POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# A Multi-Fidelity approach to Deep Kernel Learning of dynamical systems from high-dimensional data sources

Project Report for the PACS course
Mathematical Engineering

Author: **Federico Lancellotti**

Student ID: 10685832
Professor: Prof. Luca Formaggia
Teaching Assistants: Dr. Alberto Artoni, Dr. Beatrice Crippa
Advisor: Prof. Andrea Manzoni
Co-advisors: Dott. Nicolò Botteghi
Academic Year: 2023-24

# Contents

# 1 | Introduction

Data-driven discovery is at the heart of many modern methods we use to model, predict and control dynamical systems. Complex systems, such as in finance, robotics, climate, epidemiology, are not amenable to the traditional physical principles derivation, and researchers are turning to data-driven approaches for the most pressing scientific and engineering problems. The governing laws are often difficult to unveil, since these systems typically show nonlinear behaviours. On the other hand, high-dimensional measurements are able to capture the dominant underlying patterns that should be characterized and modeled for the eventual goal of sensing, prediction, estimation and control [5].

The recent achievements in Machine Learning and Deep Learning, combined with the availability of large amounts of data and computational resources, laid the foundations for new paradigms for the analysis and understanding of dynamical systems [4].

Though data are often high-dimensional, the behaviour exhibited by many of these complex systems can be effectively captured by a reduced number of latent state variables, that can describe their intrinsic low-dimensional nature. The process of mapping high-dimensional measurements into a low-dimensional latent space is often referred to as Model Order Reduction [37]. This task is traditionally tackled by the Singular Value Decomposition (SVD) [5] and more recently approached via a specific type of neural network (NN) called Autoencoder (AE), which is able to learn a nonlinear map between the high-dimensional data space and a low-dimensional state space.

Examples in which this strategy is adopted include Gaussian process surrogate modelling [15], Dynamics Mode Decomposition [41][33], sparse identification of latent dynamics (SINDy) [6], manifold learning [25] in combination with SINDy for latent coordinate discovery [8], and in combination with NN-based surrogate models for latent representation learning towards control [44][3].

In particular, the construction of efficient surrogate models is crucial for producing model proxies which can cheaply and accurately characterise a partial differential equations system. Indeed, computational costs can easily become prohibitive when parameterised time-dependent systems of PDEs are solved with detailed full-order models (FOMs) in a

multi-query context [10], such as in uncertainty quantification [7], optimal control [29][43], and parameter estimation [12].

Reduced-order models (ROMs) have been developed to construct low-dimensional representations of high-dimensional systems for a significant reduction in computational costs with controlled accuracy [38][2]. Intrusive ROM techniques explicitly incorporate full-order governing equations at the reduced level and often yield reliable and physically meaningful solutions. On the other hand, non-intrusive approaches learn reduced-order systems primarily from solution data, including from numerical or experimental data, hence being more flexible, general and relevant [10].

However, the applicability and reliability of many numerical methods may collapse when the collection of high-fidelity (HF) data for model reduction is very time-consuming to produce or expensive to obtain. When time or budget restrictions prohibit the generation of additional data, the amount of available samples may be too limited to provide satisfactory model results. In addition, it is increasingly common to encounter scenarios where a wealth of data sources are readily available, easily accessible, and/or cheaply computable, albeit not perfectly accurate. For this reason, multi-fidelity methods are often employed to incorporate information from other sources, which are ideally well-correlated with the high-fidelity data, but can be obtained at a lower cost. By leveraging correlations between different datasets, multi-fidelity methods often enable strong generalisation performance when compared to models based solely on a small amount of high-fidelity data [16].

Traditional MF regression schemes, also known as co-kriging [21][1], proved suitable for many different applications, including geostatistics [19], solving PDEs [39][20], uncertainty quantification, inference and optimitazion [32]. More recently, NNs based architectures were proposed to overcome some of the drawbacks related to the curse of dimensionality and consider nonlinear correlations between datasets of different fidelity levels [16], also in the case of time-dependent problems [10].

Moreover, most data-driven methods for modelling physical phenomena still rely on the assumption that the relevant state variables are already known. Despite the computing power at our disposal and the recent theoretical and technical novelties in artificial intelligence, the quest of identifying the hidden state variables has resisted automation and is still an open field of research [4][9].

Whether we aim to identify the hidden state variables or learn the entire system evolution from data, inferring complex dynamics from a variety of sources of information with different degrees of fidelity and accuracy is not effortless, as the identification, understanding and quantification of various uncertainties is often required. When data-driven meth-

ods consider stochasticity and uncertainties quantification, Variational Autoencoders [22] (VAEs) are often considered for learning low-dimensional states as probabilistic distributions. Samples from these distributions can be used for the construction of latent state models via Gaussian models [11].

With the present work, we introduce a multi-fidelity generalisation of the framework proposed by Botteghi et. al in [4], namely a data-driven strategy for dimensionality reduction, latent-state model learning and uncertainty quantification, based on a variety of high-dimensional measurements of different degrees of accuracy, generated by unknown dynamical systems.

In particular, we introduce an ensemble of sub-models, each associated with a level of fidelity and composed of a Deep Kernel Learning (DKL) [45][31] encoder, which combines the highly expressive NN with a kernel-based probabilistic model of Gaussian process [40] to reduce the dimensionality and quantify the uncertainty simultaneously, followed by a DKL latent-state forward model that predicts the system dynamics with quantifiable modelling uncertainty, and an NN-based decoder designed to enable reconstruction. Each sub-model produces an estimate of how many state variables the observed system is likely to have, and latent representations of both the state and the dynamics of the system. The estimate of the intrinsic dimensionality is used to bound the latent state space dimension for the sub-models at higher levels of fidelity, while the low-fidelity latent representations are adopted as additional sources of information to correct the system dynamics learnt on a reduced volume of high-fidelity data.

The major advantage of the proposed method lies in its substantial generality, both in terms of the nature of the complex systems it can learn and the simplicity of the data sources it can adopt. Indeed, deep kernels display a better expressive power with respect to traditional GPR kernels, employing a remarkable scalability to high-dimensional inputs and nonlinear behaviours. To the best of our knowledge, this is the first attempt at considering the most ordinary of high-dimensional data sources, RGB videos, in a multi-fidelity context, overcoming the need of the numerical solution, often not available in real applications. The proposed strategy is tested on a variety of cases, from the simple motion of a pendulum to a more challenging PDE problem. We believe that the hereby achieved results in efficiency, accuracy and generality are essential for data-driven physical modeling.

In the following chapters, the details of the proposed framework are presented, along with the Python implementation and main numerical results. Chapter 2 collects the key prerequisites of our model, namely GPR, DKL and MF schemes, while the theoretical

details of the architecture are described in Chapter 3. The implementation is described in details in Chapter 4 and the code can be found at [24]. The model performances on data from low-dimensional dynamical systems and PDE discretisation are presented in Chapter 5. In particular, the method is tested on its ability to reconstruct the frame, to predict the system dynamics one-step forward in time and to extrapolate in time. Finally, in Chapter 6 we summarise the achieved results, highlighting both strengths and issues of the proposed framework, concluding with possible future expansions of our work.

# 2 | Preliminaries

Consider a number of high-dimensional measurements of a dynamical system of the form $\dot{\mathbf{s}} = \mathcal{F}(\mathbf{s})$, where $\mathbf{s}$ is the state vector and it is a function of time. The data is obtained from a variety of sources, not necessarily exhibiting the same degree of fidelity to the real system. Our goal is (i) to learn a meaningful representation of the unknown states and a surrogate model for $\mathcal{F}$, overcoming both the high-dimensional nature of the available measurements and the nonlinearities displayed by the system, and (ii) to leverage not only the high fidelity data, but also the less accurate sources. To achieve such goal, in Chapter 3 we propose a comprehensive framework, that is built on a number of components.

In the following, we introduce Gaussian Process Regression [40], also known as *kriging*, a non-parametric regression technique that models the prior knowledge about a computational model with a Gaussian process [27], and that we use for data-driven surrogate modeling and uncertainty quantification (UQ). To deal with high-dimensional data, we additionally introduce DKL [45], that combines the nonlinear representational power of NNs with the reliable uncertainty estimates of Gaussian processes [27]. In particular, DKL will be employed in the architecture of both the AE and the dynamical model discussed in Chapter 3, that compose the main building blocks of our framework. Finally, we present the major intuitions and ideas behind multi-fidelity methods, with the aim at leveraging the correlation between data sources of different fidelities, i.e. the inputs of our model.

## 2.1. Gaussian Process Regression

A Gaussian process (GP) is a collection of random variables, any finite number of which have a joint Gaussian distribution [40]. A GP is completely specified by its mean function and covariance function. Let $\mathbf{x} \in \mathbb{R}^n$, we define the mean function $m(\mathbf{x})$ and the covariance/kernel function $k(\mathbf{x}, \mathbf{x}')$ of a real process $Z(\mathbf{x})$ as

$$\begin{aligned} m(\mathbf{x}) &= E[Z(\mathbf{x})]\,, \\ k(\mathbf{x}, \mathbf{x}') &= \mathbb{E}[(Z(\mathbf{x}) - m(\mathbf{x}))(Z(\mathbf{x}') - m(\mathbf{x}'))]\,, \end{aligned} \tag{2.1}$$

and we will write the GP as

$$Z(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \,. \tag{2.2}$$

In regression, we consider the training data inputs $\mathcal{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(N)}), \mathbf{x}^{(i)} \in \mathbb{R}^n$, and the corresponding training targets $\mathcal{Y} = (G(\mathbf{x}^{(1)}), G(\mathbf{x}^{(2)}), ..., G(\mathbf{x}^{(N)}))$. Assume that the prior knowledge of a computational model $G(\mathbf{x})$ can be modelled by a GP $Z(\mathbf{x}) \sim \mathcal{GP}(m, k)$, then any collection of function values $\mathbf{Z} = Z(\mathcal{X})$ has a joint Gaussian distribution

$$\mathbf{Z} = Z(\mathcal{X}) = [Z(\mathbf{x}_1), ..., Z(\mathbf{x}_n)]^\top \sim \mathcal{N}(\boldsymbol{\mu}, K_{\mathcal{X},\mathcal{X}}) \,, \tag{2.3}$$

with mean vector $\boldsymbol{\mu}_i = m(\mathbf{x}_i)$ and covariance matrix $(K_{\mathcal{X},\mathcal{X}})_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$, determined from the mean function and covariance kernel of the GP. We can parametrise the kernel function as

$$k(\mathbf{x}, \mathbf{x}') = \sigma_k^2 r(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta}) \tag{2.4}$$

where $\sigma^2$ and $\boldsymbol{\theta}$ have to be estimated [27], as we will see later.

To build up a GP model, the class of the correlation kernel $r(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta})$ needs to be set. In this regard, a popular choice of the kernel function is the automatic relevance determination (ARD) squared exponential (SE) kernel:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_k^2 \exp\left( -\frac{1}{2} \sum_{j-1}^{d} \frac{(x_j - x_j')^2}{l_j^2} \right) \tag{2.5}$$

where $\{l_j\}_{(1 \le j \le d)}$ are the length-scales along each individual input direction.

Assuming additive Gaussian noise $G(\mathbf{x})|Z(\mathbf{x}) \sim \mathcal{N}(0, \sigma_\epsilon^2)$, the predictive distribution of the GP evaluated at $N^*$ test data points $\mathcal{X}^*$ is

$$[Z(\mathcal{X}^*)|Z(\mathcal{X}) = \mathcal{Y}, \sigma^2, \boldsymbol{\theta}] \sim \mathcal{N}(\mathbb{E}[Z(\mathcal{X}^*)], C(Z(\mathcal{X}^*))) \tag{2.6}$$

where

$$\begin{aligned} \mathbb{E}[Z(\mathcal{X}^*)] &= \boldsymbol{\mu}_{\mathcal{X}^*} + K_{\mathcal{X}^*,\mathcal{X}}(K_{\mathcal{X},\mathcal{X}} + \sigma I)^{-1}\mathcal{Y} \\ C(Z(\mathcal{X}^*)) &= K_{\mathcal{X}^*,\mathcal{X}^*} + K_{\mathcal{X}^*,\mathcal{X}}(K_{\mathcal{X},\mathcal{X}} + \sigma I)^{-1}K_{\mathcal{X},\mathcal{X}^*} \end{aligned} \tag{2.7}$$

$K_{\mathcal{X}^*,\mathcal{X}}$ is an $N^* \times N$ matrix of covariances between the GP evaluated at $\mathcal{X}^*$ and $\mathcal{X}$. $\boldsymbol{\mu}_{\mathcal{X}^*}$ is the $N^* \times 1$ mean vector, and $K_{\mathcal{X},\mathcal{X}}$ is the $N \times N$ covariance matrix evaluated at training inputs $\mathcal{X}$. All covariance matrices implicitly depend on the kernel hyperparameters $\boldsymbol{\theta}$ [45].

By maximising the marginal likelihood given the training targets $\mathcal{Y}$, namely the probability of observing the data conditioned only on the kernel hyperparameters, the optimal values of $(\boldsymbol{\theta}, \sigma_\epsilon)$ can be estimated in the following way:

$$\begin{aligned}(\hat{\boldsymbol{\theta}}, \hat{\sigma}_\epsilon) &= \arg\max_{\boldsymbol{\theta},\sigma_\epsilon} \log p(\mathcal{Y}|\mathcal{X};\boldsymbol{\theta}) = \\ &= \arg\max_{\boldsymbol{\theta},\sigma_\epsilon} \Big\{ -\mathcal{Y}^\top (K_{\mathcal{X},\mathcal{X}|\boldsymbol{\theta}} + \sigma_\epsilon^2 I)^{-1}\mathcal{Y} - \log|K_{\mathcal{X},\mathcal{X}|\boldsymbol{\theta}} + \sigma_\epsilon^2 I| \Big\}\end{aligned} \tag{2.8}$$

Optimising the GP hyperparameters $\sigma$ and $\boldsymbol{\theta}$ requires to solve the linear system $(K_{\mathcal{X},\mathcal{X}} + \sigma_\epsilon^2 I)^{-1}\mathcal{Y}$ and computing the log determinant $\log|K_{\mathcal{X},\mathcal{X}|\boldsymbol{\theta}} + \sigma_\epsilon^2 I|$, which can be very expensive in the cases of high-dimensional inputs (e.g., images with thousands of pixels) or big datasets ($M \gg 1$) [4]. In this context, the standard approach is to compute the Cholesky decomposition of the $N \times N$ matrix $K_{\mathcal{X},\mathcal{X}}$, which requires $\mathcal{O}(N^3)$ operations and $\mathcal{O}(N^2)$ storage [45]. In the next section, we present an alternative solution, that improves the expressive power of traditional kernels by leveraging on neural networks to learn deep kernels.

## 2.2. Deep Kernel Learning

To mitigate the limited scalability of traditional GPs to high-dimensional inputs, often referred to as the curse of dimensionality, Deep Kernel Learning [45] (DKL) was developed. By exploiting the nonlinear expressive power of deep NNs, DKL is capable to learn compact data representations, while maintaining the probabilistic features of kernel-based GP models for UQ [4].

Starting from a base kernel $k(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta})$ with hyperparameters $\boldsymbol{\theta}$, we embed a deep NN, which represents a nonlinear mapping from the input $\mathbf{x}$ to the feature space, into the kernel function:

$$k(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta}) \to k(g(\mathbf{x}, \mathbf{w}), g(\mathbf{x}', \mathbf{w}); \boldsymbol{\theta}, \mathbf{w}) \ , \tag{2.9}$$

where $g(\mathbf{x}, \mathbf{w})$ is a non-linear mapping given by a deep architecture, such as a deep convolutional network, parametrised by weights $\mathbf{w}$. Unlike traditional kernels used in GPR, the deep kernel encapsulates a better expressive power, since the deep learning transformation $g(\mathbf{x}, \mathbf{w})$ is capable of capturing non-stationary and hierarchical structures [45]. As for traditional GPs, different kernel functions can be chosen and both the GP hyperparameters and the NN parameters are jointly trained by maximising a marginal likelihood. Therefore, DKL still suffers from computational inefficiencies, disqualifying the method for very large ($M \gg 1$) datasets [4]. Additionally, stochastic gradient descent

or other stochastic training strategies are not available for DKL models [13] if we choose non-Gaussian likelihoods, which cause the posterior to be intractable.

To overcome these limitations, Stochastic Variational Deep Kernel Learning [46] (SVDKL) was introduced. SVDKL utilises variational inference [40] to approximate the posterior distribution with the best fitting Gaussian to a set of inducing data points sampled from the posterior. Similar to the framework proposed by Botteghi et al. in [4], our model is built upon SVDKL for the same main reasons: kernel-based models as GPs offer better quantification of uncertainty [40] with respect to deterministic NNs, and it is computationally cheaper than Bayesian NNs when a deep NN architecture is integrated [23].

## 2.3. Multi-fidelity methods

So far, we have seen the advantages of GPR in terms of uncertainty quantification and data-driven surrogate modeling, and how to mitigate its main drawback by exploiting the remarkable expressive power of NNs with deep kernels. We now want to be able to consider a variety of data sources, which display different accuracy with respect to the true dynamical system, and exploit their correlation.

GPR with training data from different fidelity levels is known as co-kriging [21]. The core idea is to leverage a large amount of low-fidelity (LF) data during training, in a setting in which a limited number of high-fidelity (HF) samples are available. Since LF evaluations are cheap, either in terms of time or computational effort, the cost of training data preparation can be reduced by controlling the number of HF evaluation [16].

Assuming a linear correlation between the different fidelity levels, we can employ the Linear Model for Coregionalisation [1] (LMC), that expresses the prior of a hierarchy of $L$ fidelities as

$$Z_l(\mathbf{x}) = \sum_{i=0}^{L-1} a_{l,i} u_i(\mathbf{x}) \,, \quad l = 0, 1, ..., L-1, \tag{2.10}$$

namely, each level of solution $Z_l(\mathbf{x})$ can be written as a linear combination of $L$ independent GPs $u_i \sim \mathcal{GP}(0, k_i(\cdot,\cdot))$. The vector $\mathbf{a}_i = (a_{0,i}, ..., a_{L-1,i})^\top$, $0 \leq i \leq L-1$, collects the weights of the corresponding GP component $u_i$ and the matrix-valued kernel of the multi-fidelity GPR model assumes the form

$$K(\mathbf{x}, \mathbf{x}') = \sum_{i=0}^{L-1} \mathbf{a}_i \mathbf{a}_i^\top k_i(\mathbf{x}, \mathbf{x}') \,. \tag{2.11}$$

In the two-level case, by considering the special form $\mathbf{a}_0 = (1, 0)^\top$, $\mathbf{a}_1 = (\rho, 1)^\top$, we can

retrieve the well-known Autoregressive model AR(1)-cokriging [21] defining the prior of a LF solution $Z_0$ and a HF $Z_1$:

$$Z_0(\mathbf{x}) = u_0(\mathbf{x})\,, \tag{2.12}$$
$$Z_1(\mathbf{x}) = \rho u_0(\mathbf{x}) + u_1(\mathbf{x})\,. \tag{2.13}$$

Notice that LMC allows for much more general linear combinations, if the situation suggests a less strict hierarchical structure among the levels of fidelity. For instance, if we have two different LF sources of data that exhibit a comparable accuracy, and a single HF source, we can consider the following configuration:

$$Z_0(\mathbf{x}) = u_0(\mathbf{x})\,, \tag{2.14}$$
$$Z_1(\mathbf{x}) = u_1(\mathbf{x})\,, \tag{2.15}$$
$$Z_2(\mathbf{x}) = \rho_0 u_0(\mathbf{x}) + \rho_1 u_1(\mathbf{x}) + u_2(\mathbf{x})\,. \tag{2.16}$$

in which $\mathbf{a}_0 = (1,0,0)^\top$, $\mathbf{a}_1 = (0,1,0)^\top$, $\mathbf{a}_2 = (\rho_0, \rho_1, 1)^\top$. In this case, the models $Z_0$ and $Z_1$, associated with the two LF levels, are independent, while the HF model $Z_2$ leverages on all the three GP components. We will exploit this notion later, to evaluate our strategy on a more general hierarchical configuration of data sources.

