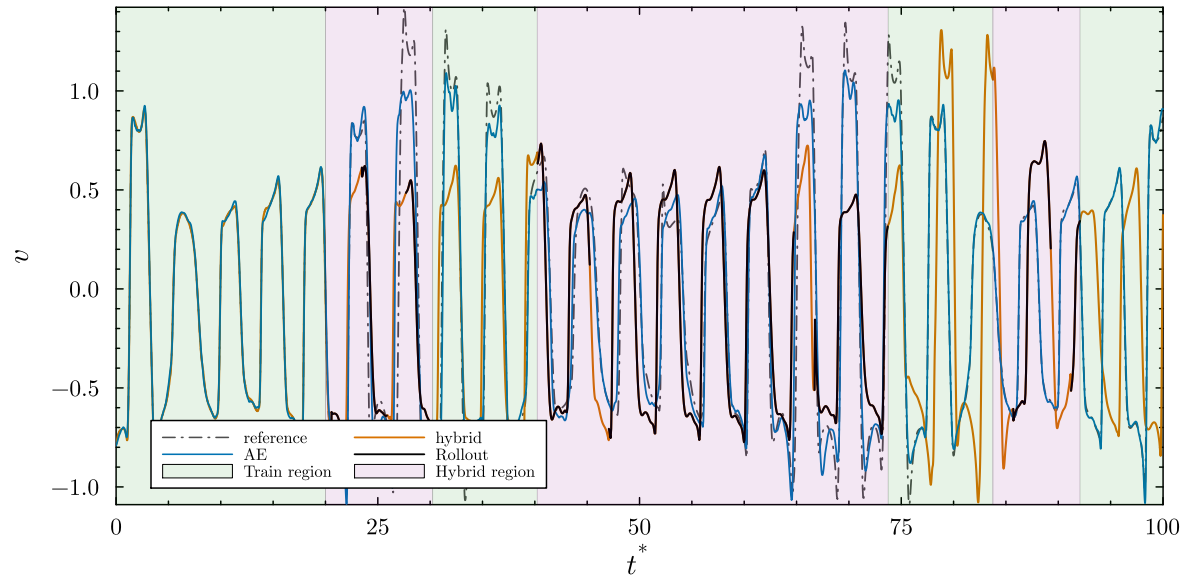



**Figure 4.7:** Transverse wake velocity $v$ at the probe $(x/L, y/L) = (2, 0.5)$ against non-dimensional time $t^*$, for the reference, the per-frame AE reconstruction, the hybrid simulation. Shaded bands mark the training and hybrid windows.

force coefficient plot in Figure 4.2; the higher harmonics of the shedding frequency, associated with the secondary wake structures displayed in Figure 3.3, are also captured by both the hybrid simulation and the AE reconstruction. The hybrid, AE and reference curves remain closely aligned through the middle frequency range, until they separate near $fL/U \approx 2$–$3$. No genuine Kolmogorov $f^{-5/3}$ inertial subrange is expected in a two-dimensional wake at $Re = 2500$; in two dimensions the forward (small-scale) cascade follows the steeper $k^{-3}$ enstrophy scaling rather than $k^{-5/3}$ [9], which under Taylor's frozen-turbulence hypothesis maps to the $f^{-3}$ reference slope shown in Figure 4.8.

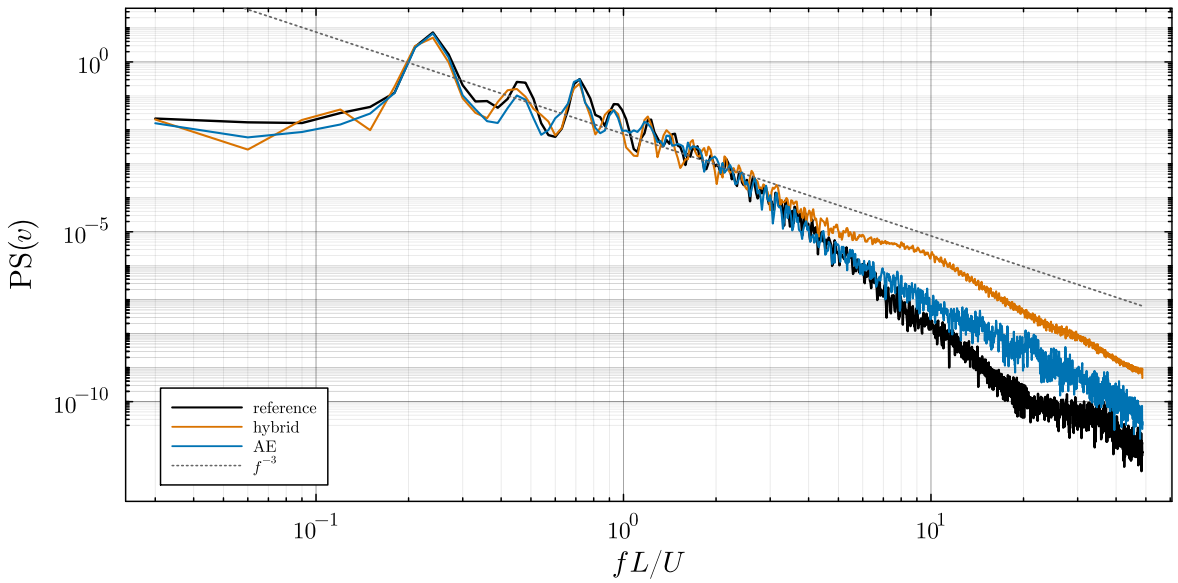



**Figure 4.8:** Power spectrum of the transverse wake velocity $\mathrm{PS}(v)$ for the reference $\mathcal{S}_{\text{ref}}$, the hybrid $\mathcal{S}$, and the per-frame autoencoder reconstruction of $\mathcal{S}_{\text{ref}}$ (AE), against the non-dimensional frequency $fL/U$. The dotted line marks an $f^{-3}$ slope, shown only as a visual reference for the 2D mid-band decay rate.

Beyond the crossover near $fL/U \approx 2$–$4$, the three spectra spread out into a clear ordering: the reference falls steeply to $\mathcal{O}(10^{-10})$, the AE reconstruction plateaus roughly two to three decades above it, and the hybrid plateaus a further decade above the AE. This ordering shows how the errors are stacked

in $\mathcal{P}$, with the AE spectrum being the lower bound for error for the hybrid simulation, as all snapshots produced by $\mathcal{P}$ must pass through it. It lies above the reference spectrum because the reconstruction error of $\psi_\theta$ adds a high-wavenumber texture from the bilinear-upsampling decoder, which only gets visible when the true spectrum $\mathrm{PS}(v)$ has fallen below it.

When looking at the spectrum of the hybrid simulation, it sits elevated above the AE spectrum, indicating that the higher-frequency energy cannot be a decoder effect, since both fields are reconstructed using the same $\psi_\theta$; instead, it is a property of the neural ODE rollouts and their coupling to WaterLily. Two mechanisms contribute to this added error: first, the AE reconstructs instantaneous latents encoded from true reference fields, whereas the hybrid decodes latent vectors integrated by $f_\phi$, which exhibit a degree of drift relative to the ground truth. Because the decoder maps a $16$-dimensional latent onto a $256 \times 256$ field, it can be locally sensitive, so that a small drift $\delta\mathbf{z}$ in the latent state is amplified into a comparatively large perturbation in physical space. Secondly, the hybrid signal is a concatenation of latent-rollout and WaterLily segments spliced at every OOD fallback, so each re-injected field and the solver's relaxation of it can introduce a discontinuity in $v$, visible as a kink in the hybrid trace in Figure 4.7. Each such discontinuity adds energy at high frequencies, lifting the tail above the reference and the AE signal and flattening its decay during the $fL/U \approx 5$–$10$ frequency band; the tail still falls off at high $fL/U$ and follows the same slope as the reference.

Despite the elevated tail of the hybrid spectrum, the added high-frequency energy remains low in absolute terms. Since it lies more than 2 decades below the shedding frequency, it would contribute negligibly to the integrated signal energy. The dominant scales, namely the primary shedding frequency and its harmonics, are reproduced accurately by both the AE reconstruction and the hybrid simulation. The spectral discrepancy is therefore confined to a low-energy, high-frequency band where the true spectrum has already decayed, and does not compromise the physical fidelity of the resolved flow. The concentration of error in the high-frequency, low-energy range, where the true signal is weakest, is a recurring difficulty for learned models advanced autoregressively over long horizons [45].

### 4.2.6. Autoencoder performance

The autoencoder is the entry and exit point of every $\tilde{\mathcal{T}}_z$ because every initial state is first compressed by $\varphi_\theta$, and every field returned to $\mathcal{S}$ is reconstructed by $\psi_\theta$. Its round-trip error (3.11) therefore sets a lower bound on the maximal accuracy $\mathcal{P}$ can attain, independent of how well the latent dynamics can be predicted. Understanding how the autoencoder performs is therefore crucial for interpreting the baseline results of Section 4.2.3, since any error already present in the reconstruction cannot be attributed to $f_\phi$.

#### Training dynamics

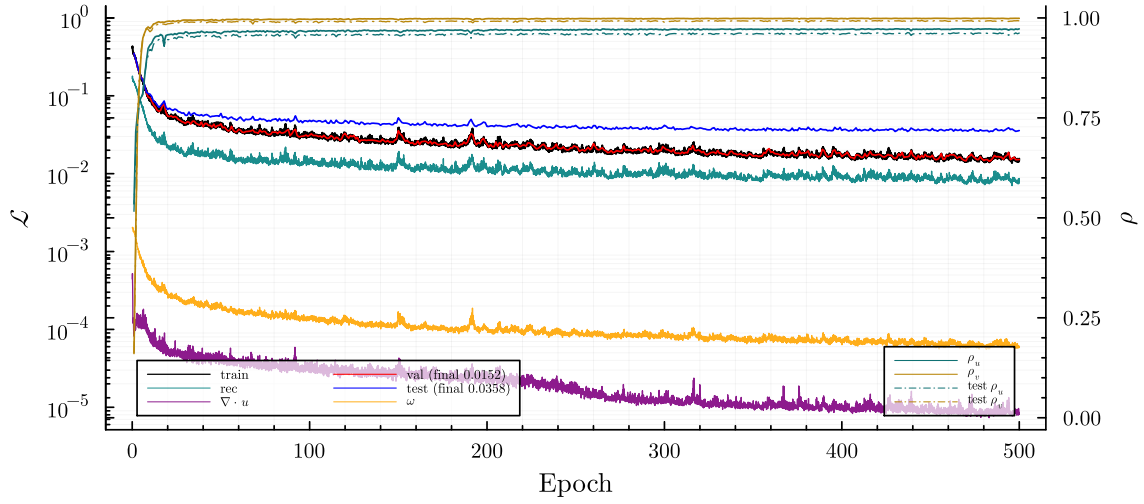



**Figure 4.9:** Initial training history of the baseline autoencoder. Left axis ($\mathcal{L}$, logarithmic): the total loss on the train, validation and test splits, together with the divergence ($\nabla \cdot \mathbf{u}$) and vorticity ($\omega$) penalty terms evaluated on the test split. Right axis ($\rho$): the reconstruction correlation coefficients $\rho_u$ and $\rho_v$ on train and test.

Figure 4.9 shows the loss history of the initial autoencoder training, using the parameters listed in Table 4.2, on the initial data-gathering trajectory sampled up to $t^*_{\text{train}} = 17.5$. The reconstruction loss on the train and validation splits converge to $\mathcal{L} \approx 1.5 \times 10^{-2}$, while the divergence and vorticity penalties settle near $10^{-5}$ and $10^{-4}$ respectively. The correlation coefficients of both velocity components also quickly approach unity, indicating that flow structure is learned rapidly. The test loss settles at $3.55 \times 10^{-2}$, which is about twice the validation loss value; this value is determined on the held-out window $t^* \in [17.5, 20]$. This gap indicates that the initial training window lacked sufficient information about the spatial dynamics of $\mathcal{S}$, suggesting that more data are required for generalisation across the entire simulation window, motivating the need to update the autoencoder weights $\theta$.

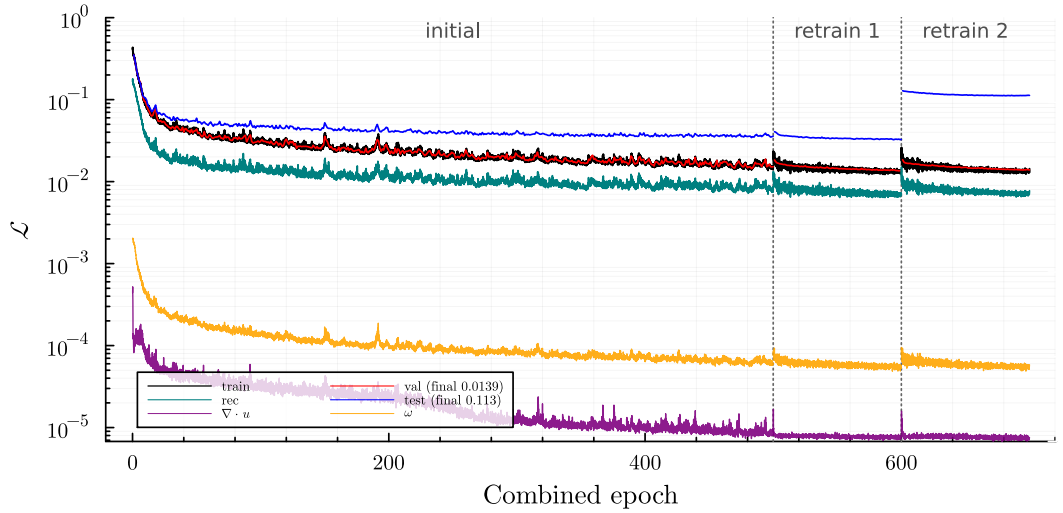



**Figure 4.10:** Combined training history of the baseline autoencoders. Left axis ($\mathcal{L}$, logarithmic): Showing how the cost function of the autoencoder varies over retraining runs, indicating the correlation of the training set to the current simulation state.

Figure 4.10 extends the loss over the 2 retraining stages, where each stage updates the weights for an additional $100$ epochs using a reduced learning rate (Section 3.7.4). The first learning rate leaves the losses mostly unchanged, but with additional data, it has learned additional dynamics, reflected in the ability to perform longer rollouts, as discussed in Section 4.2.3. The second retrain drives the test loss up by nearly an order of magnitude while train and validation hold steady: the accumulated dissipative drift (Section 4.2.3) has pushed $\mathcal{S}$ into a regime that is under-represented in the training data, which makes it so that the generalisation outside of the training regime drastically worsens. This same disparity in training data, and current state of $\mathcal{S}$, is also reflected in the immediately truncated rollouts $7$ and $9$ (Table 4.7).

**Instantaneous reconstruction performance**

The reconstruction of an instantaneous velocity field is displayed in Figure 4.11 for the two velocity components. These figures show that $\psi_\theta$ performs well in reconstructing the large scale structures of the flow, aligning with the conclusions from Section 4.2.5.

The instantaneous reconstructions are visibly smoother than the reference, most notably along the thin shear layers separating from the cylinder. These over-smoothing and the concentration of error in high-gradient regions are documented failure modes of autoencoders applied to turbulent flows, as they have shown to reproduce the large-scale coherent structures correctly but act as nonlinear filters that attenuate the small-scale, high-wavenumber content [48, 27, 34]. These smoother gradients, especially at the cylinder wall, are the direct explanation of the dissipative bias seen in the integrated forces, since the structures re-injected into $\mathcal{S}$ are systematically weaker than the reference.

This ”smoothing” of the reconstructed field is more apparent when computing the vorticity using the reconstructed velocity components. The instantaneous vorticity in Figure 4.12 shows how the effect of the more smeared out velocity gradients.

The reference shear layers and shed vortex cores are sharp and compact, whereas the reconstructed field is diffuse: the cores are spread over a wider area at reduced peak amplitude, and the thin braid of

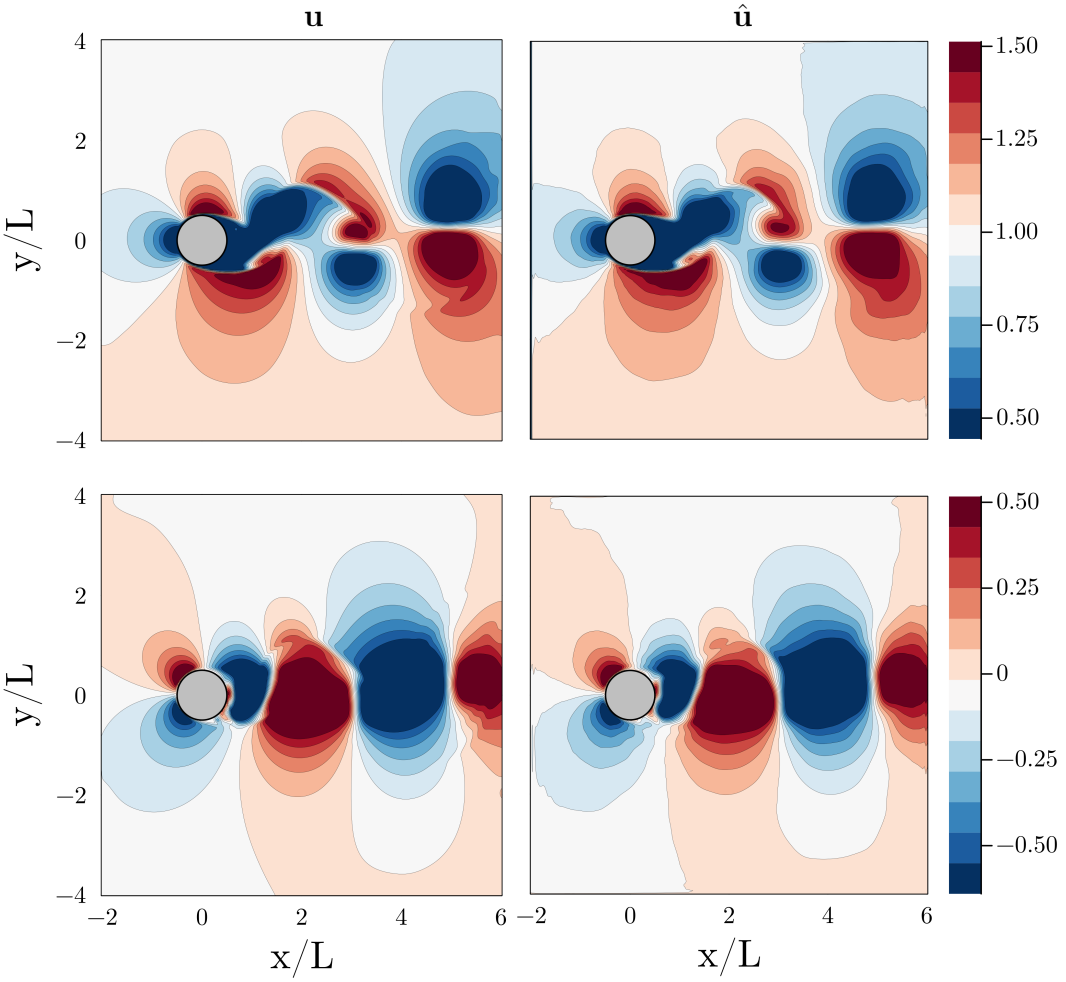



**Figure 4.11:** Instantaneous reconstruction of the velocity field by the baseline autoencoder at $t^* = 40.46$. Top row: streamwise component $u$; bottom row: transverse component $v$. Left column: WaterLily reference; right column: instantaneous autoencoder reconstruction $\hat{\mathbf{u}} = \psi_\theta(\varphi_\theta(\mathbf{u}))$

vorticity in the immediate wake is largely smeared out. Figure 4.12 also shows the addition of noisy artefacts in the near-wall region and close to the vortices, which explains the presence of the added noise observed in Section 4.2.5. A further band of spurious vorticity is also visible along the reconstructed domain boundary in Figure 4.12; this is not physical but a boundary artefact of the convolutional decoder, where the zero-padding biases the outermost cells and this small velocity deviation is amplified by the gradient operator used to compute $\omega = \hat{u}_{2,1} - \hat{u}_{1,2}$. This deviation generally stays confined to the outer edge of the domain and has no effect on the wake and near-body forces and metrics reported in this section

Overall, the autoencoder faithfully reproduces the large-scale structures in the wake and shedding cycle, with $\rho_u$, $\rho_v$ converging near unity, at the cost of attenuating the small-scale content revealed by the vorticity field. These characteristics set the pipeline's accuracy floor; investigating whether it can be lowered by varying the latent dimension, or the training procedures, is the subject of the parameter study in Section 4.3.1.

### 4.2.7. NODE performance

With the autoencoder's performance over the hybrid simulation characterised and the spatial accuracy floor thereby fixed, the learning behaviour of $f_\phi$ can now be traced through the simulation to understand the neural ODE's inference behaviour. The latent dynamics are followed from the initial multiple-shooting fit through to the retraining optimisations that the hybrid loop relies on, tracking how

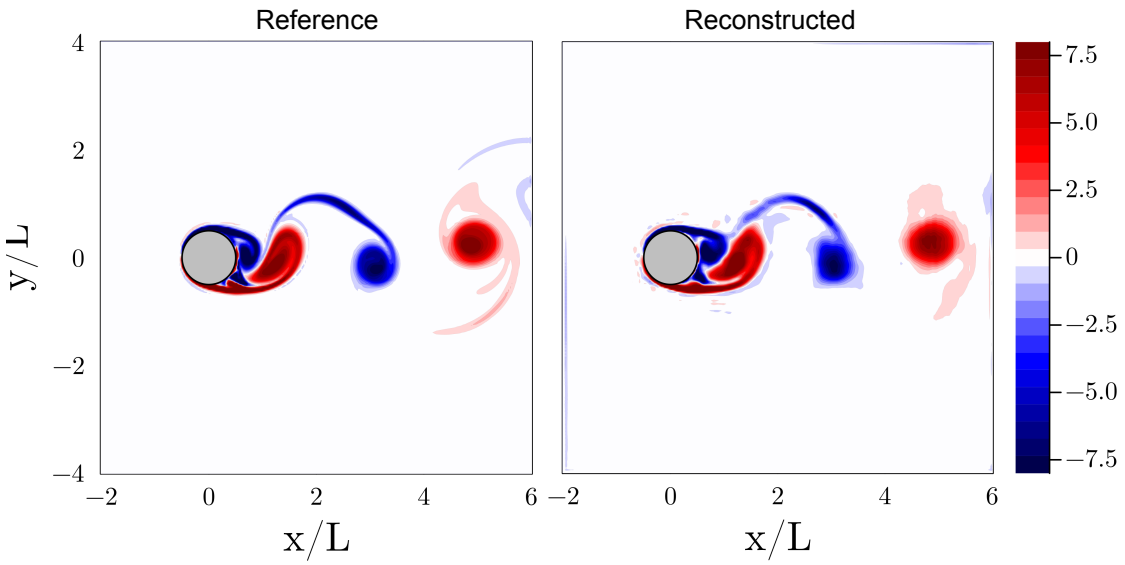
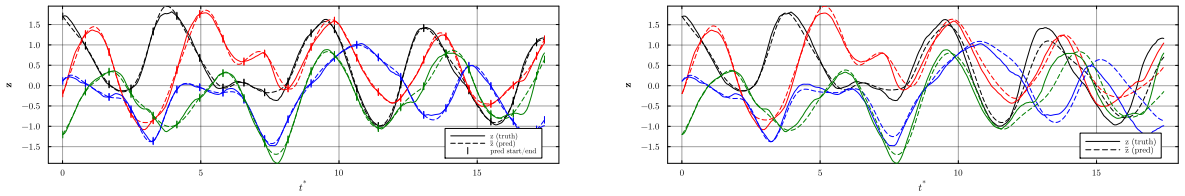



**Figure 4.12:** Instantaneous vorticity $\omega$ of the same snapshot as Figure 4.11. Left: WaterLily reference; right: autoencoder reconstruction. The reconstructed shear layers and vortex cores are diffused and exhibit reduced peak amplitudes, illustrating the autoencoder's low-pass behaviour.

their fidelity evolves as $f_\phi$ is retrained on the growing set of trajectories.

Figure 4.13a shows the initial multiple-shooting training. Within every segment, the prediction $\tilde{z}$ tracks the reference $z$ closely, and the shooting gaps at the segment boundaries have closed to the point of being barely visible. The fit is visibly converged, but it is also segment-wise: the loss (3.20) only ever penalises $f_\phi$ over a single span of $n_s$ samples at a time. The training scheme, therefore, biases $f_\phi$ towards short-horizon accuracy and reveals nothing about how the dynamics compound once the segment resets are removed.



**(a)** Converged multiple-shooting fit; markers denote the start and end of each shooting segment.

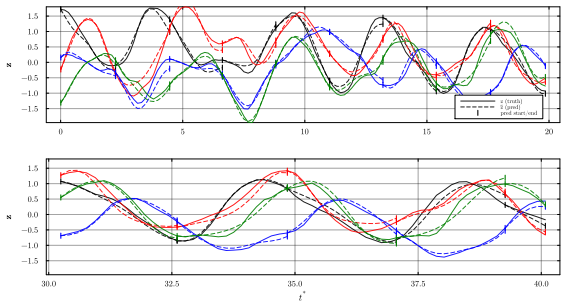
**(b)** Single rollout from $t^* = 0$ over the full training span, no segment resets.

**Figure 4.13:** Latent dynamics of the baseline $f_\phi$ over the initial training span, four representative components shown. Reference $z$ (solid) against prediction $\tilde{z}$ (dashed). (a) Converged multiple-shooting fit, with segment start/end markers; (b) unaided single rollout integrated from $t^* = 0$ over the same window.

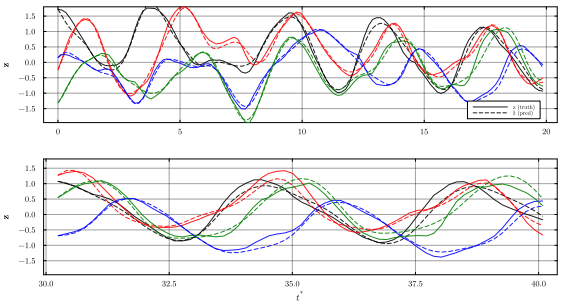
The compounding error over longer rollouts is exposed in Figure 4.13b, where $f_\phi$ is integrated from $t^* = 0$ over the whole span. The dominant shedding oscillation is recovered correctly: phase and amplitude of the leading components track the reference through the first several periods, confirming that $f_\phi$ has learned the principal latent dynamics rather than merely interpolating between segment endpoints. Towards the end of the window, however, a clear mismatch is evident, with both a phase and an amplitude deficit relative to $z$. The long rollout thus sustains the correct dynamics over a limited horizon before drifting, which gets controlled by the KNN monitor.

