

Multi-level neural networks for PDEs with uncertain parameters

Y. van Halder^a, B. Sanderse^a, B. Koren^b

^aCentrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG, Amsterdam, the Netherlands

^bEindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, the Netherlands

Abstract

A novel multi-level method for partial differential equations with uncertain parameters is proposed. The principle behind the method is that the error between grid levels in multi-level methods has a spatial structure that is by good approximation independent of the actual grid level. Our method learns this structure by employing a sequence of convolutional neural networks, that are well-suited to automatically detect local error features as latent quantities of the solution. Furthermore, by using the concept of transfer learning, the information of coarse grid levels is reused on fine grid levels in order to minimize the required number of samples on fine levels. The method outperforms state-of-the-art multi-level methods, especially in the case when complex PDEs (such as single-phase and free-surface flow problems) are concerned, or when high accuracy is required.

Keywords: neural networks, multi-fidelity, surrogate modelling, parametric PDEs

1. Introduction

Uncertainty Quantification (UQ) has become increasingly important for complex engineering applications. Determining and quantifying the influence of parametric and model-form uncertainties is essential for a wide range of applications: from turbulent flow phenomena [1, 2], aerodynamics [3], biology [4, 5] to design optimisation [6, 7, 8].

When performing non-intrusive UQ, a black-box is used to sample solutions of a deterministic model, often a PDE, in parameter space, and to interpolate these samples to construct a surrogate model [9]. The number of samples increases exponentially with the number of dimensions of the stochastic space, i.e., the number of uncertainties in the model. This *curse of dimensionality* limits the applicability of UQ algorithms, especially when the black-box is computationally expensive to sample from. Using a low-fidelity black-box alleviates the computational burden by lowering the computational time per sample, but significantly drops the accuracy of the deterministic samples. Using a high-fidelity black-box surely increases accuracy, but at the cost of a significant increase in computational time. As a remedy, multi-fidelity approaches were introduced, which use high-fidelity samples to construct the surrogate model and low-fidelity samples to explore the parameter space, and to see where to place a new high-fidelity sample [10, 11, 12, 13, 14, 15, 16, 17]. These approaches significantly reduce the number of high-fidelity samples required to construct an accurate surrogate model. Multi-level stochastic collocation (MLSC) [18, 19, 20, 21, 22, 23] extends this approach by using a hierarchy of grid resolutions, often referred to as levels, and places samples on each grid level to approximate the difference between two consecutive grid resolutions. The underlying concept of variance decay (of the difference in the solution between subsequent levels) ensures that the number of samples required to approximate the difference between two consecutive levels, reduces with increasing level [24]. This results in a significant reduction of required number of high-fidelity samples.

MLSC significantly reduces the required number of high-fidelity samples, due to the utilisation of variance decay of subsequent solution differences. Our method utilises the variance decay and is based on the fact that *subsequent errors themselves typically exhibit similar behaviour (as a function of the spatial variables) with increasing grid*

*Corresponding author

Email address: `y.van.halder@cwi.nl` (Y. van Halder)

resolution. This is a key insight that we will directly exploit in this paper as it allows us to construct a novel class of multi-level methods based on the concepts of convolutional neural networks and transfer learning.

Our proposed method is as follows. Like in the MLSC approach [18], we use the telescopic-sum identity to express the solution in terms of the solution differences between consecutive grid levels ('relative errors', in the following mostly shortly 'errors'). Our method differentiates itself in that we use the similarity of this error between consecutive grid levels to reduce the required number of samples on each level. We achieve this through an original combination of several ingredients. First, we express the relative error between grid levels as a function of the solution at the coarser level through the use of a neural network. Convolutional neural networks enable this step. Convolutional neural networks are well-suited to learn local or low-level features ('latent quantities') in the solution structure, that turn out to be similar to the truncation error commonly encountered in numerical discretisation methods for PDEs. In connection with a second, fully connected, neural network, the convolutional neural network is able to learn the error between grid levels as a function of the solution. Finally, we use transfer learning to reuse the weights and biases from previously trained neural networks at coarse levels to quickly train the neural networks at finer grid levels, thereby significantly reducing the required number of samples of high-fidelity solutions. In the remaining of this paper we refer to the proposed approach as the Multi-Level Neural Network (MLNN) method.

In summary, the highlights of the novel MLNN method for efficient surrogate construction of parametric PDEs are:

- *The error between grid levels is trained in terms of the solution through a neural network;*
- *Convolutional neural networks are used to learn local error features as latent quantities;*
- *Transfer learning is used to exploit the similarities between errors at different levels.*

This paper is outlined as follows: section 2 further introduces the problem and notation, section 3 introduces the machine learning based multi-level approach, section 4 discusses in detail how to train the neural networks, section 5 summarises the methodology, and finally, section 6 demonstrates efficiency and accuracy of the MLNN method when applied to several test-cases, ranging from the steady linear advection-diffusion equation to surrogate construction for the unsteady incompressible Navier-Stokes equations.

2. Problem Description

Quantifying the effects of uncertainties in computational engineering typically consists of three steps: (i) the input uncertainties are characterised in terms of a Probability Density Function (PDF), which follows from observations or physical evidence; (ii) the uncertainties are propagated through the model; (iii) the outputs are post-processed, where the Quantity of Interest (QoI) is expressed in terms of its statistical properties. Our goal is to solve the following stochastic problem:

$$\mathcal{L}(u; \mathbf{z}, \mathbf{x}) = 0, \quad \forall \mathbf{x} \in D, \quad \forall \mathbf{z} \in I, \quad (1)$$

where $I \subset \mathbb{R}^k$ is the space of uncertain inputs and is referred to as *random space*, $\mathbf{z} = (z_1, \dots, z_k) \in I$ the k -dimensional vector containing uncertain inputs which follows a joint PDF $\rho(\mathbf{z})$, $\mathbf{x} \in D$ the vector of spatial/temporal coordinates, $u_{\text{exc}} := u(\mathbf{z}, \mathbf{x})$ the exact solution, and \mathcal{L} a continuous differential operator. In the remainder of this work we mostly consider $\mathbf{x} = (x, y) \in D \subset \mathbb{R}^2$, corresponding to two spatial coordinates. The QoI is given by $q(\mathbf{z}, F(u(\mathbf{z}, \mathbf{x})))$, where F is a given operator.