## 2.4. Levina-Bickel algorithm

As we will see later, our framework involves dimensionality reduction, performed by a specific type of NN called AE [13]. AE are able to learn a nonlinear map between the high-dimensional data space and a low-dimensional state space, through a NN called encoder, and an inverse mapping through a decoder. Instead of fixing the latent state space hyperparameter, we prefer to automatically learn it by estimating the true system state space dimension via the Levina-Bickel algorithm [26]. In the following, we will refer to such estimate as the Intrinsic Dimensionality (ID).

The Levina-Bickel algorithm for ID estimation provides a maximum likelihood estimator (MLE) of the dimension $m$ from i.i.d. observations $\mathbf{z}_1, ..., \mathbf{z}_n$ in $\mathbb{R}^p$, where, in our framework, $\mathbf{z}_1, ..., \mathbf{z}_n$ are the latent vectors collected from the trained AE model.

The observations (latent vectors) represent an embedding of a lower-dimensional sample, i.e. $\mathbf{z}_i = g(\mathbf{s}_i)$, where $\mathbf{s}_i$ are sampled from an unknown smooth density $f$ on $\mathbb{R}^{ID}$, with unknown $ID \leq p$, and $g$ is a continuous and sufficiently smooth (but not necessarily globally smooth) mapping.

The key idea is to fix a point $\mathbf{z}$, assume $f(\mathbf{z}) \approx const$ in a small sphere $S_{\mathbf{z}}(R)$ of radius $R$ around $\mathbf{z}$, and treat the observations as a homogeneous Poisson process in $S_{\mathbf{z}}(R)$. On the basis of this observation, the Levina–Bickel algorithm derives the local ID estimator near $\mathbf{z}$ as

$$ID_{L-B}(\mathbf{z}) = \frac{1}{k-2} \sum_{j=1}^{k-1} \log \frac{T_k(\mathbf{z})}{T_j(\mathbf{z})} \quad , \tag{2.17}$$

where we are fixing the number of neighbors $k$ rather than the radius of the sphere $R$, and $T_k(\mathbf{z})$ is the Euclidean distance between $\mathbf{z}$ and the its $k$-th nearest neighbor in $\mathbf{z}_1, ..., \mathbf{z}_n$.

The actual derivation from the likelihood equations divides by $k-1$, rather than $k-2$. However, we prefer this correction to the formulation, since it makes the estimator asymptotically unbiased [26].

For some applications, one may want to evaluate local dimension estimates at every data point, or average estimated dimensions within data clusters. However, we assume that all the data points come from the same "manifold", and therefore average over all observations.

The choice of $k$ clearly affects the estimate and, in general, the sphere should be small and contain sufficiently many points, for the estimator to work well. We could, for instance, average over a range of values $k$, or implement more elaborate solutions to choose $k$ automatically. In this work, we decide to simply fix $k = 20$.

Finally, the global $ID$ estimator is calculated as

$$ID_{L-B} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{k-2} \sum_{j=1}^{k-1} \log \frac{T_k(\mathbf{z}_i)}{T_j(\mathbf{z}_i)} \quad . \tag{2.18}$$

In the next chapter, we introduce a model that conjugates the concepts introduced so far, namely the availability of a variety of data sources in a multi-fidelity fashion, with the flexibility and expressive power of SVDKL methods, to fully exploit high-dimensional measurements of nonlinear dynamical systems. We will later test it on a variety of cases, on its ability to leverage multiple data sources in the task of learning the dynamics of an unknown dynamical system.

# 3 | Methods

In this work, we consider the following nonlinear dynamical system:

$$\frac{d}{dt}\mathbf{s}(t) = \mathcal{F}(\mathbf{s}(t)), \quad \mathbf{s}(t_0) = \mathbf{s}_0, \quad t \in [t_0, t_f], \tag{3.1}$$

where $\mathbf{s}(t) \in \mathcal{S} \subset \mathbb{R}^d$ is the state vector at time $t$, $\mathcal{F} : \mathcal{S} \to \mathcal{S}$ is a nonlinear function determining the evolution of the system given the current state $\mathbf{s}(t)$, $\mathbf{s}_0$ is the initial condition and $t_0$ and $t_f$ are the initial and final time, respectively. While the state $\mathbf{s}(t)$ is not directly accessible in general and the function $\mathcal{F}$ is usually unknown, we can obtain information about the system indirectly through measurements from different sensor devices. Assuming a discretisation in time of the measurements, we indicate with $\mathbf{x}_l^{(t)}$ the measurement vector of fidelity level $l$, at the time-step $t$, and $\mathbf{x}_l^{(t+1)}$ the measurement (of the same fidelity level $l$) at time $t+1$.

Given $L$ sets of $N_l$ $d_l$-dimensional measurements $\mathbf{X}_l = (\mathbf{x}_l^1, ..., \mathbf{x}_l^{N_l}) \in \mathcal{X}_l, \mathbf{x}_l^i \in \mathbb{R}^{d_l}$, with $d_l \gg 1$ and $l$ being the fidelity level, $0 \leq l \leq L$, we consider the problem of identifying a meaningful representation of the hidden states and a surrogate model for $\mathcal{F}$, exploiting not only the HF data, but also partial or less accurate measurements, in a multi-fidelity fashion.

In practice, we will build a number of surrogate (sub-)models $Z_l$, each one trained on a specific set of measurements $\mathbf{X}_l$ and leveraging the knowledge on the system acquired by the previous less accurate models $Z_i$, $i < l$.

Each sub-model is composed of two components: a SVDKL AE, that maps each high-dimensional measurement to a distribution over the latent state space, and a SVDKL dynamical model, that learns the dynamics of the system using the latent state variables. The learnt latent representations of both the state and the dynamics are then used as inputs for the subsequent submodels, to bound the dimensionality of the latent space and correct its learning process.

Figure 3.1 shows a scheme of the model, considering two levels of fidelity, for ease of reading.

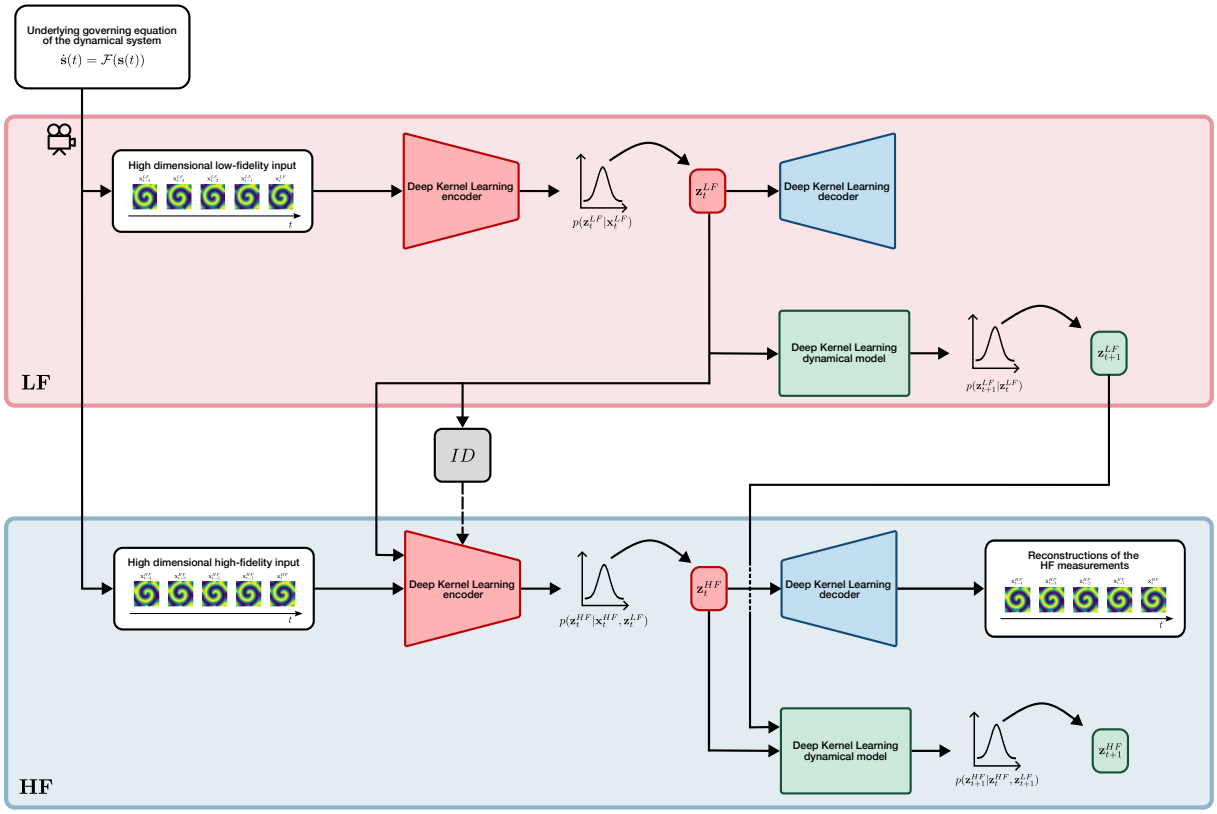



Figure 3.1: The model is composed of two main parts, receiving LF and HF inputs, respectively. Each part includes an AE and a dynamical model. The outputs of the LF sub-model, are used from the HF one and to estimate ID.

The model architecture is described in detail in the following sections, considering an arbitrary number of levels. In particular, Section 3.1 introduces the SVDKL AE, to map the high-dimensional measurements into a low-dimensional space; Section 3.2 describes the SVDKL dynamical model, that learns the dynamics of the system. The loss function used during the training phase is presented in Sections 3.3 and 3.3.1, while a strategy for the estimation of the intrinsic dimensionality of the system is proposed in Section 3.4. Finally, a generalisation beyond the strict chain hierarchy of data sources is introduced in Section 3.5.

## 3.1. Learning the hidden state

We introduce a SVDKL encoder $E_l : \mathcal{X}_l \to \mathcal{L}_l$ that maps the high-dimensional measurements and the previous latent state representation into distributions over a low-dimensional latent space $\mathcal{L}_l$, where $l$ indicates the fidelity level. A latent state sample can be obtained as:

$$z_{i,l}^{(t)} = Z_i^{E_l}(\mathbf{x}_l^{(t)}) + \varepsilon_{E_l}\,, \quad \varepsilon_{E_l} \sim \mathcal{N}(0, \sigma_{E_l}^2)\,, \tag{3.2}$$

$$Z_{l,i}^{E_l}(\mathbf{x}_l^{(t)}) \sim \mathcal{GP}(\mu(g_{E_l}(\mathbf{x}_l^{(t)}; \mathbf{w}_{E_l})), k(g_{E_l}(\mathbf{x}_l^{(t)}; \mathbf{w}_{E_l}), g_{E_l}(\mathbf{x}_l^{(t')}; \mathbf{w}_{E_l}); \boldsymbol{\theta}_{E_l,i}))\,, \quad 1 \leq i \leq |\mathbf{z}_l| \tag{3.3}$$

where $z_{i,l}^{(t)}$ is the sample from the $i^{th}$ GP with kernel $k$ and mean $\mu$, $g_{E_l}(\mathbf{x}_l^{(t)}; \mathbf{w}^{E_l})$ is the feature vector output of the NN part of the SVDKL encoder $E_l$, $\varepsilon_{E_l}$ is an independently added noise and $|\mathbf{z}_l|$ indicates the dimension of $\mathbf{z}_l$ [4].

Since we have no access to the actual state values, the parameters $(\mathbf{w}_{E_l}, \boldsymbol{\theta}_{E_l}, \sigma_{E_l}^2)$ are optimised with a decoder $D_l : \mathcal{L}_l \times \mathcal{L}_{l-1} \to \mathcal{X}_l$ that reconstructs the measurements given the latent state samples corrected with the latent state representation at the previous fidelity level, namely the input of the decoder $D_l$ is

$$\tilde{\mathbf{z}}_l = \mathbf{z}_l + \rho \mathbf{z}_{l-1}\,. \tag{3.4}$$

The reconstructions $\hat{\mathbf{x}}_l$ are also used to generate gradients for the encoder $E_l$. While the SVDKL encoder $E_l$ learns $p(\mathbf{z}_l^{(t)}|\mathbf{x}_l^{(t)})$, the decoder $D_l$ learns the inverse mapping $p(\hat{\mathbf{x}}_l^{(t)}|\tilde{\mathbf{z}}_l^{(t)})$ in which $\hat{\mathbf{x}}_l^{(t)}$ is the reconstruction of $\mathbf{x}_l^{(t)}$.

We can define the loss function as

$$\text{loss}_{E_l}(\mathbf{w}_{E_l}, \boldsymbol{\theta}_{E_l}, \sigma_{E_l}^2, \boldsymbol{\theta}_{D_l}) = \mathbb{E}_{\mathbf{x}_l^{(t)} \sim \mathbf{X}_l}[-\log p(\hat{\mathbf{x}}_l^{(t)}|\tilde{\mathbf{z}}_l^{(t)})]\,, \tag{3.5}$$

where $\hat{\mathbf{x}}_l^{(t)}|\tilde{\mathbf{z}}_l^{(t)} \sim \mathcal{N}(\mu_{\hat{\mathbf{x}}_l^{(t)}}, \Sigma_{\hat{\mathbf{x}}_l^{(t)}})$. By minimising the loss function with respect to the encoder and decoder parameters, we can obtain a compact representation of the measurements. While $|\mathbf{z}_0|$ is an hyperparameter to set, for the subsequent sub-models the dimension of the latent space is automatically obtained by estimating the intrinsic dimension of the system from the latent state representation of the previous level of fidelity, as we will discuss more in detail later.

## 3.2. Learning the latent dynamics

The second component of each sub-model is a surrogate model $F_l : \mathcal{L}_l \times \mathcal{L}_{l-1} \to \mathcal{L}_l$, still based on a SVDKL architecture, that aims to learn and predict the system evolution given the latent state variables sampled from $\mathcal{L}_l$ and the next latent state $\mathbf{z}_{l-1}^{(t+1)}$ sampled from $\mathcal{L}_{l-1}$, where $l$ is the current fidelity level. The next latent states, at fidelity level $l$, $\mathbf{z}_l^{(t+1)}$ can be sampled with $F_l$:

$$z_{i,l}^{(t+1)} = Z_i^{F_l}(\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)}) + \varepsilon_{F_l}\,, \quad \varepsilon_{F_l} \sim \mathcal{N}(0, \sigma_{F_l}^2)\,, \tag{3.6}$$

$$\begin{aligned} &Z_i^{F_l}(\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)}) \sim \\ &\quad \mathcal{GP}(\mu(g_{F_l}(\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)}; \mathbf{w}_{F_l})), k(g_{F_l}(\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)}; \mathbf{w}_{F_l}), g_{F_l}(\mathbf{z}_l^{(t')}, \mathbf{z}_{l-1}^{(t'+1)}; \mathbf{w}_{F_l}); \boldsymbol{\theta}_{F_l,i}))\,, \\ &\qquad\qquad 1 \le i \le |\mathbf{z}_l| \end{aligned} \tag{3.7}$$

where $z_{i,l}^{(t+1)}$ is sampled from the $i^{th}$ GP, $g_{F_l}(\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)}; \mathbf{w}^{F_l})$ is the feature vector output of the NN part of the SVDKL dynamical model $F_l$, $\varepsilon_{F_l}$ is an independently added noise and $|\mathbf{z}_l|$ indicates the dimension of $\mathbf{z}_l$.

Due to the unsupervised nature of our strategy, since we only have access to the sequence of measurements at different time-steps, the learning process of the dynamics employs the encoding of the measurement $\mathbf{x}_l^{(t+1)}$ through the SVDKL encoder $E_l$ to the distribution $p(\mathbf{z}_l^{(t+1)}|\mathbf{x}_l^{(t+1)})$, and uses such a distribution as target for $p(\mathbf{z}_l^{(t+1)}|\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)})$ produced by the dynamical model $F_l$. In particular, $F_l$ is trained by minimising the Kullback-Leibler divergence between the distributions $p(\mathbf{z}_l^{(t+1)}|\mathbf{x}_l^{(t+1)})$ and $p(\mathbf{z}_l^{(t+1)}|\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)})$. The loss is given by:

$$\text{loss}_{F_l}(\mathbf{w}_{F_l}, \boldsymbol{\theta}_{F_l}, \sigma_{F_l}^2) = \mathbb{E}_{\mathbf{x}_l^{(t)}, \mathbf{x}_l^{(t+1)} \sim \mathbf{X}_l}[\text{KL}[p(\mathbf{z}_l^{(t+1)}|\mathbf{x}_l^{(t+1)})||\, p(\mathbf{z}_l^{(t+1)}|\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)})]]\,, \tag{3.8}$$

where $p(\mathbf{z}_l^{(t+1)}|\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)})$ is obtained by feeding a sample from $p(\mathbf{z}_l^{(t)}|\mathbf{x}_l^{(t)})$ and from

$p(\mathbf{z}_{l-1}^{(t+1)}|\mathbf{z}_{l-1}^{(t)}, \mathbf{z}_{l-2}^{(t+1)})$ to $F_l$.

We also reintroduce the same loss function encountered in the SVDKL AE, by passing the latent state prediction $\mathbf{z}_l^{(t+1)}$ obtained from $F_l$ to the decoder $D_l$ and minimising the error between the reconstruction $\hat{\mathbf{x}}_l^{(t+1)}$ and the measurement $\mathbf{x}_l^{(t+1)}$:

$$\tilde{\text{loss}}_{F_l}(\mathbf{w}_{F_l}, \boldsymbol{\theta}_{F_l}, \sigma_{F_l}^2) = \mathbb{E}_{\mathbf{x}_l^{(t)}, \mathbf{x}_l^{(t+1)} \sim \mathbf{X}_l}[-\log p(\hat{\mathbf{x}}_l^{(t+1)}|\mathbf{z}_l^{(t+1)}, \mathbf{z}_{l-1}^{(t+1)})]\,. \tag{3.9}$$

## 3.3. Training of the models

The two components $E_l$ and $F_l$ are jointly trained, allowing the gradients of the dynamical model $F_l$ to flow through the encoder $E_l$ as well. The overall loss function is given by:

$$\begin{aligned}&\text{loss}_l(\mathbf{w}_{E_l}, \boldsymbol{\theta}_{E_l}, \sigma_{E_l}^2, \boldsymbol{\theta}_{D_l}, \mathbf{w}_{F_l}, \boldsymbol{\theta}_{F_l}, \sigma_{F_l}^2) = \\ &\mathbb{E}_{\mathbf{x}_l^{(t)}, \mathbf{x}_l^{(t+1)} \sim \mathbf{X}_l}[-\log p(\hat{\mathbf{x}}_l^{(t)}|\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t)}) + \beta \text{KL}[p(\mathbf{z}_l^{(t+1)}|\mathbf{x}_l^{(t+1)})||p(\mathbf{z}_l^{(t+1)}|\mathbf{z}_l^{(t)}, \mathbf{z}_{l-1}^{(t+1)}]+ \\ &\qquad - \log p(\hat{\mathbf{x}}_l^{(t+1)}|\mathbf{z}_l^{(t+1)}, \mathbf{z}_{l-1}^{(t+1)})]\,,\end{aligned} \tag{3.10}$$

in which $\beta$ is used to scale the contribution of the two loss terms.

Once the sub-model at fidelity level $l$ is trained, it is evaluated on a portion of the training data $\mathbf{X}_l$ available also at level of fidelity $l+1$. The learnt latent representations of both the hidden state and the dynamics are used as additional inputs during the training of the sub-model $l+1$. In the particular case of the sub-model at level $l=0$, null vectors are used as additional inputs, since no information has been learnt so far. The last sub-model $L-1$ is trained on the (few) measurements of highest accuracy, leveraging on all the low-fidelity data.