After this first fit has concluded, the hybrid loop is ready to begin rolling out and accelerating the simulation; once the monitor has raised $n_{\text{max, flags}}$ flags, retraining is triggered, and $f_\phi$ is refitted on the augmented trajectory set $\mathcal{T}_z^{(r+1)}$.

The converged multiple-shooting fit of the augmented set, Figure 4.14a, is visibly looser than the initial



**(a)** Converged multiple-shooting fit on the augmented trajectory set; markers denote shooting-segment boundaries.



**(b)** Single rollout from $t^* = 0$ across both windows after the first retraining, no segment resets.

**Figure 4.14:** Latent dynamics of $f_\phi$ after the first retraining, four representative components shown over two time windows ($t^* \in [0, 20]$, top; $t^* \in [30, 40]$, bottom). Reference $z$ (solid) against prediction $\tilde{z}$ (dashed). (a) Converged multiple-shooting fit on the augmented trajectory set, with segment start/end markers; (b) unaided single rollout from $t^* = 0$ over the same windows.

fit in Figure 4.13a: the shooting segments no longer overlap cleanly at their boundaries, and small discontinuities now appear at the group ends. This follows from the retraining scheme and is similar to the behaviour shown in Figure 4.10. The newly harvested trajectory is but a minority of the augmented set (3.21), so the averaged loss (3.20) stays dominated by the originally represented dynamics, and the shortened retraining schedule leaves not enough room for the shooting gaps to be pushed to zero.

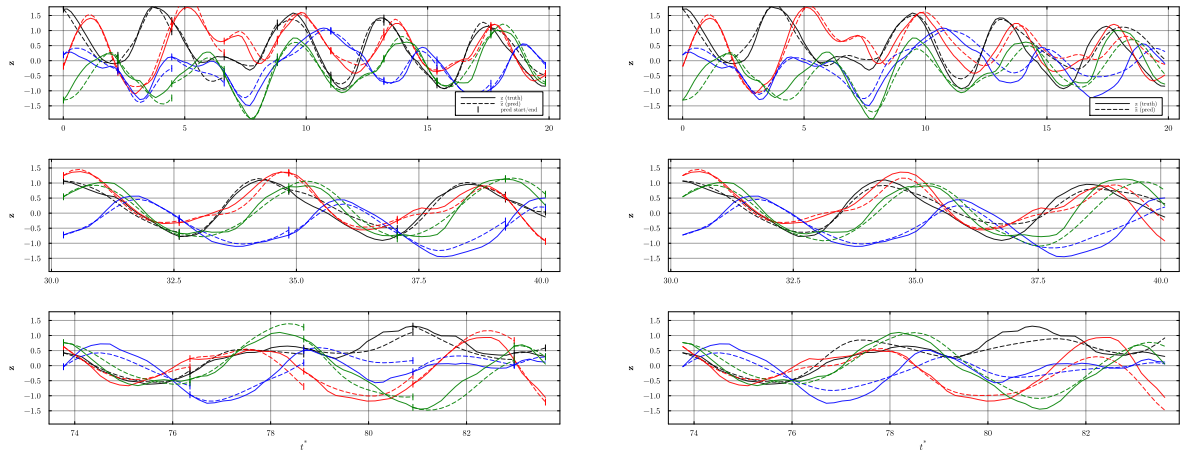
The single rollout of Figure 4.14b, however, tracks the reference markedly better than the looser segment fit would suggest: integrated from $t^* = 0$, $f_\phi$ holds phase and amplitude on the dominant components across both windows, with the mismatch only emerging towards the end of the span. Thus, the looser segment fit does not automatically translate into worse rollouts; the retrained $f_\phi$ has still absorbed the dominant latent dynamics of the extended horizon, which determine the rollout and the achievable acceleration.

After the second retraining, the trajectory set spans three disjoint chunks, shown in the bottom panels of Figure 4.15. The first two chunks remain well-fitted in both segments and the rollout, but the third is remarkably worse: its shooting segments leave visible gaps, and the rollout decorrelates early. Similar to the second retraining of the autoencoder, this is caused by the under-representation of the current flow state in the set, the chunk was harvested from the dissipative regime that rollouts 4–6 drove $\mathcal{S}$ into, and enters (3.21) as a minority, so the averaged loss (3.20) stays dominated by the older dynamics and $f_\phi$ fits it only weakly, the same effect behind the immediate KNN trigger of rollout 7.

The behaviour of $f_\phi$ across the hybrid simulation shows that the temporal error can have two origins. The first is, as mentioned in Section 3.6.1: even a well-converged fit sustains the latent dynamics only over a limited horizon before the rollout drifts from the reference, partly through the autoregressive nature of the neural ODE, and partly through the chaotic divergence of the wake itself, which is why the rollout must be gated rather than trusted indefinitely. The second matches the conclusions from Section 4.2.6: the fidelity of each retrained $f_\phi$ is set by how well the active flow regime is represented in $\mathcal{T}_z$, so the well-covered early trajectories are reproduced accurately while the under-represented later trajectories are fitted only weakly. The retraining procedure effectively transfers the dominant dynamics when the new data follow the existing distribution, but degrades when they do not. The influence of a relevant governing hyperparameter, the group size $n_s$ on the rollout fidelity is examined next in Section 4.3.2.

## 4.3. Effect of Pipeline Parameters

Having discussed the functioning of the hybrid simulation and its physical effects in detail in Section 4.2, this section aims to highlight the influence of specific parameters of each component of $\mathcal{P}$ ($\varphi_\theta$, $\psi_\theta$, $f_\phi$, $\mathcal{K}$). Certain parameters are isolated to quantify their effects, with two aims: to justify the baseline values of Table 4.4 rather than presenting them arbitrarily, and to expose the trade-offs that lie within the developed framework. The study proceeds component by component, following the order of chapter 3. For the autoencoder, the latent dimension $N_z$ and the initial number of autoencoder



**(a)** Converged multiple-shooting fit on the twice-augmented trajectory set; markers denote shooting-segment boundaries.

**(b)** Single rollout from $t^* = 0$ across all windows after the second retraining, no segment resets.

**Figure 4.15:** Latent dynamics of $f_\phi$ after the second retraining, four representative components shown over three time windows ($t^* \in [0, 20]$, top; $t^* \in [30.24, 40.24]$, middle; $t^* \in [73.77, 83.77]$, bottom). Reference $z$ (solid) against prediction $\tilde{z}$ (dashed). (a) Converged multiple-shooting fit on the augmented trajectory set, with segment start/end markers; (b) unaided single rollout from $t^* = 0$ over the same windows.

training epochs are varied to examine their influence on the hybrid simulation across its respective time window. For the Neural ODE, the influence of the multiple-shooting group size $n_s$ is displayed, since it directly governs rollout stability and $f_\phi$ learning capabilities. For the integration monitor, the effect of the relative threshold quantile $q$ is shown, as it directly influences the rollout length.

The parameters presented in this section are those that have the greatest influence on the main goal of the developed framework, namely, accelerating chaotic simulations. Because the developed framework comprises numerous parameters that can influence the acceleration and fidelity of the hybrid runs, not all are shown in this section. For an overview of all simulations that were run, and their relevant performance metrics, see Appendix B.

Each parameter is varied individually, following a one-factor-at-a-time (OFAT) analysis about the baseline of Table 4.4; this means that the interactions between the parameters are not explored. A comprehensive study of the interactions between these parameters would require a large number of pipeline training runs, which would grow exponentially with the parameter count and exceed this work's computational budget. The parameter variations below, therefore, report the effect of each parameter, with the baseline serving as a solid comparison point. The following subsections will present and compare various hybrid simulations based on the comparison metrics presented in Section 4.1.1, the exact values that were used to construct these metrics (such as $T_{\text{wall}}^{\text{h, tot}}$ and $T_{\text{wall}}^{\text{h, training}}$) are presented in Appendix B.

### 4.3.1. Autoencoder

The autoencoder uses $\varphi_\theta$ to compress instantaneous velocity snapshots $\mathbf{u}$ down to the compressed representation $\mathbf{z}$. Although it is not a time-dependent operation, it does have a significant influence on the dynamics of $\boldsymbol{\mathcal{T}}_z$, since it controls how the full-scale physics are represented on the lower-dimensional latent space. This means that $\varphi_\theta$ dictates the dynamics that $f_\phi$ has to learn, which get rolled out by the ODE solver past the training horizon as the rolled out trajectory $\tilde{\boldsymbol{\mathcal{T}}}_z$. Subsequently, these rolled-out dynamics are reconstructed in physical space by $\psi_\theta$, which in turn influences the downstream flow behaviour, which is fed back into $\mathcal{P}$ at later retrain or hybrid stages.

To assess the influence of the autoencoder, three parameters are varied: the latent dimension $N_z$, the number of initial training epochs, and the physics-informed loss weighting.

**Latent dimension**

The latent dimension $N_z$ determines the width of the bottleneck and governs how much of the flow physics is preserved by the compression operation. Nair et al. [56] has shown that for autoencoder–

neural ODE surrogates of advection-dominated flows, the recovered latent time-scales are governed primarily by the training-trajectory length rather than by $N_z$, so that the bottleneck width has little bearing on how long the latent rollout remains temporally faithful. That work, however, evaluates accuracy in the latent prediction alone; it does not assess the fidelity of the full-scale field reconstructed from that latent state. The variation in latent dimensions here, therefore, serves to expose how $N_z$ trades the achievable acceleration against the fidelity of the reconstructed latent and full-scale dynamics.

In the present hybrid setting, $N_z$ instead trades inference speed against the fidelity of the reconstructed flow statistics, as the sweep over $N_z \in \{8, 16, 32\}$ (with $N_z = 16$ the baseline) in Figure 4.16 shows.

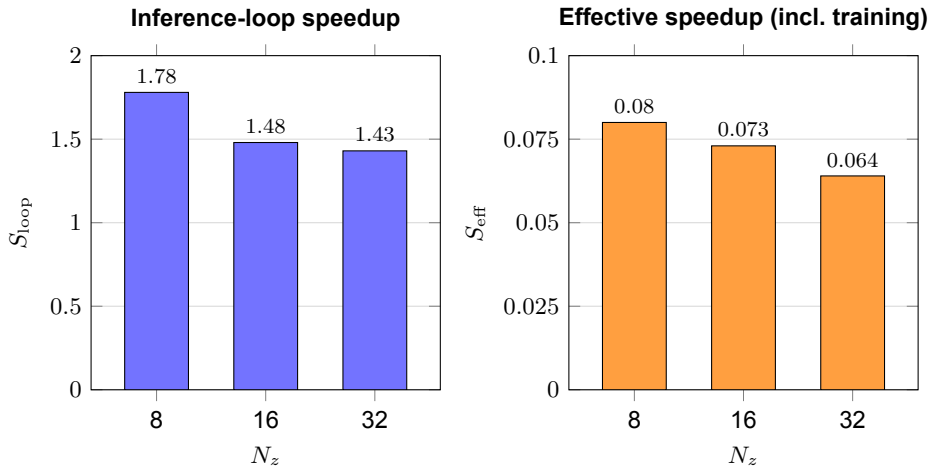



**Figure 4.16:** Effect of the autoencoder latent dimension on hybrid solver acceleration. Left: inference-loop speedup $S_{\text{loop}}$; Right: effective speedup $S_{\text{eff}}$, including training costs.

The smallest bottleneck proves to be the best of the three models in terms of acceleration performance: with $N_z = 8$, the hybrid loop achieves $S_{\text{loop}} = 1.78$, compared with $1.48$ and $1.43$ for $N_z = 16$ and $32$. The acceleration performance is largely attributable to the longer average rollout length of the smallest bottleneck, with an average of $964$ CFD steps per accepted prediction, compared to $408$ and $421$ for $N_z = 16$ and $32$, respectively. That $N_z = 16$ still attains a marginally higher $S_{\text{loop}}$ than $N_z = 32$ despite a slightly shorter mean rollout reflects the lower per-step cost of advancing and decoding the narrower latent state. The lower-dimensional latent space presents simpler dynamics for the compact MLP $f_\phi$ to learn, so the rollout $\tilde{\mathcal{T}}_z$ stays in distribution for longer before the KNN monitor flags it. This extended in-distribution horizon, and thus not any change in the intrinsic latent time-scale, is the dominant source of the higher $S_{\text{loop}}$ at small $N_z$; the latent dynamics are not slower, only easier to learn for both $f_\phi$ and the KNN monitor, which is consistent with Nair et al. [56]. However, because of the lower amount of values that are able to contain information about the chaotic flow, it is not able to produce the same result when assessed over mean flow and RST accuracy, as displayed in Figure 4.17.

As expected, a smaller latent space means that fewer of the flow's characteristics survive the encoder $\varphi_\theta$ to be reconstructed by the decoder $\psi_\theta$. Figure 4.17 shows that this loss is carried almost entirely by the second-order statistics. As the bottleneck widens, both panels improve with $N_z$, the field error $\varepsilon$ falls, and the spatial correlation $\rho$ rises for every Reynolds-stress component. The degradation is most severe for the transverse normal stress $\langle v'v' \rangle$ of the RST, whose error nearly triples relative to $N_z = 32$ ($\varepsilon = 0.027$ against $0.010$). The correlation coefficients show that the shear stress $\langle u'v' \rangle$ is the least well recovered, especially for the smallest bottleneck.

This comparison shows the trade-off the bottleneck imposes between speedup and the physical fidelity of the reconstructed turbulence statistics. At $N_z = 8$, the simpler latent dynamics produce by far the longest rollouts and the largest $S_{\text{loop}}$, but at the expense of the RST fields; the $N_z = 32$ shows that more complex latent dynamics allow for better reconstructions, but at the expense of a lesser acceleration performance. This means that the appropriate bottleneck size is thus dictated by whether statistical fidelity or acceleration behaviour is the priority for a given application.

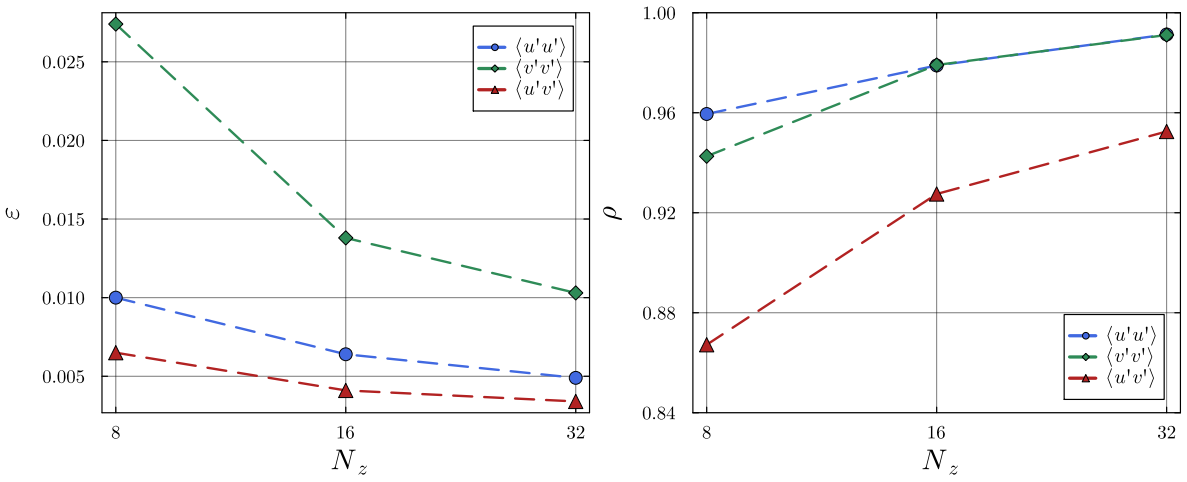



**Figure 4.17:** Effect of the autoencoder latent dimension $N_z$ on the accuracy of the reconstructed Reynolds-stress field. Left: mean absolute error $\varepsilon$ of the three independent Reynolds-stress components. Right: spatial correlation coefficient $\rho$ of the same components. Both metrics improve monotonically with $N_z$, showing how the shear stress $\langle u'v' \rangle$ is the least well-recovered component at every bottleneck width.

### Training Epochs

The number of training epochs is varied around the baseline of $500$, since the autoencoders' initial training consumes most of the wall time, as Table 4.5 shows, but it also controls how strictly the instantaneous dynamics are learned, and therefore how the fluctuating field is represented in the latent space. The epoch count $E$ is swept over $E \in \{250, 500, 750\}$, with $E = 500$ the baseline; the retraining epochs are held fixed at $100$ throughout, so the sweep is meant to show the effect of the *initial* compression quality.

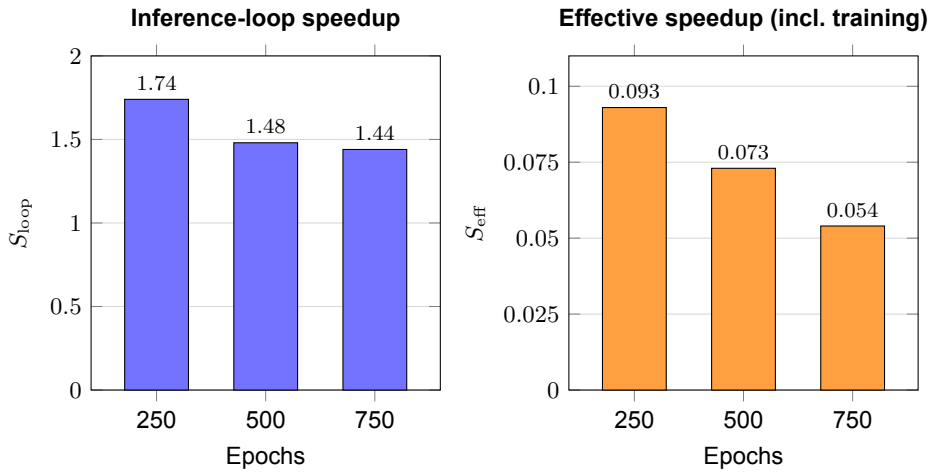



**Figure 4.18:** Effect of initial training epochs of the autoencoder on hybrid solver acceleration. Left: inference-loop speedup $S_{\text{loop}}$; Right: effective speedup $S_{\text{eff}}$, including training costs.

Figure 4.18 shows how the effect on the acceleration performance mirrors that of the latent dimensions. The least-converged model performs best: at $E = 250$, the hybrid loop reaches $S_{\text{loop}} = 1.74$, against $1.48$ at the baseline and $1.44$ at $E = 750$. Just as with the narrower bottleneck, the improvement is carried by the average rollout length: $623$ CFD steps per accepted prediction at $E = 250$, compared to $408$ and $391$ at $E = 500$ and $750$. An under-trained encoder has not yet learned the fine fluctuations, so it produces a smoother, lower-complexity latent representation; in turn, the dynamics that $f_\phi$ must reproduce are correspondingly simpler, leading to $\tilde{\mathcal{T}}_z$ remaining in distribution for longer before the KNN monitor flags it as OOD. Reducing the training budget, therefore, affects the rollout length through the same mechanism as shrinking $N_z$: it lowers the effective information that can be stored in the flow's

latent representation.

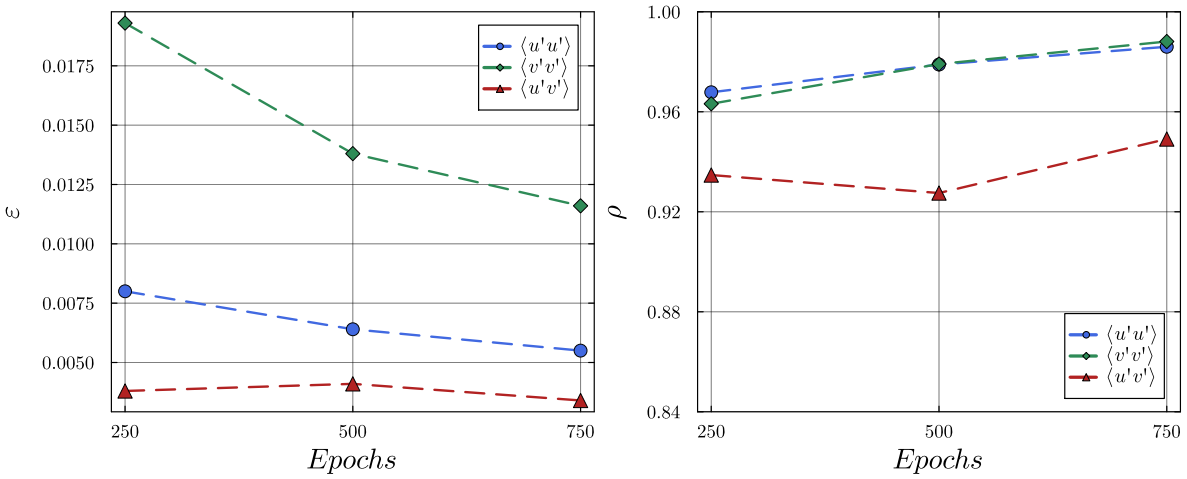



**Figure 4.19:** Effect of the initial autoencoder training epochs on the accuracy of the reconstructed Reynolds-stress field. Left: mean absolute error $\varepsilon$ of the three independent Reynolds-stress components. Right: spatial correlation coefficient $\rho$ of the same components. Increasing epoch count lowers $\varepsilon$, most strongly for $\langle v'v' \rangle$, while $\rho$ stays near unity

Figure 4.19 shows how the increase in epochs lowers the error in magnitude of the mean fluctuations, especially the transverse $\langle v'v' \rangle$ component. The spatial correlation $\rho$ is already high at $E = 250$ and barely changes. The encoder, therefore, captures the spatial pattern of the Reynolds stresses almost immediately, and the additional epochs mainly calibrate the fluctuation amplitude rather than relocate it. This behaviour can be highlighted when plotting the trajectory of the latent representation of the training window:

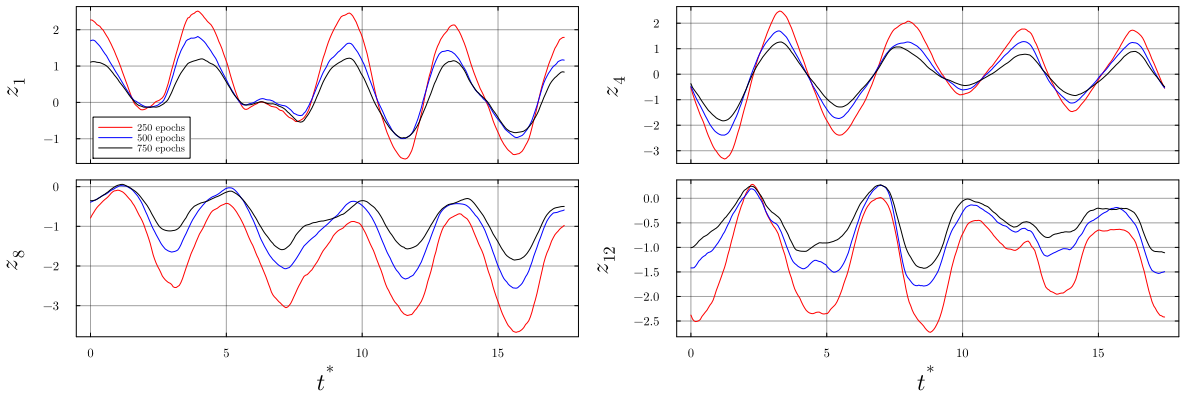



**Figure 4.20:** Four representative latent coordinates of the latent trajectory $\mathcal{T}_z$ for autoencoders trained $E = 250$ (red), $500$ (blue) and $750$ (black) epochs. The lower-epoch $\mathcal{T}_z$ has a higher amplitude than the higher-epoch trajectories, which appear to converge as $E$ increases.

Figure 4.20 confirms this conclusion. It shows how all three initial encoded trajectories trace the same dominant shedding oscillation, so the spatial pattern is shared, but the lower-epoch $\mathcal{T}_z$ oscillates at a higher amplitude that seems to converge as the number of epochs increases. The extra epochs thus mainly rescale the latent signal rather than reshape it; the rescaled amplitude is then translated to physical space by $\psi_\theta$, explaining the drop in $\varepsilon$ at near-constant $\rho$. The higher-amplitude, smoother $E = 250$ trajectory also provides the simpler dynamical target for $f_\phi$, which is why it sustains the longest average rollout.

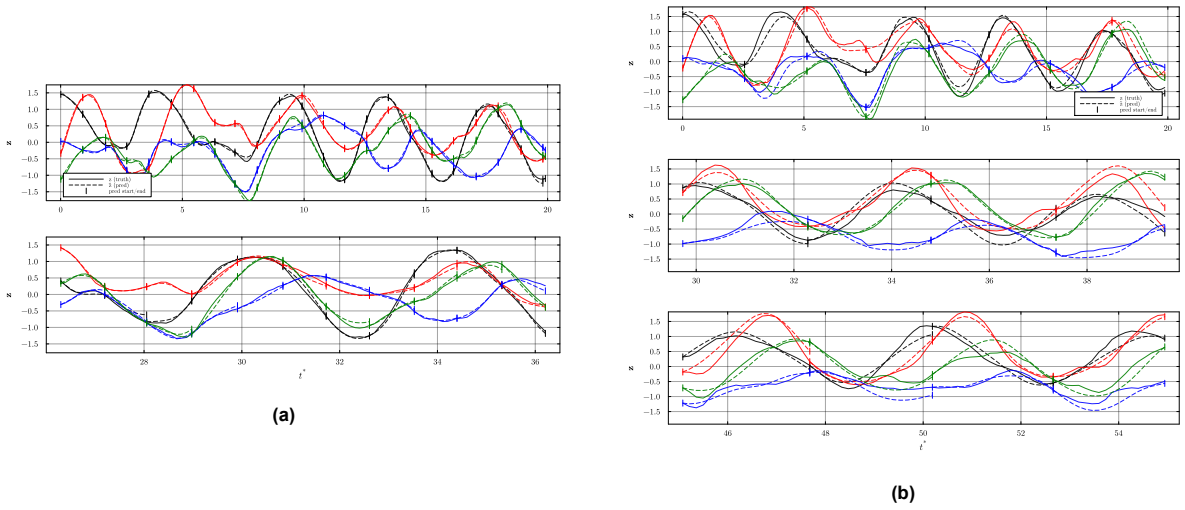
### 4.3.2. Neural ODE

The neural ODE $f_\phi$ integrates the latent state of the simulation over time during the latent rollout, so how long its prediction remains faithful to the reference sets directly determines how long the KNN monitor keeps the loop in distribution, and, with it, the achievable speedup. Two important hyperparameters of

the multiple-shooting scheme Section 3.5.2 govern this fidelity and are varied here: the group size $n_s$ and the continuity weight $\lambda_c$.