We are interested in constructing a surrogate model for q as a function of \mathbf{z} . This is done non-intrusively by sampling solutions $u_i(\mathbf{x})$ of (1) at different locations $\{\mathbf{z}_i\}_{i=1}^{N_z}$ in the random space and extracting QoI-values from these solutions, i.e.:

$$\mathcal{L}(u_i; \mathbf{z}_i, \mathbf{x}) = 0, \quad \forall \mathbf{x} \in D, \quad i = 1, \dots, N_z, \quad (2a)$$

$$q_i = q(\mathbf{z}_i) = F(u_i). \quad (2b)$$

From the set of solutions $\{q_i\}_{i=1}^{N_z}$, a surrogate $\tilde{q}(\mathbf{z})$ may be constructed, which satisfies

$$\tilde{q}(\mathbf{z}) \approx q(\mathbf{z}), \quad \forall \mathbf{z} \in I, \quad (3)$$

and statistical moments, i.e., mean and standard deviation, may be extracted from \tilde{q} . However, solving the differential-equation problems (2a) exactly is often not possible, and therefore the QoI at a given \mathbf{z}_i is obtained by solving these problems discretised in space and/or time:

$$L(\mathbf{u}(\mathbf{z}_i); \mathbf{z}_i, X) = 0, \quad (4a)$$

$$q_d(\mathbf{z}_i) = F_d(\mathbf{u}(\mathbf{z}_i)), \quad (4b)$$

where the subscript d indicates a quantity which is related to the discretised problem, L the discretised differential operator, $X \in \mathbb{R}^{N \times 2}$ the spatial/temporal computational grid, say (x_i, y_i) , consisting of N grid points, $\mathbf{u}(\mathbf{z}_i) \in \mathbb{R}^N$ the solution vector on the computational grid for $\mathbf{z} = \mathbf{z}_i$, and F_d the operator which maps the discretised solution $\mathbf{u}_d(\mathbf{z}_i)$ to the QoI $q_d(\mathbf{z}_i)$. The computational cost for solving the discretised equation is determined by the discretisation scheme and the resolution of the computational grid X , i.e., the number of grid points N . If (4a) is solved with high precision by using for instance a fine grid or a high-order discretisation scheme, then the solution is referred to as a high-fidelity solution. Contrary, when solving (4a) on a coarse grid or by using a low-order discretisation scheme, the solution is referred to as a low-fidelity solution. Other definitions in terms of reduced order models can be given for high/low-fidelity, but they require a different procedure from the one proposed here, and are therefore not considered in this paper. Therefore, the fidelity of the solution is assumed to be fully determined by the grid resolution, and we often refer to different fidelities as levels.

When performing forward propagation of input uncertainties, we have to solve the discretised equation a number of times for different values of \mathbf{z} . As a result, accurate forward propagation of uncertainties is often infeasible when high-fidelity solutions are computationally expensive to produce. Low-fidelity models may alleviate the computational expenses and make UQ feasible for problems with a complex underlying model, but the accuracy of the solutions decreases and the resulting surrogate model may be inaccurate. Our goal is: ***to construct a computationally feasible surrogate with high-fidelity accuracy by using solution values of different fidelities***. The MLNN method uses the main concept of a multi-level method; we use discrete solutions on a hierarchy of grids with increasing resolution. A surrogate is constructed, which is mainly based on a set of low-fidelity solutions, and uses a relatively low number of higher-fidelity solutions in order to enhance the accuracy of the surrogate. The MLNN method is discussed in more detail in the next two sections.

3. Machine Learning Based Multi-Level Approach

The multi-level approach is explained in this section. To begin with, we introduce notation for the different levels of solutions. The discretised operators corresponding to different levels are labelled with a superscript and are denoted as $L^{(i)}$, and the corresponding computational grids $X^{(i)}$ consisting of $N^{(i)}$ grid points. The solutions are denoted as $\mathbf{u}^{(i)}(\mathbf{z}) \in \mathbb{R}^{N^{(i)}}$ and are functions of the input uncertainties. In the MLNN method we use a hierarchy of grid resolutions consisting of a total of N_L grids. We denote with $\mathbf{u}^{(i)}(\mathbf{z})$ the solution which is computed on the coarsest grid level $X^{(1)}$, while $\mathbf{u}^{(N_L)}(\mathbf{z})$ is the solution computed on the finest grid level $X^{(N_L)}$. For simplicity the computational grids are assumed to be nested:

$$X^{(1)} \subset X^{(2)} \subset \dots \subset X^{(N_L)}. \quad (5)$$

Several multi-level strategies have been presented that use a hierarchy in grid resolutions [10], which reduces the number of high-fidelity solves required to construct an accurate surrogate model or extract accurate statistics. Consider restriction of the fine-grid solution to the coarsest grid:

$$\mathbf{u}^{(N_L)}(\mathbf{z}) \rightarrow \mathbf{u}^{(N_L)}(\mathbf{z})|_{X^{(1)}}, \quad (6)$$

where $\cdot|_{X^{(1)}}$ is the restriction operator which computes values on the coarse grid $X^{(1)}$, which may incorporate sub sampling and interpolation. The multi-level methods described in [18, 20, 21] rewrite $\mathbf{u}^{(N_L)}(\mathbf{z})$, using the telescoping-sum identity:

$$\mathbf{u}^{(N_L)}(\mathbf{z})|_{X^{(1)}} = \mathbf{u}^{(1)}(\mathbf{z}) + \sum_{i=2}^{N_L} (\mathbf{u}^{(i)}(\mathbf{z})|_{X^{(1)}} - \mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}), \quad (7)$$