### 3.3.1. Variational inference

Since the two SVDKL components exploit variational inference to approximate the aforementioned posterior distributions, we add two extra terms to the loss function, one for each SVDKL component, of the form

$$\text{loss}_{var}(\mathbf{w}, \boldsymbol{\theta}) = \text{KL}[p(\mathbf{v})||q(\mathbf{v})]\,, \tag{3.11}$$

where $p(\mathbf{v})$ is the posterior to be approximated over the inducing points $\mathbf{v}$, and $q(\mathbf{v})$ represents an approximating candidate distribution. The inducing points $\mathbf{v}$ are placed on a grid, following the intuition in the original SVDKL work [46].

## 3.4. Intrinsic dimension of the system

In order to produce a meaningful representation of the hidden state variables of the system, we would want to retrieve a number of latent variables as close as possible to the dimension of the actual state space. Similarly to the work by Chen at al. [9], we employ the Levina-Bickel algorithm [26] to estimate the intrinsic dimension (ID) of the system.

The Levina-Bickel algorithm is applied to the latent representations of the states $\mathbf{z}_l^t$ and $\mathbf{z}_l^{t+1}$ obtained from the SVDKL AE and the latent forward dynamics learnt by the SVDKL dynamical model. Finally, a rounded average of the three estimates is considered as ID estimate at fidelity level $l$.

When constructing the sub-model at level $l+1$, the ID estimate at level $l$ is employed as dimension of the latent space $|\mathbf{z}|$, restricting the new latent space to the desired dimension.

## 3.5. Beyond the chain hierarchy

So far, we have assumed a strict chain hierarchy among the levels of fidelity: the measurements $\mathbf{X}_l$ at level $l$ are supposedly less accurate than the measurements $\mathbf{X}_{l+1}$. While this is convenient in some cases, in most of practical applications it often happens that different sensors produce similarly accurate but partial measurements, imposing a less rigid configuration of the sub-models.

To address these cases we can leverage on the generality of the LMC previously mentioned. If two different LF sensors exhibit a comparable degree of accuracy, we can separately train their respective models, $Z_0$ and $Z_1$, in the classical single-fidelity fashion. Consider now a linear combination of their respective latent state representations, $\mathbf{z} = \rho_0\mathbf{z}_0 + \rho_1\mathbf{z}_1$, as additional LF input for the HF model $Z_2$.

Following this simple intuition, we can easily generalise our framework and exploit much more complex hierarchical fidelity structures of measurements.

We conclude this chapter presenting Algorithm 3.1, that summarises the main steps of the model, considering two levels of fidelity. We use the notation $\mathbf{z}$ and $\mathbf{z}^{next}$ to indicate the latent representations obtained from the SVDKL AE applied on the measurements at times $t$ and $t+1$ respectively; $\mathbf{z}^{fwd}$ indicates the latent representation obtained from the SVDKL dynamical model.

The algorithm can be easily generalised, with a slightly heavier notation, to the case of an arbitrary number of levels, not necessarily in a strict chain hierarchy.

Algorithm 3.1 Multi-Fidelity DKL Algorithm

**for** level = LF, HF **do**
  **if** level == LF **then**
    Initialise $\mathbf{z}_{LF}, \mathbf{z}_{LF}^{next}, \mathbf{z}_{LF}^{fwd} = \mathbf{0}$;
    Set latentDimension to an arbitrary value;
  **else**
    Initialise $\mathbf{z}_{LF} = \mathbf{z}$, $\mathbf{z}_{LF}^{next} = \mathbf{z}^{next}$, $\mathbf{z}_{LF}^{fwd} = \mathbf{z}^{fwd}$;
    Set latentDimension = ID;
  **end if**

  **for** epoch = 1, 2, ... **do**
    Encode $\mathbf{x} \rightarrow \mathbf{z}$, $\mathbf{x}^{next} \rightarrow \mathbf{z}^{next}$;
    Decode $\hat{\mathbf{x}} \leftarrow \mathbf{z} + \rho\mathbf{z}_{LF}$, $\hat{\mathbf{x}}^{next} \leftarrow \mathbf{z}^{next} + \rho\mathbf{z}_{LF}^{next}$;
    Estimate $\mathbf{z} + \rho\mathbf{z}_{LF}^{fwd} \rightarrow \mathbf{z}^{fwd}$;
    Decode $\hat{\mathbf{x}}^{fwd} \leftarrow \mathbf{z}^{fwd}$;
    Compute the $loss(\mathbf{x}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^{next}, \hat{\mathbf{x}}^{fwd}, \mathbf{z}^{next}, \mathbf{z}^{fwd})$;
    Backward propagate the gradients;
  **end for**

  **if** level == LF **then**
    Compute ID = $\lceil \frac{1}{3}LB(\mathbf{z}) + \frac{1}{3}LB(\mathbf{z}^{next})) + \frac{1}{3}LB(\mathbf{z}^{fwd}))\rceil$;
  **end if**
**end for**

# 4 | Implementation details

In this chapter, we introduce the Python implementation of the Multi-fidelity Deep Kernel Learning model. This module extends the code provided by Botteghi et al. in [4] to the multi-fidelity framework, considering additional test cases and introducing a new user interface.

The module is composed of three main parts, each thought to be accessed through a dedicated Python script:

1. the dataset generating part, to produce the data for the training and testing of the model;
2. the training of the model, to fit the model exploiting a dedicated training dataset;
3. the testing of the model, to test the goodness of fit of the trained model and produce some reconstructions from the dedicated testing dataset.

**Warning**: this implementation, and in particular the training section, is computationally intensive. The use of a GPU is recommended.

The code can be accessed through GitHub [24].

## 4.1. Folder structure

The Git repository contains the following files and folders:

- `README.md`, with complete instructions on how to run the code;
- `config.yaml`, to set the test case environment, the hyperparameters of the model and some test related settings;
- `generate_dataset.py`, to produce the train and test data, according to the specified test case;
- `main-gym.py`, `main-pde.py`, to initialize and train the model on the generated train dataset;

- `test-gym.py`, `test-pde.py`, `test_utils.py`, to load the weights and test the model on the test dataset, producing some plots and reconstructions;
- `requirements.txt`, that lists the required libraries to run the module;
- `/src` (folder), containing the implementation of the Python module;
- `/Data` (folder), to store the train and test datasets;
- `/Results` (folder), to store the weights of the trained model and the plots produced during the testing phase.

## 4.2. Generate the dataset

The `GenerateDataset` class provides an interface to produce the training and testing datasets, at multiple levels of fidelity and considering the desired test case. A separate pickle file containing pairs of consecutive RGB frames is produced for each dataset and it is stored in the `/Data/env_name/` folder. The user can set the test case through the `env_name` parameter in the `config.yaml` file; two major dataset environments are available: Gym and PDE.

**Gym** The Gym environment leverages on the Gymnasium library to produce one of the following datasets: Pendulum, Acrobot, MountainCarContinuous. A number of independent episodes are simulated, with the dynamical system evolving from random initial conditions. The episodes are produced in a nested fashion, with some available at all the fidelity levels and some only at lower fidelities: LF datasets are larger but less accurate.

**PDE** The PDE environment solves either the reaction-diffusion or the diffusion-advection problem, producing pairs of consecutive RGB frames representing the solution matrix at times $t$ and $t + 1$ plotted as heatmaps. For LF datasets, the solution is observed over a longer timeframe, but on a coarser mesh, and possibly on a larger set of parameters.

### 4.2.1. `GenerateDataset`

The `GenerateDataset` class is implemented as an abstract class [34], with `GenerateGym` and `GeneratePDE` inheriting from it and implementing the abstract method `generate_dataset()`. The parameters can be read from the `config.yaml` file and passed to the `GenerateDataset` object as the `args` dictionary argument.

### GenerateGym

The `GenerateGym` class allows to produce a dataset from one of the available environments of the Gymnasium library [17]: Pendulum, Acrobot and MountainCarContinuous.

The following parameters can be set in the `config.yaml` file:

- `num_episodes`: a dictionary with the number of training episodes for each level of fidelity (an episodes consists of 200 frames, 500 frames and 400 frames, respectively, for the three aforementioned test cases);
- `num_episodes_test`: a dictionary with the number of testing episodes for each level of fidelity;
- `crop`: a list of dictionaries, each associated with a fidelity level and containing:
  - `portion`: the portion of the image to retain (from 0 to 1, 0: none, 1: the entire image);
  - `pos`: the quadrant to crop (1, 2, 3 or 4);
- `occlusion`: a list of dictionaries, each associated with a fidelity level and containing:
  - `portion`: the portion of the image to cover (from 0 to 1, 0: none, 1: the entire image);
  - `pos`: the quadrant to cover (1, 2, 3 or 4);
- `obs_dim_1`: a dictionary with the length of the images, for each fidelity level;
- `obs_dim_2`: a dictionary with the height of the images, for each fidelity level.

The method `generate_dataset()` runs episodes in the chosen Gymnasium environment, adopting random initials conditions and the null action, and it produces a pickle file for each fidelity level, containing a list of dictionaries with keys:

- `obs`: a stack of the observations at times $t-1$ and $t$, as RGB images;
- `next_obs`: a stack of the observations at times $t$ and $t+1$, as RGB images;
- `terminated`: whether the episode is terminated;
- `state`: the state of the system at time $t$;
- `next_state`: the state of the system at time $t+1$.

It leverages on a number of auxiliary private methods, namely, `_new_obs()` to stack to consecutive frames and optionally add occlusion or crop the image, `_log_obs()` to build

the dictionary with two consecutive observations and auxiliary data, `_save_log()` to save the dataset as a pickle file. The Gymnasium environment is initialized by the method `_set_environment()`, called inside the constructor. Optionally, it is possible to save the dataset as png images by passing the argument `png=True` and leveraging on the method `_save_png()`

### GeneratePDE

The GeneratePDE class produces a dataset from either the reaction-diffusion or the diffusion-advection problem.

The following parameters can be set in the `config.yaml` file:

- `d`: a dictionary with the diffusion coefficient for each fidelity level;
- `mu`: a dictionary of lists, each containing the reaction or transport coefficients to use in the training set for each fidelity level;
- `mu_test`: a dictionary of lists, each containing the reaction or transport coefficients to use in the training set for each fidelity level;
- `T`: a dictionary with the training time horizon for each fidelity level;
- `T_test`: the time horizon for testing;
- `dt`: the time step;
- `n`: a dictionary with the number of grid points for each fidelity level.

Once the PDE problem is chosen, the method `generate_dataset` employs a dedicated solver from the class `PDESolver` to solve the equation at each level of fidelity and for each parameter `mu`. Since the testing set chronologically follows the training set, we consider the complete time horizon `T[level] + T_test` when solving the equation.

Solutions related to different values of mu but to the same fidelity level are stacked. Once computed, they are logged leveraging on the auxiliary private methods `_log_level()` and `_log_obs()`. Finally they are saved as pickle files containing a list of dictionaries with keys:

- `obs`: a stack of the observations at times $t$ and $t+1$, as RGB images;
- `next_obs`: a stack of the observations at times $t+1$ and $t+2$, as RGB images;
- `t`: the time $t$;
- `terminated`: whether the episode is terminated.

In this case, the RGB images are heatmaps of the numerical solution.

### 4.2.2. PDESolver

The abstract class `PDESolver` has two implemented child classes: `ReactionDiffusionSolver` and `DiffusionAdvectionSolver`.

#### ReactionDiffusionSolver

The `ReactionDiffusionSolver` class solves the lambda-omega reaction-diffusion system governed by the following equations

$$\dot{u} = \left(1 - \left(u^2 + v^2\right)\right) u + \mu \left(u^2 + v^2\right) v + d \left(u_{xx} + u_{yy}\right), \tag{4.1}$$

$$\dot{v} = -\mu \left(u^2 + v^2\right) u + \left(1 - \left(u^2 + v^2\right)\right) v + d \left(v_{xx} + v_{yy}\right) \tag{4.2}$$

defined over a spatial domain $(x, y) \in [-L, L]^2$ and a time span $t \in [0, T]$, where $\mu$ and $d$ are the reaction and diffusion parameters, respectively. The initial condition is defined as

$$u(x, y, 0) = v(x, y, 0) = \tanh\left(\sqrt{x^2 + y^2} \cos\left((x + iy) - \sqrt{x^2 + y^2}\right)\right). \tag{4.3}$$

The class leverages on the `solve_ivp()` function from the SciPy library [42], using the explicit Runge-Kutta method of order 5 to solve the system, and the `fft2()` and `ifft2()` functions to compute the laplacians. The parameters $n$ and $dt$ define the number of grid points for the spatial mesh and the time step for the evaluation in time of the evolution of the system. The method `_rhs()` computes the discretized right hand side of the equation.

#### DiffusionAdvectionSolver

The `DiffusionAdvectionSolver` class solves an advection-diffusion problem describing a fluid motion in the shallow water limit given by

$$\frac{\partial \omega}{\partial t} + \mu \left(\frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x}\right) = d\nabla^2 \omega, \tag{4.4}$$

$$\nabla^2 \psi = \omega, \tag{4.5}$$

defined over a spatial domain $(x, y) \in [-L, L]^2$ and a time span $t \in [0, T]$. Here $\omega(x, y, t)$ and $\psi(x, y, t)$ represent the vorticity and stream function, respectively, $\nabla^2 = \partial x^2 + \partial y^2$ is

the two-dimensional Laplacian, and we consider the following initial condition of vorticity:

$$\omega(x, y, 0) = \exp\left(-2x^2 - \frac{y^2}{20}\right), \quad (x, y) \in [-L, L]^2. \tag{4.6}$$

The class leverages on the Numpy [30] functions `fft2()` and `ifft2()` for the gradient and the laplacian, and on the SciPy [42] function `odeint()` to solve the system. The parameters $n$ and $dt$ define the number of grid points for the spatial mesh and the time step for the evaluation in time of the evolution of the system. The method `_rhs()` computes the discretized right hand side of the equation.

### 4.2.3. Logger

The `Logger` class logs the generated data and saves it to a pickle [35] file. In particular, the filename and the logging directory are defined inside the constructor, while the method `obslog()` appends the new observation and the method `save_obslog()` saves the logged data to a file.

### 4.2.4. utils

`save_pickle()` This function saves the input data as a pickle file, with filename as defined by the passed parameter.

`stack_frames()` This function stacks two frames together along the channel dimension and performs optional cropping and occlusion.

`crop_frame()` This function crops a frame based on the specified portion and position. The portion is a float taking values in $[0, 1]$ (if `portion=1`, the entire frame is retained), while the position is the number of the quadrant to consider (1: top-right, 2: top-left, 3: bottom-left, 4: bottom-right), cropping from the relative vertex of the original image.

`add_occlusion()` This function adds a black occlusion to the frame, of size defined by the portion $\in [0, 1]$ parameter (if `portion=1`, the entire frame is occluded). The occlusion is generated from the center of the image towards one of the four vertices, relative to the desired quadrant position (1: top-right, 2: top-left, 3: bottom-left, 4: bottom-right).

`len_of_episode()` This function returns the length of an episode in number of frames, for each of the supported Gym test cases. In particular, 200 for the Pendulum, 500 for the Acrobot, 400 for the MountainCarContinuous.

`heatmap_to_image()` This function converts a matrix into its heatmap, returned as RGB image.

## 4.3. Train the model

The training of the model is performed for each fidelity level sequentially. Each stage is composed of the construction of the $l$-th sub-model, its training and, possibly, the estimation of the intrinsic dimension. The class `BuildModel` handles each of these steps and an example of its usage can be found in the `main-gym.py` and `main-pde.py` scripts.

Some configuration parameters and model hyperparameters can be set in the `config.yaml` file and passed to the `BuildModel` object as the `args` dictionary argument.

### 4.3.1. `BuildModel`

The `BuildModel` class initializes, trains and tests the model. Its constructor accepts the following arguments:

- `args`: a dictionary with the parameters read from the `config.yaml` file;
- `use_gpu`: a bool to set whether to use the GPU;
- `test`: a bool to set the model in testing mode and select the correct input dataset;
- `flush`: a bool to optionally flush the print statements.

The following parameters can be set in the `config.yaml` file:

- `seed`: the seed, to make the results reproducible;
- `batch_size`: the size of the batch during the training;
- `max_epoch`: the maximum number of epochs for the training;
- `rho`: the weight of the LF term;
- `training`: whether to activate the training;
- `lr`, `lr_gp`, `lr_gp_lik`, `lr_gp_var`: the learning rates;
- `reg_coef`: the weight decay coefficient for the Adam optimizer;
- `k1`, `k2`: the weights of the loss function;
- `grid_size`: the grid size for the variational strategy;

- `obs_dim_1`, `obs_dim_2`, `obs_dim_3`: the dimension of each observation (typically $84 \times 84 \times 6$);
- `h_dim`: the hidden dimension size;
- `env_name`: the name of the testing case (as defined when the dataset is generated);
- `training_dataset`: the name of the data files;
- `log_interval`: the frequency of saving the partially trained model weights;
- `jitter`: the jitter value to compensate for possible numerical instabilities.

The `BuildModel` constructor also handles the instantiation of the correct DataLoader object, according to the defined testing case.

### `add_level()`

The `add_level()` method initializes the $l$-th sub-model, instantiating the likelihood for both the stochastic variational DKL autoencoder and the dynamical model, as GPyTorch [14] `MultitaskGaussianLikelihood`, and the sub-model itself.

### `train_level()`

The `train_level()` method trains the previously initialized sub-model passed as argument. In particular, it initializes the variational losses for both parts of the sub-model, as `VariationaKL` defined in `variational_inference.py`, and the optimizers. For the non variational parameters, the PyTorch [36] Adam optimizer is adopted, while the Stochastic Gradient Descent optimizer is used for the variational parameters. Schedulers to reduce the learning rate are implemented after half and three quarters of the epochs.

The training part leverages on the train function implemented in the `trainer.py` file, run for a defined number of epochs, and it is performed with the GPyTorch `cholesky_jitter` setting to cure possible numerical instabilities. The weights of the model are saved in a dedicated `.pth` file.

### `get_latent_representations()`

This method evaluates the trained sub-model and returns the latent representation computed from the training dataset. The evaluation is performed in chunks, to avoid memory saturation when the dataset is large, and the dimension of each chunk can be set as argument of the method. The latent representations `z` (of the observation at time $t$), `z_next` (of the observation at time $t+1$) and `z_fwd` (the one-step forward prediction) are

returned.

#### eval_level()

This methods evaluates the trained sub-model and return both the latent representations `z`, `z_next`, `z_fwd` and the relative reconstructions `mu` and `mu_fwd`. The evaluation is performed on the data imported when the object is instantiated (training set if `test=False`, testing set if `test=True`), therefore it is recommended to use this method only on small datasets to avoid memory saturation.

### 4.3.2. DataLoader

The `DataLoader` is implemented as one parent `BaseDataLoader` class and two specialized classes, `GymDataLoader` and `PDEDataLoader`. Their role is to load and manipulate the dataset, including sampling batches for the training.

The common attributes, such as the observations and the latent representations, are handled by the parent classes, from which `GymDataLoader` and `PDEDataLoader` inherit and complete the methods addressing the specifics of the test case they aim to cover.

The constructor reads the data passed as argument, converting it to a PyTorch Tensor and normalizing it. Three additional methods are implemented:

- `sample_batch()`: to randomly sample a batch of desired size;
- `sample_batch_with_idx()`: to sample a batch from a given list of indices;
- `get_all_samples()`: to get all the samples (useful when evaluating the model).

The batches are returned as dictionaries with keys: `obs`, `next_obs`, `z_LF`, `z_next_LF`, `z_fwd_LF`. Both `obs` and `next_obs` are composed of two RGB images concatenated along the color channel. The child class `GymDataLoader` returns also the states of the system at time $t$ and $t+1$ as keys `state` and `next_state`; the child class `PDEDataLoader` returns also the current time instant $t$ as key `t`.

### 4.3.3. trainer

The `train()` function loops over a number of random batches: at each iteration, the gradients are initialized, a forward pass of the model is performed and the loss components are updated. Once the loss is computed, the gradients are backward propagated and the parameters are updated.