**Group size**

The group size determines the length of the independent integrations that the trajectory is partitioned into: a small $n_s$ keeps the optimisation stable but multiplies the shooting boundaries and leaves little continuous latent horizon to be learnt, biasing the latent dynamics towards shorter rollouts. A lower $n_s$ also allows finer latent dynamical features to be learned by $f_\phi$. A larger $n_s$ will thus pass a smoother version of the latent dynamics to be learnt, with lower fidelity, but gives $f_\phi$ more time horizon to rollout over, which could improve the rollout length.



(a) (b)

**Figure 4.21:** Converged multiple-shooting fits for all training and retraining loops. (a) $n_s = 10$ and (b) $n_s = 20$; reference $z$ (solid) against prediction $\tilde{z}$ (dashed), four representative latent components over the training windows. The smaller group tracks the finer oscillation detail within each segment; the larger group recovers a smoother, lower-fidelity trajectory. Baseline fit is shown in Figure 4.15

Figure 4.21 shows the converged fits for $n_s = 10$ and $n_s = 20$. The small group reproduces the finer features of the latent oscillation more tightly within each segment, where the large group fits a smoother trajectory that loses some of the higher-frequency fidelity of the latent trajectory. The tighter fit of the small group lets $f_\phi$ produce in-distribution rollouts for longer at inference: the smallest group $n_s = 10$ sustains the longest unaided rollouts, $948$ steps per prediction, against $686$ for $n_s = 20$ and $408$ for the baseline $n_s = 15$, and correspondingly requires a single parameter update, only switches back to CFD stabilisation $6$ times, against $7$ and $9$. This carries directly into the acceleration of Figure 4.22: $n_s = 10$ attains $S_{\text{loop}} = 1.65$ and $S_{\text{eff}} = 0.093$, the highest of the three. The dependence on $n_s$ proves to be not strictly monotonic, since the baseline rollouts are on average shorter than those of $n_s = 20$, despite its intermediate group size. The actual optimum for the group size is rather hard to determine with a single run per configuration on a chaotically diverging wake, because of run-to-run variability in when the out-of-distribution monitor first triggers relative to the shedding phase. The advantage of the small group is nonetheless large and consistent across both speedup measures.

The longer rollouts come at a cost in accuracy, but the full-domain mean-flow and Reynolds-stress statistics are not the relevant measure of that cost here. These fields are accumulated over the entire simulation window, the bulk of which is advanced by the exact solver, so a configuration that produces shorter rollouts spends a larger fraction of its steps in pure CFD and has its field statistics inflated accordingly, independently of how well $f_\phi$ reconstructed the flow during the latent rollouts. Comparing the reconstructed RST and mean-flow fields across group sizes, therefore, rewards the configuration that relied on the surrogate least, rather than the one that reconstructed the flow most faithfully, and this latter configuration is set aside here. The important takeaway from the group size is that a smaller group is generally the more beneficial choice for the framework: by partitioning the trajectory into shorter segments, the optimiser resolves the finer features of the training dynamics within each segment and passes a more faithful representation of those dynamics to $f_\phi$, as seen in the tighter fit of Figure 4.21a.

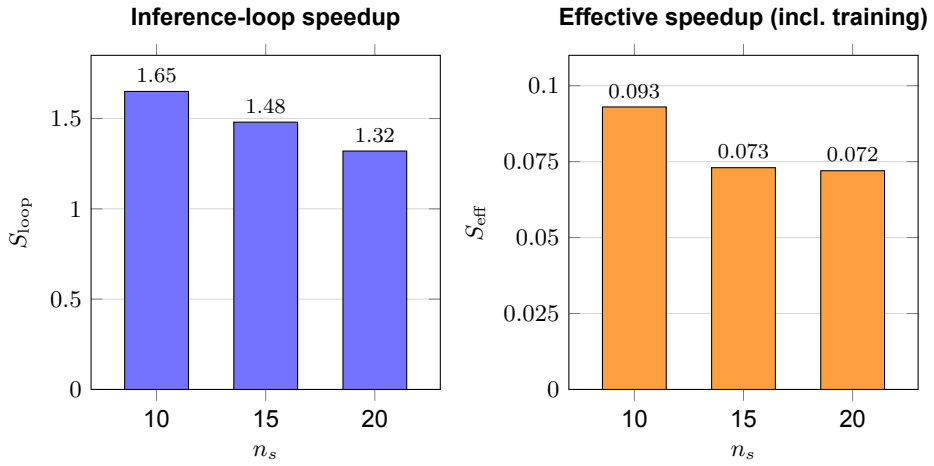



**Figure 4.22:** Effect of multiple shooting group size on hybrid solver acceleration. Left: inference-loop speedup $S_{\text{loop}}$; Right: effective speedup $S_{\text{eff}}$, including training costs.

This better-captured surrogate is what sustains the longest unaided rollouts and the highest speedup of the three configurations, so for the acceleration objective that motivates the framework, the smaller group is preferable, provided its reduced accuracy in the hybrid regions remains acceptable for the application at hand.

### 4.3.3. Integration monitor

The integration monitor is the only pipeline component that influences the acceleration without itself being a learned feature: it decides at every prediction attempt how far a rollout is allowed to advance before WaterLily stabilises the simulation (Section 3.6.2). Rollout length is one of the most dominant influences on $S_{\text{loop}}$, and the monitor precisely terminates a rollout when its score crosses $\tau$ (Table 4.7). The threshold quantile $q$ directly determines how strict the threshold is, thereby controlling the maximal rollout length. To investigate its influence, the quantile is varied over $q \in \{0.9, 0.95, 0.99\}$. Tightening the gate (lower $q$, and in turn $\tau$) flags the rollout earlier and shortens the average steps-per-prediction; this trades away $S_{\text{loop}}$ for a stricter guarantee that the decoded state has not drifted far from the training distribution $\mathcal{T}_z$. A less direct effect is that the threshold governs how quickly OOD flags accumulate towards $n_{\text{max, flags}}$, so the monitor also shifts the retraining intervals, thereby affecting the maximal achievable speedup, $S_{\text{eff}}$.

**Threshold variation**

As desired in the rollout, tightening the KNN quantile results in stricter gating and shorter rollouts. The average rollout length drops from $408$ steps at the base setting ($q = 0.999$) to $307$ at $q = 0.95$ and $178$ at $q = 0.9$, which is exactly what a lower $\tau$ should do: a tighter threshold notices the latent state drifting away from $\mathcal{T}_z$ sooner, so the conventional solver takes back control after less neural ODE steps. Figure 4.23 shows how both speedup metrics follow the same increasing path as the rollout length does for a higher quantile setting.

An important point about the stricter rollout gating is that the monitor at $q = 0.9$ triggers one more retraining cycle than the monitors at $q = 0.95$ and $q = 0.999$, which trigger the same number of retraining cycles. Since training the ML components dominates the wall-time budget, this single extra cycle is what drags $S_{\text{eff}}$ down to $0.047$, compared with $0.055$ at $q = 0.95$ and $0.073$ at $q = 0.999$. The smaller gap between the latter two is driven instead by the shorter rollouts at $q = 0.95$, which force the conventional solver to do more of the work. A tight KNN threshold, therefore, erodes the achievable acceleration through two channels: shorter rollouts that hand more of the workload back to the slower conventional solver, and more frequent flagging that triggers additional retraining.

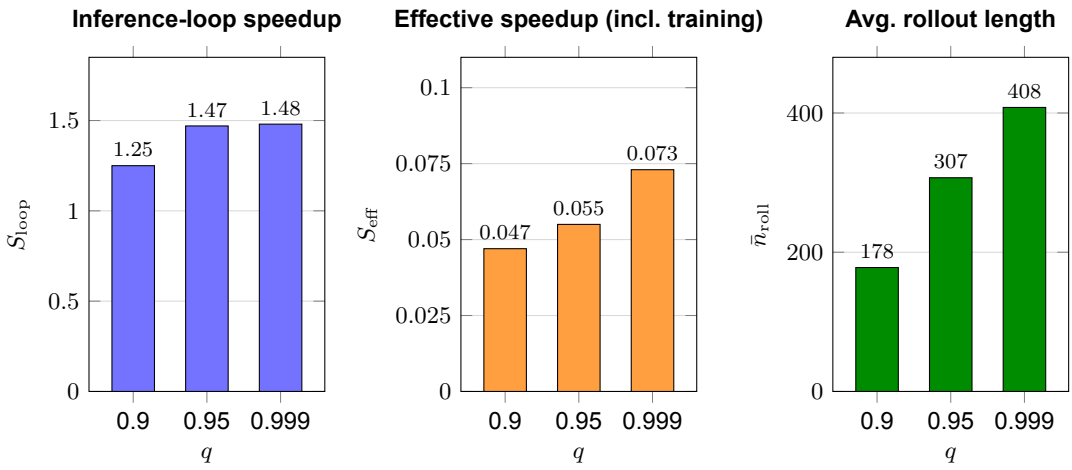



**Figure 4.23:** Effect of KNN threshold quantile $q$ on hybrid solver acceleration. Left: inference-loop speedup $S_{\text{loop}}$; middle: effective speedup $S_{\text{eff}}$, including training costs; right: average rollout length $\bar{n}_{\text{roll}}$ in CFD steps per accepted prediction.

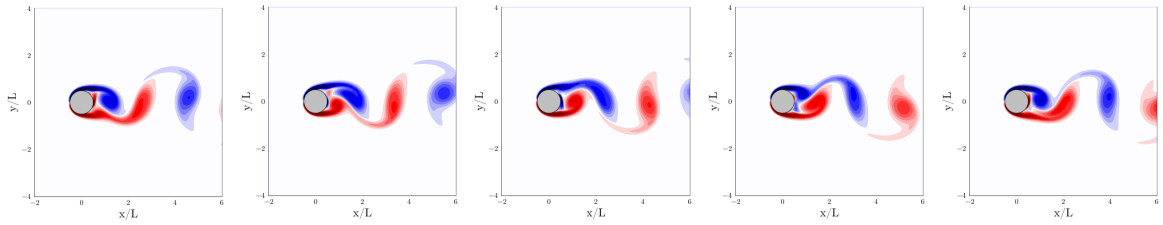
## 4.4. Generalisation

The results presented so far were obtained at a single operating point, namely the cylinder wake at $Re = 2500$ on a $256 \times 256$ grid, a regime deliberately chosen to test the limits of the autoencoder and neural ODE under a manageable computational cost. This section steps away from that point to inspect how the achieved acceleration and physical fidelity respond to the two parameters that define the problem, the flow regime and the grid resolution. The Reynolds number is first reduced to a lower shedding regime to assess the pipeline's behaviour, where the dynamics are simpler and more periodic. Following this, the grid is refined and the Reynolds number is doubled to $Re = 5000$ to stress test how the achieved acceleration scales with problem size and how the framework handles more complex flow dynamics.

Unless stated otherwise, all parameters are kept identical to those of the base case reported in Section 4.1; the autoencoder architecture, latent dimension $N_z = 16$, the multiple-shooting settings ($n_s = 15$, $\lambda_c = 200$), and the KNN monitor calibration are left unchanged, so that any difference in the results below is just an effect of the change in flow regime or grid resolution alone.

### 4.4.1. Re 250

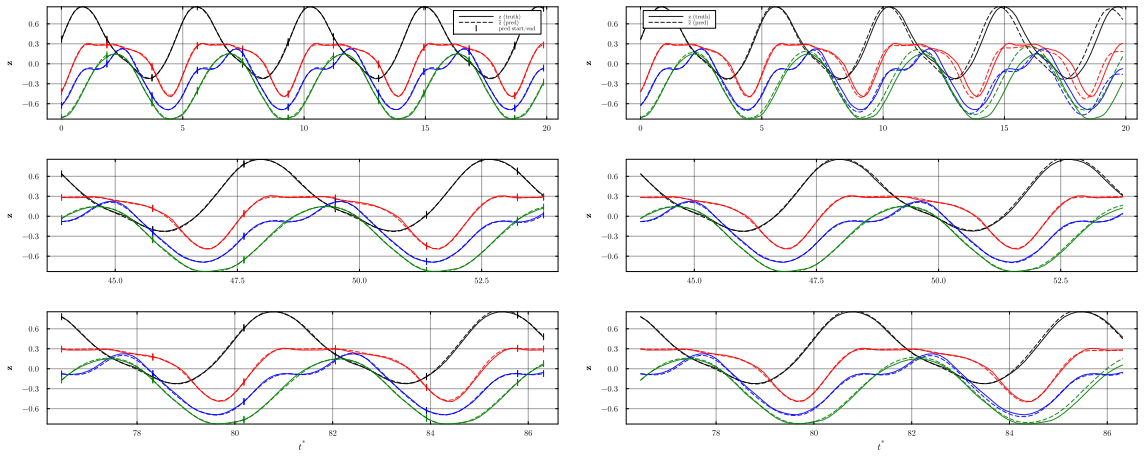
At $Re = 250$, the cylinder wake sheds as a clean periodic cycle with a single shedding frequency and less abrupt gradients between flow structures. Figure 4.24 displays the vorticity field of the new flow regime in 5 consecutive snapshots.




**(a)** $t^*_n$ **(b)** $t^*_{n+\Delta t}$ **(c)** $t^*_{n+2\Delta t}$ **(d)** $t^*_{n+3\Delta t}$ **(e)** $t^*_{n+4\Delta t}$

**Figure 4.24:** Vorticity field $\omega^{L}/_{U}$ of five consecutive snapshots at $Re = 250$. Indicating positive (red) and negative (blue) vorticity. Showing the clean periodic shedding

This new flow regime will prove to be no challenge to $\mathcal{P}$, which is able to capture this regime with high fidelity: the latent dynamics $f_\phi$ are able to track the encoded trajectory $\mathcal{T}_z$ closely throughout the training window. Figure 4.25 shows the cleanly periodic latent trajectories, the multiple shooting fits and a single training span rollout of the latent dynamics.



**(a)** Converged multiple-shooting fit on the augmented trajectory set; markers denote shooting-segment boundaries.

**(b)** Single rollout from $t^* = 0$ across both windows after the first retraining, no segment resets.

**Figure 4.25:** Latent dynamics of $f_\phi$ at $Re = 250$, four representative components shown over all time windows. Reference $z$ (solid) against prediction $\tilde{z}$ (dashed). (a) Converged multiple-shooting fit on the augmented trajectory set, with segment start/end markers; (b) unaided single rollout from $t^* = 0$ over the same windows.

This high-quality fit, however, is double-edged. Because of the periodic nature of the wake, the latent vectors form a low-variance distribution in the $N_z$-dimensional space, so the distances between latent vectors follow this distribution tightly. The neighbour distances on which the KNN monitor $\mathcal{K}$ is calibrated are therefore small, and the resulting threshold $\tau$ becomes overly tight, allowing almost no drift of the neural ODE. This behaviour is confirmed in Figure 4.26 together with Figure 4.10.

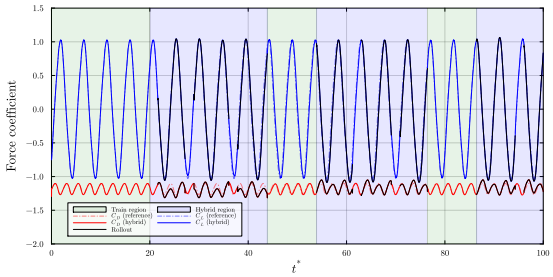


**Figure 4.26:** Force coefficients comparison for the $Re = 250$ hybrid run. Showing that the simpler wake dynamics are quickly learnt, but longer rollouts are prevented by a strict KNN threshold.

| # | NODE model | Steps | $\Delta t^*$ | $t^*_{\text{r, end}}$ | score | $\tau$ |
|---|---|---|---|---|---|---|
| 1 | initial | 505 | 5.516 | 27.176 | 0.0753 | 0.0720 |
| 2 | initial | 647 | 8.889 | 36.065 | 0.0722 | 0.0720 |
| 3 | initial | 550 | 7.825 | 43.890 | 0.0749 | 0.0720 |
| 4 | 1st retrain | 667 | 17.319 | 61.209 | 0.0657 | 0.0654 |
| 5 | 1st retrain | 558 | 7.909 | 69.118 | 0.0682 | 0.0654 |
| 6 | 1st retrain | 505 | 7.327 | 76.445 | 0.0657 | 0.0654 |
| 7 | 2nd retrain | 716 | 17.847 | 94.292 | 0.0741 | 0.0736 |
| 8 | 2nd retrain | 358 | 5.724 | 100.017 | – | 0.0736 |

**Table 4.10:** Per-rollout summary of the KNN-gated NODE inference at $Re = 250$. Every accepted rollout terminates marginally above the active threshold $\tau$; rollout 8 is capped by the $t^* = 100$ cutoff while still in distribution.

Figure 4.26 does show that $\psi_\theta$ is able to reconstruct the gradients close to the cylinder wall rather well. Since the force coefficients match exactly those of the reference simulation. Showing that for this reduced Reynolds number, the relevant velocity gradients are less harsh, and can thus be reconstructed rather well compared to the higher Reynolds number contour plots seen in Figure 4.12.

When analysing the inference-loop speedup over the complete simulation window $t^* \in [0, 100]$ is

$$S_{\text{loop}} = \frac{T^{\text{ref}}_{\text{wall}}}{T^{\text{h, tot}}_{\text{wall}}} = \frac{114.41\,\text{s}}{62.50\,\text{s}} = 1.83, \tag{4.8}$$

with the hybrid total splitting into $T^{\text{h, cfd}}_{\text{wall}} \approx 61.50\,\text{s}$ of stabilising CFD steps and $T^{\text{h, rollout}}_{\text{wall}} = 1.00\,\text{s}$ for the eight accepted latent rollouts of Figure 4.10. The rollouts themselves make up only $1.6\%$ of the loop wall time; the denominator is almost entirely WaterLily CFD steps. Just as in the base simulation, $\mathcal{P}$ proves to be an order of magnitude faster per unit simulated time than the solver it replaces, but the tight OOD gate forces roughly half of the simulation window to be covered by CFD steps between rollouts,

diluting the realised $S_{\text{loop}}$ to $1.83$. The acceleration potential of the simpler $Re = 250$ dynamics is real but largely unrealised, gated by rollout length rather than by the cost of the latent dynamics $f_\phi$.

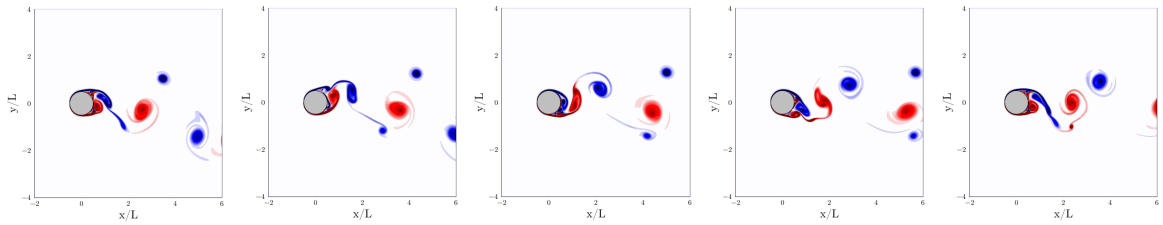
When adding the training cost of the networks, the effective speedup of the complete workflow can be realised.

$$S_{\text{eff}} = \frac{T_{\text{wall}}^{\text{ref}}}{T_{\text{wall}}^{\text{h, tot}} + T_{\text{wall}}^{\text{h, training}}} = \frac{114.41\,\text{s}}{62.50\,\text{s} + 1266\,\text{s}} \approx 0.086, \tag{4.9}$$

where the training wall time cost $T_{\text{wall}}^{\text{h, training}} = 21.1\,\text{min}$ covers the initial autoencoder and NODE training and the two retrain phases. As in the base case, the workflow is much slower than plain WaterLily once training is included in the speedup factor, since the training costs still dwarf the short reference run. The effective speedup is comparable to that of the base case, because the autoencoder dominates the training budget and takes a similar amount of wall time in both regimes.

### 4.4.2. Re 5000

As established in Section 4.2.2, the per-CTU advantage of the latent rollout is tied to the base-case resolution: since the WaterLily solver is memory-bound, its cost per CTU grows with the grid size [98], whereas a single rollout of $\mathcal{P}$ is fixed by the latent dimension $N_z = 16$ and is independent of the grid. The $256^2$ baseline used throughout this work is therefore a comparatively inexpensive and dynamically simple case for demonstrating the gain. To assess the hybrid loop on a simultaneously more expensive and more demanding regime, the reference case is re-run with an increase in Reynolds number to $Re = 5000$ and the grid refined to $512^2$, keeping the domain and cylinder geometry of Figure 3.1 fixed. These two changes act on different parts of the framework. The finer grid raises the cost per CTU of the reference solver while leaving the rollout cost untouched, which should widen the achievable acceleration margin, whereas the higher Reynolds number drives the wake from the quasi-periodic shedding of the baseline towards a more complex shedding state with sharper near-body shear layers and more complex structures in the wake, as displayed in Figure 4.27.



(a) $t_n^*$ (b) $t_{n+\Delta t}^*$ (c) $t_{n+2\Delta t}^*$ (d) $t_{n+3\Delta t}^*$ (e) $t_{n+4\Delta t}^*$

**Figure 4.27:** Vorticity field $\omega^{L/U}$ of five consecutive snapshots at $Re = 5000$. Indicating positive (red) and negative (blue) vorticity. The refined grid and higher Reynolds number result in more complex wake structures that are more sharply resolved than those of the $256^2$ baseline.

**Wall time performance**

Following the wall-clock breakdown in Section 4.2.2, the $Re = 5000$ hybrid simulation is broken down into its segments, whose wall times are presented in Table 4.11. Just as in the base simulation, the initial autoencoder training consumes a very high fraction of $\mathcal{S}$'s total wall time, just over $50\%$ of the total, and once the two retraining cycles are added, building and updating the networks accounts for close to $90\%$ of the $3217$ s pipeline. The initial data-gathering, the initial data-gathering, and the three snapshot-gathering runs make up most of what remains, while the latent rollouts themselves contribute well under $1\%$; the only non-negligible inference-loop cost is the single extended CFD stretch in the second hybrid section ($92.3\,\text{s}$), where the OOD monitor held control with the solver rather than the rollout.

As mentioned, the rollout cost barely changes with the grid. It is fixed by the latent dimension $N_z = 16$ and stays around $48$ ms/CTU, the same order as the base run, because it never touches the full field. The WaterLily time integrations, on the other hand, now cost $4842$ ms/CTU, against $866$ ms/CTU at $256^2$. This large cost difference between rollout and CFD is even more noticeable when inspecting the acceleration metrics, $S_{\text{loop}}$:

| Phase | Component | Time (s) | % of total |
|---|---|---|---|
| Initial data-gathering | – | 84.77 | 2.64 |
| Train | AE | 1662.00 | 51.66 |
| | NODE | 222.00 | 6.90 |
| Hybrid section 1 | CFD + rollout | 3.781 | 0.12 |
| | CFD + rollout | 4.066 | 0.13 |
| | CFD + rollout | 3.986 | 0.12 |
| Retrain 1 | Gather snapshots | 52.57 | 1.63 |
| | AE | 348.00 | 10.82 |
| | NODE | 156.00 | 4.85 |
| Hybrid section 2 | CFD + rollout | 0.038 | 0.00 |
| | CFD + rollout | 3.959 | 0.12 |
| | CFD + rollout | 4.776 | 0.15 |
| Retrain 2 | Gather snapshots | 50.30 | 1.56 |
| | AE | 390.00 | 12.12 |
| | NODE | 192.00 | 5.97 |
| Hybrid section 3 | CFD + rollout | 0.148 | 0.00 |
| | CFD + rollout | 4.223 | 0.13 |
| | CFD + rollout | 5.002 | 0.16 |
| Retrain 3 | Gather snapshots | 29.69 | 0.92 |
| **Total** | | **3217.31** | **100.00** |

**Table 4.11:** Wall-clock breakdown of the hybrid simulation pipeline ($Re = 5000$ run).

$$S_{\text{loop}} = \frac{T_{\text{wall}}^{\text{ref}}}{T_{\text{wall}}^{\text{h, tot}}} = \frac{651.83\,\text{s}}{314.94\,\text{s}} = 2.07, \tag{4.10}$$

Showing that the hybrid loop uses the low-cost rollouts to its full effect, where it is able to achieve a $2.07$ speedup compared to the reference simulation. To assess the speedup of the hybrid loop compared to the training cost of the ML components of $\mathcal{P}$, the training wall time is added to the comparison in $S_{\text{eff}}$ with $T_{\text{wall}}^{\text{h, training}} = 3042$ s of training.

$$S_{\text{eff}} = \frac{T_{\text{wall}}^{\text{ref}}}{T_{\text{wall}}^{\text{h, tot}} + T_{\text{wall}}^{\text{h, training}}} = \frac{651.83\,\text{s}}{314.94\,\text{s} + 3042\,\text{s}} \approx 0.19, \tag{4.11}$$

A single finer run is therefore still much slower than simply running $\mathcal{S}_{\text{ref}}$, because the training cost is not repaid within one simulation window. But it still performs better than the base case, which achieved an effective speedup of $0.073$, indicating that refining the grid begins to close part of the gap between the full pipeline and running the reference simulation.

At a lower resolution, this demonstrates that the cost of the framework is dominated by one-off training rather than by the accelerated integration: the inference loop is already cheaper than the reference, and the more expensive the reference, the larger that margin becomes. Closing the remaining gap to break-even no longer requires a faster rollout; it requires either reusing the trained $\mathcal{P}$ across runs or overlapping training with the live simulation, together with a less strict integration monitor to convert the available loop speedup into realised wall-time savings.