In case of nested grids, (5), the restriction operator solely uses sub sampling. Restricting to the coarsest level is not necessary, as we can also prolongate the coarse solutions to the finest level computational grid $X^{(N_L)}$, but in many cases the coarsest grid suffices to accurately calculate the QoI. To circumvent interpolation of coarse-grid solution values to the fine grid and reduce the degrees of freedom in the neural networks that are used later, we opt to restrict solution values to the coarsest level, but the MLNN method can be extended to account for prolongation to the finest level as well. For convenience, we denote the error between two levels as

$$\mathbf{e}^{(i)}(\mathbf{z}) := \mathbf{u}^{(i)}(\mathbf{z})|_{X^{(i)}} - \mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(i)}} . \quad (8)$$

The goal of a multi-level method is to approximate the error between two levels $\mathbf{e}^{(i)}$, where we assume that $\mathbf{e}^{(i)} \rightarrow 0$ as $i \rightarrow \infty$. As a result, the variance decay in the magnitude of the terms in (7) decreases the number of samples required to approximate $\mathbf{e}^{(i)}(\mathbf{z})$ with increasing level i [18, 25], which reduces the total computational cost for constructing an accurate surrogate.

The MLNN method is also based on identity (7), but does not use interpolation for constructing the approximations for $\mathbf{e}^{(i)}$ as used in [18]. Instead we use a convolutional neural network that approximates $\mathbf{e}^{(i)}$ in terms of a set of local low-level features, also known as latent quantities. To show that a representation of $\mathbf{e}^{(i)}$ in a set of latent quantities is justified, the error $\mathbf{e}^{(i)}$ can be written as:

$$\begin{aligned} \mathbf{e}^{(i)} &= \mathbf{u}^{(i)}(\mathbf{z})|_{X^{(i)}} - \mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(i)}} \\ &= (\mathbf{u}^{(i)}(\mathbf{z})|_{X^{(i)}} - \mathbf{u}_{\text{exc}}(\mathbf{z})|_{X^{(i)}}) - (\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(i)}} - \mathbf{u}_{\text{exc}}(\mathbf{z})|_{X^{(i)}}) \\ &= \mathbf{E}^{(i)}|_{X^{(i)}} - \mathbf{E}^{(i-1)}|_{X^{(i)}} , \end{aligned} \quad (9)$$

where $\mathbf{E}^{(i)} = \mathbf{u}^{(i)}(\mathbf{z}) - \mathbf{u}_{\text{exc}}(\mathbf{z})|_{X^{(i)}}$ denotes the global error between the solution on level i and the exact solution (sub sampled on $X^{(i)}$). The exact solution is often not known and therefore (9) is not directly useful. However, for a linear discretisation operator $L^{(i)}$, we can write:

$$\begin{aligned} L^{(i)}\mathbf{E}^{(i)} &= L^{(i)}(\mathbf{u}^{(i)} - \mathbf{u}_{\text{exc}}|_{X^{(i)}}) \\ &= L^{(i)}\mathbf{u}^{(i)} - L^{(i)}(\mathbf{u}_{\text{exc}}|_{X^{(i)}}) \\ &= -L^{(i)}(\mathbf{u}_{\text{exc}}|_{X^{(i)}}) \\ &= -\boldsymbol{\tau}^{(i)} , \end{aligned} \quad (10)$$

where $\boldsymbol{\tau}$ is the local truncation error. This can be rewritten as:

$$\mathbf{E}^{(i)} = -\left(L^{(i)}\right)^{-1} \boldsymbol{\tau}^{(i)} , \quad (11)$$

The local truncation error is the error when substituting the exact solution into the discretised operator $L^{(i)}$ and is computed locally, for instance by performing Taylor-series expansions on the schemes that are used to discretise the differential operator \mathcal{L} . The inverse operator $\left(L^{(i)}\right)^{-1}$ maps the local truncation error to the global error $\mathbf{E}^{(i)}$. For linear discretisation of linear PDEs this results in the form:

$$\boldsymbol{\tau}^{(i)} = \sum_{k=1}^{\infty} c_k (\mathbf{u}_{\text{exc}})_{\partial,k}(\mathbf{z})|_{X^{(i)}} , \quad (12)$$

where $(\mathbf{u}_{\text{exc}})_{\partial,k}$ is a sequence of partial derivatives of the exact solution with respect to spatial/temporal coordinates. The truncation error can be interpreted as a set of latent quantities, which are local or low-level features that can be used to effectively express the error in the solution. However, this only holds for linear differential equations and is therefore a rather restrictive assumption.

Our key idea is to construct an approximation for $\mathbf{e}^{(i)}$ in terms of a set of latent quantities. This set of latent quantities is obtained by applying a non-linear transformation of the solution values on a coarser level. For linear differential equations, the truncation error is a good candidate for such a set of latent quantities, but it does not generalise

to non-linear problems. Therefore, we propose a new algorithm that approximates $\mathbf{e}^{(i)}(\mathbf{z})$, $i = 2, \dots, N_L$ using a neural network approach. The *How* and *Why* of this approach are discussed in more detail next.

How a neural network is used to approximate $\mathbf{e}^{(i)}(\mathbf{z})$

The neural network architecture that is used for approximating $\mathbf{e}^{(i)}(\mathbf{z})$ is shown in figure 1. The neural network consists of two stages:

- Stage 1: Obtain latent/inferred quantities from the solution vector $\mathbf{u}^{(i)}(\mathbf{z})$, that can be mapped efficiently to $\mathbf{e}^{(i)}(\mathbf{z})$*
- Stage 2: Construct mapping from latent quantities to $\mathbf{e}^{(i)}(\mathbf{z})$*

The neural network takes as input the sub-sampled solution values $\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}$ and is trained such that it outputs $\mathbf{e}^{(i)}(\mathbf{z})$.