Since the autoencoder and the forward model are jointly trained, the gradients of the dynamical model flow through the autoencoder as well.

### 4.3.4. `models`

The final model is implemented by the `SVDKL_AE_latent_dyn()` class, contained inside the `models.py` file, along with the various layers it is composed of. The implementation is an expansion of the model used in [4] that supports the multi-fidelity case, accepting as argument also the low-fidelity latent representations of the input observations. The latent space dimension is either 20, for the sub-model 0, or $ID$, for the following sub-models, as estimated from the previously learnt $(l-1)$-th latent representation.

`SVDKL_AE_latent_dyn` As introduced in the previous chapter, this model is composed of two main layers: a stochastic variational deep kernel learning autoencoder and a forward dynamical deep kernel learning model. The forward pass evaluates the SVDKL AE on both the current and the next observation, and the forward model on the latent representation of the current observation. Finally, the forward latent representation is decoded to an image reconstruction, later used by the loss function.

Two additional methods are implemented to predict the dynamics and the mean dynamics of the system. In particular, the forward model is evaluated and the output forward latent representation is decoded to an image reconstruction.

`SVDKL_AE` The stochastic variational deep kernel learning autoencoder is composed of an encoder, a decoder and a gaussian process layer. The forward pass encodes the input and corrects the resulting latent representation with the corresponding low fidelity one. The features are then passed through the gaussian process layer, which produces a predictive latent state distribution $p(\mathbf{z}_t|\mathbf{x}_t)$. Finally, the latent state vector $\mathbf{z}_t$ is sampled and decoded.

`Encoder` The encoder is composed of four convolutional layers, with 32 filters per layer. The convolutional filters are of size $(3 \times 3)$ and shifted across the images with stride 1. Batch normalization is also used after the 2nd and 4th convolutional layers. The output features of the last convolutional layer are flattened and fed to two final fully connected layers. Each layer has ELU activations, except the last fully-connected layer with a linear activation.

`Decoder` The decoder is composed of a linear fully-connected layer and four transpose convolutional layers with 32 filters each. The convolutional filters are of size $(3 \times 3)$ and shifted across the images with stride 1. Batch normalization is used after the 2nd and 4th convolutional layers, and ELU activations are employed for all the layers except the last one.

`GaussianProcessLayer` The gaussian process layer class inherits from the GPyTorch [14] `ApproximateGP` to implement a GP with constant mean and ARD-SE kernel, which produces a latent state distribution $p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{z}_t^{LF})$. The class leverages on the GPyTorch `GridInterpolationVariationalStrategy`, with the Cholesky variational distribution implemented by the same library.

`Forward_DKLModel` The forward deep kernel learning dynamical model is composed of a neural network and a gaussian process layer. Analogously to the SVDKL encoder, the output features of the neural network are fed to independent GPs to produce a next state distribution $p(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{z}_{t+1}^{LF})$. Again, we sample the next latent states $\mathbf{z}_{t+1}$.

`ForwardModel` The forward model is a traditional neural network composed of 3 fully-connected layers of size 512, 512, and $ID$ (or 20, for the lowest fidelity level), respectively, with ELU activations except the final layer with a linear activation.

### losses

The binary cross entropy and the Kullback-Leibler divergence are implemented as components of the loss function, exploited during the training stage, leveraging on the PyTorch [36] functionals and distributions. The BCE is used to compute the loss between an input and a target tensor, in particular between the observation at time $t$ and its reconstruction, to train the autoencoder, and between the next observation and the decoded forward prediction, to train the dynamical model. The KL divergence is used to compute the loss between two distributions, in particular between the likelihood from the autoencoder trained on the next observation and the likelihood of the dynamical model, to train the latter. We employ the KL balancing, for balancing how much the prior is pulled towards the posterior and vice versa. The implementation is basically the same as the original code in [4].

#### variational_inference

As variational terms of the loss, for both the autoencoder and the dynamical model, we use the `VariationalKL` class implemented in `variational_inference.py`, inherited from the original code by Botteghi et al. in [4]. These two terms are initialized by the `BuildModel` class and updated by the train function during the training stage.

The `VariationalKL` class inherits from the `_ApproximateMarginalLogLikelihood` class, which expands the GPyTorch [14] `MarginalLogLikelihood` class by approximating the marginal log-likelihood. In particular, it gets the KL divergence from the model and return the difference between that and the prior.

#### intrinsic_dimension

A function `eval_id()` to estimate the intrinsic dimension of a dataset is implemented in the `intrinsic_dimension.py` file. It leverages on the Levina-Bickel method, separately implemented in the `Levina_Bickel()` function, along with the k-nearest neighbors algorithm. The function `estimate_ID()` computes the rounded mean of the three estimates of $ID$ from $\mathbf{z}$, $\mathbf{z}_{next}$ and $\mathbf{z}_{fwd}$.

### 4.3.5. utils

`load_pickle()` This function loads the input data from a pickle file, with filename as defined by the passed parameter.

`align_pde()` This function computes indices to align in time two PDE datasets of different levels of fidelity. Indeed, we usually assume that the LF simulations are run for a longer time, but the relative latent representations used to train the HF model should be aligned with the HF data.

`check_indices()` This function checks with an assert if the indices passed as argument are inside the bounds of the input tensor.

## 4.4. Test the model

Once the model is trained, we can test it on an unseen dataset to measure its accuracy in reconstructing and predicting the dynamics of the model. A typical workflow involves instantiating the model as a `BuildModel` object, passing the argument `test=True`, and reconstructing its levels as done during the training stage. The methods `add_level()`,

`test_level()` and `eval_level()` are used in this case.

In a typical setup, we would want to evaluate the level of highest fidelity; to do so, we should use its image reconstructions and the respective LF latent representations, outputs of the previous sub-model(s).

The scripts `test-gym.py` and `test-pde.py` provide a possible workflow to test the results, on the Gym and PDE cases respectively. In particular, it plots the latent variables as functions of time, using the function `plot_latent_dims()`, and the reconstructions at times $t$ and $t+1$ (by the autoencoder and the forward model, respectively), with the relative MSE with respect to the true measurement. A gif is also generated, to better show the evolution in time of the system and the behavior of the model. This testing stage leverages on the Matplotlib library [28] for the plots and exploits the `predict_dynamics_mean()` of the model to compute the forward reconstructions.

By iterating on this method, it is possible to extrapolate in time in an autoregressive fashion, and study the predictive capabilities of the model further away from the seen measurements.

The scripts `test-gym.py` and `test-pde.py` store the results inside the same directory of the trained weights, in a `/plots` subfolder. The following parameters can be set in the `config.yaml` file and passed as `args` to the `BuildModel` object, in addition to the same used for training:

- `testing_dataset`: the name of the data files;
- `results_folder`: the path where to save the model weights;
- `weights_filename`: the name of the file where to store the model weights.

Moreover, the testing script makes use of both the initial latent space dimension and the ID estimate, which should be stored during the training stage and loaded in the testing stage.

### 4.4.1. `BuildModel`

`test_level()` This method loads both the (testing) dataset and the trained weights of the sub-model. If the LF latent representations are not passed as arguments, dummy ones are defined instead. Finally, the data loader is instantiated. Both the data loader and the sub-model are returned, ready to be evaluated with the `eval_level()` method. This method assumes an already instantiated sub-model, using the `add_level()` method.

### 4.4.2. utils

`plot_frame()` This function plots a given frame of the dataset, optionally displaying to the screen and/or storing as a `.png` file. The plots are generated using the Matplotlib library [28].

`plot_latent_dims()` This function receives as input the latent representations of the dataset as a matrix $N \times D$, where $N$ is the number of samples and $D$ is the number of dimensions. The first `dims` dimensions are plotted as functions of time, with a curve for each episode. The length of the episodes and how many to be plotted are specified as arguments. The plots are generated using the Matplotlib library [28]. The function can optionally display the plots to screen and/or save them as `.png` images.

`get_length()` This function return the length of a variable. In particular, it returns 1 if the variable is an int or a float; if the variable is a list, it returns its length.

`generate_gif()` This function loads a number of `.png` images from a given local path and renders them as a `.gif` file, using the Imageio library [18]. The function assumes that the images have progressive integers as filenames.

`plot_error()` This function plots the errors, passed as list, as functions of time, and saves the plot as `.png` image to the desired filepath.

# 5 | Results

In this chapter we assess the model performance on data from low-dimensional dynamical systems, and from high-dimensional dynamical systems from PDE discretisation. In particular, we consider the simple pendulum system, the reaction-diffusion and the diffusion-advection PDE problems.

First, the theoretical details of the problems are introduced, followed by some considerations on the Levina-Bickel estimate of the intrinsic dimension and the behaviour of the latent variables. Then, the model performances are measured in terms of accuracy of the frame reconstruction and one-step forward prediction in time of the dynamics. Additionally, we study the model ability to extrapolate the system evolution in time. In particular, we examine a variety of parameter values, both seen and unseen during the training stage.

## 5.1. Low-dimensional dynamical systems

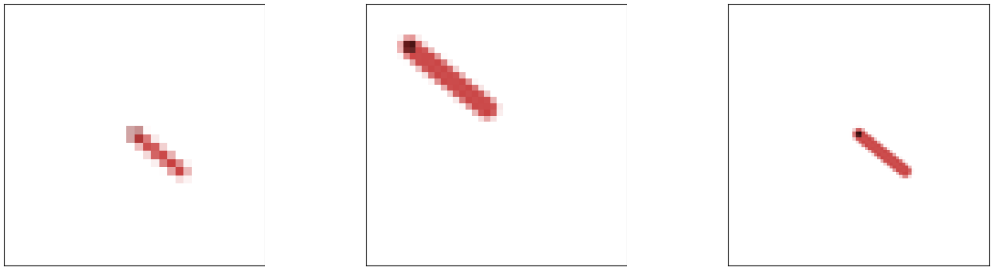
In this section, the main results from the simple pendulum test case are presented. As input data we consider high-dimensional RGB images obtained through RGB cameras of different fidelity, with the system evolution simulated by the Gymnasium Python library [17].

### 5.1.1. Simple pendulum

To demonstrate the predictive capabilities of our framework, the first test-case we consider is the simple pendulum described by the following equation:

$$\ddot{\phi}(t) = -\frac{g}{l} \sin \phi(t) \,, \tag{5.1}$$

where $\phi$ is the angle of the pendulum, $\ddot{\phi}$ is the angular acceleration, $l$ is the length, $g$ is the gravitational acceleration. The starting state is a random angle and a random angular



(a) Fidelity level 0 (b) Fidelity level 1 (c) Fidelity level 2

Figure 5.1: Samples from the simple pendulum pendulum dataset at the same time instant $t$. (a) shows a low-resolution frame; (b) shows an high-resolution but cropped frame, to simulate partial measurements; (c) shows an high-resolution frame.

velocity, in particular

$$\phi(0) \sim \mathcal{U}([-\pi, \pi]) \,, \tag{5.2a}$$

$$\dot{\phi}(0) \sim \mathcal{U}([-1, 1]) \,. \tag{5.2b}$$

Our goal is to approximate the time-dependent state of the pendulum $x_1 = \phi, x_2 = \dot{\phi}$, considering a test set of new episodes, unseen during training.

### 5.1.2. Dataset structure

We adopt three fidelity levels that differ in the resolution of the frame. In particular, the first fidelity level is composed of a number of (full) RGB images of size $l_0 \times l_0 \times 3$; the second level is composed of cropped RGB images of size $l_1 \times l_1 \times 3$, considering the bottom-right 60% portion of the image; the highest level of fidelity collects full RGB images of high resolution size $l_2 \times l_2 \times 3$, where $l_2 > l_0$.

Each dataset is composed of a number $N^{level}$ of episodes, each composed of 200 consecutive frames, with $N^0 > N^1 > N^2$. The initial conditions are randomly initialised at the beginning of each episode.

Figure 5.1 shows three frames from the multi-fidelity datasets, at levels 0, 1 and 2, respectively. Table 5.1 lists the configuration parameters used for generating the dataset.

### 5.1.3. ID and latent variables

Our framework is able to produce an efficient and reliable low-dimensional surrogate of the pendulum system. Indeed, the Levina-Bickel algorithm applied on the two LF

| Parameters | level 0 | level 1 | level 2 |
|---|---|---|---|
| $N$ | 150 | 80 | 50 |
| $l$ | 32 | 42 | 84 |
| $portion$ | 1 | 0.6 | 1 |
| $N_{test}$ | 3 | 3 | 3 |

Table 5.1: List of configuration parameters for the pendulum dataset.

latent representations estimates $ID = 3$, which is slightly larger than the true state-space dimension (=2), but significantly lower than the initial latent space dimension (=20), allowing us to obtain a compact 3-dimensional latent state space for the high fidelity model.

Figure 5.2 shows the three latent variables $\theta_1^2(t)$, $\theta_2^2(t)$ and $\theta_3^2(t)$ as functions of time, for each test episode. We can recognize some periodical patterns in all three variables, as expected. However, only the first two exhibit remarkably low uncertainty, while $\theta_3^2(t)$ displays a larger variance.

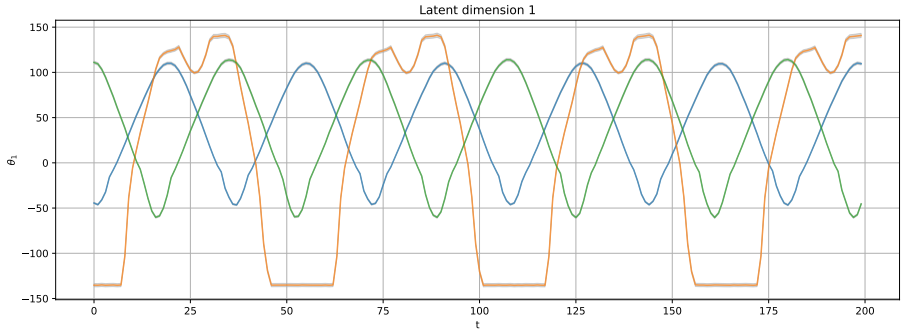
### 5.1.4. Reconstruction and one-step forward prediction

In this subsection we illustrate the effectiveness of the model in reconstructing the frames and predicting forward in time, on new and unseen measurements, generated using a different seed.

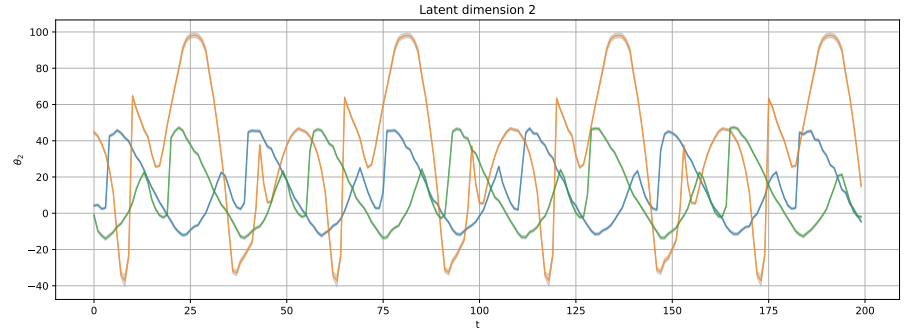
Figure 5.3 shows the reconstruction of the frame $\mathbf{x}_t^{HF}$, produced by the SVDKL Autoencoder, and the one-step forward prediction $\hat{\mathbf{x}}_{t+dt}^{HF}$, generated by the SVDKL Dynamical model, with their respective absolute errors. In particular, it considers some frames from one of the testing episodes.

Visually, both the reconstructions and the predictions are consistent with the actual measurement, without blurring too much the shape of the pendulum. Indeed, the absolute errors are usually displaying a marginal inaccuracy near the border of the shape and the reconstruction MSE is consistently small ($< 3e-3$), with some periodic fluctuations, as shown in Figure 5.4. With respect to the forward predictions, the MSE reaches larger maxima, up to $6e-3$.

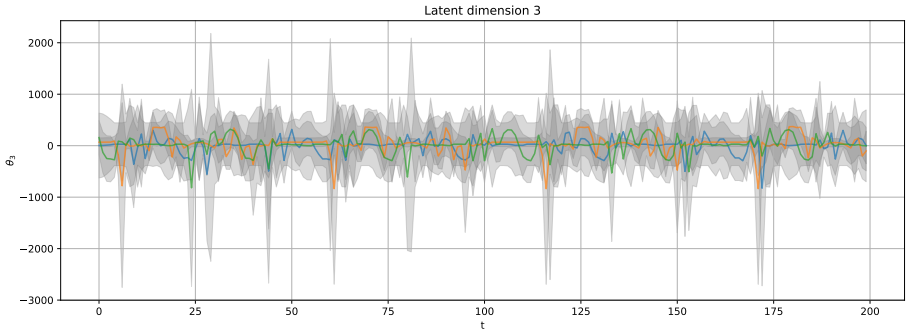
Overall, the model is able to capture most of the system dynamics.




(a) $\theta_1^2(t)$




(b) $\theta_2^2(t)$




(c) $\theta_3^2(t)$

Figure 5.2: Latent variables of the fidelity level 2 autoencoder for the three test episodes, for the pendulum case. The uncertainty bands are given by ± two standard deviation in the predictive distribution. All three variables present some periodical patterns, but while $\theta_1^2(t)$ and $\theta_2^2(t)$ exhibit remarkably low uncertainty, $\theta_3^2(t)$ displays a larger variance.

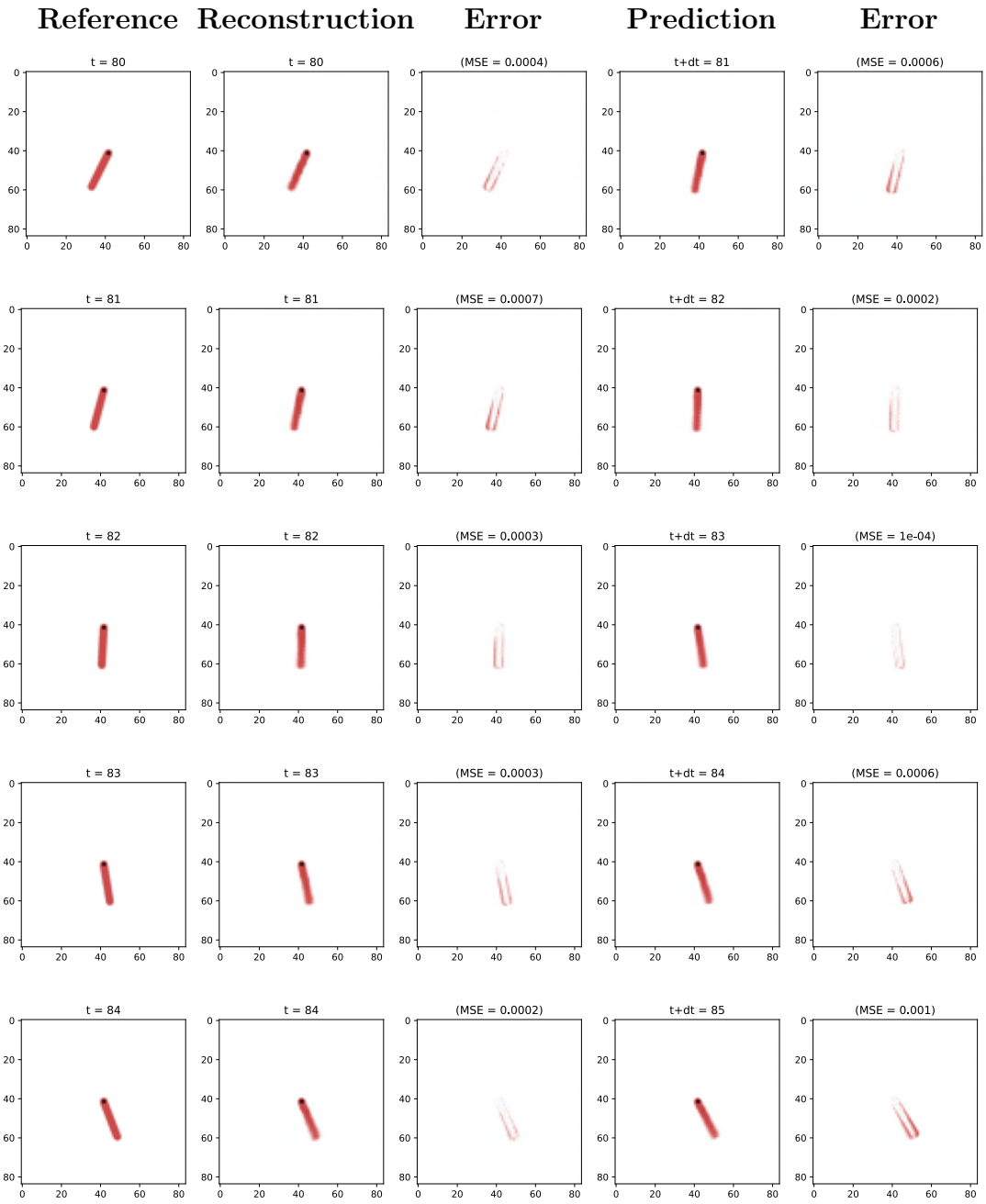



Figure 5.3: Reconstructions and predictions one-step forward in time, with respective absolute errors, of the first test episode, for the pendulum case. Both reconstruction and forward prediction are consistent and accurate during the full oscillation, when considering small angles $\phi$.