**Spatial and physical fidelity**

The finer simulation is passed to $\mathcal{P}$ without any architectural change: the architectures of $\varphi_\theta$, $\psi_\theta$ and $f_\phi$ of Section 4.2 are reused as-is. The only change to the architecture is the input and output dimensions

of the autoencoder, so that they can correctly pass the new, finer grid down to the latent space without dimension mismatches. An advantage of the grid refinement is that the decoder's fixed cell-width smoothing now spans half the physical width, compared to the base architecture. This means that the thin separating shear layers that dominated the base-grid reconstruction error (Figure 4.11) are resolved by roughly twice as many cells and blurred less in physical space. Because the autoencoder can now capture finer dynamics to a much greater degree, the cost function converges much faster on the validation set, as displayed in Figure 4.28

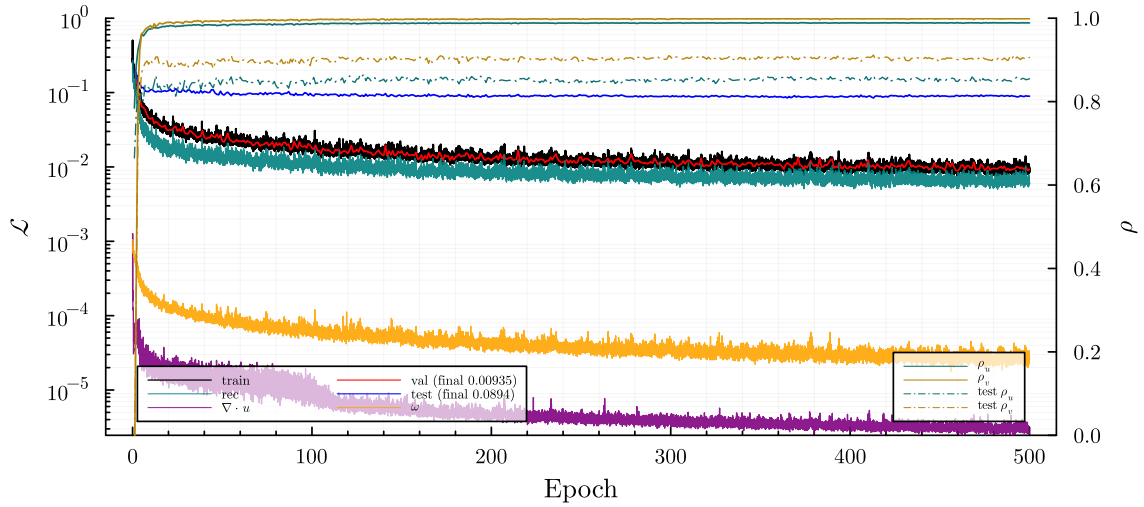



**Figure 4.28:** Initial training history of the autoencoder at the $Re = 5000, 512^2$ operating point. Left axis ($\mathcal{L}$, logarithmic): total loss on the train, validation and test splits, together with the divergence ($\nabla \cdot \mathbf{u}$) and vorticity ($\omega$) penalties evaluated on the test split. Right axis ($\rho$): reconstruction correlation coefficients $\rho_u$, $\rho_v$ on train and test. The validation and test losses converge to $9.6 \times 10^{-3}$ and $8.9 \times 10^{-2}$ respectively, the persistent gap between them marking the autoencoder's failure to generalise to the more chaotic wake.

Comparing the first autoencoder training to that of the base case in Figure 4.9, the AE is able to achieve a lower loss on the train and validation set, but the poor performance of the test set proves that the more chaotic wake, due to the increased Reynolds number, is not captured by the train/validation window alone. The gap between the converged train/validation loss and the test loss is markedly wider than the factor of two seen at base resolution, indicating that the held-out window $t^* \in [17.5,\ 20]$ contains wake states the autoencoder has not encountered during training. Whereas the base-case wake repeats structures that the network has already learnt, the $Re = 5000$ test split presents the autoencoder with spatial configurations not represented in the earlier shedding cycles, so $\varphi_\theta$ and $\psi_\theta$ generalise poorly beyond the training horizon despite the lower in-sample loss. The lower train/validation loss is therefore not evidence of a better fit but of a wake whose aperiodicity the four training cycles cannot span: the autoencoder reconstructs what it has seen more tightly, yet fails to extrapolate to unseen states, which, for a wake this chaotic, makes generalisation intrinsically harder.

However, Figure 4.28 shows that the early plateau of the autoencoder training shows that a large part of the training budget is spent unnecessarily. Unlike the base case, the test loss barely changes over the performed epochs; it settles at its higher level within the first $\sim 100$ epochs and does not improve thereafter, so the remaining epochs only tighten the in-sample fit without recovering any accuracy on the held-out window. Because the generalisation performance at this Reynolds number is set by the spatial information contained in the training set rather than by the length of training, the epochs spent past the plateau buy nothing on the unseen wake states and can be removed at no cost to the test-set reconstruction. Since the initial autoencoder consumes just over half of the workflow budget (Table 4.11), early-stopping it at the convergence knee, together with the same reduction applied to the two retraining cycles, translates directly into a higher effective speedup. If the initial autoencoder were trained for only $\sim 100$ epochs, and the retraining runs cut to the same fraction, the complete training time would fall from $3042\,\mathrm{s}$ to roughly $1100\,\mathrm{s}$ while leaving the reconstruction accuracy unchanged, since that accuracy was already reached at the plateau. This yields a projected effective speedup of:

$$S_{\text{eff}} = \frac{T_{\text{wall}}^{\text{ref}}}{T_{\text{wall}}^{\text{h, tot}} + T_{\text{wall}}^{\text{h, training}}} = \frac{651.83\,\text{s}}{314.94\,\text{s} + 1100\,\text{s}} \approx 0.46, \tag{4.12}$$

This speedup value shows that trimming the autoencoder training schedule alone moves $S_{\text{eff}}$ from $0.19$ using the initial training scheme to $0.46$, an order-of-magnitude gain that moves the integrated workflow from roughly ten times slower than the reference to within a factor of two of break-even, without touching the inference loop itself. It demonstrates that the cost of the framework is dominated by one-off training rather than by the accelerated integration: the inference loop is already cheaper than the reference, and the finer the grid, the larger that margin becomes. A single from-scratch run does not yet break even, but closing the remaining gap no longer requires a faster rollout; it requires either better use of the trained $\mathcal{P}$ across runs or overlapping training with the live simulation to convert the available loop speedup into actual wall-time savings

However, it is important to stress that this trimming does not improve the physical fidelity of the run; the generalisation gap and the force errors it produces are unchanged. The speedup is recovered by stopping at the same accuracy sooner, not by reaching a better one.

As the generalisation gap in Figure 4.28 indicates, $\mathcal{P}$ will not generalise well outside of the training set, because the test loss of the AE sets the ceiling of the achievable accuracy of the acceleration framework. Where this becomes physically visible is the loading on the cylinder: a faithful velocity reconstruction is the prerequisite for the hybrid loop to reproduce the forces accurately, since the coefficients are computed from the decoded velocity $\hat{\mathbf{u}}$, so the near-body reconstruction error feeds directly into the pressure projection and the integrated force. At $Re = 5000$ this reconstruction is no longer faithful, and the consequence is shown in Figure 4.29, where the hybrid force coefficients are compared against the reference over the simulation window.

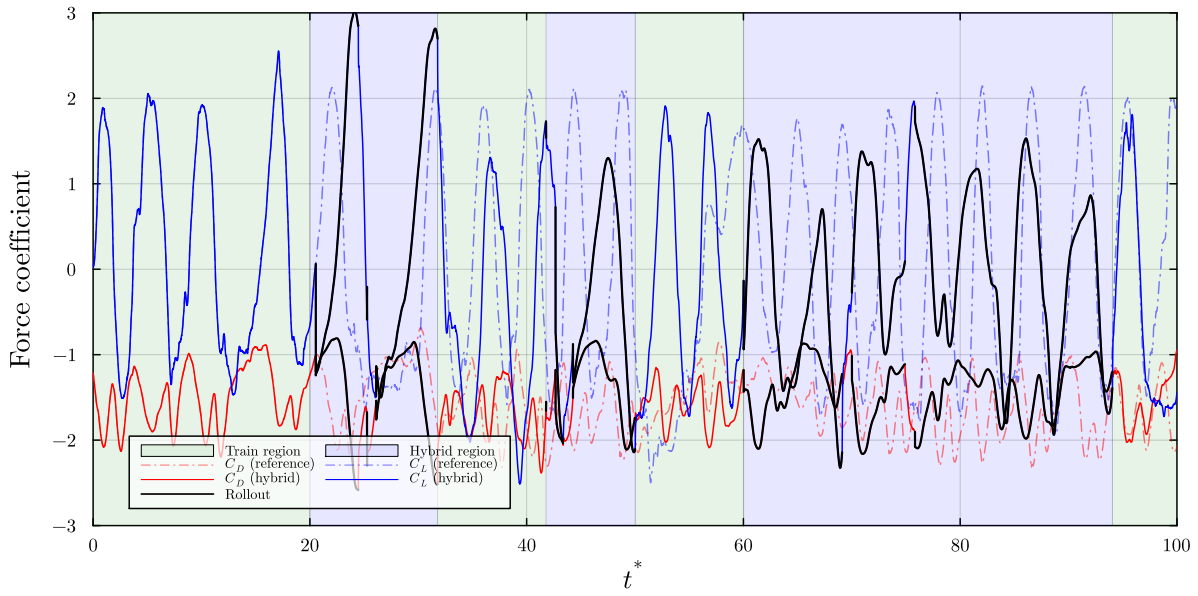
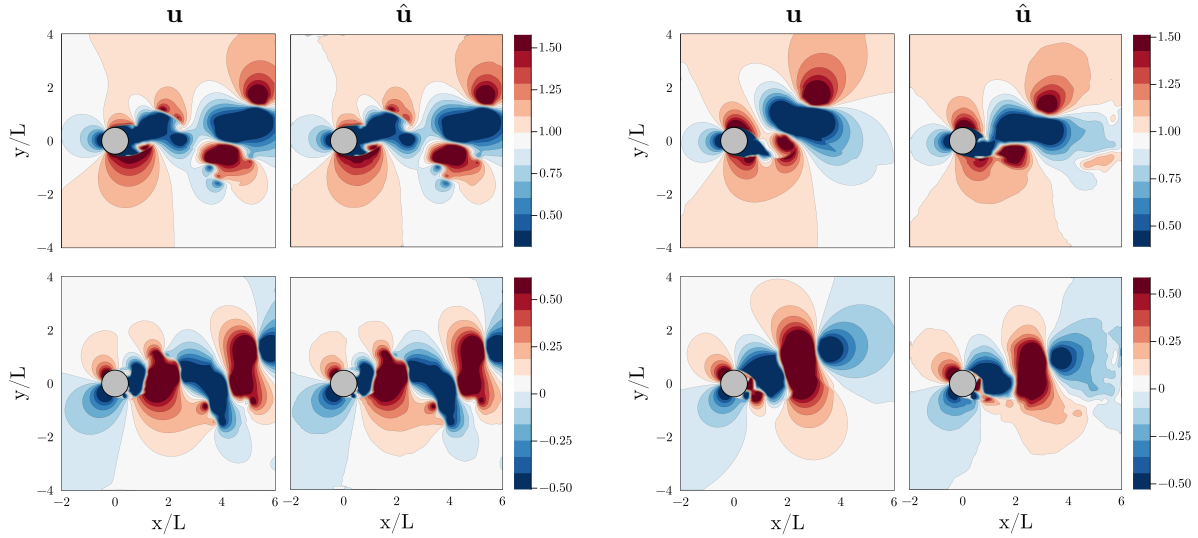



**Figure 4.29:** Lift ($C_L$) and drag ($C_D$) coefficients over $t^*$ for the $Re = 5000$, $512^2$ hybrid run. Reference (dash-dotted) against the hybrid loop (solid); the black trace is a single unaided rollout from $t^* = 0$ with no OOD resets. Green bands mark the training region, blue bands the hybrid regions. The mean drag is held across the window, but the lift loses both amplitude and shedding signature within the hybrid regions.

Figure 4.29 shows that the hybrid loop holds the time-mean drag but fails to reproduce the lift fluctuation: within the hybrid regions, the $C_L$ trace drifts in both amplitude and phase, and the unaided black rollout diverges from the reference entirely. This failure traces back to the near-body flow this regime produces. At $Re = 5000$, the vortices form and shed through a thin, sharply varying near-wall layer, visible in the closely spaced shear structures of Figure 4.27, and it is precisely these complex high-gradient near-body fluctuations that the autoencoder struggles to reconstruct. The more complex the near-body

fluctuations, the larger this error, so the lift, which integrates them directly, is the coefficient that suffers most.

The instantaneous reconstructions in Figure 4.30 show the same generalisation gap in physical space. The in-sample snapshot (a) retains the sharp shear-layer structure of the reference, whereas the held-out snapshot (b) is visibly smoothed: the near-wake vortices lose definition and the thin separating layers blur out. This is the spatial signature of the loss gap in Figure 4.28: the decoder reproduces the wake configurations it saw during training but regresses toward a smeared, lower-gradient field on states from the unseen window. Because the force coefficients are integrated from the decoded velocity, this near-body smoothing feeds directly into the loading, and it is the high-gradient near-wall layer, which the lift integrates most directly, that carries the amplitude and phase errors shown in Figure 4.29.




**(a)** In-sample snapshot

**(b)** Held out snapshot

**Figure 4.30:** Instantaneous velocity reconstruction by the autoencoder at the $Re = 5000$, $512^2$ operating point for (a) an in-sample snapshot ($t^* = 6.84$) and (b) a held-out snapshot ($t^* = 19.74$). Rows: streamwise $u$ (top), transverse $v$ (bottom); within each panel, left is the WaterLily reference $\mathbf{u}$, right the reconstruction $\hat{\mathbf{u}} = \psi_\theta(\varphi_\theta(\mathbf{u}))$. The held-out field is visibly smoothed, indicating poor generalisation beyond the training distribution.

Compared to the instantaneous reconstruction, the time-averaged statistics in Table 4.12b seem to paint a better picture; the corresponding mean-velocity and Reynolds-stress contour fields, with their reference and difference maps, are shown in Appendix C in Figure C.2 and Figure C.1. The mean velocity components and all three Reynolds-stress fields correlate with the reference at $\rho > 0.96$, with mean absolute errors no larger than $0.023$, and the time-mean drag is held to within $2.6\%$ over the full window. The interpretation is that $\mathcal{P}$ has learnt the statistically converged structure of the wake, even where it cannot track the instantaneous field: averaging over many shedding states recovers the low-order statistics, while the high-gradient near-body fluctuations that set the instantaneous loading are too complex to be learnt by $\mathcal{P}$. This is why the mean quantities come out well while the forces do not. The hybrid-region errors of $8.5\%$ in $\langle C_D \rangle$ and $9.2\%$ in $C_L^{\text{rms}}$ (Table 4.12a) are the cost of that same near-body error, and they are the honest figures here, since the smaller full-window errors are diluted by the reference segments. The integrated force metrics at this operating point, therefore, cannot be interpreted as quantitatively reliable, even though the time-averaged behaviour holds up.

| Quantity | Ref. | Hybrid | $e_{\text{abs}}$ | $e_{\text{rel}}$ [%] |
|---|---|---|---|---|
| *Full window* ($t^* \in [0, 100]$) | | | | |
| $\langle C_D \rangle$ | $-1.56768$ | $-1.52634$ | $0.04134$ | $2.637$ |
| $C_L^{\text{rms}}$ | $1.30170$ | $1.23656$ | $0.06515$ | $5.005$ |
| *Hybrid regions only* | | | | |
| $\langle C_D \rangle$ | $-1.57821$ | $-1.44368$ | $0.13452$ | $8.524$ |
| $C_L^{\text{rms}}$ | $1.29928$ | $1.17933$ | $0.11995$ | $9.232$ |

**(a)** Integrated force coefficients: time-mean drag $\langle C_D \rangle$ and r.m.s. lift $C_L^{\text{rms}}$ against the reference, with absolute and relative errors.

| Quantity | $\varepsilon$ (MAE) | $\rho$ |
|---|---|---|
| *Mean flow* | | |
| $\langle u \rangle$ | $0.023$ | $0.9773$ |
| $\langle v \rangle$ | $0.012$ | $0.9761$ |
| *Reynolds stresses* | | |
| $\langle u'u' \rangle$ | $0.010$ | $0.9825$ |
| $\langle v'v' \rangle$ | $0.015$ | $0.9932$ |
| $\langle u'v' \rangle$ | $0.006$ | $0.9626$ |

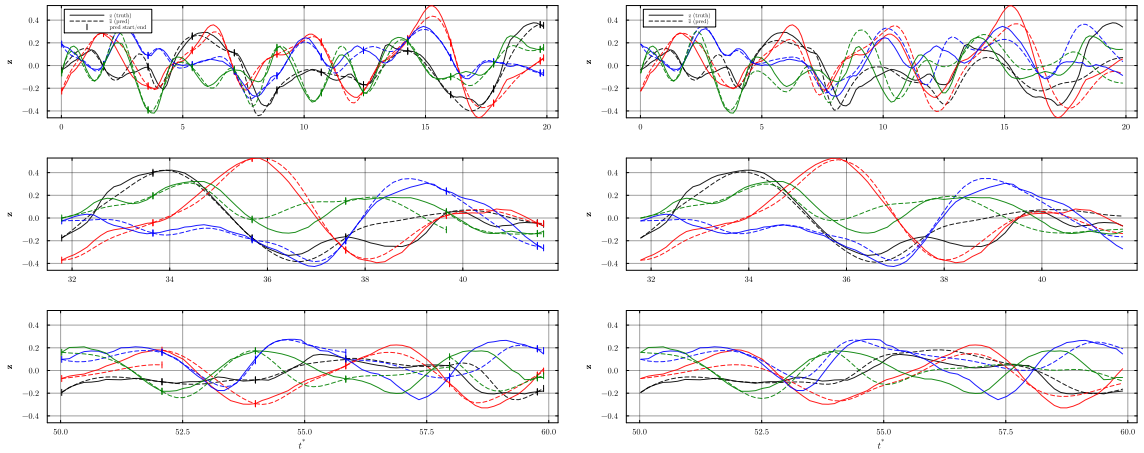
**(b)** Time-averaged field statistics over $\Omega$: mean absolute error $\varepsilon$ and spatial correlation $\rho$.

**Table 4.12:** Hybrid-loop accuracy at $Re = 5000$. (a) integrated loading; (b) mean-flow and Reynolds-stress fields.

All in all, the performance of $\mathcal{P}$ shows a split outcome at this operating point: the autoencoder reconstructs the wake states it has seen but smooths those drawn from the held-out window, so the instantaneous loading is unreliable when trying to predict outside of the training window; while the time-averaged velocity and Reynolds-stress fields remain faithful to the reference. The source of error at this operating point cannot be blamed on the autoencoder alone, since the loading is only as good as the decoded velocity field, and that field is in turn only as good as the latent trajectory the Neural ODE supplies, so the reconstruction ceiling set here does not yet distinguish error baked in by the autoencoder from drift accumulated during the latent integration.

### Latent Dynamics

The latent dynamics inherit the underlying wake's chaos. Figure 4.31 shows that the reference latent components carry no clean periodicity: unlike Figure 4.25, where each component traces a near-sinusoidal orbit at the shedding frequency, the $Re = 5000$ trajectories are irregular in both amplitude and phase, with no simple frequency to which the components return. The chaos that hampered the instantaneous velocity reconstruction is therefore already present in the latent state $\mathbf{z} = \varphi_\theta(\mathbf{u})$ itself.



**(a)** Converged multiple-shooting fit on the augmented trajectory set; markers denote shooting-segment boundaries.

**(b)** Single rollout from $t^* = 0$ across both windows after the second retraining, no segment resets.

**Figure 4.31:** Latent dynamics of $f_\phi$ of the $Re = 5000$, $512^2$ operating point, four representative components shown over all time windows. Reference $z$ (solid) against prediction $\tilde{z}$ (dashed). (a) Converged multiple-shooting fit on the augmented trajectory set, with segment start/end markers; (b) unaided single rollout from $t^* = 0$ over the same windows.

Against this reference, $f_\phi$ recovers the latent dynamics reasonably well under multiple shooting but not during free rollout. In Figure 4.31a, the converged multiple-shooting fit tracks the reference closely within each shooting segment, but this good visual performance is mainly due to the multiple-shooting training: each segment is re-anchored to the reference latent state at its start (the segment markers). The single rollout from $t^* = 0$ in Figure 4.31b is the unflattering test, and there the prediction $\hat{\mathbf{z}}$ drifts away from the reference within a few windows. The single-shot rollouts track the learned frequencies to some extent but diverge at the end of each trajectory window. These unguarded rollouts show that

$f_\phi$ has learnt the dynamics of the latent state, but due to the chaotic nature of the rollout, rollout drift can set in quickly. This is similar to the way the force rollout of Figure 4.29 also drifts to unknown states. The latent divergence and the force divergence are therefore connected behaviours.

The chaotic nature of the latent dynamics is what loosens the KNN monitor. Because the latent trajectories never settle onto a tight orbit, the training set populates a broad, high-variance region of latent space, and the KNN gating threshold $\tau$ derived from it is correspondingly large; it also climbs across the retraining stages (Table 4.13), from $\tau = 0.614$ for the initial model to $0.898$ and $0.902$ after the first and second retrains, as each augmented training set widens the accepted region further. A large $\tau$ is a lenient gate: a rollout must drift far before its score crosses the threshold and triggers a reset.

| # | NODE model | Steps | $\Delta t^*$ | $t^*_{\text{r, end}}$ | score | $\tau$ |
|---|---|---|---|---|---|---|
| 1 | initial | 717 | 4.470 | 24.470 | 0.615 | 0.614 |
| 2 | initial | 1 | 0.811 | 25.281 | 0.855 | 0.614 |
| 3 | initial | 1035 | 6.502 | 31.783 | 0.615 | 0.614 |
| 4 | 1st retrain | 2 | 10.021 | 41.804 | 0.899 | 0.898 |
| 5 | 1st retrain | 126 | 1.556 | 43.360 | 0.899 | 0.898 |
| 6 | 1st retrain | 1049 | 6.656 | 50.017 | 0.898 | 0.898 |
| 7 | 2nd retrain | 1661 | 19.094 | 69.110 | 0.903 | 0.902 |
| 8 | 2nd retrain | 895 | 5.802 | 74.912 | 0.903 | 0.902 |
| 9 | 2nd retrain | 3329 | 19.130 | 94.042 | 0.904 | 0.902 |

**Table 4.13:** Per-rollout summary of the KNN-gated NODE inference at $Re = 5000$. Every accepted rollout terminates just above the active threshold $\tau$.

The consequence is visible in the `Steps` and $\Delta t^*$ columns, where accepted rollouts are allowed to run for hundreds to thousands of integration steps, the longest covering $\Delta t^* \approx 19$ time units in a single uninterrupted rollout. This long rollout at the end of the simulation window is, however, not entirely erroneous, as Figure 4.29 shows; it still traces the reference, but it is significantly more diffuse than the reference, showing how the rollout matches one of the characteristic frequencies of the shedding regime, indicating that $f_\phi$ has managed to learn this specific flow behaviour. Which is exactly what Figure 4.29 also indicated with the matching, but diverging, unguarded rollouts

The latent dynamics at the $Re = 5000$, $512^2$ operating point expose a contradiction in the KNN gating strategy. The chaotic latent space sets $\tau$ from the spread of the training distribution, so the threshold scales with how disordered that distribution is. This means that a well-performing autoencoder that produces a clean, near-periodic wake yields a tight latent cluster, a small $\tau$, and a strict gate that resets often, thereby capping rollout length and limiting the achievable speedup, as in Section 4.4.1. The chaotic $Re = 5000$ wake does the opposite: its complex latent distribution inflates $\tau$, the gate turns lenient, and rollouts are allowed to run for thousands of steps, exactly the regime in which $f_\phi$ accumulates the most drift. The monitor is therefore most permissive where the dynamics are hardest to predict and most restrictive where they are easiest to predict, which is the wrong way round. This is not a tuning error but a consequence of deriving the threshold from in-distribution variance: the harder the flow is to learn, the more freedom the gate hands the part of the pipeline least able to use it.

# 5

# Discussion

The preceding chapter demonstrates that the proposed hybrid pipeline reproduces the statistical observables of the $Re = 2500$ cylinder wake: the time-averaged velocity field, the three independent Reynolds-stress components, and the integrated force coefficients, all with appropriate physical accuracy relative to the reference simulation. On their own, these results show that latent neural ODE rollouts can replace part of the conventional time integration without compromising the quantities that matter most to the flow. They do not, however, say whether the replacement was worth it. This is what the acceleration metrics $S_{\text{loop}}$ and $S_{\text{eff}}$ aim to show. This chapter aims to provide context for the acceleration and accuracy results reported in the previous section, to specify the conditions under which the reported speedup actually holds, and to clarify where it does not. Several of those results look favourable or unfavourable only because certain choices were kept fixed, such as the grid size and the hardware on which the pipeline was run. These choices are examined and discussed in turn to adequately assess the results of this thesis.

## 5.1. Interpreting the acceleration metrics

The speedup metrics constructed in this work are key indicators of the current framework's functionality. The hybrid-loop speedup for the base simulation $S_{\text{loop}} = T_{\text{wall}}^{\text{ref}}/T_{\text{wall}}^{\text{h, tot}} \approx 1.48$ measures the cost of hybrid integration *once the networks already exist*; the effective speedup $S_{\text{eff}} = T_{\text{wall}}^{\text{ref}}/(T_{\text{wall}}^{\text{h, tot}} + T_{\text{wall}}^{\text{h, training}}) \approx 0.07$ charges the workflow for building those networks. The gap between the two is the central pitfall of this work: for a single run over the integrated window, with two full retraining procedures (Table 4.5; the third update is data-only), the complete pipeline is roughly an order of magnitude *slower* than conventionally advancing the reference solver. The per-CTU rollout is some $84.6$ times cheaper than conventional time integration of a single CTU of CFD (Figure 4.1a), but this advantage is dwarfed by the fixed cost of training: the autoencoder and neural ODE together consume $682$ s, equivalent to roughly $1018$ CTU of reference integration, against the $100$ CTU actually simulated.

The disparity between the two metrics is itself informative. The loop speedup $S_{\text{loop}} > 1$ confirms that, once trained, $\mathcal{P}$ integrates the flow more cheaply per CTU than the reference solver; the effective speedup $S_{\text{eff}} < 1$ shows that this per-step speedup is still overshadowed by the cost of network training. These two metrics together do show that $\mathcal{P}$ holds the capacity for genuine end-to-end acceleration, but the realisation of it depends on shifting the balance between these costs, either by reducing the training expense through smarter network training regimes and better reuse of the pipeline, or by raising the cost per CTU of the reference simulation so that each replaced step is worth more.

Generalisation experiments act on the two factors that set $S_{\text{loop}}$ separately: the cost of each replaced step and the number of steps a rollout replaces. A finer grid raises the per-CTU cost of the memory-bound reference solver [98] while leaving the latent-rollout cost fixed, so each replaced step is worth more and the $S_{\text{loop}}$ ceiling rises. The change in Reynolds number changes the number of steps that get replaced by $\mathcal{P}$, but in an inverse way: lower $Re$ yields smoother, lower-dimensional dynamics that $f_\phi$ could in principle follow for longer, yet the realised rollout length is governed by the KNN gate rather than

by $f_\phi$ alone. An increase in Reynolds number makes the latent dynamics more complex, making them harder for $f_\phi$ to learn and leading to increased drift. But the KNN monitor adopts a more permissive gate due to increased variance in the latent training set, which enables longer rollouts and thus better acceleration performance.