Stage 1, How?

The first part of the neural network that corresponds to stage 1 consists of a number of sequential convolutional layers. The filters that are used in convolutional neural networks are equivalent to a sequence of local finite difference operators whose coefficients are determined during the training procedure. As a result, the convolutional layers are used to efficiently combine the solution values into latent quantities.

Stage 1, Why?

Instead of constructing an approximation for $\mathbf{e}^{(i)}(\mathbf{z})$ directly (as done for example in [18]), we first construct a set of latent quantities from the solution vector $\mathbf{u}^{(i)}(\mathbf{z})|_{X^{(1)}}$, which are then mapped to $\mathbf{e}^{(i)}(\mathbf{z})$. We learn from equations (10)-(11), that a mapping between the solution vector and $\mathbf{e}^{(i)}(\mathbf{z})$ exists in terms of the truncation error, which consists of a set of latent quantities. As a result, the MLNN method uses a sequence of convolutional layers to find a set of latent quantities that can be used by stage 2 of the neural network to efficiently construct an approximation for $\mathbf{e}^{(i)}(\mathbf{z})$.

Stage 2, How?

After propagating the input $\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}$ through the convolutional layers, the output is flattened into a single vector and feeds the second stage of the neural network. The second stage consists of multiple fully-connected layers that apply a non-linear transformation that maps the previously obtained latent quantities to the desired quantity $\mathbf{e}^{(i)}(\mathbf{z})$. Furthermore, as we want to approximate the dependency of $\mathbf{e}^{(i)}$ on \mathbf{z} , \mathbf{z} is included in the neural network by concatenating the values of \mathbf{z} to the flattened convolutional output.

Stage 2, Why?

Solely using convolutional layers is not preferred due to their inherent local nature, making them only useful for extracting low-level features. The latent quantities obtained in stage 1 need to be mapped to the desired quantity $\mathbf{e}^{(i)}(\mathbf{z})$. As in equation (12), this requires a global and possibly non-linear transformation. It is known that fully-connected neural networks are so-called universal-function approximators [26, 27, 28] and are therefore perfectly suited for constructing the non-linear mapping between the latent quantities and $\mathbf{e}^{(i)}(\mathbf{z})$.

Stage 1 + Stage 2

The full architecture for a scalar partial differential equation with two space dimensions and uncertainties \mathbf{z} is shown schematically in figure 1.

The architecture for the coupled vector partial differential equation case requires a multi-channel input, i.e., one channel for each of the calculated quantities, which is explained in more detail in section 4. We denote the neural network that maps $\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}$ to $\mathbf{e}^{(i)}(\mathbf{z})$ as,

$$\mathbf{e}^{(i)}(\mathbf{z}) \approx P^{(i)}(\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}). \quad (13)$$

To clarify, the neural network $P^{(i)}$ requires a training set that comprises pairs of solutions $(\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}, \mathbf{u}^{(i)}(\mathbf{z})|_{X^{(1)}})$, where $\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}$ is used as input for the neural network, while $\mathbf{u}^{(i)}(\mathbf{z})|_{X^{(1)}}$ enters the cost function during training. This is discussed in more detail in section 4. In order to utilise the telescopic-sum identity (7), we need approximations

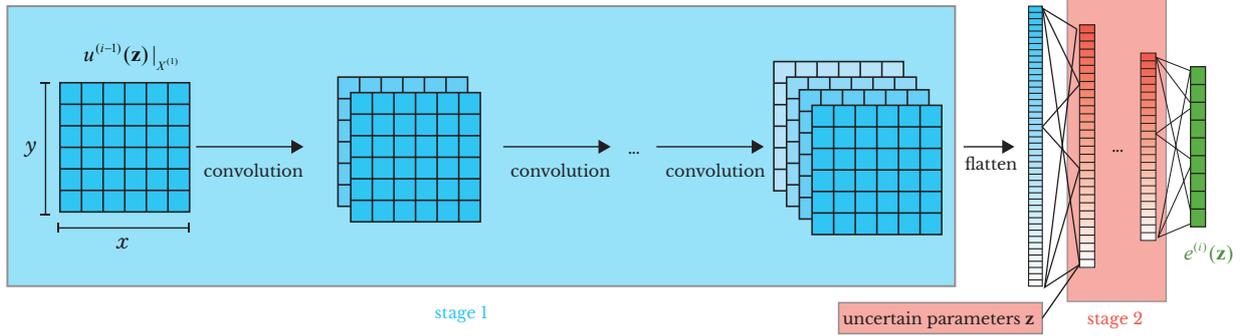


Figure 1: The convolutional and fully-connected network architecture for approximating $\mathbf{e}^{(i)}$ for a scalar partial differential equation with two space dimensions and uncertainties \mathbf{z} .

for $\mathbf{e}^{(i)}(\mathbf{z})$ with $i = 2, \dots, N_L$ and therefore we train $N_L - 1$ neural networks ($P^{(2)}, \dots, P^{(N_L)}$) of the form shown in figure 1. After training the neural networks, we can approximate the high-fidelity solution as:

$$\mathbf{u}^{(N_L)}(\mathbf{z})|_{X^{(1)}} \approx \mathbf{u}^{(1)}(\mathbf{z}) + \sum_{i=2}^{N_L} P^{(i)} \left(\mathbf{u}^{(1)}(\mathbf{z}) + \sum_{j=2}^{i-1} P^{(j)} \left(\mathbf{u}^{(1)}(\mathbf{z}) + \sum_{k=2}^{j-1} P^{(k)}(\dots) \right) \right), \quad (14)$$

where we used (7) and recursively used:

$$\mathbf{u}^{(i)}(\mathbf{z})|_{X^{(1)}} = \mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}} + \mathbf{e}^{(i)} \approx \mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}} + P^{(i)}(\mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}). \quad (15)$$

We denote the approximate high-fidelity solution as $\tilde{\mathbf{u}}^{(N_L)}(\mathbf{z}) \approx \mathbf{u}^{(N_L)}(\mathbf{z})|_{X^{(1)}}$. Notice that we only need $\mathbf{u}^{(1)}(\mathbf{z})|_{X^{(1)}}$ to construct an approximation for $\mathbf{u}^{(N_L)}(\mathbf{z})|_{X^{(1)}}$. The architecture, hyperparameters and training procedure of the neural networks will be discussed in more detail in section 4.