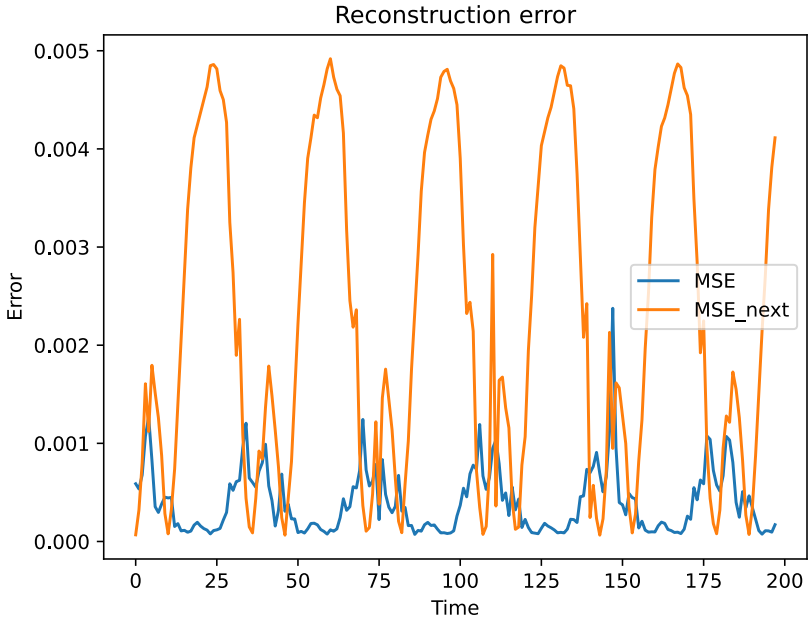



Figure 5.4: MSEs of the reconstruction and forward prediction of the first test episode, for the pendulum case, with respect to the relative true measurements, as function of time. Both MSE display a periodic behavior, while maintaining within acceptable values (<5e-3).

### 5.1.5. Extrapolation in time

If we iterate the one-step forward prediction, by feeding the predicted latent representation to the next iteration, we can extrapolate in time. Figure 5.5 shows some frames, obtained extrapolating in time from the first frame at $t = 0$.

Qualitatively speaking, the prediction remains consistent for a few iterations, only to degrade for the rest of the iterations. Degraded frames present a full, not blurred, shape of the pendulum, but stuck at a previous state of the evolution.

Figure 5.5 shows the first six consistent predictions, that capture the essence of the system dynamics with a full oscillation. However, the absolute error (right column of Figure 5.5) shows that the actual dynamics is slower and it is outpaced by the faster extrapolated one.

The extrapolation MSE, presented in Figure 5.6, is consistently very high, with the exception of sporadic small values that coincide with the actual states close to the constant predicted one, mimicking a broken clock.

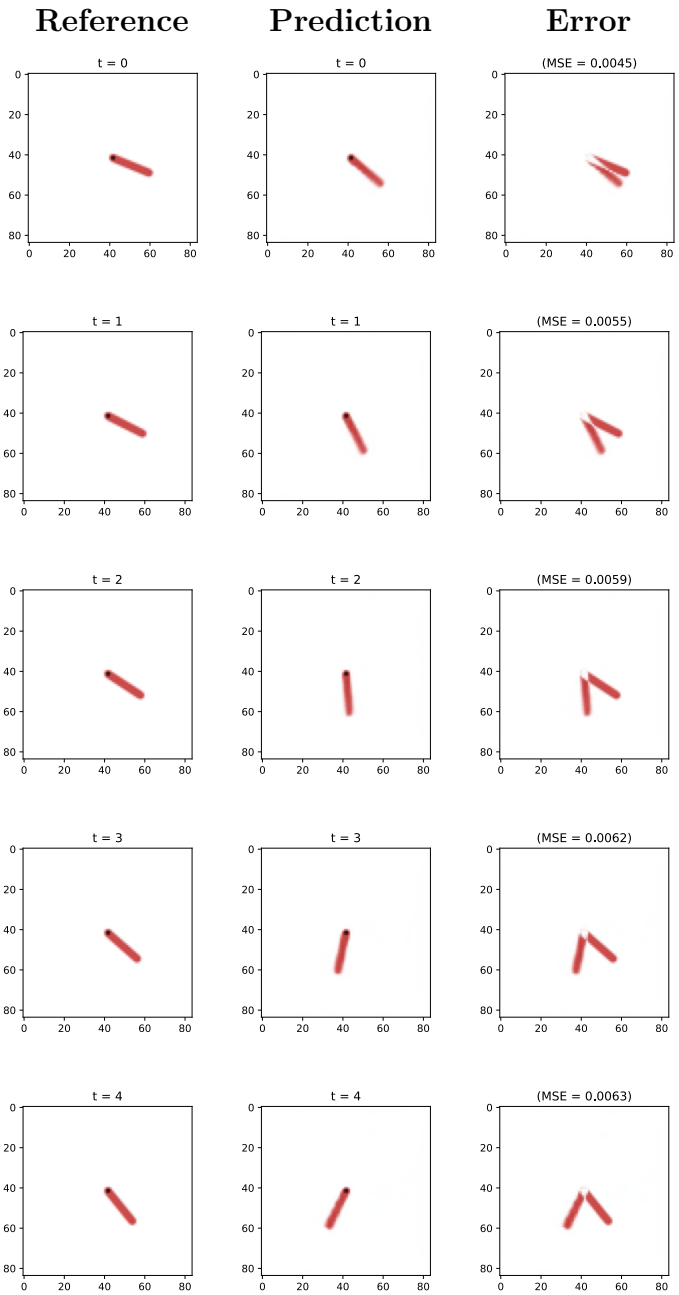



Figure 5.5: Extrapolation in time, for the pendulum case, for a few iterations, with the respective absolute error. While the oscillating dynamics is essentially captured, the extrapolated movement of the pendulum is faster than the measured $\dot{\phi}$.

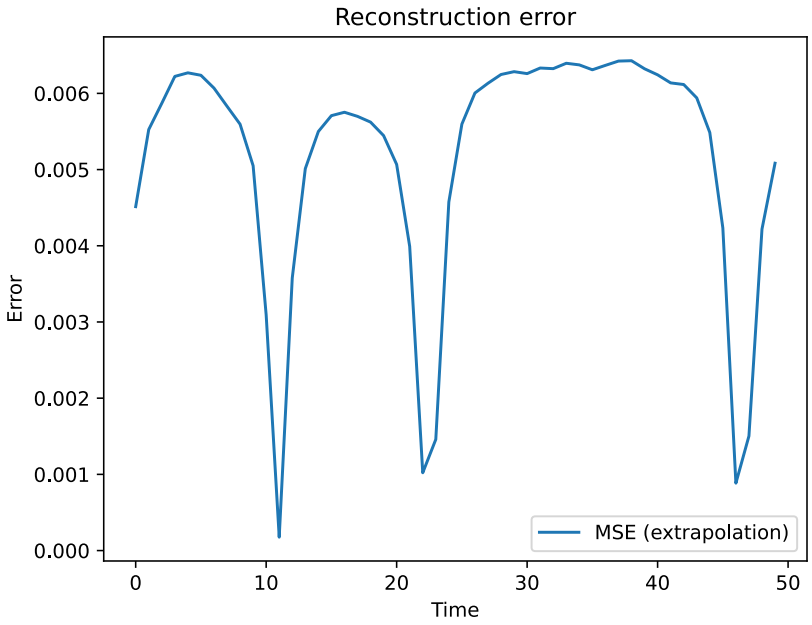



Figure 5.6: MSE of the extrapolation in time, for the pendulum case. The error is consistently high, with sporadic minima that coincide with the predicted constant state that the pendulum periodically returns to.

### 5.1.6. Some considerations

Our framework proves very effective in learning the dynamics of low dimensional dynamical systems, as the pendulum, from high-dimensional measurements. Additionally, the Levina-Bickel algorithm produces an efficient but relatively exhaustive representation of the system state space.

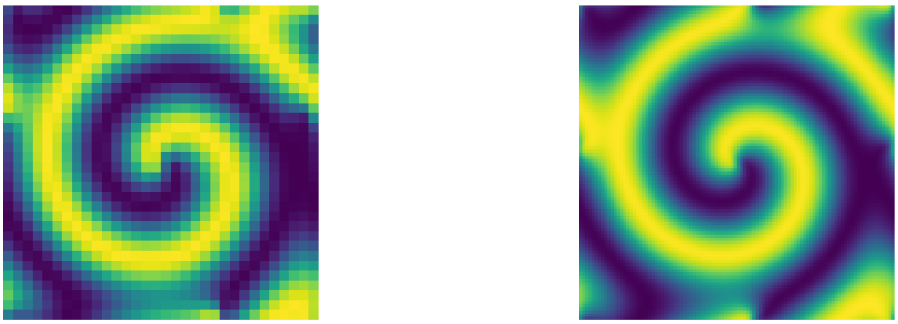
The model is able to leverage a variety of data sources of different fidelity to capture and predict trajectories of the system, unseen during the training stage. However, for this particular class of problems, the task of extrapolation in time remains open, with the model still lacking accuracy.

From a computational standpoint, on the hardware at our disposal, the training phase required a few days of work, considering the dataset size reported in Table 5.1. Overall, we expect this to be a computationally taxing task, since the model follows an unsupervised data-driven approach and requires large datasets.

## 5.2. High-dimensional dynamical systems from PDE discretisation

In this section we consider measurements from high-dimensional dynamical systems from PDE discretisation, in particular, presenting the numerical results from the reaction-diffusion and the diffusion-advection problems. In this case, input data are high-dimensional RGB images, that represent the heatmap of the numerical solution, obtained through a numerical solver for a variety of parameters. In this context, with respect to more accurate measurements, low fidelity datasets are characterised by coarser meshes, but longer periods of observation and larger parameter sets.

### 5.2.1. Reaction-diffusion problem



(a) Low fidelity sample (b) High fidelity sample

Figure 5.7: Samples from the reaction-diffusion dataset: (a) and (b) show two frames from the LF and HF data, respectively, at the same time instant $t$.

We consider a lambda-omega reaction-diffusion system defined by the following equations

$$\dot{u} = \left(1 - \left(u^2 + v^2\right)\right) u + \mu \left(u^2 + v^2\right) v + d \left(u_{xx} + u_{yy}\right), \tag{5.3a}$$

$$\dot{v} = -\mu \left(u^2 + v^2\right) u + \left(1 - \left(u^2 + v^2\right)\right) v + d \left(v_{xx} + v_{yy}\right) \tag{5.3b}$$

defined over a spatial domain $(x, y) \in [-L, L]^2$ and a time span $t \in [0, T]$, where $\mu$ and $d$ are the reaction and diffusion parameters, respectively. The initial condition is defined as

$$u(x, y, 0) = v(x, y, 0) = \tanh\left(\sqrt{x^2 + y^2} \cos\left((x + iy) - \sqrt{x^2 + y^2}\right)\right). \tag{5.4}$$

We are interested in approximating the solution components $u$ and $v$ as functions of the varying reaction parameter $\mu \in \mathcal{P} = [0.5, 1.5]$, with a fixed diffusion coefficient $d$.

### 5.2.2. Diffusion-advection problem

(a) Low fidelity sample (b) High fidelity sample

Figure 5.8: Samples from the diffusion-advection dataset: (a) and (b) show two frames from the LF and HF data, respectively, at the same time instant $t$.

We consider a diffusion-advection problem describing a fluid motion in the shallow water limit given by

$$\frac{\partial \omega}{\partial t} + \mu \left( \frac{\partial \psi}{\partial x} \frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y} \frac{\partial \omega}{\partial x} \right) = d \nabla^2 \omega, \tag{5.5a}$$

$$\nabla^2 \psi = \omega \,, \tag{5.5b}$$

defined over a spatial domain $(x, y) \in [-L, L]^2$ and a time span $t \in [0, T]$. Here $\omega(x, y, t)$ and $\psi(x, y, t)$ represent the vorticity and stream function, respectively, $\nabla^2 = \partial x^2 + \partial y^2$ is the two-dimensional Laplacian, and $d$ is the diffusion coefficient. We consider the following initial condition of vorticity:

$$\omega(x, y, 0) = \exp\left( -2x^2 - \frac{y^2}{20} \right), \quad (x, y) \in [-L, L]^2. \tag{5.6}$$

Our goal is to approximate the time-dependent vorticity field $\omega$ as the parameter $\mu$ varies over $\mathcal{P} = [1, 5]$.

### 5.2.3. Dataset structure

We train the model on a small amount of HF data, with respect to both the time length of observation and the number of considered parameter $\mu$ values, and on a larger amount of

cheaper LF data. In particular, LF data are associated with a coarser grid of discretisation ($n^{LF} < n^{HF}$), but they are observed for a longer time window, $T^{LF}_{train} > T^{HF}_{train}$.

After the training stage, the model is tested on new measurements of the system for the time window $[T^{HF}_{train}, T^{HF}_{train} + T_{test}]$, for values of the parameter $\mu$ seen by the HF sub-model during training, but also extrapolating over $\mathcal{P}$.

Figures 5.7 and 5.8 show two frames from the LF and HF datasets, from the reaction-diffusion and diffusion-advection cases, respectively. Tables 5.2 and 5.3 list the configuration parameters used for generating the reaction-diffusion and diffusion-advection datasets, respectively.

| **Parameters** | **LF** | **HF** |
|---|---|---|
| $n$ | 32 | 100 |
| $d$ | 0.05 | 0.05 |
| $\mu_{train}$ | {0.5, 0.75, 1, 0.6, 1.15, 1.375} | {0.5, 0.75, 1} |
| $T_{train}$ | 80 | 40 |
| $L$ | 10 | |
| $dt$ | 0.01 | |
| $\mu_{test}$ | {0.5, 1, 0.6, 1.15} | |
| $T_{test}$ | 10 | |

Table 5.2: List of configuration parameters for the reaction-diffusion dataset.

| **Parameters** | **LF** | **HF** |
|---|---|---|
| $n$ | 32 | 100 |
| $d$ | 0.001 | 0.001 |
| $\mu_{train}$ | {1.5, 2.5, 3.5, 4.5} | {1.5, 2.5} |
| $T_{train}$ | 30 | 10 |
| $L$ | 5 | |
| $dt$ | 0.01 | |
| $\mu_{test}$ | {1.5, 2.5, 3.5} | |
| $T_{test}$ | 5 | |

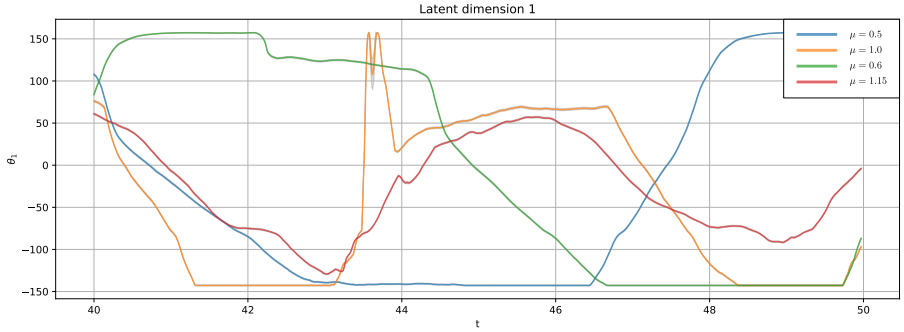
Table 5.3: List of configuration parameters for the diffusion-advection dataset.

### 5.2.4. ID and latent variables

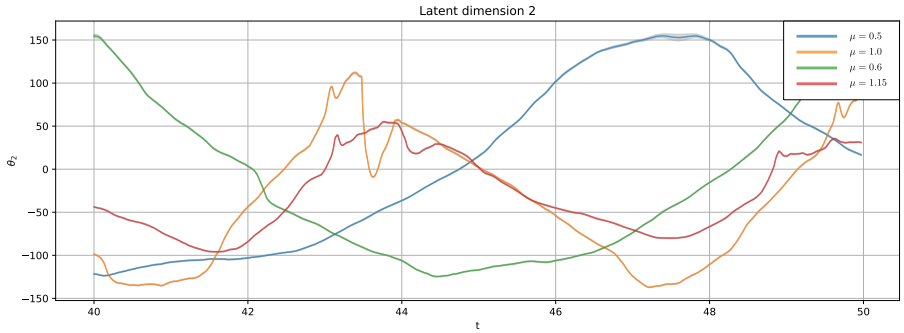
Our workflow exploits the Levina-Bickel algorithm (see Chapter 2) to compute an estimate of the intrinsic dimension of the dynamical system and produce an efficient and reliable low-dimensional surrogate model. In both the reaction-diffusion and diffusion-advection cases, the intrinsic dimension is estimated $ID = 2$, hence restricting the latent state

space for the HF model to a two-dimensional space, which is consistent with our prior theoretical knowledge.

Figures 5.9 and 5.10 show the latent variables $\theta_1^{HF}(t)$ and $\theta_2^{HF}(t)$ as functions of time, for each value of $\mu \in \mathcal{P}_{test}$, for the reaction-diffusion and diffusion-advection systems, respectively. While it is challenging to interpret their behavior, we can assess their significance from the accuracy of the reconstructions and forward predictions the model produces (reported in the following subsections).




(a) $\theta_1^{HF}(t)$



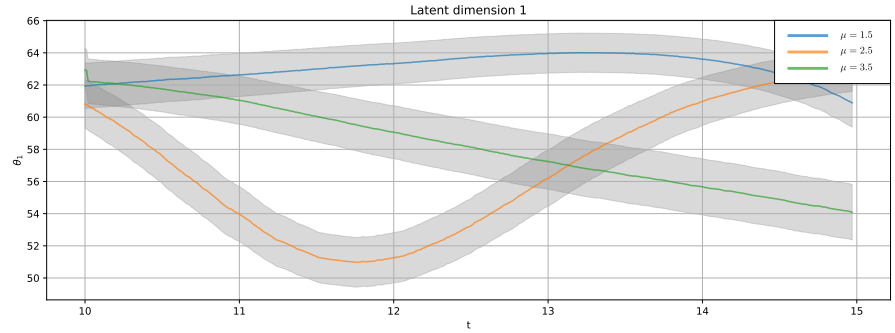

(b) $\theta_2^{HF}(t)$

Figure 5.9: Latent variables of the HF autoencoder during the test time window, for each value of the parameter $\mu$, for the reaction-diffusion case.