The honest statement of the results of the developed framework is therefore conditional: the hybrid loop is faster than CFD per step, but the workflow as a whole would only serve as a true acceleration method for longer, more expensive or repeated simulation windows. This conditionality also has an upside. Because the rollout cost is fixed by the latent dimension while the cost of a CFD step grows with the simulation size, the per-CTU cost margin widens as the reference solver becomes more expensive, as demonstrated in the finer-mesh case of Section 4.4.2. The regimes in which $\mathcal{P}$ is worth its training cost are therefore precisely the more demanding regimes, where the CFD step is costly enough that saving and replacing it repays the training overhead more quickly. Given that the KNN monitor is adequately fit to the training set, this is discussed further in Section 5.5.

## 5.2. The role of hardware in the reported gain

A second important discussion note is hardware-related. With the exception of the autoencoder training, every component in $\mathcal{P}$ (the WaterLily reference, the warmup and stabilisation CFD steps, the neural ODE training, and the latent rollout) is executed on CPU. This means that the $S_{\text{loop}}$ is a clean CPU-versus-CPU comparison on identical hardware, which is appropriate for isolating the algorithmic gain. However, since WaterLily is a backend-agnostic solver, it can run on GPU architectures, where its benchmarks report approximately 2 orders of magnitude of speedup on GPU relative to the serial CPU execution [98].

If the reference had been run on a GPU, the per-CTU CFD cost would fall while the cost of a latent rollout would remain the same, eroding the performance of $S_{\text{loop}}$. This erosion is, however, strongly grid-dependent: the $\sim 2$ orders of magnitude reported in [98] are an asymptotic figure, reached only once the GPU memory is completely utilised at large grids, whereas the present $256^2$ case ( $\sim 0.07$ M DOF) lies below the smallest benchmarked grid and would see a considerably smaller GPU advantage. Even at this reduced factor, because the rollout cost is fixed, GPU execution of the reference would push $S_{\text{loop}}$ towards, and plausibly below, break-even on equal hardware. The GPU comparison should therefore be read as the most demanding case the hybrid loop can be asked to handle, not as a representative one.

Additionally, the accelerated workflow was deliberately kept on the CPU, mainly because training $f_\phi$ is not expected to benefit greatly from GPU acceleration: the network is small, and the cost is dominated by the sequential ODE solve and the reverse-mode differentiation through it, which parallelise poorly across the latent batch. This means that for GPU operation of $\mathcal{P}$, data would constantly need to be transferred between the CPU and GPU memory, adding an overhead to $\mathcal{P}$ that would spoil the maximal acceleration that it could possibly achieve.

However, this does not invalidate the approach. WaterLily was selected as the development vehicle for this framework precisely because it has differentiable and backend-agnostic properties that enable the coupling described in Section 3.7.3. Many of the CFD solvers used in practice, however, are not yet able to utilise GPU architectures; for these, CPU execution would be the only path to acceleration, as described in this work. The GPU comparison above should therefore serve as the most demanding case the hybrid loop can be asked to face, rather than the representative one: in any setting where the reference solver is CPU-bound by construction, the per-step gain reported through $S_{\text{loop}}$ is the gain that genuinely applies, and WaterLily serves here only as a convenient and well-characterised stand-in for a broader class of solvers.

## 5.3. Sensitivity across parameter space

The developed framework exposes a large number of coupled hyperparameters, and the various runs in Section A.2 varied only a small number of them around a single baseline. The autoencoder alone introduces a large number of parameters that could influence the eventual solution, more than those addressed in this work (kernel size, activation functions, optimisation algorithm, etc.). The addition of the neural ODE does not make the problem easier; it adds a whole new set of parameters that

are coupled with those of the autoencoder. This coupling is the central difficulty. The latent geometry that $f_\phi$ must learn is itself produced by $\varphi_\theta$, so a change to the autoencoder (a wider bottleneck $N_z$, a different physics weighting $(\lambda_{\text{div}}, \lambda_{\text{curl}})$, or simply a longer training budget) reshapes the very trajectories $\boldsymbol{\mathcal{T}}_z$ on which the multiple-shooting loss, the group size $n_s$, and the continuity weight $\lambda_c$ act. The KNN monitor sits one step further downstream of this: its threshold $\tau$ and neighbour count $k$ are calibrated on $\boldsymbol{\mathcal{T}}_z$, so any shift in the latent representation propagates through to the OOD decision and, with it, the rollout length that ultimately governs $S_{\text{loop}}$. The parameters of $\mathcal{P}$, therefore, cannot be regarded as independent dials; an optimum found for one component is conditional on the state of the others.

The sensitivity study of Section 4.3 respects this only partially. The one-factor-at-a-time (OFAT) procedure cannot reveal the complex interactions between the parameters of $\mathcal{P}$. A full accounting of parameter interactions would require sampling the entire hyperparameter space, and the number of pipeline trainings this demands grows exponentially with the parameter count, which places it beyond the computational scope of this work. The results reported here should accordingly be interpreted as a marginal sensitivity study around a single operating point, not as a converged optimum. Meaning that the framework should be understood as one whose accessible performance is, in all likelihood, understated because the true optimal performance is not yet found.

A further consequence of this coupling is visible in the comparison metrics themselves. The configurations that produced the largest $S_{\text{loop}}$ were not the most accurate but the least: the smallest bottleneck ($N_z = 8$) and the most lightly trained autoencoder both extended the average rollout, because an under-resolved encoder yields a lower-complexity latent signal that $f_\phi$ is able to keep within the training distribution for longer before the KNN monitor flags it. The same configurations were the worst at recovering the second-order statistics, with the shear stress $\langle u'v' \rangle$ and the transverse normal stress $\langle v'v' \rangle$ degrading most. Acceleration and statistical fidelity, therefore, pull against one another, and $S_{\text{loop}}$ on its own is a misleading objective: maximising it rewards discarding precisely the turbulent information $\mathcal{P}$ wants to preserve. The appropriate target is the trade-off between the two, and the operating point on that curve is dictated by whether acceleration or statistical accuracy is the priority for a given application, rather than by either metric alone.

## 5.4. Generalisation to the $Re = 5000$ regime

The $Re = 5000$, $512^2$ case (Section 4.4.2) tests $\mathcal{P}$ on a simultaneously more expensive and more demanding operating point. The two changes act on different parts of the framework but were applied together: the grid refinement raises the per-CTU cost of the memory-bound reference solver while leaving the rollout cost fixed, and the higher Reynolds number drives the wake from quasi-periodic shedding into a more complex state.

On cost, the grid refinement behaves as the memory-bound argument predicts: the reference rises to $4842\,\text{ms}$/CTU against $866\,\text{ms}$/CTU at $256^2$ while the rollout stays fixed at $\sim 48\,\text{ms}$/CTU, lifting $S_{\text{loop}}$ to $2.07$ and $S_{\text{eff}}$ to $\approx 0.19$ against $0.073$ at the base point, so the economic claim of Section 5.1 strengthens here. On fidelity, the same run exposes the limit of the compression: the autoencoder no longer generalises beyond its training window (validation and test losses of $9.6 \times 10^{-3}$ and $8.9 \times 10^{-2}$, Figure 4.28), and the reconstruction cannot reconstruct the complex, high-gradient, near-wall layer that get integrated to the lift forces (Figure 4.29). The favourable $S_{\text{loop}}$ and the degraded forces are two faces of one run, where the reconstruction, not the rollout, becomes the binding error source.

Compounding the poor generalisation outside the training window, the encoder's latent dynamics mimic the chaotic nature of the wake. This makes the training of $f_\phi$ harder due to the absence of the clean, near-periodic shedding signal that the $Re = 2500$ latent trajectory offered the network to lock onto. Where the quasi-periodic base wake gave $f_\phi$ a low-dimensional, repetitive target it could follow well beyond the training horizon, the $Re = 5000$ latent trajectory is highly chaotic, so the compact network has no stable structure to extrapolate, and the rollout diverges from the reference sooner. This is where the regime begins to expose the limit of the neural ODE itself: a more chaotic flow yields latent dynamics whose predictability, rather than the width of the bottleneck encoding them, sets how far $f_\phi$ can be trusted to advance the state.

## 5.5. Integration Monitor

The generalisation to $Re = 250$ and $Re = 5000$ expose a structural flaw in the KNN gate that is invisible at the base point. The threshold $\tau$ is set as a quantile of the in-distribution nearest-neighbour distances of $\mathcal{T}_z$, so it scales with how dispersed the training distribution is. The well-resolved $Re = 250$ wake is cleanly periodic and low-variance: its latent cluster is tight, $\tau \approx 0.07$, and the gate is so strict that every rollout is clipped just past the threshold and reset, holding the realised speedup below what $f_\phi$'s near-perfect fit could otherwise sustain (Section 4.4.1). The badly-resolved $Re = 5000$ wake does the opposite: its chaotic latent distribution inflates $\tau$ from $0.614$ to $0.902$ across the retraining stages, the gate turns lenient, and rollouts are allowed to run for thousands of steps, exactly where $f_\phi$ accumulates the most drift (Section 4.4.2).

The integration monitor is therefore more permissive where the dynamics are hardest to learn and predict, and most restrictive where they are easiest, which is the wrong way around. Since the acceleration framework must preserve accuracy, it should penalise incorrect rollouts in the regimes where the dynamics are most complex, where prediction errors are made quickly. This is not a tuning error but a direct consequence of deriving the trigger from training-set variance: the harder the flow is to learn, the wider its latent distribution, and the more latitude the gate gives to the part of the pipeline least able to use it. The deeper issue is that the KNN score measures proximity to $\mathcal{T}_z$ in latent space, not the physical correctness of the decoded field. A latent point can sit comfortably inside a broad cluster while decoding to a physically wrong field, as at $Re = 5000$, and can drift marginally past a tight $\tau$ while decoding to a perfectly acceptable field, as at $Re = 250$.

Among these generalisation cases, the base case occupies the middle ground. The $Re = 2500$, $256^2$ latent distribution carries enough variance that $\tau$ gives the rollout a useful margin to drift, rather than clipping a well-fitted rollout the moment it leaves the training cloud as at $Re = 250$; while the latent space stays well-resolved enough that a genuinely unseen wake state still falls outside the cluster and crosses the threshold before the decoded field becomes unphysical, unlike at $Re = 5000$. The monitor behaves as intended here, but this is a property of the operating point, not the gate: the base-case variance happens to land in the narrow window where a threshold read off the training spread coincides with a physically sensible one.

Circling back to 3.6.1, the choice of the KNN score was deliberate, and in hindsight double-edged. It was selected precisely because it imposes no distributional assumption on the latent states: since the encoder is retrained per flow and yields a different latent geometry each time, any assumed parametric distribution would not transfer across cases, whereas a non-parametric neighbour distance adapts to whatever cluster the training data happens to form [90]. This generalisability leads to inverse gating behaviour over $Re$ variation, because a threshold read off the latent trajectory's own spread inherits the spread's dependence on the flow. A monitor gated on a physical residual instead, such as the decoded-field divergence $\langle\, |\hat{u}_{i,i}| \,\rangle$, would not share this defect: it judges every rollout against the same physical tolerance regardless of latent geometry, imposing accuracy directly rather than by proxy. This route was not chosen in this work due to the cost of decoding and evaluating a residual in physical space at every step, which is almost as expensive as advancing WaterLily itself, which would erase the speedup the rollout exists to provide, so a practical physical monitor hinges on finding a residual proxy cheap enough to check often.

# 6

# Conclusion

This work set out to lower the computational cost of unsteady CFD simulations by replacing the most expensive part of the solver loop, the time integration of the discretised Navier–Stokes equations, with a less costly latent integration. The central research question asked how effectively a Neural ODE can accelerate this step without compromising accuracy. To answer it, a hybrid framework was built around the WaterLily solver: a convolutional autoencoder compresses the augmented flow state $(\mathbf{u}, \boldsymbol{\mu}_0^\epsilon)$ into a latent vector, a Neural ODE $f_\phi$ advances that state in time, and a KNN out-of-distribution monitor (Section 3.6.1) decides at every step whether the latent rollout can be trusted or whether control is handed back to the conventional solver. The framework was developed and tested on unsteady two-dimensional flow past a circular cylinder at $Re = 2500$.

At the base operating point, a $256^2$ grid with latent dimension $N_z = 16$ and multiple-shooting group $n_s = 15$, the hybrid loop behaved as intended: it interleaved cheap latent rollouts with conventional CFD steps, with the KNN monitor accepting a rollout until drift exceeded its reference threshold, at which point WaterLily resumed to stabilise the flow. This produced an inference-loop speedup of $S_{\text{loop}} = 1.48$ (Section 4.2.2), confirming that, once trained, the latent rollout integrates the flow more cheaply per step than the reference solver. The same base case also exposed the central limitation of the workflow: once the one-off cost of training the networks was charged to the comparison, the effective speedup fell to $S_{\text{eff}} = 0.07$, an order of magnitude slower than simply running WaterLily, because that training cost was never repaid within a single short simulation window. The disparity between the two metrics is discussed in detail in Section 5.1.

All the parameter studies pointed to the same underlying lever. Narrowing the latent bottleneck, shortening the autoencoder training schedule, or tightening the multiple-shooting group each simplified the dynamics the Neural ODE had to learn, thereby lengthening the average rollout and raising $S_{\text{loop}}$, but at the cost of coarsening the reconstructed turbulence statistics. Speed and physical fidelity are therefore traded against one another. Various methods influence this trade, either directly through a stricter rollout monitor, which gates how far a rollout is allowed to drift before control is handed back, or indirectly through the autoencoder and the latent dimension, which set how much dynamical detail the compression retains and therefore how long $f_\phi$ can stay in distribution before being flagged.

The base-case economics showed that the performance of $\mathcal{P}$ is tied to the operating point rather than to the method itself, and the two generalisation experiments move it in opposite directions. Lowering the Reynolds number to $Re = 250$ produced a clean, low-variance wake that $f_\phi$ reconstructs almost perfectly; the realised speedup, however, is not set by that fit but by the KNN monitor, whose tight latent distribution forces a strict threshold $\tau$ that clips each rollout shortly after it begins and holds the achievable acceleration below what $f_\phi$ could otherwise sustain (Section 4.4.1). Raising the Reynolds number to $Re = 5000$ on a refined $512^2$ grid acts through the cost of each replaced step instead: because WaterLily is memory-bound while the latent-rollout cost is fixed by $N_z$, every step the rollout replaces is worth more at higher resolution, lifting $S_{\text{eff}}$ from the base-case $0.07$ to $0.19$ (Section 4.4.2). That same regime, however, exposes the fundamental limitation of the workflow: the autoencoder no

longer generalises beyond its training window, so the decoded near-body field is not predicted correctly, and the integrated loading becomes unreliable in hybrid regions even as the speedup improves. The favourable economics and the degraded forces are two faces of the same coin.

## 6.1. Answering the Research Question

**How effectively can neural ordinary differential equations accelerate the time integration step in CFD solvers for unsteady flow simulations without compromising accuracy?**

The central research question can be answered from two perspectives. The developed framework behaves differently depending on the time window used to assess its performance. When assessing the neural ODE rollout per step, the answer is a clear yes: once $\mathcal{P}$ is trained, the latent rollout integrates the flow more cheaply than the reference solver. The accuracy of the time integration is held not by the Neural ODE being uniformly accurate, but by the KNN monitor returning control to the solver whenever a rollout drifts past its threshold. In that sense, the framework accelerates time integration *and* maintains controllable accuracy, which is what the research question sought.

Assessing the total cost of the framework is conditional and provides a more honest view of the acceleration scheme. At the $256^2$ base case, the effective speedup was $S_{\text{eff}} = 0.07$, an order of magnitude slower than the reference solver once the network training was charged in the hybrid wall time. This gap is not a property of the developed method but of the operating point: because the rollout cost is fixed while a WaterLily step becomes more expensive with grid size, a costlier reference narrows it, with the $Re = 5000$, $512^2$ case raising $S_{\text{eff}}$ to $0.19$ without yet reaching break-even (Section 4.4.2). At that point, however, the limiting factor shifts from cost to fidelity: the autoencoder's reconstruction ceiling, rather than the rollout, becomes the dominant source of error. The Neural ODE coupled with the autoencoder is therefore an effective accelerator in regimes where a conventional CFD step is sufficiently costly to justify learning to replace it, provided that the compression can still represent the flow at that operating point.

### 6.1.1. Physical accuracy across regimes, Reynolds numbers and geometries

**How well does the neural ODE acceleration framework preserve the flow's statistical observables when transferred across Reynolds numbers and grid resolution?**

Across the regimes tested, the domain-wide statistical observables transfer robustly, while the instantaneous loading does not always behave the same way. At the base $Re = 2500$, $256^2$ point the mean-velocity and Reynolds-stress fields match the reference at $\rho \geq 0.99$, and the shedding frequency is recovered to within the cycle-to-cycle spread ($St_{\text{hyb}} = 0.233$ against $St_{\text{ref}} = 0.236$); the clean $Re = 250$ wake is reconstructed equally faithfully; and even at the demanding $Re = 5000$, $512^2$ point the mean flow and all three Reynolds stresses still correlate at $\rho > 0.96$ with mean absolute errors no larger than $0.023$. The averaged fields survive the transfer because integrating over many shedding states recovers the low-order structure of the wake.

The two axes act on the framework in opposite directions. Refining the grid is the minor change: at $512^2$ the decoder's cell-width smoothing spans half the physical width compared to the base case, so the thin separating shear layers are resolved by more cells and blurred less, and the autoencoder reaches a lower in-sample loss and converges faster on the train and validation splits than at $256^2$. Raising the Reynolds number is where the framework begins to struggle: the more chaotic wake presents near-body states that the short training window hasn't learned yet, so the held-out loss widens into a generalisation gap that the grid refinement cannot close. Since both changes were applied together at $Re = 5000$, the deployed fidelity is set by the Reynolds number, not the resolution.

This is what limits the integrated forces. The lift fluctuations show this the best, with hybrid-region $C_L^{\text{rms}}$ errors of order $9$–$11\%$, and at $Re = 5000$ the mean drag degrades with it ($8.5\%$ in the hybrid regions), because the coefficients integrate the high-gradient near-body field that the autoencoder struggles to learn once the wake is too chaotic for a short training window. The binding constraint on transfer is therefore the autoencoder's reconstruction ceiling, set by the Reynolds number rather than the grid; the rollout and the monitor sit downstream of the encoder and thus inherit its error, but are not the leading cause of statistical force error in the acceleration regime.

### 6.1.2. Sensitivity to architecture and training data quality

**How sensitive is the solution accuracy of neural ODEs to network architecture and training data quality?**

The method is bound by a plethora of parameters that influence its accuracy and behaviour, so a dedicated sensitivity study is a logical next step; current knowledge is limited to one-at-a-time sweeps around a single baseline. What those sweeps did make clear is that what matters most is how well the current simulation state is represented in the training trajectories $\mathcal{T}_z$. Since $f_\phi$ and the KNN monitor both draw their logic from $\mathcal{T}_z$, the rollout runs long when the active regime overlaps the training distribution and is flagged almost immediately when it does not. Rollout length is therefore a direct reflection of that overlap, and accuracy depends less on raw data volume than on whether the data covers the state the simulation currently occupies.

With respect to sensitivity to architecture, the framework is most sensitive to whatever determines the complexity of the latent dynamics $f_\phi$ must learn. For example, the latent dimension size $N_z$. A narrower bottleneck simplifies those dynamics and allows for longer rollouts, but discards wake detail, thereby polluting the reconstructed statistics, whereas a wider bottleneck recovers fidelity at the cost of acceleration. Architecture, therefore, does not have a single optimum but sets the position on the speed–fidelity trade, which is why a joint sweep is needed to locate it for a given accuracy target.

### 6.1.3. Controlling the physical integrity of the acceleration framework

**How can the quality of a neural ODE rollout be monitored at runtime to determine when its predictions can be trusted?**

The integration length and accuracy are controlled by the KNN out-of-distribution score: each predicted latent state is compared to its k nearest neighbour in $\mathcal{T}_z$ and control is returned to the solver once that distance crosses a quantile threshold $\tau$. As a runtime trust signal, the KNN gate has clear advantages: it is non-parametric and thus imposes no distributional assumption on the latent geometry, which is re-learnt per flow; it is cheap enough to evaluate at every step; and, at the base operating point, it gates the rollout at a physically sensible drift tolerance.

However, the generalisation across flow regimes has highlighted its disadvantages: because $\tau$ is determined by the spread of the training trajectory, it governs how close the rollout remains to the training distribution, thereby indirectly rather than directly controlling accuracy. This makes the gate scale with flow complexity rather than with error: the clean $Re = 250$ wake yields a tight cluster and a gate so strict it clips well-fitted rollouts, while the chaotic $Re = 5000$ wake inflates $\tau$ and admits rollouts that have already drifted into a physically wrong state (Section 5.5). The rollout can therefore be trusted whenever the latent variance is sufficiently moderate to allow the predicted trajectory to drift while remaining close to the reference trajectories, just as at the base operating point, but the KNN score alone cannot certify the rollout when that coincidence breaks down. An integration control that is independent of the flow case would perhaps better be constructed by something the latent distance does not see, either the physics of the decoded field or the drift of the latent distribution in terms of variance or standard deviation, as set out in Section 6.2.

## 6.2. Recommendations and Future Work

The methods developed in this work provide a promising basis for future work; the complex parameter landscape offers a wide range of interactions that could improve the acceleration performance of $\mathcal{P}$.

### 6.2.1. Training set and current simulation state correlation

The framework's retraining functionality is useful only when the current simulation state is well represented in the retraining data. When the simulation drifted into an uncovered regime, the new snapshots entered as a minority, the loss stayed dominated by the older dynamics, and the update barely shifted the fit. The number of new data points sampled should therefore not be fixed, but a function of the distributional distance between the current state and the training set, so that the networks receive a relevant update. In addition to flexible sampling, a study into whether to retrain on the augmented set or on the new trajectory alone could be relevant for further research.

### 6.2.2. Parameter Sensitivity analysis

The parameter studies were one-at-a-time sweeps with a single run per configuration. This exposed the dominant lever, but it cannot locate an optimum, especially since a single run on a chaotic wake carries large variance depending on where the monitor first flags. A joint sweep with repeated runs is recommended, as it would capture the parameter interactions that a one-at-a-time study misses and the genuine optimum of the speed–fidelity trade for a given accuracy target.

### 6.2.3. Transfer to 3D simulation

Since the unfavourable economics are a property of the operating point rather than the method, three-dimensional flow is the most promising extension: the per-step cost is much higher, and the available compression ratio is larger, so the margin, which has already widened from $256^2$ to $512^2$, should widen further. The catch is that a volumetric autoencoder increases the training cost, which already keeps $S_{\text{eff}}$ below unity. Whether the larger per-step advantage justifies the heavier training, ideally on a GPU backend (Section 5.2), could be an interesting subject for further research.

### 6.2.4. Geometric generalisation

Every result here was obtained on the circular cylinder, so accuracy across bluff, asymmetric or irregular geometries could further demonstrate the framework's generalisability. Conditioning the networks on the Reynolds number, so that one pipeline spans multiple regimes rather than a single one, would further reduce the training cost, which dominates the workflow, and move the method from a case-specific accelerator to a more general one.

### 6.2.5. Runtime integration monitoring

The generalisation experiments have shown that the integration monitor leaves performance on the table in terms of accuracy and acceleration. The KNN gate is a reasonable metric when the latent distribution carries enough variance to give a usable drift margin, but because $\tau$ is read off the spread of $\mathcal{T}_z$, it certifies latent proximity rather than physical correctness and gates inversely with flow complexity (Section 5.5).

To solutions could be tested: The first is a monitor gated on a physical residual of the decoded field, such as the mean absolute divergence $\langle\, |\hat{u}_{i,i}| \,\rangle$, which holds every rollout to the same physical tolerance regardless of latent geometry; given a scheme is developed alongside it that doesn't incur large reconstruction and computational costs in physical space. Secondly, a monitor the latent drift distributionally rather than through a single nearest-neighbour distance, tracking how the statistics of the rolled-out state, or of its rate of change, depart from the training trajectories to catch a coherent drift that a pointwise distance lets pass. A combination of these 2 approaches could also be tested, in which a cheap distributional check is backed by occasional physical residual verification.

# References


[1] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 2)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. 2024.

[2] Charu C. Aggarwal. "An Introduction to Neural Networks". In: *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0_1. URL: https://doi.org/10.1007/978-3-319-94463-0_1.

[3] E. Ajuria et al. "Towards a hybrid computational strategy based on deep learning for incompressible flows". In: *AIAA AVIATION 2020 FORUM*. Vol. 1 PartF. American Institute of Aeronautics and Astronautics Inc, AIAA, 2020, pp. 1–17. ISBN: 9781624105982. DOI: 10.2514/6.2020-3058.

[4] ANSYS, Inc. *Unleashing the Power of Multiple GPUs for CFD Simulations*. Accessed: 2025-06-17. ANSYS, Inc. 2023. URL: https://www.ansys.com/blog/unleashing-the-power-of-multiple-gpus-for-cfd-simulations.

[5] Jorge Bardina, Joel H. Ferziger, and William C. Reynolds. "Improved Subgrid-Scale Models for Large-Eddy Simulation". In: *AIAA Journal* 18.4 (1980). Seminal work proposing the scale-similarity SGS model for LES., pp. 1061–1068. DOI: 10.2514/3.50819. URL: https://doi.org/10.2514/3.50819.

[6] Andrea Beck, David Flad, and Claus Dieter Munz. "Deep neural networks for data-driven LES closure models". In: *Journal of Computational Physics* 398 (Dec. 2019). ISSN: 10902716. DOI: 10.1016/j.jcp.2019.108910.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 1st ed. Springer, 2007. ISBN: 0387310738.

[8] H.G. Bock and K.J. Plitt. "A Multiple Shooting Algorithm for Direct Solution of Optimal Control Problems*". In: *IFAC Proceedings Volumes* 17.2 (1984). 9th IFAC World Congress: A Bridge Between Control Science and Technology, Budapest, Hungary, 2-6 July 1984, pp. 1603–1608. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)61205-9. URL: https://www.sciencedirect.com/science/article/pii/S1474667017612059.

[9] Guido Boffetta and Robert E. Ecke. "Two-dimensional turbulence". In: *Annual Review of Fluid Mechanics* 44 (2011), pp. 427–451. ISSN: 00664189. DOI: 10.1146/annurev-fluid-120710-101240.