How the required number of samples for training $P^{(i)}$ is reduced

The combination of convolutional layers and fully-connected layers allows for learning complex non-linear mappings. The filter coefficients that are determined during training of the convolutional layers, have significantly less degrees of freedom when compared to fully-connected layers with similar input-output dimensions. As a result, the number of unknowns that need to be determined during training is significantly lower than in a network architecture with the same number of layers using solely fully-connected layers. This drastically decreases the required number of training samples in the MLNN method.

Second, it can be shown that $\mathbf{e}^{(i-1)}(\mathbf{z})$ and $\mathbf{e}^{(i)}(\mathbf{z})$ have a similar spatial shape (apart from a scaling factor). This concept is summarised in the following theorem:

Theorem 1. Assume the solution \mathbf{u} of the discretised problem (4a) can be expanded as follows:

$$\mathbf{u}^{(h)} = \mathbf{u}_{exc}|_{X^h} + h^d \mathbf{v}|_{X^h} + O(h^{d+1}), \quad (16)$$

where h is the grid resolution, d is the order of convergence with $d \geq 1$, and \mathbf{v} is a function independent of h . Then the errors $\mathbf{e}^{(h/2)} := \mathbf{u}^{(h/2)}|_{X^h} - \mathbf{u}^{(h)}$ and $\mathbf{e}^{(h/4)} := \mathbf{u}^{(h/4)}|_{X^h} - \mathbf{u}^{(h/2)}|_{X^h}$ satisfy

$$c\mathbf{e}^{(h/4)} = \mathbf{e}^{(h/2)} + O(h^k), \quad (17)$$

where $k \geq 1$ and c is a constant independent of h .

PROOF. Using the expansion (16), we can write

$$\mathbf{e}^{(h/2)} = \left(\left(\frac{h}{2} \right)^d - h^d \right) \mathbf{v}|_{X^h} + O(h^{d+1}), \quad (18)$$

$$\mathbf{e}^{(h/4)} = \left(\left(\frac{h}{4} \right)^d - \left(\frac{h}{2} \right)^d \right) \mathbf{v}|_{X^h} + O(h^{d+1}). \quad (19)$$

Multiplying the second equation with 2^d results in

$$2^d \mathbf{e}^{(h/4)} = \left(\left(\frac{h}{2} \right)^d - h^d \right) \mathbf{v}|_{X^h} + 2^d O(h^{d+1}) = \mathbf{e}^{(h/2)} + O(h^{d+1}), \quad (20)$$

which is of the form displayed in equation (17) with $k = d + 1$. \square

The existence of expansion (16) is a crucial assumption in this theorem. It has been proven in [29] (theorem 2.1) that this expansion exists for a large class of (discretisations of) linear differential equations under relatively mild conditions. Roughly speaking, these conditions are: existence and uniqueness of the discrete and continuous problem, and sufficient smoothness of the terms appearing in the discretised equations.

Theorem 1 indicates that if an expansion of the form (16) holds, then information of a previously trained neural network mapping $P^{(i-1)}$ can be used to more efficiently train $P^{(i)}$. The concept of reusing parts of a previously trained neural network is not new in the field of machine learning, and is known as transfer learning [30]. Transfer learning is used to circumvent the need for sampling many solutions on the higher levels by using weights and biases from the previously trained neural network $P^{(i-1)}$.

The concept of similarity of error profiles, as expressed by theorem 1, only holds for linear differential equations, as it relies on assumption (16). However, we numerically show in section 6 that the concept of similarity of error profiles also holds to a large extent for non-linear differential equations, which makes transfer learning applicable for non-linear cases as well.

4. Training the Neural Networks

In order to train the neural networks $(P^{(2)}, \dots, P^{(N_L)})$, we need to specify:

- I **Architecture**: layer architecture, activation functions, and hyperparameters
- II **Training/validation data**: data set for training and validating the neural networks
- III **Hyperparameter tuning**: finding proper hyperparameter values

We will discuss these three components separately in the next subsections.

I. Neural Network Architecture

As mentioned before we use a combination of convolutional layers and fully-connected layers, and the architecture is split in two stages.

1.1. Convolutional Layers

The name ‘‘convolutional layer’’ indicates that the network employs a mathematical operation called convolution. A convolutional layer is simply a fully-connected neural network layer in which the dense matrix multiplication is replaced by a convolution operator (multiplication with a sparse matrix) [31], where filters are convolved over the space of current values. In general, an n D convolutional layer accepts n_c channels of dimension $k_1 \times k_2 \times \dots \times k_n$ as input, and convolves a filter of dimension $f_1 \times f_2 \times \dots \times f_n \times n_c$ over the input channels, adds a bias, and applies a (non-)linear transformation to produce the output. This procedure is shown schematically in figure 2.