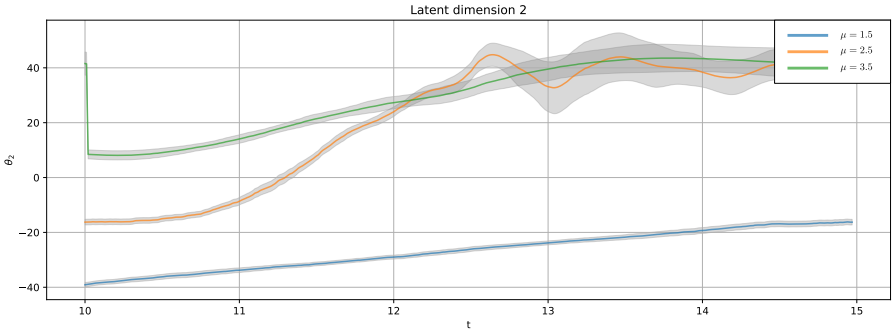
### 5.2.5. Reconstruction and one-step forward prediction

In this subsection we illustrate the effectiveness of the model, in reconstructing the frames and predicting forward in time, for testing parameter value $\mu$ seen during the training phase, but considering the system evolution over an unseen time period. In particular, for the reaction-diffusion problem, we test the model for $\mu = 0.5$, $t \in [40, 50]$. For the diffusion-advection problem, we consider $\mu = 1.5$, $t \in [10, 15]$.

Figures 5.11 and 5.13 show the reconstructed trajectories $\hat{\mathbf{x}}_t^{HF}$ and their one-step for-




(a) $\theta_1^{HF}(t)$




(b) $\theta_2^{HF}(t)$

Figure 5.10: Latent variables of the HF autoencoder during the test time window, for each value of the parameter $\mu$, for the diffusion-advection case.

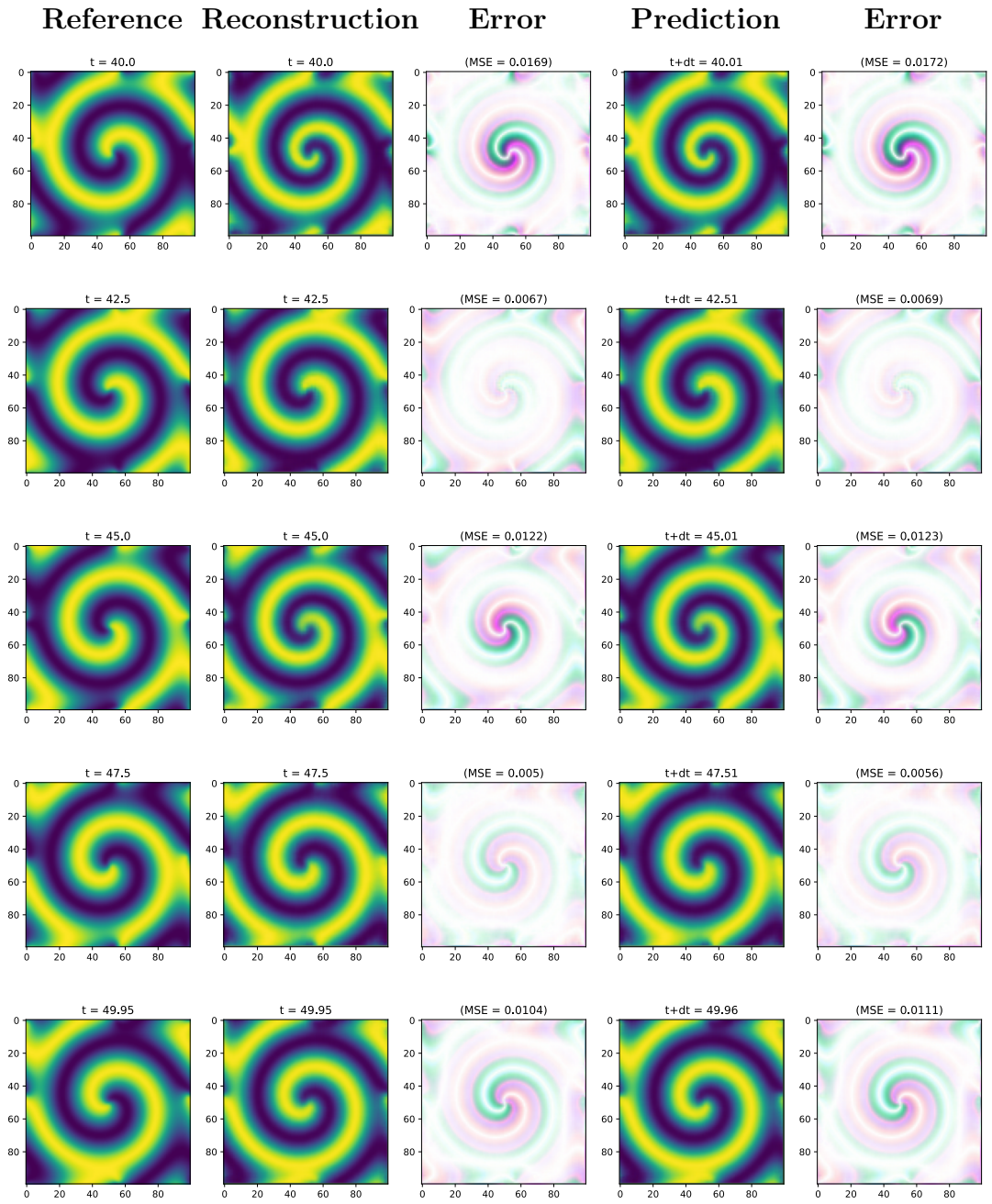



Figure 5.11: Reconstructions and predictions one-step forward in time, with respective absolute errors, for $\mu = 0.5$, reaction-diffusion test case. Both reconstructions and forward predictions are always consistent, with an absolute error that is sometimes more evident where the gradient is higher.

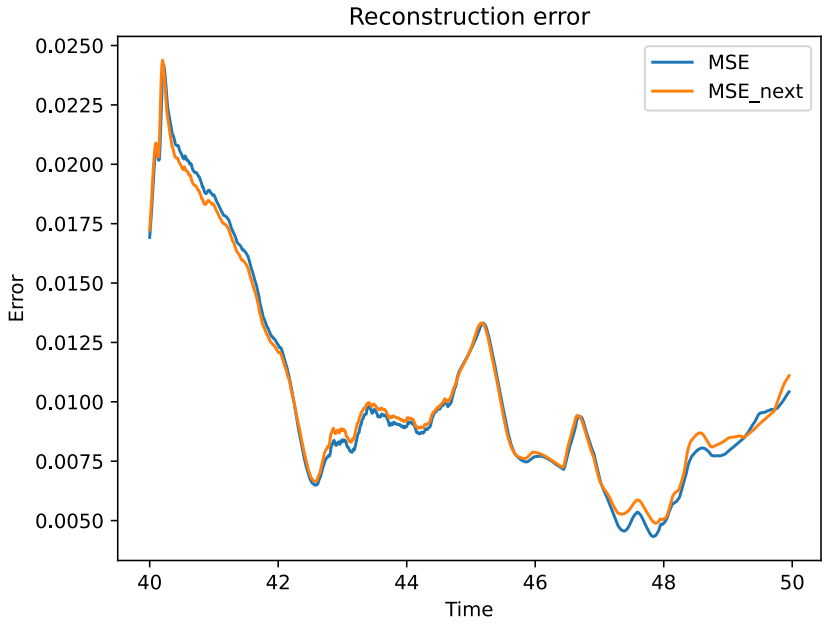



Figure 5.12: MSEs of the reconstruction and forward prediction for $\mu = 0.5$, reaction-diffusion test case, with respect to the relative true measurements, as function of time. The MSEs of both reconstructions and forward predictions assume similar values over the testing time window and decrease in time, overall, reaching a minimum of $MSE \approx 5e-3$ .

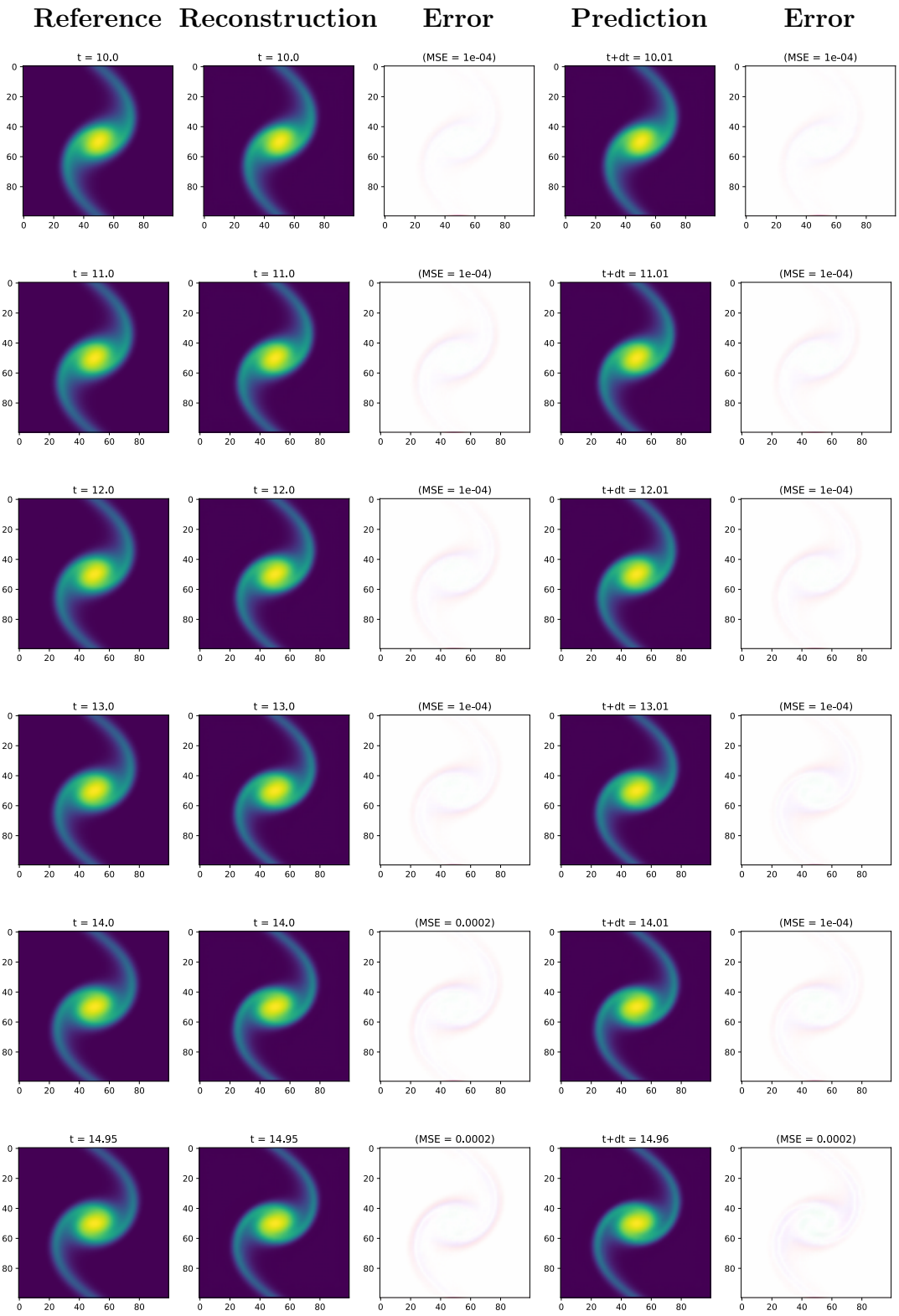



Figure 5.13: Reconstructions and predictions one-step forward in time, with respective absolute errors, for $\mu = 1.5$, for the diffusion-advection case. We notice that the MSE is always very small ($< 1e-3$).

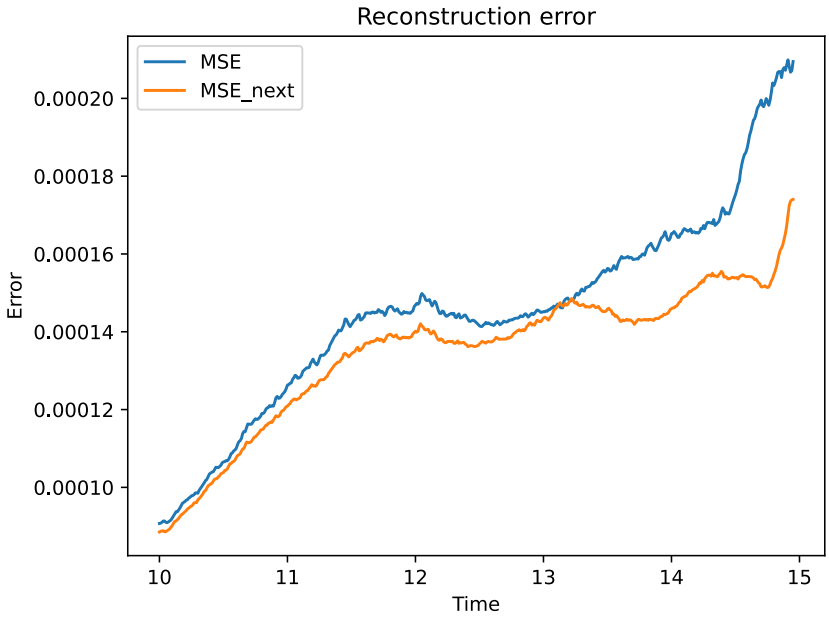



Figure 5.14: MSEs of the reconstruction and forward prediction for $\mu = 1.5$, for the diffusion-advection case, with respect to the relative true measurements, as function of time. The error grows in time, but stays $< 2e-4$.

ward prediction $\hat{\mathbf{x}}^{HF}_{t+dt}$, with their respective absolute errors, for the reaction-diffusion and diffusion-advection problems, respectively.

In both the cases, the reconstruction part of the model seems to be remarkably effective, producing consistent and very accurate trajectories. Indeed, the absolute errors display some minor departures from the actual measurements only where the gradients are larger, and the computational task more challenging. Additionally, the one-step forward prediction remains consistent across the tested time windows, exhibiting very accurate predicted frames for both the PDE systems.

The MSEs of both reconstruction and forward prediction, for the reaction-diffusion case, show a rapid decrease over the first section of the time windows, and later some fluctuations around $MSE \approx 1e-2$, as shown in Figure 5.12. On the contrary, with respect to the diffusion-advection case, the MSEs grow in time, but staying always below $MSE < 2e-4$, as shown in Figure 5.14.

Overall, the model consistently exhibits low errors for both the considered cases and for both the prediction tasks.

### 5.2.6. Extrapolation in time

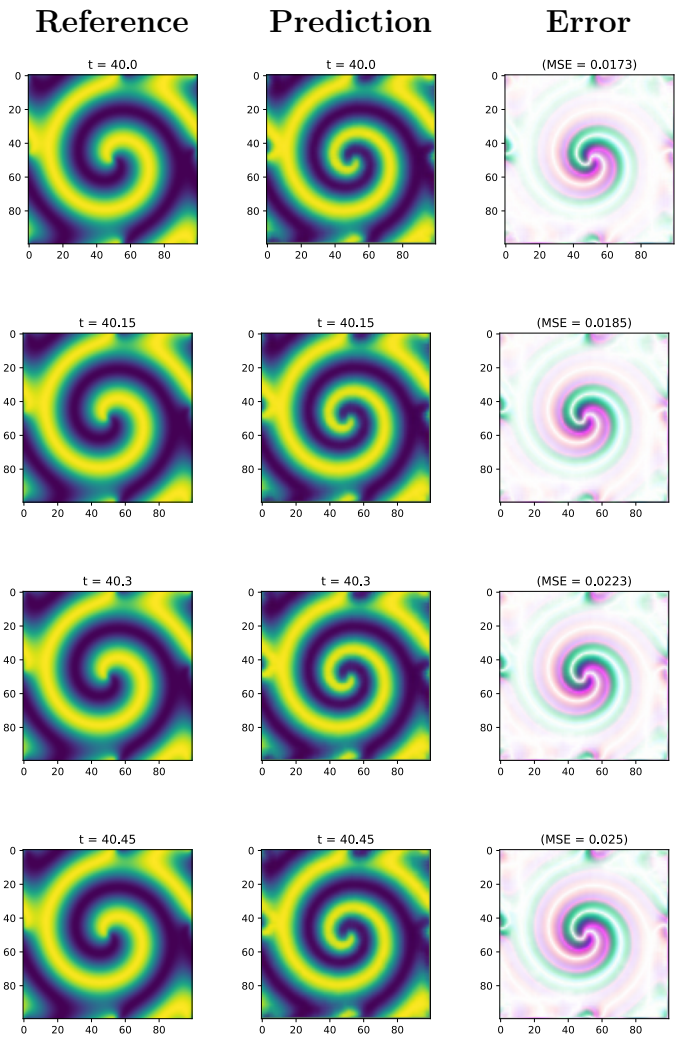



Figure 5.15: Extrapolation in time for $\mu = 0.5$, reaction-diffusion test case, for a few consecutive iterations, with the respective absolute error. The extrapolation in time is always consistent, with an absolute error observable only around the most internal spiral.

If we iterate the one-step forward prediction, by feeding the predicted latent representation to the next iteration, we can extrapolate in time. Figures 5.15 and 5.17 show some extrapolated frames, starting from the observation at time $t = 40$ and $t = 10$, for the reaction-diffusion and diffusion-advection cases, respectively.

Qualitatively speaking, the prediction remains consistent across the entire testing time window. The absolute errors are visually satisfying, highlighting only a slight departure from the actual frame around the most internal part of the spiral, for the reaction-diffusion case, and where the gradient is larger, for the diffusion-advection case.

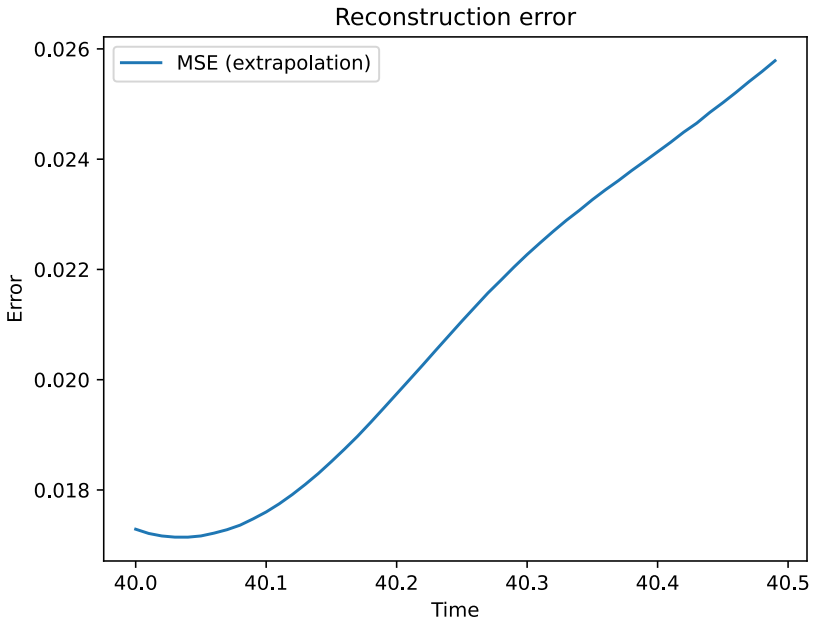



Figure 5.16: MSE of the extrapolation in time, for $\mu = 0.5$, reaction-diffusion test case. The error stays mostly constant for the first 10 iterations, and then it grows linearly.

As shown in Figures 5.16 and 5.18, the MSE increases linearly over time, in both the cases, with the exception of the first few iterations, but always staying within reasonable bounds. Overall, the model extrapolates in time remarkably well for both the considered PDE cases.

### 5.2.7. Extrapolation over the parameter space

We now consider values of the parameter $\mu$ unseen during the training phase of the HF sub-model, therefore extrapolating over the parameter space $\mathcal{P}$. In particular, with respect to the reaction-diffusion problem, we test the model for $\mu = 0.6$, for $t \in [40, 50]$; for the diffusion-advection system, we consider the parameter value $\mu = 3.5$, for $t \in [10, 15]$.