[10] Steven L Brunton, Bernd R Noack, and Petros Koumoutsakos. "Machine Learning for Fluid Mechanics". In: *Annual Review of Fluid Mechanics Downloaded from www.annualreviews.org. Guest* 7 (2020), p. 29. DOI: 10.1146/annurev-fluid-010719. URL: https://doi.org/10.1146/annurev-fluid-010719-.

[11] Andriy Burkov. *Machine Learning Engineering*. First edition. True Positive Inc., 2020. ISBN: 9781999579579. URL: http://www.mlebook.com/.

[12] Jiayi Cai et al. "Revisiting Tensor Basis Neural Network for Reynolds stress modeling: Application to plane channel and square duct flows". In: *Computers and Fluids* 275 (May 2024). ISSN: 00457930. DOI: 10.1016/j.compfluid.2024.106246.

[13] John Y. Chen. "GPU technology trends and future requirements". In: *2009 IEEE International Electron Devices Meeting (IEDM)*. 2009, pp. 1–6. DOI: 10.1109/IEDM.2009.5424433.

[14] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". In: (June 2018). URL: http://arxiv.org/abs/1806.07366.

[15] Ruilin Chen, Xiaowei Jin, and Hui Li. "A machine learning based solver for pressure Poisson equations". In: *Theoretical and Applied Mechanics Letters* 12 (5 Sept. 2022). ISSN: 20950349. DOI: 10.1016/j.taml.2022.100362.

[16] Eric Chillón et al. “Acceleration of an algebraic multigrid pressure solver using graph neural networks”. In: (June 2026). URL: http://arxiv.org/abs/2606.19251.

[17] Clay Mathematics Institute. *Millennium Prize Problems: Navier-Stokes Equation*. Online; accessed 6 June 2025. https://www.claymath.org/millennium/navier-stokes-equation. 2000.

[18] Gary Coleman and Richard Sandberg. “A Primer on Direct Numerical Simulation of Turbulence – Methods, Procedures and Guidelines”. In: (Mar. 2010).

[19] Thomas Daniel. “Machine learning for nonlinear model order reduction”. 2021UPSLM039. PhD thesis. 2021. URL: http://www.theses.fr/2021UPSLM039/document.

[20] James W. Deardorff. “A numerical study of three-dimensional turbulent channel flow at large Reynolds numbers”. In: *Journal of Fluid Mechanics* 41.2 (1970), pp. 453–480. DOI: 10.1017/S0022112070000691.

[21] James Donnelly, Alireza Daneshkhah, and Soroush Abolfathi. “Physics-informed neural networks as surrogate models of hydrodynamic simulators”. In: *Science of the Total Environment* 912 (Feb. 2024). ISSN: 18791026. DOI: 10.1016/j.scitotenv.2023.168814.

[22] Hamidreza Eivazi et al. “Physics-informed neural networks for solving Reynolds-averaged Navier–Stokes equations”. In: *Physics of Fluids* 34.7 (July 2022), p. 075117. ISSN: 1070-6631. DOI: 10.1063/5.0095270. eprint: https://pubs.aip.org/aip/pof/article-pdf/doi/10.1063/5.0095270/19816179/075117_1_online.pdf. URL: https://doi.org/10.1063/5.0095270.

[23] Hamidreza Eivazi et al. “Towards extraction of orthogonal and parsimonious non-linear modes from turbulent flows”. In: *Expert Systems with Applications* 202 (Sept. 2022). ISSN: 09574174. DOI: 10.1016/j.eswa.2022.117038.

[24] Massimo Germano et al. “A Dynamic Subgrid-Scale Eddy Viscosity Model”. In: *Physics of Fluids A* 3.7 (1991). Proposes the dynamic procedure for computing the Smagorinsky coefficient., pp. 1760–1765. DOI: 10.1063/1.857955. URL: https://doi.org/10.1063/1.857955.

[25] Paul Ghanem et al. *Learning Physics Informed Neural ODEs with Partial Measurements*. Tech. rep. 2025. URL: www.aaai.org.

[26] Sharath S. Girimaji. *Turbulence closure modeling with machine learning: a foundational physics perspective*. July 2024. DOI: 10.1088/1367-2630/ad6689.

[27] Andrew Glaws, Ryan King, and Michael Sprague. “Deep learning for in situ data compression of large turbulent flow simulations”. In: *Physical Review Fluids* 5 (11 Nov. 2020). ISSN: 2469990X. DOI: 10.1103/PhysRevFluids.5.114602.

[28] Myeong Seok Go, Jae Hyuk Lim, and Seungchul Lee. “Physics-informed neural network-based surrogate model for a virtual thermal sensor with real-time simulation”. In: *International Journal of Heat and Mass Transfer* 214 (Nov. 2023). ISSN: 00179310. DOI: 10.1016/j.ijheatmasstransfer.2023.124392.

[29] Xiaoxiao Guo, Wei Li, and Francesco Iorio. “Convolutional Neural Networks for Steady Flow Approximation”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 481–490. ISBN: 9781450342322. DOI: 10.1145/2939672.2939738. URL: https://doi.org/10.1145/2939672.2939738.

[30] Ioan Hadade and Luca di Mare. “Modern multicore and manycore architectures: Modelling, optimisation and benchmarking a multiblock CFD code”. In: *Computer Physics Communications* 205 (Aug. 2016), pp. 32–47. ISSN: 00104655. DOI: 10.1016/j.cpc.2016.04.006.

[31] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: (Dec. 2015). URL: http://arxiv.org/abs/1512.03385.

[32] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[33] Md Lokman Hosain and Rebei Bel Fdhila. “Literature Review of Accelerated CFD Simulation Methods towards Online Application”. In: *Energy Procedia*. Vol. 75. Elsevier Ltd, 2015, pp. 3307–3314. DOI: 10.1016/j.egypro.2015.07.714.

[34] Joongoo Jeon, Juhyeong Lee, and Sung Joong Kim. “Finite volume method network for the acceleration of unsteady computational fluid dynamics: Non-reacting and reacting flows”. In: *International Journal of Energy Research* 46 (8 June 2022), pp. 10770–10795. ISSN: 1099114X. DOI: 10.1002/er.7879.

[35] Joongoo Jeon et al. “Residual-based physics-informed transfer learning: A hybrid method for accelerating long-term CFD simulations via deep learning”. In: *International Journal of Heat and Mass Transfer* 220 (2024), p. 124900. ISSN: 0017-9310. DOI: https://doi.org/10.1016/j.ijheatmasstransfer.2023.124900. URL: https://www.sciencedirect.com/science/article/pii/S0017931023010451.

[36] Dmitrii Kochkov et al. “Machine learning-accelerated computational fluid dynamics”. In: (2021). DOI: 10.1073/pnas.2101784118/-/DCSupplemental.y.

[37] A. Kolmogorov. “The Local Structure of Turbulence in Incompressible Viscous Fluid for Very Large Reynolds’ Numbers”. In: *Akademiia Nauk SSSR Doklady* 30 (Jan. 1941), pp. 301–305.

[38] Marius Kurz, Philipp Offenhäuser, and Andrea Beck. “Deep reinforcement learning for turbulence modeling in large eddy simulations”. In: *International Journal of Heat and Fluid Flow* 99 (Feb. 2023). ISSN: 0142727X. DOI: 10.1016/j.ijheatfluidflow.2022.109094.

[39] B. E. Launder and D. B. Spalding. “The numerical computation of turbulent flows”. In: *Computer Methods in Applied Mechanics and Engineering* 3 (2 Mar. 1974), pp. 269–289. ISSN: 0045-7825. DOI: 10.1016/0045-7825(74)90029-2.

[40] Brian E Launder, GJ Reece, and Wolfgang Rodi. “Progress in the development of a Reynolds-stress turbulence closure”. In: *Journal of Fluid Mechanics* 68.3 (1975), pp. 537–566.

[41] Kimin Lee et al. *A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks*. 2018. arXiv: 1807.03888 [stat.ML]. URL: https://arxiv.org/abs/1807.03888.

[42] Sangseung Lee and Donghyun You. “Data-driven prediction of unsteady flow over a circular cylinder using deep learning”. In: *Journal of Fluid Mechanics* 879 (2019), pp. 217–254. DOI: 10.1017/jfm.2019.700.

[43] D. K. Lilly. “A Proposed Modification of the Germano Subgrid-Scale Closure Method”. In: *Physics of Fluids A* 4.3 (1992). Refines Germano’s dynamic model by introducing least-squares minimization to stabilize $C_s$., pp. 633–635. DOI: 10.1063/1.858280. URL: https://doi.org/10.1063/1.858280.

[44] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. “Reynolds averaged turbulence modelling using deep neural networks with embedded invariance”. In: *Journal of Fluid Mechanics* 807 (2016), pp. 155–166. DOI: 10.1017/jfm.2016.615.

[45] Phillip Lippe et al. “PDE-Refiner: Achieving Accurate Long Rollouts with Neural PDE Solvers”. In: (Oct. 2023). URL: http://arxiv.org/abs/2308.05732.

[46] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: (Jan. 2019). URL: http://arxiv.org/abs/1711.05101.

[47] Lu Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3 (3 Mar. 2021), pp. 218–229. ISSN: 25225839. DOI: 10.1038/s42256-021-00302-5.

[48] Mengxue Lu et al. “Physics-assisted multi-scale convolutional autoencoder for turbulence reduced-order modeling”. In: *Physics of Fluids* 37 (1 Jan. 2025). ISSN: 10897666. DOI: 10.1063/5.0248929.

[49] Audrey P. Maertens and Gabriel D. Weymouth. “Accurate Cartesian-grid simulations of near-body flows at intermediate Reynolds numbers”. In: *Computer Methods in Applied Mechanics and Engineering* 283 (Jan. 2015), pp. 106–129. ISSN: 00457825. DOI: 10.1016/j.cma.2014.09.007.

[50] Zhiping Mao, Ameya D. Jagtap, and George Em Karniadakis. “Physics-informed neural networks for high-speed flows”. In: *Computer Methods in Applied Mechanics and Engineering* 360 (Mar. 2020). ISSN: 00457825. DOI: 10.1016/j.cma.2019.112789.

[51] L. G. Margolin, W. J. Rider, and F. F. Grinstein. “Modeling turbulent flow with implicit LES”. In: *Journal of Turbulence* 7 (2006), pp. 1–27. ISSN: 14685248. DOI: 10.1080/14685240500331595.

[52] Igor L. Markov. *Limits on fundamental limits to computation*. Aug. 2014. DOI: 10.1038/nature13570.

[53] Arvind Mohan et al. “Compressed Convolutional LSTM: An Efficient Deep Learning framework to Model High Fidelity 3D Turbulence”. In: (Mar. 2019). URL: http://arxiv.org/abs/1903.00033.

[54] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (Apr. 1965). Also known as ”Moore’s Law” paper, pp. 114–117.

[55] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics : An Advanced Introduction with OpenFOAM® and Matlab*. Vol. 113. Fluid Mechanics and its Applications. Cham: Springer, 2016. ISBN: 978-3-319-16873-9. DOI: 10.1007/978-3-319-16874-6.

[56] Ashish S. Nair et al. “Understanding Latent Timescales in Neural Ordinary Differential Equation Models for Advection-Dominated Dynamical Systems”. In: (Mar. 2024). URL: http://arxiv.org/abs/2403.02224.

[57] NashTech Global. *Getting Familiar with Activation Function and Its Types*. https://blog.nashtechglobal.com/getting-familiar-with-activation-function-and-its-types/. Accessed: 2025-07-01. 2023.

[58] F Nicoud and F Ducros. *Subgrid-Scale Stress Modelling Based on the Square of the Velocity Gradient Tensor*. Tech. rep. 1999, pp. 183–200.

[59] FTM Nieuwstadt, Bendiks Jan Boersma, and Jerry Westerweel. *Turbulence: Introduction to Theory and Applications of Turbulent Flows*. English. Springer, 2016. ISBN: 978-3-319-31597-3. DOI: 10.1007/978-3-319-31599-7.

[60] Octavi Obiols-Sales et al. “CFDNet: A deep learning-based accelerator for fluid simulations”. In: *Proceedings of the International Conference on Supercomputing*. Association for Computing Machinery, June 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392772.

[61] Octavi Obiols-Sales et al. “SURFNet: Super-resolution of Turbulent Flows with Transfer Learning using Small Datasets”. In: (Aug. 2021). URL: http://arxiv.org/abs/2108.07667.

[62] Pranshu Pant et al. “Deep learning for reduced order modelling and efficient temporal evolution of fluid simulations”. In: *Physics of Fluids* 33 (10 Oct. 2021). ISSN: 10897666. DOI: 10.1063/5.0062546.

[63] Philippe Parnaudeau et al. “Experimental and numerical studies of the flow over a circular cylinder at Reynolds number 3900”. In: *Physics of Fluids* 20 (8 2008). ISSN: 10706631. DOI: 10.1063/1.2957018.

[64] U. Piomelli. “Large eddy simulations in 2030 and beyond”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372 (2022 Aug. 2014). ISSN: 1364503X. DOI: 10.1098/rsta.2013.0320.

[65] Stephen B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[66] Nasim Rahaman et al. “On the Spectral Bias of Neural Networks”. In: (May 2019). URL: http://arxiv.org/abs/1806.08734.

[67] M. Raissi, P. Perdikaris, and G. E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. ISSN: 10902716. DOI: 10.1016/j.jcp.2018.10.045.

[68] Osborne Reynolds. “IV. On the dynamical theory of incompressible viscous fluids and the determination of the criterion”. In: *Philosophical Transactions of the Royal Society of London. (A.)* 186 (1895), pp. 123–164. DOI: 10.1098/rsta.1895.0004. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.1895.0004. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.1895.0004.

[69] Ferry Roelofs and A. Shams. “CFD—Introduction”. In: *Thermal Hydraulics Aspects of Liquid Metal Cooled Nuclear Reactors* (Jan. 2019), pp. 213–218. DOI: 10.1016/B978-0-08-101980-1.00006-5.

[70] Carlos J. G. Rojas, Andreas Dengel, and Mateus Dias Ribeiro. “Reduced-order Model for Fluid Flows via Neural Ordinary Differential Equations”. In: (Feb. 2021). URL: http://arxiv.org/abs/2102.02248.

[71] Wybe Rozema et al. “Minimum-dissipation models for large-eddy simulation”. In: *Physics of Fluids* 27.8 (Aug. 2015), p. 085107. ISSN: 1070-6631. DOI: 10.1063/1.4928700. eprint: https://pubs.aip.org/aip/pof/article-pdf/doi/10.1063/1.4928700/15691134/085107_1_online.pdf. URL: https://doi.org/10.1063/1.4928700.

[72] Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. “Latent ODEs for Irregularly-Sampled Time Series”. In: (July 2019). URL: http://arxiv.org/abs/1907.03907.

[73] C L Rumsey and G N Coleman. *NASA Symposium on Turbulence Modeling: Roadblocks, and the Potential for Machine Learning*. Tech. rep. 2022. URL: http://www.sti.nasa.gov.

[74] Benjamin Sanderse et al. *Scientific machine learning for closure models in multiscale problems: a review*. 2024. arXiv: 2403.02913 [math.NA]. URL: https://arxiv.org/abs/2403.02913.

[75] Shibani Santurkar et al. “How Does Batch Normalization Help Optimization?” In: (Apr. 2019). URL: http://arxiv.org/abs/1805.11604.

[76] U. Schumann. “Subgrid scale model for finite difference simulations of turbulent flows in plane channels and annuli”. In: *Journal of Computational Physics* 18 (4 Aug. 1975), pp. 376–404. ISSN: 0021-9991. DOI: 10.1016/0021-9991(75)90093-5.

[77] Vikash Sehwag, Mung Chiang, and Prateek Mittal. “SSD: A Unified Framework for Self-Supervised Outlier Detection”. In: (Mar. 2021). URL: http://arxiv.org/abs/2103.12051.

[78] John Shalf. *The future of computing beyond Moore’s Law*. Mar. 2020. DOI: 10.1098/rsta.2019.0061.

[79] Varun Shankar et al. “Validation and parameterization of a novel physics-constrained neural dynamics model applied to turbulent fluid flow”. In: *Physics of Fluids* 34 (11 Nov. 2022). ISSN: 10897666. DOI: 10.1063/5.0122115.

[80] Aleksei Sholokhov et al. “Physics-informed neural ODE (PINODE): embedding physics into models using collocation points”. In: *Scientific Reports* 13 (1 Dec. 2023). ISSN: 20452322. DOI: 10.1038/s41598-023-36799-6.

[81] Justin Sirignano and Jonathan F. MacArt. “Deep learning closure models for large-eddy simulation of flows around bluff bodies”. In: *Journal of Fluid Mechanics* 966 (July 2023). ISSN: 14697645. DOI: 10.1017/jfm.2023.446.

[82] J. Smagorinski. “GENERAL CIRCULATION EXPERIMENTS WITH THE PRIMITIVE EQUATIONS: I. THE BASIC EXPERIMENT”. In: *Monthly Weather Review* 91.3 (1963), pp. 99–164. DOI: 10.1175/1520-0493(1963)091<0099:GCEWTP>2.3.CO;2. URL: https://journals.ametsoc.org/view/journals/mwre/91/3/1520-0493_1963_091_0099_gcewtp_2_3_co_2.xml.

[83] Paulo Sousa, Alexandre Afonso, and Carlos Veiga Rodrigues. “Application of machine learning to model the pressure poisson equation for fluid flow on generic geometries”. In: *Neural Computing and Applications* 36 (26 Sept. 2024), pp. 16581–16606. ISSN: 14333058. DOI: 10.1007/s00521-024-09935-0.

[84] P R Spalart. *Strategies for turbulence modelling and simulations*. URL: www.elsevier.com/locate/ijh.

[85] P. R. Spalart and V. Venkatakrishnan. “On the role and challenges of CFD in the aerospace industry”. In: *Aeronautical Journal* 120 (1223 Jan. 2016). computational cost can be used in introduction of thesis, pp. 209–232. ISSN: 00019240. DOI: 10.1017/aer.2015.10.

[86] Philippe Spalart. *AN OLD-FASHIONED FRAMEWORK FOR MACHINE LEARNING IN TURBULENCE MODELING*. Tech. rep.

[87] Philippe R Spalart and Steven R Allmaras. “A one-equation turbulence model for aerodynamic flows”. In: *AIAA Paper* 92.439 (1992), pp. 1–22.

[88] Charles G Speziale, Sutanu Sarkar, and Thomas B Gatski. “Modelling the pressure-strain correlation of turbulence: an invariant dynamical systems approach”. In: *Journal of Fluid Mechanics* 227 (1991), pp. 245–272.

[89] Luning Sun et al. “Surrogate Modeling for Fluid Flows Based on Physics-Constrained Deep Learning Without Simulation Data”. In: (July 2019). DOI: 10.1016/j.cma.2019.112732. URL: http://arxiv.org/abs/1906.02382%20http://dx.doi.org/10.1016/j.cma.2019.112732.

[90] Yiyou Sun et al. “Out-of-Distribution Detection with Deep Nearest Neighbors”. In: (Dec. 2022). URL: http://arxiv.org/abs/2204.06507.

[91] Jihoon Tack et al. “CSI: Novelty Detection via Contrastive Learning on Distributionally Shifted Instances”. In: (Oct. 2020). URL: http://arxiv.org/abs/2007.08176.

[92] TOP500 Project. *TOP500 Supercomputer Sites*. Accessed: 2025-06-09. 2025. URL: https://www.top500.org.

[93] Evren Mert Turan and Johannes Jaschke. “Multiple Shooting for Training Neural Differential Equations on Time Series”. In: *IEEE Control Systems Letters* 6 (2022), pp. 1897–1902. ISSN: 24751456. DOI: 10.1109/LCSYS.2021.3135835.

[94] Kiwon Um et al. *Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers*. Tech. rep. URL: https://github.com/tum-pbs/Solver-in-the-Loop..

[95] Ricardo Vinuesa and Steven L. Brunton. “Enhancing computational fluid dynamics with machine learning”. In: *Nature Computational Science* 2 (6 June 2022), pp. 358–366. ISSN: 26628457. DOI: 10.1038/s43588-022-00264-7.

[96] A. W. Vreman. “The filtering analog of the variational multiscale method in large-eddy simulation”. In: *Physics of Fluids* 15 (8 2003). ISSN: 10706631. DOI: 10.1063/1.1595102.

[97] Haixin Wang et al. “Recent Advances on Machine Learning for Computational Fluid Dynamics: A Survey”. In: (Aug. 2024). URL: http://arxiv.org/abs/2408.12171.

[98] Gabriel D. Weymouth and Bernat Font. “WaterLily.jl: A differentiable and backend-agnostic Julia solver for incompressible viscous flow around dynamic bodies”. In: *Computer Physics Communications* 315 (2025), p. 109748. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2025.109748. URL: https://www.sciencedirect.com/science/article/pii/S0010465525002504.

[99] Gabriel D. Weymouth and Marin Lauber. “Using Biot-Savart boundary conditions for unbounded external flow on Eulerian meshes”. In: (Mar. 2025). URL: http://arxiv.org/abs/2404.09034.

[100] David C Wilcox. “Reassessment of the scale-determining equation for advanced turbulence models”. In: *AIAA Journal* 26.11 (1988), pp. 1299–1310.

[101] Yue Xiang et al. “GPU Acceleration of CFD Algorithm: HSMAC and SIMPLE”. In: *Procedia Computer Science*. Vol. 108. Elsevier B.V., 2017, pp. 1982–1989. DOI: 10.1016/j.procs.2017.05.124.

[102] Yang Zhiyin. “Large-eddy simulation: Past, present and the future”. In: *Chinese Journal of Aeronautics* 28 (1 Feb. 2015), pp. 11–24. ISSN: 1000-9361. DOI: 10.1016/J.CJA.2014.12.007.

# A

# Autoencoder Architecture and Hyperparameter Selection

## A.1. Autoencoder Sweep

Relevant autoencoder hyperparameters were selected by performing a grid search of the five-dimensional configuration space spanned by physics-loss weights $\lambda_{\text{div}}, \lambda_{\text{curl}} \in \{0, 10^2, 10^4\}$, the latent dimension $N_z \in \{8, 16, 32\}$, the number of convolutional blocks $n_{\text{conv}} \in \{4, 5, 6\}$ and the number of dense bottleneck layers $n_{\text{dense}} \in \{1, 2, 3\}$, giving $3^5 = 243$ configurations. Each configuration was trained for $200$ epochs on the $\mathrm{Re} = 2500$ cylinder dataset under otherwise identical settings, and evaluated on a held-out test split using four metrics: the reconstruction error $\langle\, |\mathbf{u} - \hat{\mathbf{u}}| \,\rangle$, the mean absolute divergence of the reconstructed field $\langle\, |\hat{u}_{i,i}| \,\rangle$ the vorticity error $\langle\, |\omega - \hat{\omega}| \,\rangle$. Each configuration was ranked by score, which is an unweighted average of evaluation metrics, so that a lower score denotes a configuration that is jointly accurate in reconstruction and in the physical quantities of interest. The complete ranking is reported in Table A.1.

The baseline configuration that was used in this work corresponds to $N_z = 16$ with $n_{\text{conv}} = 5$, $n_{\text{dense}} = 1$ and $\lambda_{\text{div}} = \lambda_{\text{curl}} = 10^2$, and is highlighted in the table. Although it does not attain the global minimal score, it is the highest-ranked configuration that retains both physics regularisers and the compact $16$-dimensional latent space. This provides ample room to vary these parameters and study their effects on the overall hybrid operation.