Figure 2 shows a 2D convolutional layer for a 1-channel input. The number of channels of the input corresponds to the number of quantities that is solved for in the underlying PDE. E.g., when solving 2D incompressible Navier-Stokes, we solve for the two velocity components and the pressure, which results in a 3-channel input. In this work, a

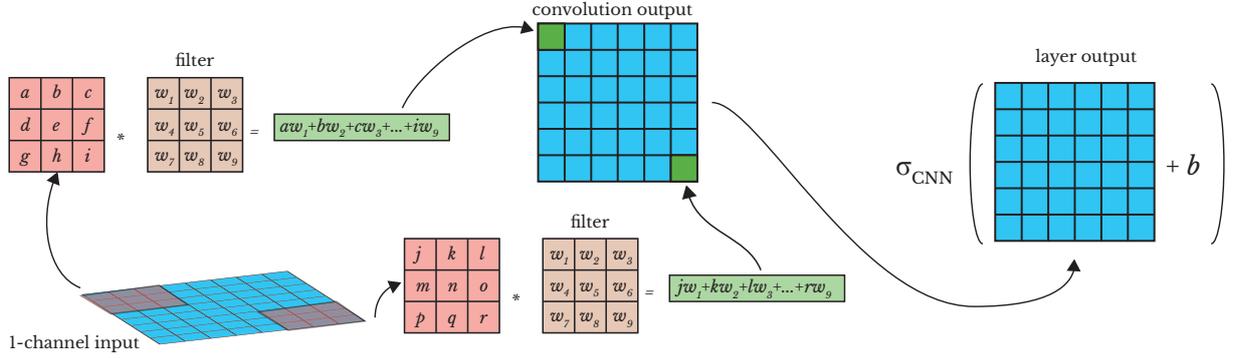


Figure 2: The 2D convolutional layer with a 1-channel input.

single 3x3 filter is used with 9 unknown filter weights w_j , which is a commonly used filter-width that yields the best results on our test cases (see section 6). In addition, a multi-channel input results in a multi-channel filter with more unknowns. It is common practice to use more than one filter, which results in an output with multiple channels, i.e., one channel associated to the convolution of one filter. Notice that the output of a convolutional layer is in general smaller than the input, as the filter cannot cross the boundaries of the input channel. Therefore, the dimensions of the output depend on the input dimensions and the filter dimensions. As a remedy, we can pad the input channel with zero values at the boundaries in order to obtain an output which has the same dimensions as the input channel. This is often referred to as zero-padding, and will be used in this paper to keep the output dimensions of the convolutional layers the same. After convolving the filters, a trainable bias coefficient is added to each output channel and the resulting values are activated with a non-linear activation function σ_{CNN} . The activation function σ for the convolutional neural network (CNN) is chosen to be the Rectified Linear Unit (ReLU):

$$\sigma_{\text{CNN}}(x) = \max(0, x), \quad (21)$$

because it does not suffer from the vanishing gradient problem and allows for a sparse representation of the output [31]. The ReLU activation function may suffer from so-called dying neurons [31]. Other variants, e.g., leaky-Relu or ELU, may be used to resolve the dying neuron problem. However, the sparse representation when using ReLU is beneficial and is therefore employed in the remainder of this work. The number of convolutional layers N_{CNN} to choose, depends on the intrinsic complexity of the dataset and is treated as a hyperparameter in the MLNN method. Lastly, increasing the number of filters with a factor 2 for consecutive CNN layers is a good starting point to tune the neural network [31], and this strategy is therefore used in this paper.

1.2. Fully-Connected Layers

After the input has been propagated through the convolutional layers, we flatten the output of the last convolutional layer into a single vector, and the uncertainties \mathbf{z} are concatenated. This vector is the start of the fully-connected part, where we apply a non-linear transformation to the output of the convolutional part. This part of the neural architecture is characterised by the number of fully-connected layers N_{FC} , the number of neurons per layer n , and the activation function σ_{FC} . The parameters N_{FC} and n are treated as hyperparameters and will be tuned during the training process. Additionally, a ReLU activation function is used for the fully-connected layers. Once we have propagated the output of the convolutional part through the N_{FC} fully-connected layers, we output a vector which has the same dimensions as the input of the network. The output layer uses a linear activation function, because it allows for unbounded output values.

I.3. Cost Function

The goal of training the neural network is to find the weights \mathbf{w} and biases \mathbf{b} , of both the convolutional and fully-connected parts, which minimise the following cost functions:

$$c^{(i)}(\mathbf{w}, \mathbf{b}) = \sum_{(\mathbf{u}^{(i)}, \mathbf{u}^{(i-1)}) \in T^{(i)}} \|P^{(i)}(\mathbf{u}^{(i-1)}|_{X^{(1)}}) - (\mathbf{u}^{(i)}|_{X^{(1)}} - \mathbf{u}^{(i-1)}|_{X^{(1)}})\|_2^2, \quad i = 2, \dots, N_L, \quad (22)$$

where $T^{(i)}$ is the training set. The construction of this training set is discussed in section II in more detail.

Overfitting may occur due to the large amount of unknowns, i.e., convolutional weights/biases and fully-connected layer weights/biases. In order to prevent overfitting, we use an l_2 -regularisation, which introduces a new hyperparameter λ and the altered cost functions:

$$c_\lambda^{(i)}(\mathbf{w}, \mathbf{b}) = \sum_{(\mathbf{u}^{(i)}, \mathbf{u}^{(i-1)}) \in T} \|P^{(i)}(\mathbf{u}^{(i-1)}|_{X^{(1)}}) - (\mathbf{u}^{(i)}|_{X^{(1)}} - \mathbf{u}^{(i-1)}|_{X^{(1)}})\|_2^2 + \lambda \|\mathbf{w}\|_2^2, \quad i = 2, \dots, N_L. \quad (23)$$

After training the neural network, we validate if the neural network generalises to unseen data by computing the validation error:

$$v^{(i)} = \sum_{(\mathbf{u}^{(i)}, \mathbf{u}^{(i-1)}) \in V^{(i)}} \|P^{(i)}(\mathbf{u}^{(i-1)}) - (\mathbf{u}^{(i)} - \mathbf{u}^{(i-1)})\|_2^2, \quad i = 2, \dots, N_L, \quad (24)$$

where $V^{(i)}$ is the validation set. The validation error is used to tune the hyperparameters of the network architecture. Additionally, a test set can be used to check if the neural network with proper hyperparameters generalises well outside the training/validation set. However, using a test set results in an increase of required number of samples, and as generalisation is only important within or close to the range of training/validation data, we choose to solely use a training and validation set.