When extrapolating over the parameter space, the model still produces predictions consistent with the measured trajectories, as we can see in Figures 5.19 and 5.21. Both the image reconstruction and the one-step forward prediction seem particularly effective. Some accuracy is lost, especially in the diffusion-advection case, for which the absolute error is particularly visible, but always presenting acceptable values of the MSEs.

The behaviour of the MSE over time is reported in Figures 5.20 and 5.21, for the reaction-diffusion and diffusion-advection cases, respectively. The former displays some fluctua-

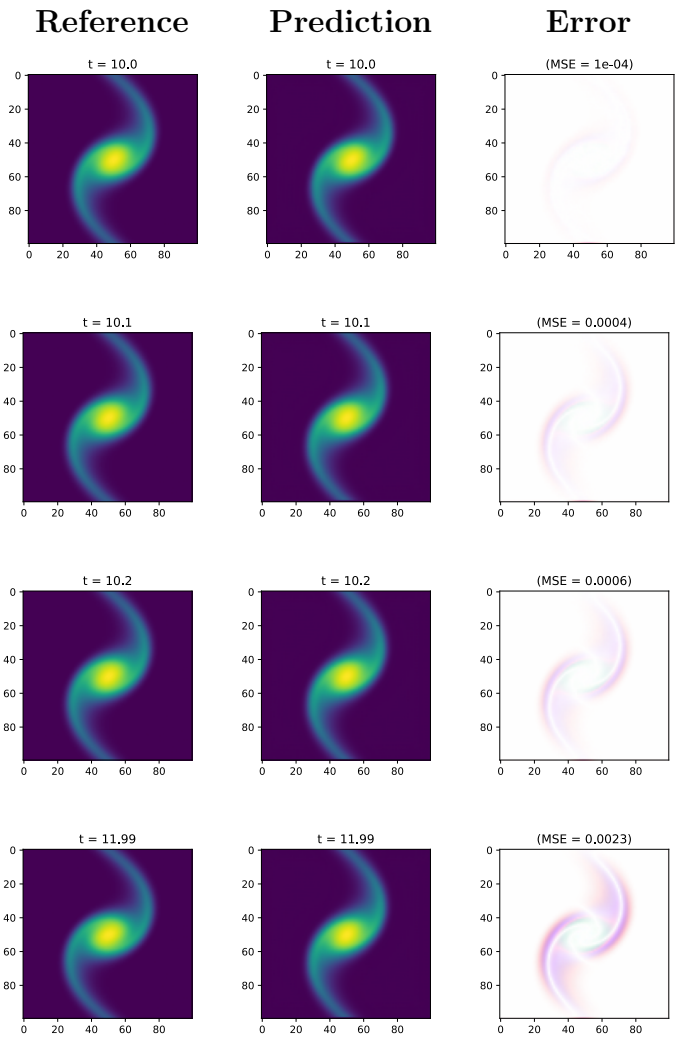



Figure 5.17: Extrapolation in time for $\mu = 1.5$, for the diffusion-advection case, for 200 iterations, with the respective absolute error. The predictions are consistent, while gradually less accurate forward in time.

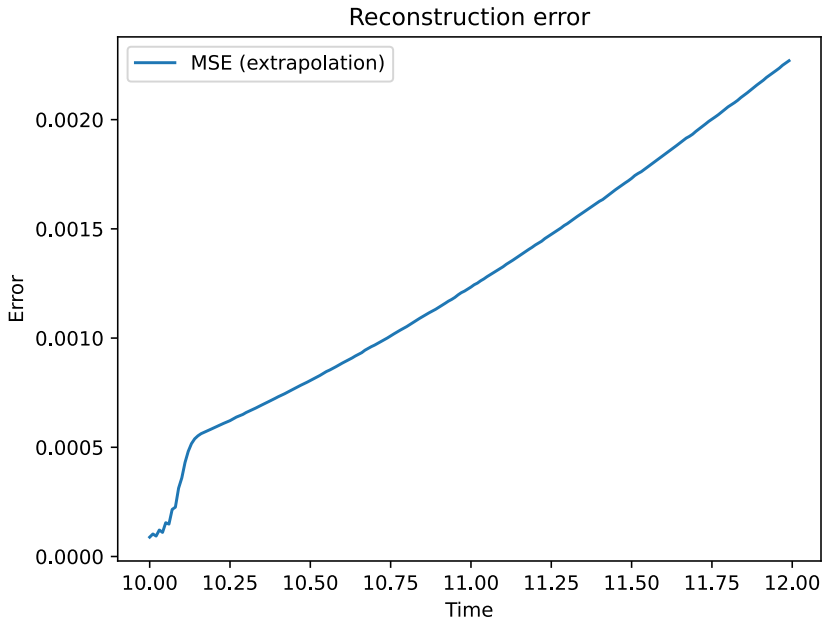



Figure 5.18: MSE of the extrapolation in time, for $\mu = 1.5$, for the diffusion-advection case. It grows exponentially at the beginning, and then linearly until the end, but always $< 5e-3$.

tions, for both the curves, with a spike in the middle of the time windows, slightly above $MSE \approx 0.02$. For the latter, the MSE linearly decreases over time, for both reconstruction and forward prediction, down to $MSE \approx 7e-3$.

If we extrapolate also over time, the forward predictions remain consistent and reasonably accurate in the reaction-diffusion case, as shows in Figure 5.23. Indeed, the plots of the absolute error show a modest discrepancy with the actual measurements that accentuate in time ($MSE \approx 0.015$), especially near the boundary. The MSE is reported in 5.24 and appear to be modest, overall, up to a maximum of $MSE \approx 0.02$).

In the diffusion-advection case, the extrapolation degrades after a number of iterations, as shows in Figure 5.25, with the model predicting a mean state of the system to minimize the error. This behavior suggests that the model has not learned the full dynamics of the system, but it is still able to produce a moderately meaningful prediction. Figure 5.26 shows the MSE, which decreases in time, possibly because the model minimises the error by predicting a constant mean state, closer to the state the trajectory is tending to.

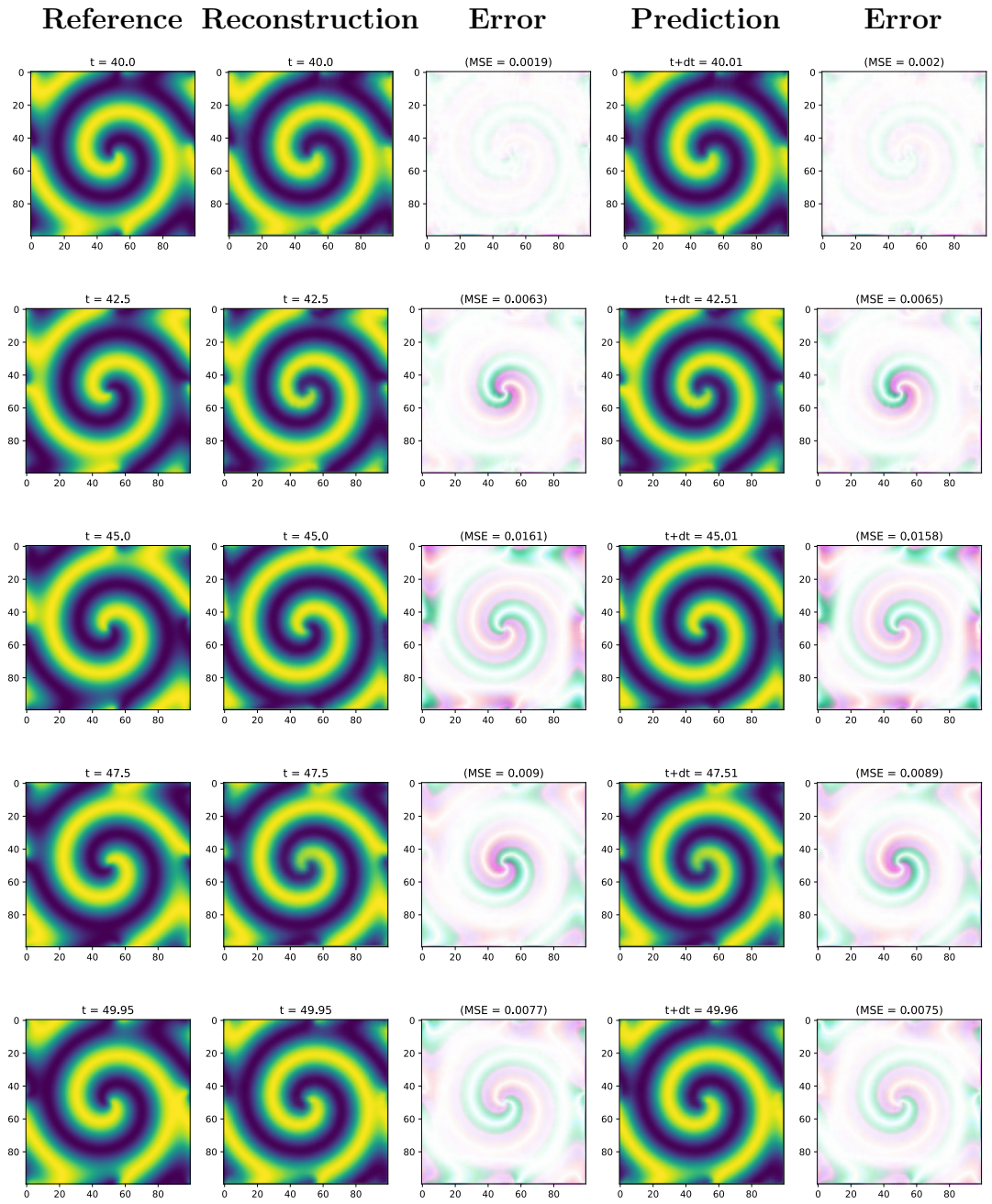



Figure 5.19: Reconstructions and predictions one-step forward in time, with respective absolute errors, for $\mu = 0.6$, reaction-diffusion test case. Both reconstructions and forward predictions are visually consistent. The plots of the absolute error show a small departure in accuracy near the central spiral, for both reconstructions and forward predictions.

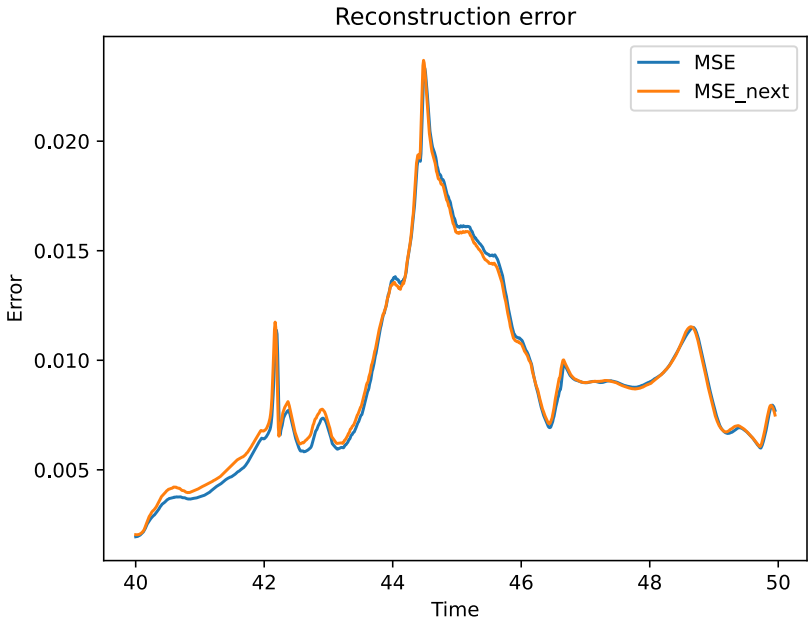



Figure 5.20: MSEs of the reconstruction and forward prediction for $\mu = 0.6$, reaction-diffusion test case, with respect to the relative true measurements, as function of time. The two errors show similar behaviors, growing during the first half of the time window, and decreasing over the second half.

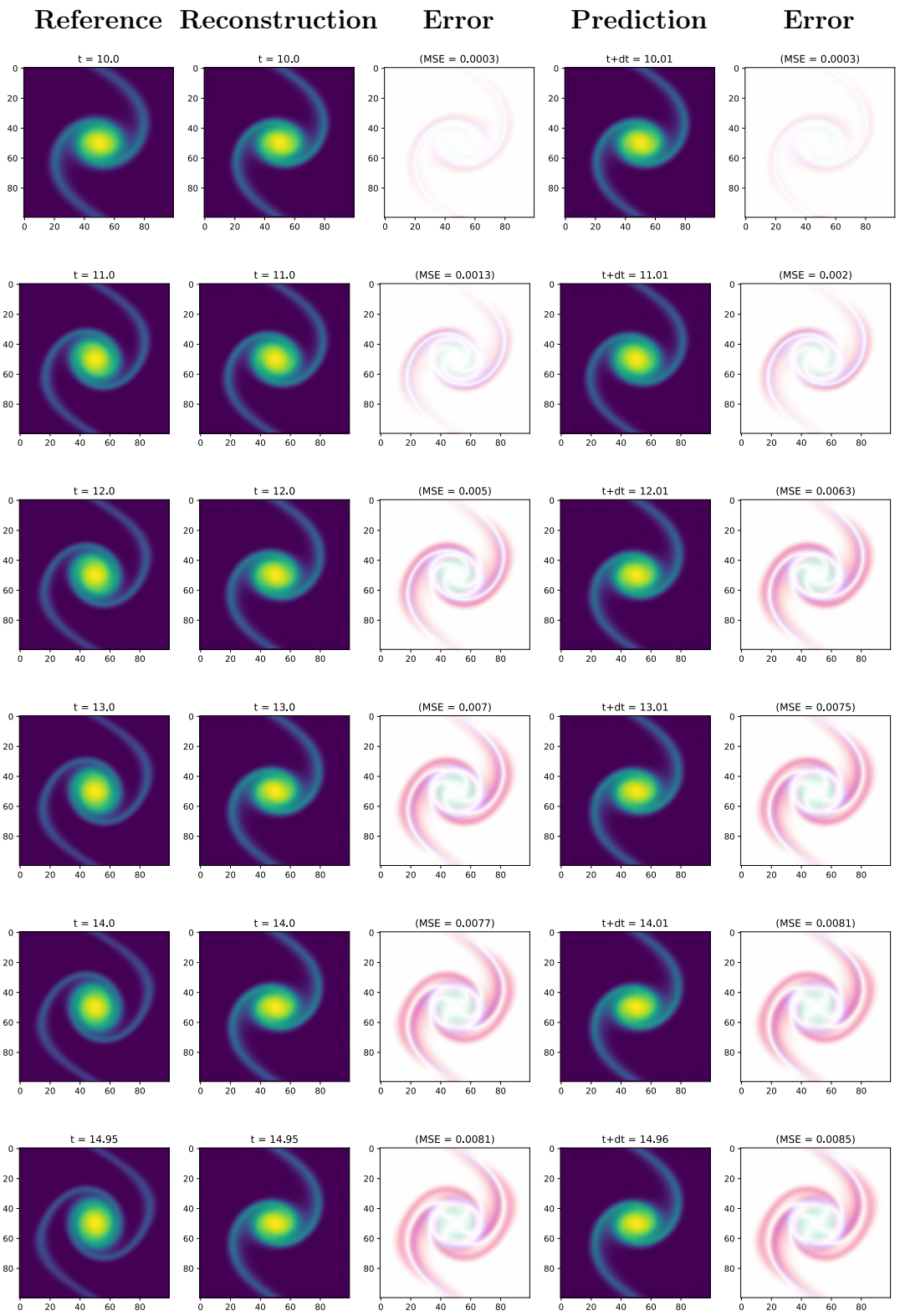



Figure 5.21: Reconstructions and predictions one-step forward in time, with respective absolute errors, for $\mu = 3.5$, for the diffusion-advection case. Both reconstructions and predictions seem visually consistent with the evolution of the system, but the error is significant.

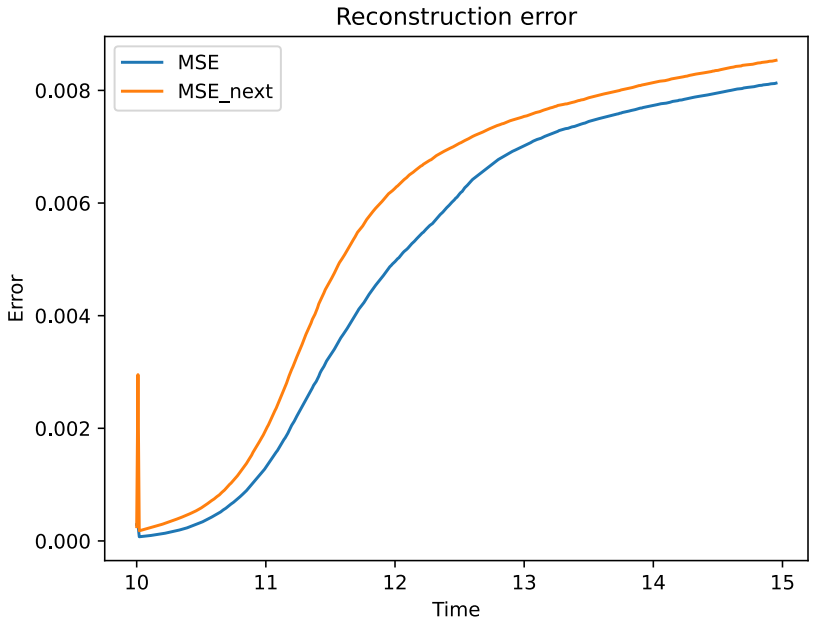



Figure 5.22: MSEs of the reconstruction and forward prediction for $\mu = 3.5$, for the diffusion-advection case, with respect to the relative true measurements, as function of time. Curiously, the MSEs decrease in time, possibly because the system has almost completed its evolution and it is easier to predict.

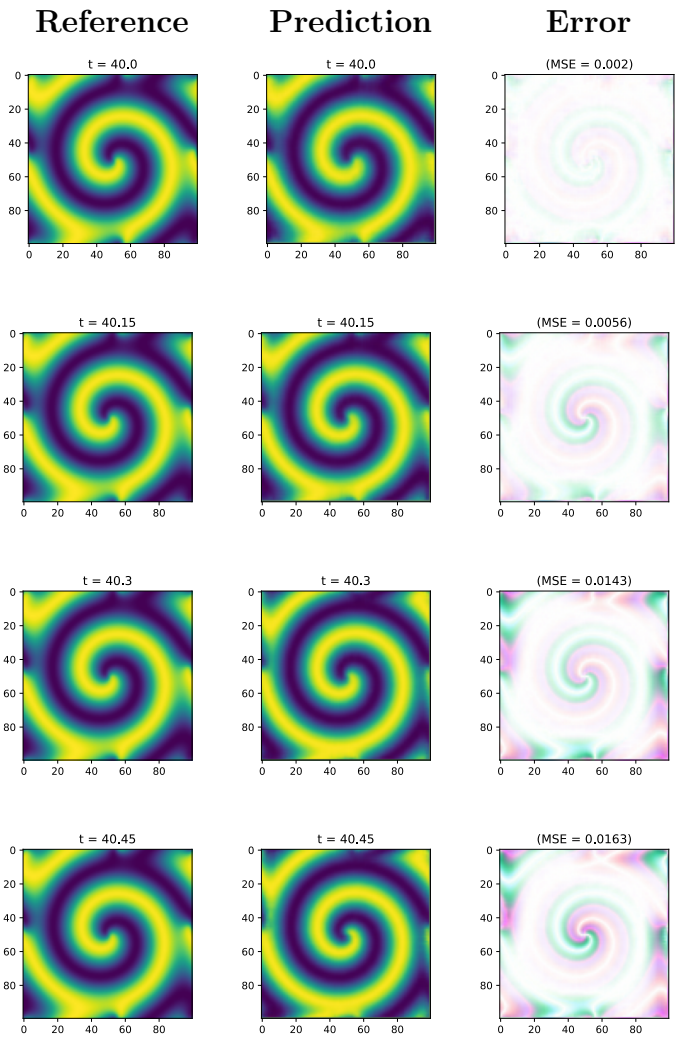



Figure 5.23: Extrapolation in time for $\mu = 0.6$, for a few iterations, with the respective absolute error. The extrapolated frames are always consistent with the true evolution and the absolute error is noticeable only near the boundary.