**Table A.1:** Autoencoder hyperparameter grid sweep over $\lambda_{\text{div}}, \lambda_{\text{curl}} \in \{0, 10^2, 10^4\}$, latent dimension $N_z \in \{8, 16, 32\}$, convolutional blocks $n_{\text{conv}} \in \{4, 5, 6\}$ and dense bottleneck layers $n_{\text{dense}} \in \{1, 2, 3\}$ (243 configurations). All metrics are evaluated on the held-out test split; rows are ordered by ascending composite score. The shaded row marks the baseline configuration adopted in this work.

| Rank | $\lambda_{\text{div}}$ | $\lambda_{\text{curl}}$ | $N_z$ | $n_{\text{conv}}$ | $n_{\text{dense}}$ | MAE ($\times 10^{-2}$) | div. ($\times 10^{-3}$) | curl ($\times 10^{-3}$) | Score ($\times 10^{-2}$) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 100 | 0 | 32 | 5 | 1 | 3.00 | 1.17 | 6.35 | 1.251 |
| 2 | 100 | 100 | 32 | 5 | 1 | 3.03 | 1.88 | 5.40 | 1.253 |
| 3 | 0 | 0 | 16 | 5 | 1 | 2.99 | 2.05 | 6.00 | 1.264 |
| 4 | 0 | 0 | 32 | 5 | 1 | 2.98 | 2.13 | 5.98 | 1.265 |
| 5 | 10000 | 100 | 32 | 5 | 1 | 3.07 | 0.72 | 6.74 | 1.272 |
| 6 | 100 | 0 | 16 | 5 | 1 | 3.08 | 1.15 | 6.27 | 1.274 |
| 7 | 100 | 0 | 16 | 6 | 1 | 3.09 | 1.23 | 6.57 | 1.289 |
| 8 | 100 | 0 | 32 | 4 | 1 | 3.13 | 1.28 | 6.54 | 1.305 |
| 9 | 100 | 100 | 16 | 5 | 1 | 3.17 | 2.09 | 5.61 | 1.313 |
| 10 | 0 | 0 | 32 | 4 | 1 | 3.13 | 1.94 | 6.15 | 1.314 |
| 11 | 100 | 100 | 32 | 6 | 1 | 3.21 | 1.74 | 5.64 | 1.316 |
| 12 | 100 | 100 | 32 | 4 | 1 | 3.22 | 1.96 | 5.45 | 1.321 |
| 13 | 100 | 0 | 8 | 6 | 1 | 3.18 | 1.19 | 6.71 | 1.323 |
| 14 | 0 | 0 | 16 | 4 | 1 | 3.14 | 2.02 | 6.33 | 1.326 |

*continued on next page*

Table A.1 – continued from previous page

| Rank | $\lambda_{\text{div}}$ | $\lambda_{\text{curl}}$ | $N_z$ | $n_{\text{conv}}$ | $n_{\text{dense}}$ | MAE ($\times 10^{-2}$) | div. ($\times 10^{-3}$) | curl ($\times 10^{-3}$) | Score ($\times 10^{-2}$) |
|---|---|---|---|---|---|---|---|---|---|
| 15 | 0 | 100 | 32 | 4 | 1 | 3.15 | 3.23 | 5.45 | 1.338 |
| 16 | 100 | 100 | 16 | 6 | 1 | 3.27 | 1.91 | 5.66 | 1.343 |
| 17 | 100 | 0 | 8 | 5 | 1 | 3.26 | 1.27 | 6.53 | 1.346 |
| 18 | 100 | 100 | 16 | 4 | 1 | 3.27 | 2.08 | 5.63 | 1.348 |
| 19 | 100 | 100 | 8 | 6 | 1 | 3.29 | 2.09 | 5.80 | 1.359 |
| 20 | 10000 | 100 | 16 | 5 | 1 | 3.31 | 0.73 | 7.01 | 1.360 |
| 21 | 100 | 0 | 32 | 6 | 1 | 3.30 | 1.14 | 6.72 | 1.361 |
| 22 | 0 | 100 | 16 | 5 | 1 | 3.22 | 3.29 | 5.37 | 1.361 |
| 23 | 0 | 0 | 8 | 5 | 1 | 3.24 | 2.15 | 6.45 | 1.366 |
| 24 | 0 | 100 | 32 | 5 | 1 | 3.21 | 3.43 | 5.42 | 1.366 |
| 25 | 0 | 0 | 16 | 6 | 1 | 3.22 | 2.11 | 6.68 | 1.367 |
| 26 | 0 | 100 | 8 | 6 | 1 | 3.29 | 2.93 | 5.44 | 1.376 |
| 27 | 100 | 0 | 16 | 4 | 1 | 3.32 | 1.41 | 6.75 | 1.378 |
| 28 | 0 | 100 | 8 | 5 | 1 | 3.25 | 3.35 | 5.60 | 1.382 |
| 29 | 100 | 100 | 8 | 5 | 1 | 3.35 | 2.28 | 5.81 | 1.386 |
| 30 | 0 | 0 | 32 | 6 | 1 | 3.32 | 1.97 | 6.48 | 1.389 |
| 31 | 0 | 100 | 16 | 4 | 1 | 3.32 | 3.24 | 5.70 | 1.406 |
| 32 | 0 | 0 | 8 | 6 | 1 | 3.38 | 1.76 | 6.64 | 1.407 |
| 33 | 0 | 100 | 16 | 6 | 1 | 3.30 | 4.01 | 5.56 | 1.421 |
| 34 | 100 | 0 | 8 | 4 | 1 | 3.41 | 1.58 | 7.05 | 1.426 |
| 35 | 100 | 100 | 32 | 5 | 2 | 3.53 | 1.82 | 5.86 | 1.431 |
| 36 | 0 | 0 | 8 | 4 | 1 | 3.36 | 2.62 | 6.79 | 1.432 |
| 37 | 100 | 100 | 16 | 5 | 2 | 3.48 | 1.98 | 6.21 | 1.434 |
| 38 | 10000 | 100 | 32 | 4 | 1 | 3.52 | 0.81 | 7.08 | 1.436 |
| 39 | 0 | 100 | 32 | 6 | 1 | 3.39 | 3.74 | 5.67 | 1.444 |
| 40 | 10000 | 100 | 8 | 6 | 1 | 3.58 | 0.74 | 7.30 | 1.462 |
| 41 | 100 | 0 | 32 | 5 | 2 | 3.56 | 1.30 | 7.03 | 1.465 |
| 42 | 100 | 100 | 8 | 4 | 1 | 3.55 | 2.47 | 6.33 | 1.476 |
| 43 | 100 | 0 | 16 | 5 | 2 | 3.64 | 1.21 | 7.26 | 1.494 |
| 44 | 100 | 0 | 8 | 6 | 2 | 3.64 | 1.36 | 7.12 | 1.495 |
| 45 | 0 | 0 | 16 | 5 | 2 | 3.53 | 2.69 | 6.96 | 1.499 |
| 46 | 0 | 0 | 8 | 5 | 2 | 3.57 | 2.54 | 7.12 | 1.513 |
| 47 | 0 | 0 | 32 | 5 | 2 | 3.64 | 2.34 | 6.77 | 1.517 |
| 48 | 10000 | 100 | 32 | 6 | 1 | 3.68 | 0.72 | 8.03 | 1.518 |
| 49 | 0 | 100 | 16 | 5 | 2 | 3.61 | 3.67 | 5.81 | 1.519 |
| 50 | 100 | 100 | 16 | 6 | 2 | 3.75 | 1.95 | 6.44 | 1.529 |
| 51 | 100 | 0 | 32 | 6 | 2 | 3.79 | 1.17 | 6.98 | 1.535 |
| 52 | 0 | 0 | 32 | 4 | 2 | 3.59 | 3.41 | 6.89 | 1.541 |
| 53 | 0 | 0 | 16 | 4 | 2 | 3.67 | 3.27 | 7.12 | 1.569 |
| 54 | 100 | 0 | 8 | 5 | 2 | 3.72 | 1.86 | 8.06 | 1.569 |
| 55 | 100 | 0 | 16 | 6 | 2 | 3.90 | 1.08 | 7.18 | 1.577 |
| 56 | 0 | 100 | 8 | 6 | 2 | 3.73 | 4.28 | 6.18 | 1.592 |
| 57 | 0 | 0 | 16 | 6 | 2 | 3.91 | 1.88 | 6.84 | 1.595 |
| 58 | 0 | 100 | 8 | 4 | 1 | 3.70 | 4.27 | 6.58 | 1.596 |
| 59 | 0 | 0 | 32 | 6 | 2 | 3.92 | 1.91 | 6.86 | 1.599 |
| 60 | 10000 | 0 | 16 | 5 | 1 | 3.82 | 0.57 | 9.42 | 1.605 |
| 61 | 0 | 0 | 32 | 5 | 3 | 3.88 | 2.41 | 6.94 | 1.606 |
| 62 | 10000 | 0 | 32 | 5 | 1 | 3.85 | 0.57 | 9.17 | 1.609 |
| 63 | 100 | 100 | 32 | 6 | 2 | 3.89 | 2.77 | 6.80 | 1.614 |
| 64 | 100 | 0 | 16 | 4 | 2 | 3.78 | 2.49 | 8.11 | 1.615 |
| 65 | 100 | 100 | 8 | 6 | 2 | 4.02 | 1.83 | 6.54 | 1.618 |
| 66 | 0 | 100 | 32 | 5 | 2 | 3.89 | 3.74 | 5.92 | 1.618 |
| 67 | 100 | 100 | 32 | 4 | 2 | 3.89 | 3.14 | 6.90 | 1.630 |
| 68 | 100 | 0 | 32 | 5 | 3 | 4.02 | 1.30 | 7.48 | 1.634 |
| 69 | 0 | 100 | 16 | 6 | 2 | 3.87 | 4.00 | 6.42 | 1.637 |
| 70 | 100 | 0 | 16 | 5 | 3 | 4.04 | 1.29 | 7.51 | 1.639 |
| 71 | 10000 | 10000 | 32 | 4 | 1 | 4.16 | 2.05 | 5.69 | 1.646 |
| 72 | 100 | 0 | 32 | 4 | 2 | 3.97 | 2.05 | 7.84 | 1.654 |
| 73 | 0 | 0 | 16 | 5 | 3 | 4.03 | 2.56 | 7.25 | 1.671 |
| 74 | 0 | 0 | 8 | 6 | 2 | 4.08 | 2.39 | 7.10 | 1.676 |
| 75 | 10000 | 100 | 16 | 4 | 1 | 4.15 | 0.84 | 7.99 | 1.676 |
| 76 | 0 | 100 | 32 | 6 | 2 | 4.00 | 4.15 | 6.44 | 1.685 |
| 77 | 0 | 0 | 32 | 6 | 3 | 4.13 | 2.06 | 7.17 | 1.685 |
| 78 | 0 | 0 | 8 | 5 | 3 | 4.12 | 2.77 | 7.24 | 1.708 |
| 79 | 100 | 100 | 16 | 5 | 3 | 4.29 | 1.94 | 6.50 | 1.712 |
| 80 | 10000 | 10000 | 32 | 5 | 1 | 4.40 | 2.12 | 5.51 | 1.720 |

*continued on next page*

Table A.1 – continued from previous page

| Rank | $\lambda_{\text{div}}$ | $\lambda_{\text{curl}}$ | $N_z$ | $n_{\text{conv}}$ | $n_{\text{dense}}$ | MAE ($\times 10^{-2}$) | div. ($\times 10^{-3}$) | curl ($\times 10^{-3}$) | Score ($\times 10^{-2}$) |
|---|---|---|---|---|---|---|---|---|---|
| 81 | 0 | 0 | 8 | 6 | 3 | 4.20 | 2.24 | 7.43 | 1.722 |
| 82 | 100 | 100 | 8 | 5 | 2 | 4.03 | 3.83 | 7.88 | 1.733 |
| 83 | 10000 | 100 | 16 | 6 | 1 | 4.27 | 0.78 | 8.70 | 1.738 |
| 84 | 10000 | 100 | 8 | 5 | 1 | 4.28 | 0.87 | 8.50 | 1.739 |
| 85 | 0 | 100 | 32 | 4 | 2 | 3.92 | 6.27 | 6.91 | 1.747 |
| 86 | 0 | 0 | 8 | 4 | 3 | 4.16 | 3.41 | 7.54 | 1.750 |
| 87 | 0 | 0 | 16 | 4 | 3 | 4.19 | 3.07 | 7.58 | 1.751 |
| 88 | 0 | 100 | 16 | 4 | 2 | 3.94 | 6.64 | 7.02 | 1.770 |
| 89 | 0 | 100 | 8 | 5 | 2 | 4.05 | 5.78 | 6.87 | 1.771 |
| 90 | 10000 | 10000 | 16 | 5 | 1 | 4.57 | 2.31 | 5.43 | 1.781 |
| 91 | 100 | 0 | 8 | 6 | 3 | 4.52 | 1.02 | 7.62 | 1.796 |
| 92 | 10000 | 0 | 32 | 4 | 1 | 4.36 | 0.58 | 10.19 | 1.813 |
| 93 | 0 | 0 | 32 | 4 | 3 | 4.39 | 3.09 | 7.71 | 1.823 |
| 94 | 0 | 0 | 16 | 6 | 3 | 4.44 | 2.57 | 7.86 | 1.829 |
| 95 | 10000 | 0 | 8 | 6 | 1 | 4.45 | 0.58 | 9.94 | 1.833 |
| 96 | 0 | 100 | 16 | 6 | 3 | 4.42 | 4.03 | 6.82 | 1.836 |
| 97 | 100 | 100 | 32 | 6 | 3 | 4.54 | 2.52 | 7.35 | 1.842 |
| 98 | 0 | 100 | 32 | 4 | 3 | 4.42 | 4.75 | 6.77 | 1.858 |
| 99 | 100 | 100 | 16 | 4 | 2 | 4.44 | 4.35 | 7.77 | 1.885 |
| 100 | 100 | 0 | 8 | 5 | 3 | 4.58 | 1.94 | 9.03 | 1.891 |
| 101 | 100 | 100 | 16 | 6 | 3 | 4.66 | 2.95 | 7.60 | 1.906 |
| 102 | 0 | 100 | 16 | 4 | 3 | 4.45 | 6.01 | 6.94 | 1.916 |
| 103 | 10000 | 10000 | 8 | 6 | 1 | 4.93 | 2.33 | 6.09 | 1.924 |
| 104 | 0 | 100 | 16 | 5 | 3 | 4.59 | 5.01 | 7.03 | 1.930 |
| 105 | 100 | 0 | 16 | 6 | 3 | 4.71 | 1.84 | 9.31 | 1.943 |
| 106 | 10000 | 100 | 8 | 6 | 2 | 4.88 | 0.81 | 8.93 | 1.951 |
| 107 | 0 | 100 | 32 | 5 | 3 | 4.77 | 3.75 | 7.16 | 1.955 |
| 108 | 0 | 100 | 32 | 6 | 3 | 4.72 | 4.62 | 7.09 | 1.964 |
| 109 | 0 | 100 | 8 | 6 | 3 | 4.81 | 3.87 | 7.14 | 1.970 |
| 110 | 100 | 0 | 32 | 6 | 3 | 4.90 | 1.60 | 9.07 | 1.990 |
| 111 | 100 | 100 | 8 | 6 | 3 | 5.05 | 1.99 | 7.43 | 1.997 |
| 112 | 10000 | 10000 | 16 | 4 | 1 | 5.14 | 2.42 | 6.14 | 2.000 |
| 113 | 10000 | 0 | 32 | 6 | 1 | 4.84 | 0.57 | 11.75 | 2.025 |
| 114 | 10000 | 0 | 16 | 4 | 1 | 5.05 | 0.53 | 10.74 | 2.058 |
| 115 | 10000 | 100 | 32 | 6 | 2 | 5.15 | 0.82 | 9.50 | 2.060 |
| 116 | 100 | 0 | 16 | 4 | 3 | 4.84 | 2.81 | 11.26 | 2.082 |
| 117 | 10000 | 100 | 32 | 6 | 3 | 5.22 | 0.76 | 9.62 | 2.084 |
| 118 | 100 | 0 | 32 | 4 | 3 | 4.96 | 2.56 | 10.78 | 2.099 |
| 119 | 100 | 100 | 8 | 5 | 3 | 5.25 | 3.11 | 7.62 | 2.109 |
| 120 | 10000 | 10000 | 8 | 5 | 1 | 5.51 | 2.36 | 6.04 | 2.115 |
| 121 | 0 | 0 | 8 | 4 | 2 | 4.80 | 7.27 | 9.04 | 2.142 |
| 122 | 100 | 0 | 8 | 4 | 2 | 5.04 | 3.30 | 11.08 | 2.158 |
| 123 | 10000 | 10000 | 8 | 4 | 1 | 5.61 | 2.67 | 6.34 | 2.172 |
| 124 | 100 | 100 | 32 | 4 | 3 | 5.50 | 4.29 | 8.72 | 2.266 |
| 125 | 100 | 100 | 32 | 5 | 3 | 5.75 | 2.81 | 8.61 | 2.298 |
| 126 | 10000 | 10000 | 32 | 5 | 2 | 6.03 | 2.40 | 6.74 | 2.314 |
| 127 | 10000 | 100 | 8 | 6 | 3 | 5.78 | 0.82 | 10.93 | 2.318 |
| 128 | 10000 | 0 | 8 | 5 | 1 | 5.69 | 0.61 | 12.10 | 2.321 |
| 129 | 10000 | 100 | 16 | 5 | 2 | 5.94 | 0.96 | 11.26 | 2.386 |
| 130 | 10000 | 0 | 16 | 6 | 1 | 5.94 | 0.65 | 11.95 | 2.400 |
| 131 | 0 | 100 | 8 | 5 | 3 | 5.65 | 8.74 | 7.62 | 2.427 |
| 132 | 10000 | 0 | 8 | 5 | 3 | 6.12 | 0.68 | 12.75 | 2.488 |
| 133 | 10000 | 0 | 32 | 6 | 3 | 6.25 | 0.53 | 12.38 | 2.513 |
| 134 | 100 | 0 | 8 | 4 | 3 | 6.05 | 4.64 | 10.43 | 2.520 |
| 135 | 100 | 100 | 8 | 4 | 2 | 6.35 | 4.56 | 8.60 | 2.555 |
| 136 | 100 | 100 | 16 | 4 | 3 | 6.21 | 5.77 | 9.24 | 2.572 |
| 137 | 10000 | 0 | 16 | 6 | 2 | 6.43 | 0.60 | 12.64 | 2.586 |
| 138 | 10000 | 10000 | 8 | 5 | 2 | 6.57 | 4.26 | 8.07 | 2.601 |
| 139 | 10000 | 0 | 8 | 6 | 3 | 6.62 | 0.54 | 13.60 | 2.677 |
| 140 | 100 | 100 | 8 | 4 | 3 | 6.49 | 6.71 | 9.34 | 2.699 |
| 141 | 10000 | 0 | 8 | 6 | 2 | 6.76 | 0.57 | 12.77 | 2.699 |
| 142 | 10000 | 10000 | 8 | 6 | 3 | 7.21 | 2.78 | 7.64 | 2.751 |
| 143 | 10000 | 100 | 8 | 4 | 1 | 7.28 | 0.75 | 9.93 | 2.781 |
| 144 | 0 | 100 | 8 | 4 | 3 | 6.42 | 11.66 | 8.90 | 2.826 |
| 145 | 0 | 100 | 8 | 4 | 2 | 6.22 | 15.70 | 7.89 | 2.861 |
| 146 | 10000 | 100 | 16 | 6 | 3 | 7.39 | 0.83 | 11.46 | 2.872 |

*continued on next page*

Table A.1 – continued from previous page

| Rank | $\lambda_{\text{div}}$ | $\lambda_{\text{curl}}$ | $N_z$ | $n_{\text{conv}}$ | $n_{\text{dense}}$ | MAE ($\times 10^{-2}$) | div. ($\times 10^{-3}$) | curl ($\times 10^{-3}$) | Score ($\times 10^{-2}$) |
|---|---|---|---|---|---|---|---|---|---|
| 147 | 10000 | 100 | 8 | 5 | 3 | 7.78 | 1.06 | 11.22 | 3.003 |
| 148 | 10000 | 0 | 16 | 6 | 3 | 7.94 | 0.60 | 13.31 | 3.112 |
| 149 | 10000 | 10000 | 8 | 4 | 2 | 8.30 | 3.95 | 8.14 | 3.168 |
| 150 | 10000 | 100 | 32 | 5 | 2 | 8.22 | 0.92 | 12.35 | 3.183 |
| 151 | 10000 | 0 | 8 | 4 | 1 | 8.24 | 0.59 | 12.83 | 3.192 |
| 152 | 100 | 10000 | 32 | 5 | 2 | 8.42 | 10.82 | 6.26 | 3.376 |
| 153 | 10000 | 10000 | 32 | 6 | 1 | 9.20 | 2.80 | 6.76 | 3.385 |
| 154 | 100 | 10000 | 32 | 4 | 1 | 9.53 | 10.41 | 6.09 | 3.727 |
| 155 | 100 | 10000 | 16 | 4 | 1 | 9.55 | 10.34 | 6.22 | 3.734 |
| 156 | 10000 | 10000 | 16 | 6 | 2 | 10.26 | 2.46 | 7.36 | 3.747 |
| 157 | 10000 | 0 | 16 | 5 | 2 | 9.72 | 0.70 | 14.83 | 3.759 |
| 158 | 10000 | 10000 | 8 | 6 | 2 | 10.17 | 3.93 | 7.94 | 3.787 |
| 159 | 0 | 10000 | 8 | 6 | 1 | 9.83 | 11.87 | 6.36 | 3.883 |
| 160 | 100 | 10000 | 8 | 4 | 1 | 9.99 | 11.88 | 6.69 | 3.948 |
| 161 | 100 | 10000 | 8 | 6 | 1 | 10.17 | 10.81 | 6.29 | 3.959 |
| 162 | 10000 | 10000 | 8 | 4 | 3 | 10.48 | 6.85 | 9.25 | 4.029 |
| 163 | 10000 | 10000 | 8 | 5 | 3 | 11.18 | 3.71 | 8.55 | 4.136 |
| 164 | 10000 | 100 | 32 | 4 | 2 | 11.36 | 1.36 | 11.54 | 4.216 |
| 165 | 10000 | 10000 | 32 | 6 | 2 | 11.89 | 1.60 | 7.81 | 4.277 |
| 166 | 100 | 10000 | 32 | 4 | 2 | 11.34 | 11.74 | 7.00 | 4.404 |
| 167 | 10000 | 0 | 32 | 5 | 2 | 11.67 | 0.63 | 15.32 | 4.421 |
| 168 | 10000 | 10000 | 16 | 5 | 3 | 12.21 | 2.64 | 8.55 | 4.442 |
| 169 | 10000 | 10000 | 16 | 6 | 1 | 12.53 | 3.20 | 6.99 | 4.515 |
| 170 | 100 | 10000 | 32 | 5 | 1 | 11.74 | 12.78 | 6.23 | 4.548 |
| 171 | 10000 | 10000 | 16 | 5 | 2 | 12.88 | 3.18 | 7.32 | 4.645 |
| 172 | 10000 | 10000 | 32 | 4 | 2 | 12.68 | 4.64 | 7.98 | 4.648 |
| 173 | 0 | 10000 | 32 | 4 | 1 | 11.99 | 15.01 | 6.20 | 4.705 |
| 174 | 100 | 10000 | 8 | 6 | 3 | 12.16 | 12.64 | 7.49 | 4.724 |
| 175 | 10000 | 100 | 16 | 5 | 3 | 12.96 | 0.75 | 12.19 | 4.750 |
| 176 | 10000 | 10000 | 16 | 4 | 2 | 12.88 | 5.32 | 8.57 | 4.756 |
| 177 | 0 | 10000 | 16 | 4 | 1 | 11.93 | 17.84 | 6.50 | 4.787 |
| 178 | 100 | 10000 | 16 | 5 | 1 | 12.58 | 12.07 | 6.22 | 4.804 |
| 179 | 10000 | 100 | 16 | 4 | 2 | 13.33 | 1.34 | 11.84 | 4.882 |
| 180 | 10000 | 100 | 8 | 5 | 2 | 13.21 | 1.11 | 13.37 | 4.885 |
| 181 | 0 | 10000 | 32 | 4 | 2 | 12.56 | 15.15 | 6.79 | 4.919 |
| 182 | 10000 | 100 | 8 | 4 | 3 | 13.45 | 1.05 | 12.76 | 4.944 |
| 183 | 100 | 10000 | 16 | 6 | 1 | 13.07 | 11.75 | 6.38 | 4.961 |
| 184 | 0 | 10000 | 16 | 6 | 1 | 12.79 | 15.09 | 6.14 | 4.970 |
| 185 | 10000 | 10000 | 32 | 4 | 3 | 13.38 | 6.57 | 9.41 | 4.994 |
| 186 | 10000 | 10000 | 32 | 6 | 3 | 14.02 | 0.93 | 10.15 | 5.044 |
| 187 | 100 | 10000 | 16 | 4 | 2 | 13.05 | 14.21 | 7.62 | 5.077 |
| 188 | 0 | 10000 | 32 | 5 | 1 | 13.09 | 16.11 | 6.19 | 5.106 |
| 189 | 10000 | 10000 | 16 | 6 | 3 | 14.15 | 4.14 | 8.75 | 5.147 |
| 190 | 10000 | 0 | 32 | 4 | 2 | 14.22 | 0.96 | 12.47 | 5.187 |
| 191 | 0 | 10000 | 8 | 6 | 2 | 13.23 | 16.94 | 7.11 | 5.212 |
| 192 | 10000 | 100 | 32 | 5 | 3 | 14.54 | 0.64 | 12.33 | 5.280 |
| 193 | 100 | 10000 | 16 | 5 | 3 | 14.07 | 10.87 | 7.00 | 5.284 |
| 194 | 10000 | 0 | 8 | 5 | 2 | 14.35 | 0.85 | 14.41 | 5.293 |
| 195 | 10000 | 10000 | 16 | 4 | 3 | 14.22 | 7.11 | 9.58 | 5.295 |
| 196 | 100 | 10000 | 16 | 6 | 3 | 13.84 | 14.15 | 6.88 | 5.314 |
| 197 | 10000 | 10000 | 32 | 5 | 3 | 15.07 | 1.14 | 8.19 | 5.334 |
| 198 | 10000 | 0 | 32 | 5 | 3 | 14.72 | 0.39 | 13.31 | 5.365 |
| 199 | 10000 | 0 | 8 | 4 | 3 | 14.65 | 1.54 | 13.70 | 5.390 |
| 200 | 100 | 10000 | 32 | 6 | 1 | 14.37 | 12.80 | 6.42 | 5.430 |
| 201 | 10000 | 0 | 16 | 4 | 2 | 14.86 | 1.21 | 13.21 | 5.432 |
| 202 | 100 | 10000 | 8 | 5 | 1 | 14.51 | 11.87 | 6.49 | 5.450 |
| 203 | 0 | 10000 | 8 | 4 | 1 | 14.06 | 18.59 | 6.62 | 5.527 |
| 204 | 10000 | 100 | 8 | 4 | 2 | 15.44 | 1.15 | 13.31 | 5.628 |
| 205 | 10000 | 0 | 32 | 4 | 3 | 15.26 | 1.06 | 15.32 | 5.634 |
| 206 | 10000 | 0 | 32 | 6 | 2 | 15.82 | 0.48 | 13.65 | 5.745 |
| 207 | 10000 | 0 | 8 | 4 | 2 | 15.85 | 1.11 | 13.87 | 5.782 |
| 208 | 100 | 10000 | 8 | 6 | 2 | 15.31 | 13.90 | 7.24 | 5.808 |
| 209 | 0 | 10000 | 16 | 5 | 1 | 15.05 | 17.96 | 6.07 | 5.818 |
| 210 | 10000 | 0 | 16 | 5 | 3 | 16.08 | 0.58 | 14.95 | 5.877 |
| 211 | 10000 | 100 | 16 | 6 | 2 | 16.19 | 0.59 | 13.86 | 5.878 |
| 212 | 10000 | 100 | 16 | 4 | 3 | 16.96 | 1.45 | 14.18 | 6.175 |