I.4. Transfer Learning

The amount of training data required for obtaining a proper set of weights and biases may become infeasible when increasing the level, as it requires a training set that is constructed by sampling many high-fidelity solutions. When training $P^{(i)}$, $i = 3, \dots, N_L$, transfer learning is used to circumvent the need for sampling many solutions on the higher levels, by using weights and biases from the previously trained neural network $P^{(i-1)}$. Transfer learning can be performed in multiple ways [30]. We choose to fix the weights and biases of the previously trained neural network $P^{(i-1)}$ and add a small number of fully-connected layers at the end, while keeping the rest of the architecture the same. As the input for $P^{(i)}$, $i = 2, \dots, N_L$ is a solution vector, which is defined on the same grid $X^{(1)}$, the learned convolutional filters are still expected to compute a proper set of latent quantities when increasing the level i . Therefore, the transfer learning approach leaves the convolutional part unaltered for all $P^{(i)}$, $i = 3, \dots, N_L$. The transfer learning procedure is schematically shown in figure 3. A single fully-connected layer is added with the same activation function as the

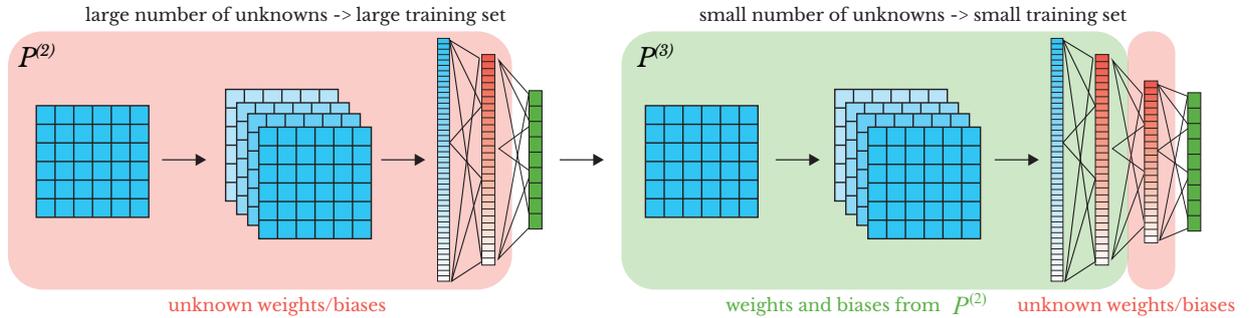


Figure 3: Transfer learning procedure.

other fully-connected layers, but with a hyperparameter n representing the number of neurons in the new layer.

II. Training/Validation Data

The neural networks $\{P^{(i)}\}_{i=2}^{N_L}$ are trained to approximate the errors $\mathbf{e}^{(i)}(\mathbf{z})|_{X^{(1)}} = \mathbf{u}^{(i)}(\mathbf{z})|_{X^{(1)}} - \mathbf{u}^{(i-1)}(\mathbf{z})|_{X^{(1)}}$. The training set for training $P^{(i)}$ requires solution values for both $\mathbf{u}^{(i-1)}(\mathbf{z})$ and $\mathbf{u}^{(i)}(\mathbf{z})$, sampled at multiple values for \mathbf{z} . The training set that is used for training $P^{(i)}$ is denoted as:

$$T^{(i)} := \left\{ \left(\mathbf{u}^{(i)}(\mathbf{z}_k)|_{X^{(1)}}, \mathbf{u}^{(i-1)}(\mathbf{z}_k)|_{X^{(1)}} \right) \mid k = 1, \dots, N_{\text{training}} \right\}. \quad (25)$$

In order to validate if the neural network generalises to values not in the training set, we validate our neural network on the set:

$$V^{(i)} := \left\{ \left(\mathbf{u}^{(i)}(\mathbf{z}_k)|_{X^{(1)}}, \mathbf{u}^{(i-1)}(\mathbf{z}_k)|_{X^{(1)}} \right) \mid k = 1, \dots, N_{\text{validate}} \right\}. \quad (26)$$

The amount and quality of data that is used to train and validate the neural network is paramount. How to ensure that we obtain a proper set of weights and biases that generalises well to unseen values, is discussed in this section. The required amount of training data depends on the complexity of the response to be approximated and on the total number of unknowns in the neural network architecture, i.e., the weights and biases. The numbers of unknown weights and biases in the neural networks $\{P^{(i)}\}_{i=2}^{N_L}$ decrease significantly by using the transfer learning approach described in section I. As a result, we need a large amount of training samples when training $P^{(2)}$, which is feasible when assuming that low-fidelity solutions are computationally cheap to sample. As opposed to this, sampling high-fidelity solutions for training $P^{(N_L)}$ is computationally expensive, but we require significantly less training samples, due to variance decay in $\mathbf{e}^{(i)}$ for increasing i [25] and the small amount of unknown weights and biases induced by transfer learning. The quality of the training data is determined by the sample locations \mathbf{z}_k in (26). As determining a proper size of the training/validation set is difficult [31], we employ a sampling strategy which iteratively adds new samples $\mathbf{u}^{(i)}(\mathbf{z}_k)$ when the neural network does not generalise well to the validation set.

The training procedures for the neural networks $\{P^{(i)}\}_{i=2}^{N_L}$ are explained in more detail below.