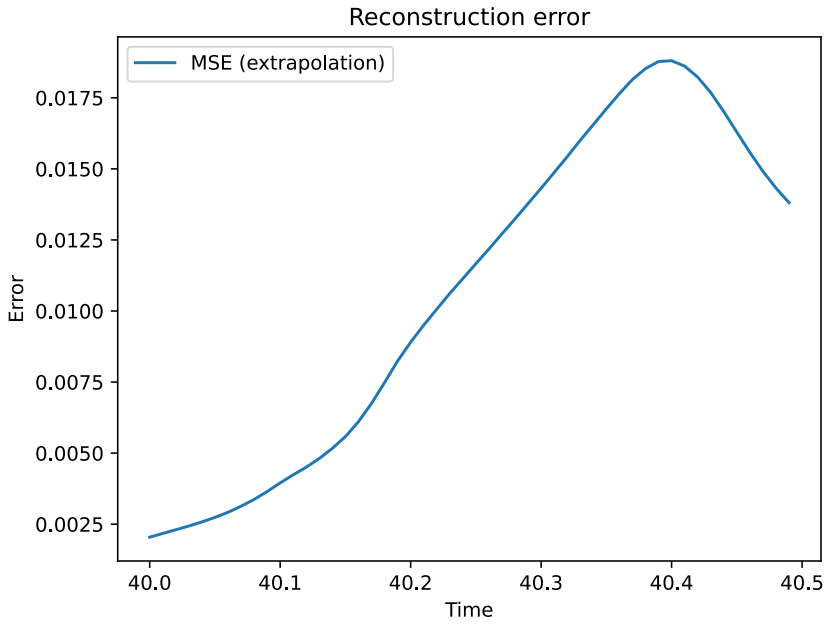



Figure 5.24: MSE of the extrapolation in time, for $\mu = 0.6$. Once again, the error grows rapidly at the beginning, and then it slightly decreases.

## 5.3. Some considerations

When considering data from PDEs, our model proves to be remarkably effective. The results presented in this chapter exhibit notable consistence and accuracy both when reconstructing the frame and predicting the dynamics one-step forward in time, highlighting the efficiency of both the comprehensive architecture and the Levina-Bickel estimator.

The efficiency of the model is also highlighted by the significantly low uncertainty exhibited when predicting the latent state.

At the cost of some foreseeable and minor defects where the gradient is larger, our framework still performs reasonably well when extrapolating over the parameter space. In general, the MSEs is bounded within acceptable values.

The more challenging task of extrapolation in time is also tackled with distinctive results, especially considering how efficient the dimensionality reduction proves to be.

Being a completely unsupervised data-driven approach, the model requires large datasets, that for this class of problems translates in a small step of time discretisation. The more evident consequence is that two successive frames often show only a slight evolution, possibly alleviating part of the challenge of the inference on the dynamics. Moreover, the

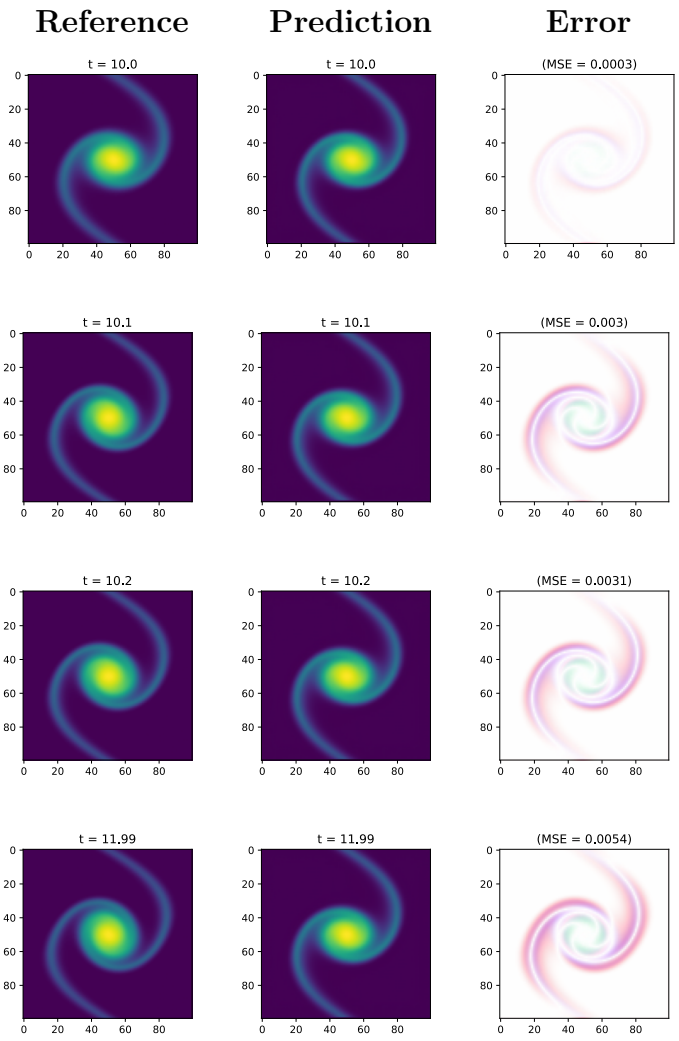



Figure 5.25: Extrapolation in time for $\mu = 3.5$, for the diffusion-advection case, for 200 iterations, with the respective absolute error. Visually, the extrapolated predictions degrade after a few iterations.

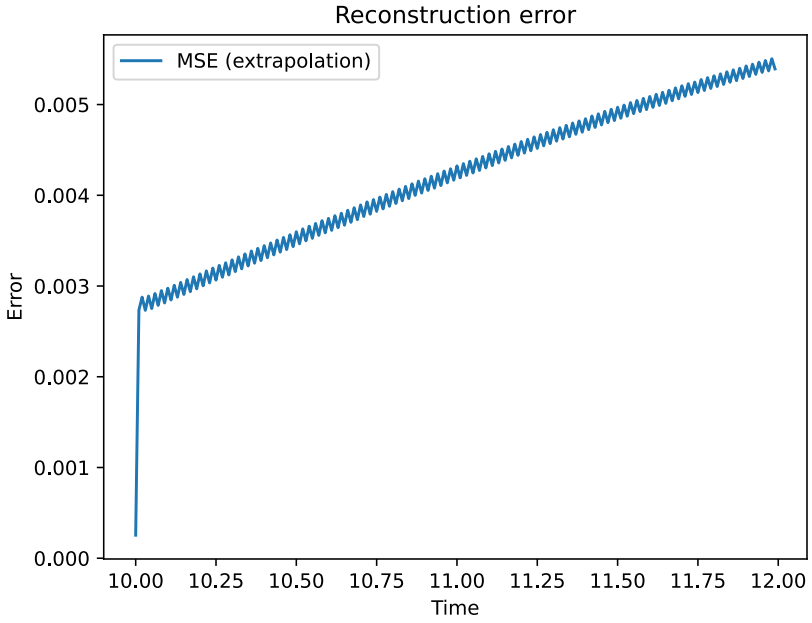



Figure 5.26: MSE of the extrapolation in time, for $\mu = 3.5$, for the diffusion-advection case. The MSE decreases over time, possibly suggesting a regression towards the mean to retain a moderate accuracy.

training stage is computationally taxing and it required a few days to complete with the HPC hardware at our disposal.

In conclusion, the proposed framework proves very effective at learning the unknown state and dynamics of PDE systems, exhibiting notable levels of accuracy for all the tested configurations.

# 6 | Conclusions and future developments

With this work, we proposed a comprehensive framework for unsupervised data-driven discovery of low dimensional dynamical systems. By conjugating the expressive capability and flexibility of Stochastic Variational Deep Kernel Learning with Multi-Fidelity schemes, the model is able to exploit large datasets of high dimensional measurements coming from a variety of data sources, exhibiting different degrees of fidelity to the true system.

The Python implementation introduced here is, indeed, capable of considering a wide range of model configurations and test cases, supporting the generation of datasets, the intuitive initialisation of the model considering an arbitrary complex hierarchical structure, and the immediate testing of its performances.

The model proved to be effective at most of its goals, including dimensionality reduction, uncertainty quantification and latent dynamics learning, showing good capabilities of generating interpretable latent representations of a large class of problems. In particular, the introduction of the Levina-Bickel algorithm in the workflow proposed by Botteghi et al. [4] enables the estimation of an efficient and compact latent state representation, close to the true state space.

Additionally, our framework allows accurate data-driven unsupervised learning from ordinary RGB videos, to the best of our knowledge for the first time in a Multi-Fidelity context. Indeed, the results discussed in the previous chapter show that the model is able to effectively replace a considerable amount of high-fidelity data with the less expensive counterpart, while maintaining notable accuracy and reducing the overall cost of measurements.

Future developments should consider a wider range of data sources, possibly leveraging both high and low-dimensional measurements, to further generalise the supported class of sensing devices and test cases. Moreover, the model may consider different time steps of discretisation among the fidelity levels, to exploit a larger group of LF data. Overall,

any generalisation of this framework should consider the goal of reducing the training cost and possibly improving the model capability of extrapolating in time.

# Bibliography

[1] M. A. Alvarez, L. Rosasco, and N. D. Lawrence. Kernels for Vector-Valued Functions: a Review, Apr. 2012. URL http://arxiv.org/abs/1106.6251. arXiv:1106.6251 [cs, math, stat].

[2] A. C. Antoulas, D. C. Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. In V. Olshevsky, editor, *A survey of model reduction methods for large-scale systems*, volume 280, pages 193–219, Providence, Rhode Island, 2001. American Mathematical Society. ISBN 9780821819210 9780821878705. doi: 10.1090/conm/280/04630. URL http://www.ams.org/conm/280/.

[3] J.-A. M. Assael, N. Wahlström, T. B. Schön, and M. P. Deisenroth. Data-Efficient Learning of Feedback Policies from Image Pixels using Deep Dynamical Models, Oct. 2015. URL http://arxiv.org/abs/1510.02173. arXiv:1510.02173 [cs, stat].

[4] N. Botteghi, M. Guo, and C. Brune. Deep Kernel Learning of Dynamical Models from High-Dimensional Noisy Data. *Scientific Reports*, 12(1):21530, Dec. 2022. ISSN 2045-2322. doi: 10.1038/s41598-022-25362-4. URL http://arxiv.org/abs/2208.12975. arXiv:2208.12975 [cs, stat].

[5] S. L. Brunton and J. N. Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, Cambridge, 2019. doi: 10.1017/9781108380690. URL https://www.cambridge.org/core/books/datadriven-science-and-engineering/77D52B171B60A496EAFE4DB662ADC36E.

[6] S. L. Brunton, J. L. Proctor, and J. N. Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, Apr. 2016. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.1517384113. URL https://pnas.org/doi/full/10.1073/pnas.1517384113.

[7] T. Bui-Thanh, K. Willcox, and O. Ghattas. Parametric Reduced-Order Models for Probabilistic Analysis of Unsteady Aerodynamic Applications. *AIAA Journal*, 46

(10):2520–2529, Oct. 2008. ISSN 0001-1452, 1533-385X. doi: 10.2514/1.35850. URL https://arc.aiaa.org/doi/10.2514/1.35850.

[8] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116(45):22445–22451, Nov. 2019. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.1906995116. URL https://pnas.org/doi/full/10.1073/pnas.1906995116.

[9] B. Chen, K. Huang, S. Raghupathi, I. Chandratreya, Q. Du, and H. Lipson. Discovering State Variables Hidden in Experimental Data, Dec. 2021. URL http://arxiv.org/abs/2112.10755. arXiv:2112.10755 [physics].

[10] P. Conti, M. Guo, A. Manzoni, A. Frangi, S. L. Brunton, and J. N. Kutz. Multi-fidelity reduced-order surrogate modeling, Sept. 2023. URL http://arxiv.org/abs/2309.00325. arXiv:2309.00325 [cs, math].

[11] A. Doerr, C. Daniel, M. Schiegg, N.-T. Duy, S. Schaal, M. Toussaint, and T. Sebastian. Probabilistic Recurrent State-Space Models. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1280–1289. PMLR, July 2018. URL https://proceedings.mlr.press/v80/doerr18a.html.

[12] M. Frangos, Y. Marzouk, K. Willcox, and B. van Bloemen Waanders. Surrogate and reduced-order modeling: a comparison of approaches for large-scale statistical inverse problems [Chapter 7]. *Prof. Willcox via Barbara Williams*, Jan. 2010. URL https://dspace.mit.edu/handle/1721.1/105500.

[13] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] GPyTorch Developers. *GPyTorch Documentation*, 2024. URL https://docs.gpytorch.ai/en/stable/. Accessed: 2024-06-19.

[15] M. Guo and J. S. Hesthaven. Data-driven reduced order modeling for time-dependent problems. *Computer Methods in Applied Mechanics and Engineering*, 345:75–99, Mar. 2019. ISSN 0045-7825. doi: 10.1016/j.cma.2018.10.029. URL https://www.sciencedirect.com/science/article/pii/S0045782518305334.

[16] M. Guo, A. Manzoni, M. Amendt, P. Conti, and J. S. Hesthaven. Multi-fidelity regression using artificial neural networks: efficient approximation of parameter-dependent output quantities. *Computer Methods in Applied Mechanics and Engineering*, 389:114378, Feb. 2022. ISSN 00457825. doi: 10.1016/j.cma.2021.114378. URL http://arxiv.org/abs/2102.13403. arXiv:2102.13403 [cs, math].

[17] Gymnasium Developers. *Gymnasium Documentation*, 2024. URL `https://gymnasium.farama.org`. Accessed: 2024-06-24.

[18] Imageio Developers. *Imageio Documentation*, 2024. URL `https://imageio.readthedocs.io/en/stable/`. Accessed: 2024-06-22.

[19] A. G. Journel. *Fundamentals of Geostatistics in Five Lessons*, volume 8 of *Short Courses in Geology*. American Geophysical Union (AGU), 1989.

[20] M. Kast, M. Guo, and J. S. Hesthaven. A non-intrusive multifidelity method for the reduced order modeling of nonlinear problems. *Computational Methods in Applied Mechanics and Engineering*, 364, 2020.

[21] M. C. Kennedy and A. O'Hagan. Predicting the Output from a Complex Computer Code When Fast Approximations Are Available. *Biometrika*, 87(1):1–13, 2000. ISSN 0006-3444. URL `https://www.jstor.org/stable/2673557`.

[22] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes, Dec. 2022. URL `http://arxiv.org/abs/1312.6114`. arXiv:1312.6114 [cs, stat].

[23] I. Kononenko. Bayesian neural networks. *Biological Cybernetics*, 61(5):361–370, Sept. 1989. ISSN 0340-1200, 1432-0770. doi: 10.1007/BF00200801. URL `http://link.springer.com/10.1007/BF00200801`.

[24] F. Lancellotti. Multi-fidelity deep kernel learning, 2024. URL `https://github.com/federico-lancellotti/multi-fidelity-deep-kernel-learning`. Accessed: 2024-06-17.

[25] K. Lee and K. Carlberg. Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders, June 2019. URL `http://arxiv.org/abs/1812.08373`. arXiv:1812.08373 [cs].

[26] E. Levina and P. Bickel. Maximum Likelihood Estimation of Intrinsic Dimension. In *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004. URL `https://papers.nips.cc/paper_files/paper/2004/hash/74934548253bcab8490ebd74afed7031-Abstract.html`.

[27] A. Manzoni. *Computational Statistics class notes*. 2023.

[28] Matplotlib Developers. *Matplotlib Documentation*, 2024. URL `https://matplotlib.org`. Accessed: 2024-06-22.

[29] F. Negri, G. Rozza, A. Manzoni, and A. Quarteroni. Reduced Basis Method for Parametrized Elliptic Optimal Control Problems. *SIAM Journal on Scien-*

*tific Computing*, 35(5):A2316–A2340, Jan. 2013. ISSN 1064-8275, 1095-7197. doi: 10.1137/120894737. URL http://epubs.siam.org/doi/10.1137/120894737.

[30] NumPy Developers. *NumPy Documentation*, 2024. URL https://numpy.org/doc/stable/index.html. Accessed: 2024-06-20.

[31] S. W. Ober, C. E. Rasmussen, and M. van der Wilk. The Promises and Pitfalls of Deep Kernel Learning, July 2021. URL http://arxiv.org/abs/2102.12108. arXiv:2102.12108 [cs, stat].

[32] B. Peherstorfer, K. Willcox, and M. Gunzburger. Survey of multifidelity methods in uncertainty propagation, inference, and optimization. *SIAM Review*, 60(3):550–591, 2018.

[33] J. L. Proctor, S. L. Brunton, and J. N. Kutz. Dynamic Mode Decomposition with Control. *SIAM Journal on Applied Dynamical Systems*, 15(1):142–161, Jan. 2016. ISSN 1536-0040. doi: 10.1137/15M1013857. URL http://epubs.siam.org/doi/10.1137/15M1013857.

[34] Python Software Foundation. *ABC - Abstract Base Classes*, 2024. URL https://docs.python.org/3/library/abc.html. Accessed: 2024-06-19.

[35] Python Software Foundation. *pickle — Python object serialization*, 2024. URL https://docs.python.org/3/library/pickle.html. Accessed: 2024-06-20.

[36] PyTorch Developers. *PyTorch Documentation*, 2024. URL https://pytorch.org/docs/stable/index.html. Accessed: 2024-06-19.

[37] A. Quarteroni and G. Rozza, editors. *Reduced Order Methods for Modeling and Computational Reduction*. Springer International Publishing, Cham, 2014. ISBN 9783319020891 9783319020907. doi: 10.1007/978-3-319-02090-7. URL http://link.springer.com/10.1007/978-3-319-02090-7.

[38] A. Quarteroni, A. Manzoni, and F. Negri. *Reduced Basis Methods for Partial Differential Equations*, volume 92 of *UNITEXT*. Springer International Publishing, Cham, 2016. ISBN 9783319154305 9783319154312. doi: 10.1007/978-3-319-15431-2. URL http://link.springer.com/10.1007/978-3-319-15431-2.

[39] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Inferring solutions of differential equations using noisy multi-fidelity data. *Journal of Computational Physics*, 335: 736–746, 2017.

[40] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Ma-*

*chine Learning*. The MIT Press, 2005. ISBN 9780262256834. doi: 10.7551/mitpress/3206.001.0001. URL https://direct.mit.edu/books/book/2320/gaussian-processes-for-machine-learning.

[41] P. J. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, Aug. 2010. ISSN 1469-7645, 0022-1120. doi: 10.1017/S0022112010001217. URL https://www.cambridge.org/core/journals/journal-of-fluid-mechanics/article/abs/dynamic-mode-decomposition-of-numerical-and-experimental-data/AA4C763B525515AD4521A6CC5E10DBD4.

[42] SciPy Developers. *SciPy Documentation*, 2024. URL https://docs.scipy.org/doc/scipy/. Accessed: 2024-06-19.

[43] C. Sinigaglia, D. E. Quadrelli, A. Manzoni, and F. Braghin. Fast active thermal cloaking through PDE-constrained optimization and reduced-order modeling. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 478(2258): 20210813, Feb. 2022. ISSN 1364-5021, 1471-2946. doi: 10.1098/rspa.2021.0813. URL http://arxiv.org/abs/2110.10845. arXiv:2110.10845 [math].

[44] N. Wahlström, T. B. Schön, and M. P. Deisenroth. From Pixels to Torques: Policy Learning with Deep Dynamical Models, June 2015. URL http://arxiv.org/abs/1502.02251. arXiv:1502.02251 [cs, stat].

[45] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing. Deep Kernel Learning. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 370–378. PMLR, May 2016. URL https://proceedings.mlr.press/v51/wilson16.html.

[46] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing. Stochastic Variational Deep Kernel Learning, Nov. 2016. URL http://arxiv.org/abs/1611.00336. arXiv:1611.00336 [cs, stat].

# List of Figures

# List of Tables