*continued on next page*

Table A.1 – continued from previous page

| Rank | $\lambda_{\text{div}}$ | $\lambda_{\text{curl}}$ | $N_z$ | $n_{\text{conv}}$ | $n_{\text{dense}}$ | MAE ($\times 10^{-2}$) | div. ($\times 10^{-3}$) | curl ($\times 10^{-3}$) | Score ($\times 10^{-2}$) |
|---|---|---|---|---|---|---|---|---|---|
| 213 | 10000 | 100 | 32 | 4 | 3 | 17.08 | 1.54 | 14.42 | 6.225 |
| 214 | 0 | 10000 | 8 | 6 | 3 | 16.51 | 16.02 | 7.46 | 6.287 |
| 215 | 100 | 10000 | 32 | 6 | 3 | 16.85 | 13.29 | 8.35 | 6.337 |
| 216 | 100 | 10000 | 32 | 6 | 2 | 17.23 | 15.14 | 6.88 | 6.477 |
| 217 | 0 | 10000 | 8 | 5 | 1 | 17.31 | 16.37 | 6.06 | 6.516 |
| 218 | 0 | 10000 | 16 | 5 | 2 | 17.13 | 19.00 | 6.53 | 6.562 |
| 219 | 100 | 10000 | 32 | 5 | 3 | 17.96 | 13.74 | 6.90 | 6.676 |
| 220 | 10000 | 0 | 16 | 4 | 3 | 18.46 | 1.31 | 15.79 | 6.723 |
| 221 | 0 | 10000 | 16 | 4 | 2 | 17.94 | 20.34 | 7.71 | 6.916 |
| 222 | 100 | 10000 | 16 | 5 | 2 | 18.62 | 15.06 | 6.96 | 6.942 |
| 223 | 100 | 10000 | 16 | 6 | 2 | 18.43 | 22.50 | 8.61 | 7.180 |
| 224 | 0 | 10000 | 32 | 6 | 1 | 19.79 | 14.17 | 5.69 | 7.258 |
| 225 | 0 | 10000 | 32 | 6 | 2 | 19.42 | 20.19 | 6.85 | 7.376 |
| 226 | 100 | 10000 | 16 | 4 | 3 | 19.91 | 17.09 | 8.54 | 7.493 |
| 227 | 0 | 10000 | 32 | 6 | 3 | 19.51 | 24.60 | 8.67 | 7.611 |
| 228 | 100 | 10000 | 8 | 4 | 2 | 22.39 | 12.01 | 8.45 | 8.146 |
| 229 | 0 | 10000 | 16 | 5 | 3 | 21.84 | 20.79 | 7.46 | 8.220 |
| 230 | 100 | 10000 | 32 | 4 | 3 | 22.71 | 13.63 | 9.70 | 8.347 |
| 231 | 0 | 10000 | 32 | 5 | 3 | 22.63 | 24.17 | 7.31 | 8.594 |
| 232 | 0 | 10000 | 32 | 5 | 2 | 23.08 | 22.76 | 7.24 | 8.692 |
| 233 | 0 | 10000 | 32 | 4 | 3 | 24.79 | 20.34 | 9.98 | 9.275 |
| 234 | 0 | 10000 | 8 | 4 | 2 | 25.56 | 24.08 | 8.30 | 9.598 |
| 235 | 0 | 10000 | 16 | 4 | 3 | 29.29 | 23.24 | 9.53 | 10.855 |
| 236 | 100 | 10000 | 8 | 5 | 3 | 30.90 | 22.69 | 8.36 | 11.336 |
| 237 | 100 | 10000 | 8 | 5 | 2 | 30.03 | 33.42 | 9.02 | 11.424 |
| 238 | 0 | 10000 | 8 | 5 | 2 | 29.83 | 40.62 | 8.65 | 11.587 |
| 239 | 0 | 10000 | 16 | 6 | 3 | 31.60 | 34.89 | 8.82 | 11.991 |
| 240 | 0 | 10000 | 8 | 5 | 3 | 32.35 | 31.64 | 8.47 | 12.122 |
| 241 | 100 | 10000 | 8 | 4 | 3 | 36.68 | 45.23 | 9.64 | 14.057 |
| 242 | 0 | 10000 | 16 | 6 | 2 | 38.84 | 52.69 | 7.72 | 14.961 |
| 243 | 0 | 10000 | 8 | 4 | 3 | 47.88 | 68.33 | 9.67 | 18.562 |

## A.2. Baseline Architecture

The layer-by-layer composition of the baseline autoencoder is given in Table A.2, displaying the amount of trainable parameters the best model of the grid sweep in Table A.1. The encoder applies five convolutional blocks that successively halve the spatial resolution while doubling the channel count, compressing the $256 \times 256 \times 4$ augmented input to an $8 \times 8 \times 128$ feature map; this is flattened and projected by a single dense layer onto the $16$-dimensional latent state. This path is mirrored in the decoder, recovering the full scale velocity field using 5 sequential bilinear upsampling blocks before projecting and smoothing described in Section 3.4.1. The two dense layers dominate the parameter budget, accounting for roughly $58\%$ of the $467\,476$ trainable parameters, while the three interior BatchNorm layers contribute the $227$ non-trainable running statistics.

| Component | Layer | Output size | Parameters |
|---|---|---|---|
| *Encoder* | | | |
| Input | | $256 \times 256 \times 4$ | |
| Block 1 | Conv $3 \times 3$, ReLU, MaxPool | $128 \times 128 \times 8$ | 296 |
| Block 2 | Conv $3 \times 3$, BN, ReLU, MaxPool | $64 \times 64 \times 16$ | 1 200 |
| Block 3 | Conv $3 \times 3$, BN, ReLU, MaxPool | $32 \times 32 \times 32$ | 4 704 |
| Block 4 | Conv $3 \times 3$, BN, ReLU, MaxPool | $16 \times 16 \times 64$ | 18 624 |
| Block 5 | Conv $3 \times 3$, ReLU, MaxPool | $8 \times 8 \times 128$ | 73 856 |
| Flatten | | 8192 | |
| Dense 1 | Linear | 16 | 131 088 |
| *Latent space* $\mathbf{z} \in \mathbb{R}^{16}$ | | | |
| *Decoder* | | | |
| Dense 2 | Linear, Reshape | $8 \times 8 \times 128$ | 139 264 |
| Block 6 | Upsample, Conv $3 \times 3$, GELU | $16 \times 16 \times 64$ | 73 792 |
| Block 7 | Upsample, Conv $3 \times 3$, GELU | $32 \times 32 \times 32$ | 18 464 |
| Block 8 | Upsample, Conv $3 \times 3$, GELU | $64 \times 64 \times 16$ | 4 624 |
| Block 9 | Upsample, Conv $3 \times 3$, GELU | $128 \times 128 \times 8$ | 1 160 |
| Block 10 | Upsample, Conv $3 \times 3$, GELU | $256 \times 256 \times 4$ | 292 |
| Projection | Conv $3 \times 3$ | $256 \times 256 \times 2$ | 74 |
| Smoothing | Conv $3 \times 3$ | $256 \times 256 \times 2$ | 38 |
| **Total** | | | **467 476** |

**Table A.2:** Layer-by-layer summary of the autoencoder architecture ($n_{\text{conv}} = 5$, $n_{\text{dense}} = 1$, $C_{\text{base}} = 8$, $N_z = 16$). Spatial dimensions refer to the feature map size after each operation. The total number of trainable parameters is 467 476, with an additional 227 non-trainable BatchNorm running statistics. The two dense bottleneck layers account for approximately 58% of the total parameter count.

# B

# Pipeline Parameter Variaton

**Table B.1:** Parameter-variation sweep: wall times, acceleration, force coefficients and errors, and mean/Reynolds-stress field statistics for each run. Column labels denote the varied hyperparameter relative to the `base` configuration.

| | | Epochs | | batch | | $N_z$ | | $(\lambda_{\text{div}}, \lambda_{\text{curl}})$ | | | | $n_s$ | | $\lambda_c$ | | $q$ | | $k$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | base | 250 | 750 | 4 | 8 | 8 | 32 | (0, 0) | (100, 0) | (0, 100) | (1000, 1000) | 10 | 20 | 100 | 400 | 0.9 | 0.95 | 3 | 10 |
| **Wall times [s]** | | | | | | | | | | | | | | | | | | | |
| $T_{\text{ref}}^{\text{wall}}$ | 86.63 | 103.4 | 93.16 | 90.30 | 90.87 | 122.1 | 98.74 | 105.1 | 108.7 | 94.43 | 90.02 | 135.5 | 134.6 | 87.46 | 86.44 | 86.24 | 88.18 | 167.4 | 169.7 |
| $T_{h,\text{tot}}^{\text{wall}}$ | 58.56 | 49.70 | 60.12 | 70.33 | 62.26 | 48.59 | 60.21 | 63.15 | 62.61 | 50.34 | 62.67 | 52.45 | 65.62 | 42.45 | 51.60 | 69.17 | 58.87 | 107.2 | 86.70 |
| $T_{h,\text{train}}$ | 1128.0 | 876.0 | 1530.0 | 1560.0 | 1338.0 | 1032.0 | 1374.0 | 1200.0 | 1194.0 | 1014.0 | 1308.0 | 882.0 | 1140.0 | 1050.0 | 1068.0 | 1746.0 | 1506.0 | 1362.0 | 1086.0 |
| **Acceleration metrics** | | | | | | | | | | | | | | | | | | | |
| $S_{\text{loop}}$ | 1.479 | 1.743 | 1.441 | 1.232 | 1.391 | 1.783 | 1.439 | 1.372 | 1.384 | 1.721 | 1.382 | 1.652 | 1.320 | 2.041 | 1.679 | 1.252 | 1.472 | 0.8079 | 0.9992 |
| $S_{\text{eff}}$ | 0.0730 | 0.0936 | 0.0545 | 0.0531 | 0.0619 | 0.0802 | 0.0604 | 0.0686 | 0.0689 | 0.0814 | 0.0632 | 0.0927 | 0.0719 | 0.0793 | 0.0774 | 0.0477 | 0.0554 | 0.0590 | 0.0739 |
| **Force coefficients — whole simulation** | | | | | | | | | | | | | | | | | | | |
| $\overline{C_D}$ ref. | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 | -1.588 |
| $C_L^{\text{rms}}$ ref. | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 | 1.185 |
| $\overline{C_D}$ hyb. | -1.586 | -1.741 | -1.615 | -1.564 | -1.617 | -1.616 | -1.688 | -1.855 | -1.824 | -1.578 | -1.557 | -1.661 | -1.738 | -1.730 | -1.647 | -1.661 | -1.624 | -1.602 | -1.633 |
| $C_L^{\text{rms}}$ hyb. | 1.144 | 1.078 | 1.149 | 1.140 | 1.213 | 1.152 | 1.086 | 1.063 | 1.067 | 1.098 | 1.083 | 1.081 | 1.153 | 1.090 | 1.047 | 1.160 | 1.161 | 1.180 | 1.133 |
| **Force coefficients — hybrid regions only** | | | | | | | | | | | | | | | | | | | |
| $\overline{C_D}$ ref. | -1.582 | -1.581 | -1.579 | -1.584 | -1.577 | -1.583 | -1.589 | -1.577 | -1.585 | -1.582 | -1.582 | -1.583 | -1.579 | -1.581 | -1.582 | -1.590 | -1.580 | -1.591 | -1.575 |
| $C_L^{\text{rms}}$ ref. | 1.184 | 1.180 | 1.182 | 1.187 | 1.183 | 1.191 | 1.185 | 1.185 | 1.184 | 1.186 | 1.169 | 1.183 | 1.182 | 1.181 | 1.182 | 1.176 | 1.187 | 1.175 | 1.185 |
| $\overline{C_D}$ hyb. | -1.585 | -1.798 | -1.622 | -1.542 | -1.639 | -1.628 | -1.761 | -2.019 | -2.032 | -1.569 | -1.546 | -1.689 | -1.817 | -1.782 | -1.667 | -1.732 | -1.662 | -1.615 | -1.660 |
| $C_L^{\text{rms}}$ hyb. | 1.050 | 1.026 | 1.121 | 1.106 | 1.228 | 1.119 | 1.036 | 0.9688 | 0.9095 | 1.055 | 1.015 | 1.027 | 1.128 | 1.047 | 0.9768 | 1.145 | 1.117 | 1.180 | 1.100 |
| **Force errors — whole simulation** | | | | | | | | | | | | | | | | | | | |
| $\lvert\Delta C_D\rvert$ | 0.0027 | 0.1524 | 0.0271 | 0.0241 | 0.0290 | 0.0274 | 0.0998 | 0.2671 | 0.2360 | 0.0101 | 0.0317 | 0.0728 | 0.1496 | 0.1418 | 0.0592 | 0.0724 | 0.0353 | 0.0139 | 0.0444 |
| $\lvert\Delta C_L\rvert$ | 0.0412 | 0.1075 | 0.0364 | 0.0449 | 0.0283 | 0.0329 | 0.0990 | 0.1217 | 0.1178 | 0.0870 | 0.1023 | 0.1039 | 0.0318 | 0.0948 | 0.1379 | 0.0255 | 0.0238 | 0.0052 | 0.0520 |
| rel. err $C_D$ [%] | 0.1681 | 9.595 | 1.704 | 1.520 | 1.824 | 1.728 | 6.281 | 16.81 | 14.86 | 0.6365 | 1.994 | 4.582 | 9.416 | 8.930 | 3.728 | 4.558 | 2.225 | 0.8744 | 2.796 |
| rel. err $C_L$ [%] | 3.479 | 9.073 | 3.069 | 3.791 | 2.387 | 2.773 | 8.356 | 10.27 | 9.937 | 7.342 | 8.635 | 8.765 | 2.684 | 7.998 | 11.63 | 2.151 | 2.004 | 0.4350 | 4.388 |
| **Force errors — hybrid regions only** | | | | | | | | | | | | | | | | | | | |
| $\lvert\Delta C_D\rvert$ | 0.0027 | 0.2175 | 0.0429 | 0.0421 | 0.0613 | 0.0451 | 0.1724 | 0.4416 | 0.4470 | 0.0134 | 0.0359 | 0.1060 | 0.2384 | 0.2013 | 0.0852 | 0.1425 | 0.0826 | 0.0249 | 0.0851 |
| $\lvert\Delta C_L\rvert$ | 0.1346 | 0.1548 | 0.0612 | 0.0805 | 0.0446 | 0.0723 | 0.1486 | 0.2160 | 0.2744 | 0.1306 | 0.1540 | 0.1560 | 0.0542 | 0.1345 | 0.2056 | 0.0310 | 0.0694 | 0.0047 | 0.0849 |
| rel. err $C_D$ [%] | 0.1694 | 13.76 | 2.719 | 2.656 | 3.885 | 2.849 | 10.85 | 28.00 | 28.21 | 0.8490 | 2.268 | 6.695 | 15.10 | 12.73 | 5.387 | 8.964 | 5.226 | 1.568 | 5.402 |
| rel. err $C_L$ [%] | 11.36 | 13.11 | 5.173 | 6.787 | 3.771 | 6.069 | 12.55 | 18.23 | 23.18 | 11.01 | 13.17 | 13.19 | 4.586 | 11.39 | 17.39 | 2.634 | 5.845 | 0.4028 | 7.168 |
| **Mean-flow field statistics** | | | | | | | | | | | | | | | | | | | |
| $\langle u \rangle$ MAE | 0.0094 | 0.0150 | 0.0080 | 0.0070 | 0.0060 | 0.0060 | 0.0050 | 0.0100 | 0.0080 | 0.0070 | 0.0090 | 0.0100 | 0.0120 | 0.0120 | 0.0100 | 0.0090 | 0.0090 | 0.0040 | 0.0090 |
| $\langle v \rangle$ MAE | 0.0053 | 0.0090 | 0.0060 | 0.0060 | 0.0040 | 0.0040 | 0.0030 | 0.0050 | 0.0030 | 0.0060 | 0.0060 | 0.0070 | 0.0060 | 0.0080 | 0.0070 | 0.0060 | 0.0050 | 0.0020 | 0.0040 |
| $\langle u \rangle$ corr. | 0.9932 | 0.9858 | 0.9958 | 0.9966 | 0.9980 | 0.9976 | 0.9988 | 0.9921 | 0.9968 | 0.9970 | 0.9949 | 0.9915 | 0.9857 | 0.9879 | 0.9907 | 0.9936 | 0.9944 | 0.9982 | 0.9936 |
| $\langle v \rangle$ corr. | 0.9928 | 0.9827 | 0.9926 | 0.9934 | 0.9966 | 0.9959 | 0.9980 | 0.9928 | 0.9959 | 0.9932 | 0.9922 | 0.9866 | 0.9865 | 0.9856 | 0.9863 | 0.9934 | 0.9945 | 0.9986 | 0.9954 |
| **Reynolds-stress (RST) field statistics** | | | | | | | | | | | | | | | | | | | |
| $\langle u'u' \rangle$ MAE | 0.0064 | 0.0100 | 0.0070 | 0.0060 | 0.0040 | 0.0070 | 0.0030 | 0.0060 | 0.0040 | 0.0060 | 0.0090 | 0.0110 | 0.0080 | 0.0100 | 0.0100 | 0.0060 | 0.0050 | 0.0030 | 0.0050 |
| $\langle v'v' \rangle$ MAE | 0.0138 | 0.0250 | 0.0210 | 0.0110 | 0.0090 | 0.0180 | 0.0080 | 0.0130 | 0.0110 | 0.0140 | 0.0230 | 0.0280 | 0.0180 | 0.0210 | 0.0230 | 0.0120 | 0.0120 | 0.0060 | 0.0100 |
| $\langle u'v' \rangle$ MAE | 0.0041 | 0.0060 | 0.0040 | 0.0020 | 0.0020 | 0.0040 | 0.0020 | 0.0030 | 0.0020 | 0.0030 | 0.0040 | 0.0050 | 0.0050 | 0.0050 | 0.0050 | 0.0030 | 0.0040 | 0.0020 | 0.0030 |
| $\langle u'u' \rangle$ corr. | 0.9789 | 0.9519 | 0.9778 | 0.9832 | 0.9922 | 0.9819 | 0.9952 | 0.9770 | 0.9916 | 0.9865 | 0.9701 | 0.9530 | 0.9603 | 0.9492 | 0.9534 | 0.9853 | 0.9879 | 0.9960 | 0.9869 |
| $\langle v'v' \rangle$ corr. | 0.9791 | 0.9441 | 0.9673 | 0.9897 | 0.9916 | 0.9713 | 0.9937 | 0.9853 | 0.9910 | 0.9801 | 0.9501 | 0.9360 | 0.9777 | 0.9610 | 0.9482 | 0.9869 | 0.9849 | 0.9963 | 0.9912 |
| $\langle u'v' \rangle$ corr. | 0.9275 | 0.8727 | 0.9409 | 0.9761 | 0.9853 | 0.9399 | 0.9767 | 0.9594 | 0.9876 | 0.9537 | 0.9353 | 0.9075 | 0.8969 | 0.8784 | 0.8875 | 0.9525 | 0.9364 | 0.9860 | 0.9659 |
| **Context / cross-check** | | | | | | | | | | | | | | | | | | | |
| Ref. steps | 8773 | 9888 | 9127 | 8773 | 8832 | 11787 | 9481 | 10005 | 10499 | 9142 | 8773 | 8773 | 8773 | 8773 | 8773 | 8774 | 8773 | 8774 | 8774 |
| Hyb. steps | 5813 | 5565 | 6117 | 6818 | 6176 | 5803 | 6102 | 6558 | 6568 | 5421 | 6167 | 3271 | 4142 | 4076 | 4932 | 6507 | 5506 | 6251 | 5350 |
| # NODE pred. | 9 | 9 | 10 | 12 | 11 | 8 | 10 | 12 | 11 | 9 | 11 | 6 | 7 | 7 | 9 | 13 | 11 | 12 | 9 |
| Avg. rollout | 408 | 623 | 391 | 210 | 314 | 964 | 421 | 374 | 470 | 537 | 290 | 948 | 686 | 681 | 425 | 178 | 307 | 271 | 492 |
| Grand total [min] | 28.6 | 17.2 | 29.6 | 30.5 | 25.6 | 20.2 | 26.5 | 23.3 | 23.5 | 19.5 | 25.6 | 20.6 | 24.0 | 20.5 | 23.0 | 36.5 | 31.4 | 29.0 | 22.9 |

# C

# Re 5000 Time Averaged Fields

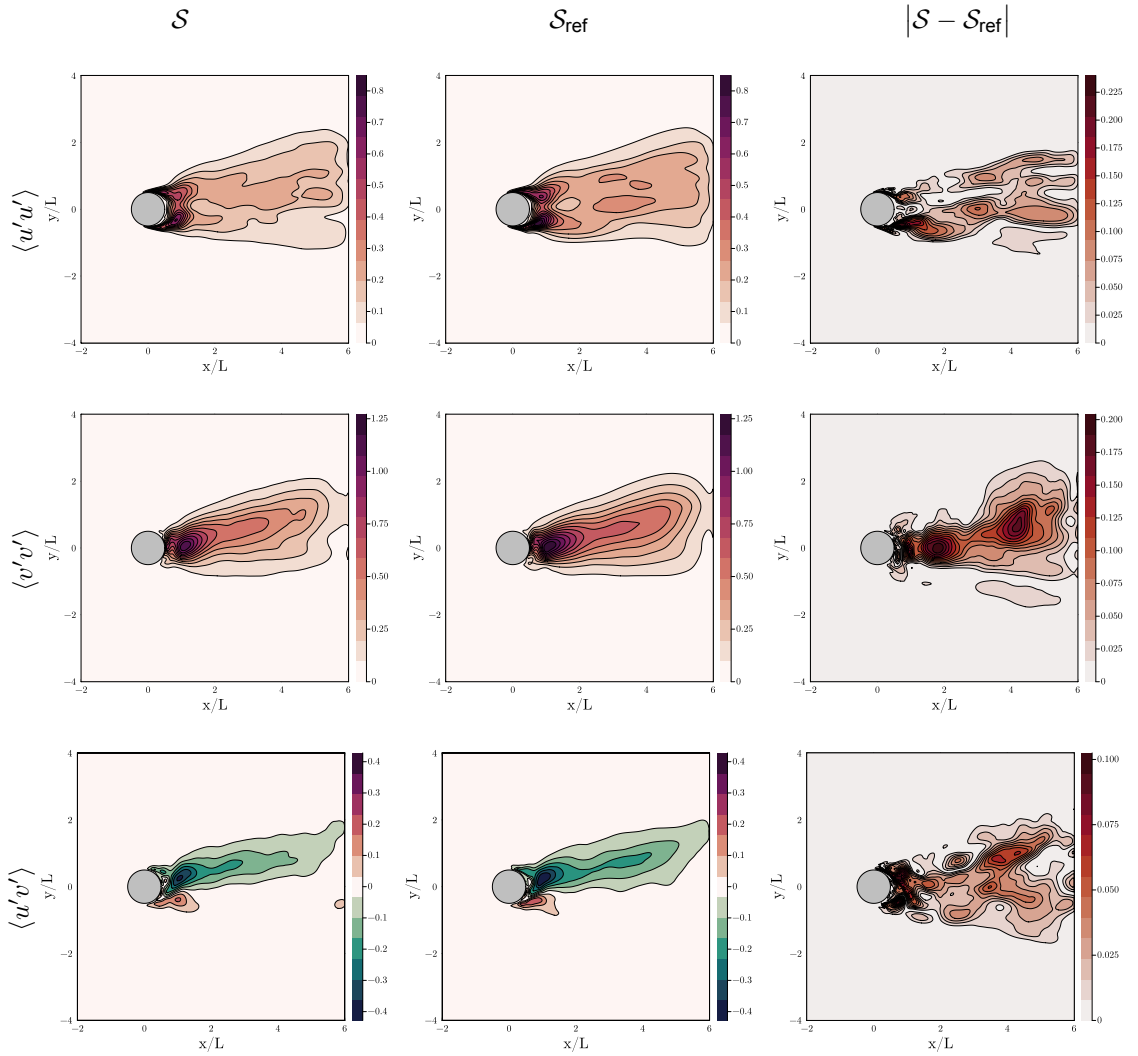



**Figure C.1:** Time-averaged Reynolds-stress components at $Re = 5000$, top to bottom $\langle u'u' \rangle$, $\langle v'v' \rangle$, $\langle u'v' \rangle$. Columns: hybrid estimate $\mathcal{S}$, reference solver $\mathcal{S}_{\text{ref}}$, and absolute difference $|\mathcal{S} - \mathcal{S}_{\text{ref}}|$. The hybrid recovers the spatial structure of each component, with residuals concentrated in the energetic near wake. Lengths scaled by $L$, velocities by $U$.

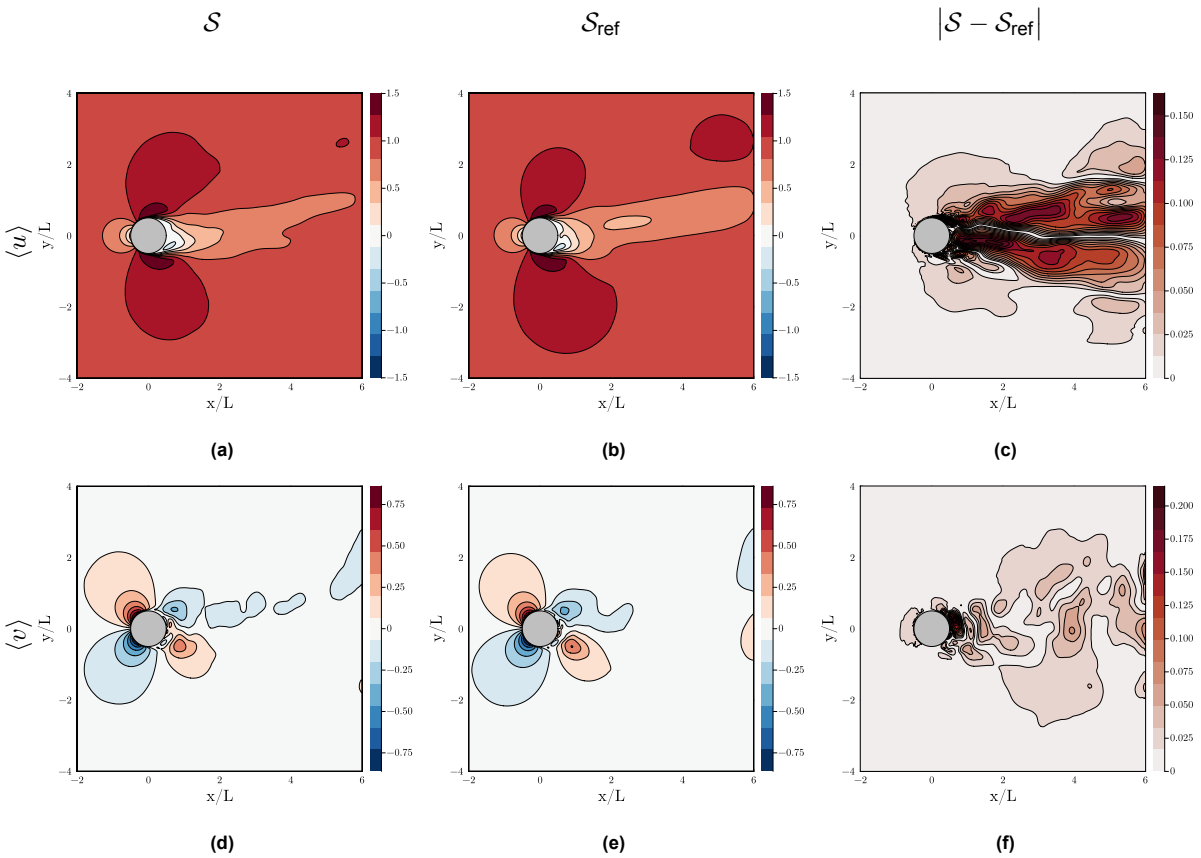



**Figure C.2:** Time-averaged velocity components at $Re = 5000$: $\langle u \rangle$ (top) and $\langle v \rangle$ (bottom). Columns: hybrid $\mathcal{S}$, reference $\mathcal{S}_{\text{ref}}$, and absolute difference $\left|\mathcal{S} - \mathcal{S}_{\text{ref}}\right|$. The dominant discrepancy is a coherent band along the wake centreline in $\langle u \rangle$. Lengths scaled by $L$, velocities by $U$.