Training $P^{(2)}$

The first neural network that needs to be trained is $P^{(2)}$. When training $P^{(2)}$, we approximate the error

$$\mathbf{e}^{(2)}(\mathbf{z})|_{X^{(1)}} = \mathbf{u}^{(2)}(\mathbf{z})|_{X^{(1)}} - \mathbf{u}^{(1)}(\mathbf{z}), \quad (27)$$

and to construct $T^{(2)}$, the sample locations \mathbf{z}_k need to be specified. As the low-fidelity solutions $\mathbf{u}^{(1)}(\mathbf{z})$ and $\mathbf{u}^{(2)}(\mathbf{z})$ are assumed to be computationally cheap to sample from, we place many samples on the first two levels. In this work, the initial sample set comprises a relatively large set of $10^{\dim(I)}$ solutions in the random space I , which shows to give good results for our test cases in section 6. However, as we employ an iterative sampling strategy, the final size of the sample set is tailored to the underlying problem, which makes a proper size of the initial sample set less significant. The locations are defined using Monte Carlo sampling according to the underlying PDF of the uncertainties $\rho(\mathbf{z})$. As transfer learning can not be used for training $P^{(2)}$, the number of unknown weights and biases is large compared to $P^{(i)}$, $i = 3, \dots, N_L$. As a result, the initial sample set should be large enough to properly train the neural network, but also small enough to be feasible in high-dimensional random spaces. The sample set is split using the 80/20%-rule in a training set $T^{(2)}$ and a validation set $V^{(2)}$. Splitting the full sample set according to this 80/20% ratio is known as the Pareto Principle [31] and is commonly used in machine learning for determining the size of the training and validation sets. After minimising (23), we tune the hyperparameters and pick the network architecture which results in the smallest validation error $v_{\min}^{(i)}$. This is discussed in more detail in section III. Training is stopped if $v_{\min}^{(i)}$ satisfies:

$$v_{\min}^{(i)} < \varepsilon, \quad (28)$$

where ε is a specified threshold. The threshold ε indicates how well the trained neural network generalises to data that is not contained in the training set. If the stopping criterion is not met, we increase the sample set by sampling another $10^{\dim(I)}$ new PDE solutions. The newly obtained larger sample set is again split randomly in a training/validation set following the 80/20%-rule, which are used to retrain the neural network. This process of training, validating, enlarging the training set is repeated until (28) is satisfied. Notice that each time the sample set is increased, the neural network needs to be retrained, which is assumed to be a relatively cheap operation when compared to computing a high-fidelity solution, as the weights and biases of the previous training can be used as a very good initial guess for the retraining.

Training $P^{(i)}$, $i > 2$

Constructing a large training set when increasing i is often not feasible. Therefore, the combination of transfer learning and iterative sample set construction is used to limit the required number of training samples. Training the relatively small neural networks is fast in general. We iteratively increase the size of the training set during the training procedure. We again start with a small sample set that is used for training/validation of the neural network. After training the neural network we check if (28) is met, if not, then new samples are added to the sample set, which effectively increases the size of the training/validation set. This process is repeated until (28) is satisfied.

Initially, the sample set comprises $2^{\dim(I)}$ randomly placed samples in the space I , which is split in a training set $T^{(i)}$ and validation set $V^{(i)}$ following the 80/20%-rule. The size of the initial sample set is smaller when compared to the initial sample set used for training $P^{(2)}$ to alleviate computational burden when sampling the computationally expensive higher-fidelity solutions. The number of samples in the initial sample set is chosen to be small in order for the MLNN method to still be feasible for high-dimensional random spaces. As mentioned before, the use of an iterative sampling strategy makes the size of the initial sample set insignificant. Furthermore, notice that we only need to sample solutions for $\mathbf{u}^{(i)}$ and $\mathbf{u}^{(1)}$, as accurate approximations of $\mathbf{u}^{(i-1)}$ can be constructed using (14) with $N_L = i - 1$.

Nesting of samples on different levels removes the need for sampling additional solutions for $\mathbf{u}^{(i-1)}$, but introduces correlated expectations on the different levels, which is unwanted [24]. Strategies to reuse samples on different levels have recently been proposed [32], and could be used to further enhance the MLNN method, but are not required to show the basic methodology that we propose.

After training the neural network, we check if criterion (28) is met. If the stopping criterion is not met, we increase the sample set by sampling $2^{\dim(I)}$ new PDE solutions. The newly obtained larger sample set is again split randomly in a training/validation set following the 80/20%-rule, which is used to retrain the neural network. This process of training, validating, enlarging the training set, is repeated until we satisfy (28).

III. Hyperparameter Tuning

When training the neural networks, values for the hyperparameters need to be specified. A complete list of hyperparameters in this work is shown below:

- Hyperparameters when training $P^{(2)}$:
 - λ (regularisation parameter)
 - N_{CNN} (number of convolutional layers)
 - N_{FC} (number of fully-connected layers)
 - n (number of neurons per fully-connected layer)
- Hyperparameters when training $P^{(i)}$, $i > 2$:
 - $\lambda^{(i)}$ (regularisation parameter)
 - $n^{(i)}$ (number of neurons in the added fully-connected layer)

To tune these hyperparameters, we use a grid search, which specifies pre-defined values for each parameter, and constructs a tensor grid of all possible combinations of hyperparameter values. A neural network is then trained for each combination of hyperparameter values. To limit the total number of neural networks we have to train, we pick only 3 values for each hyperparameter, which are commonly used in deep-learning approaches:

- $\lambda \in \{0, 10^{-6}, 10^{-3}\}$,
- $N_{\text{CNN}} \in \{2, 4, 6\}$,
- $N_{\text{FC}} \in \{1, 3, 5\}$,
- $n \in \{\lfloor \frac{N^{(1)}}{2} \rfloor, N^{(1)}, 2N^{(1)}\}$.

This range of values for the hyperparameters has been shown to work well for many cases [30, 31, 33, 34, 35, 36] and is therefore employed in this paper.

5. Complete algorithm

A schematic representation of the MLNN method is shown in figure 4.

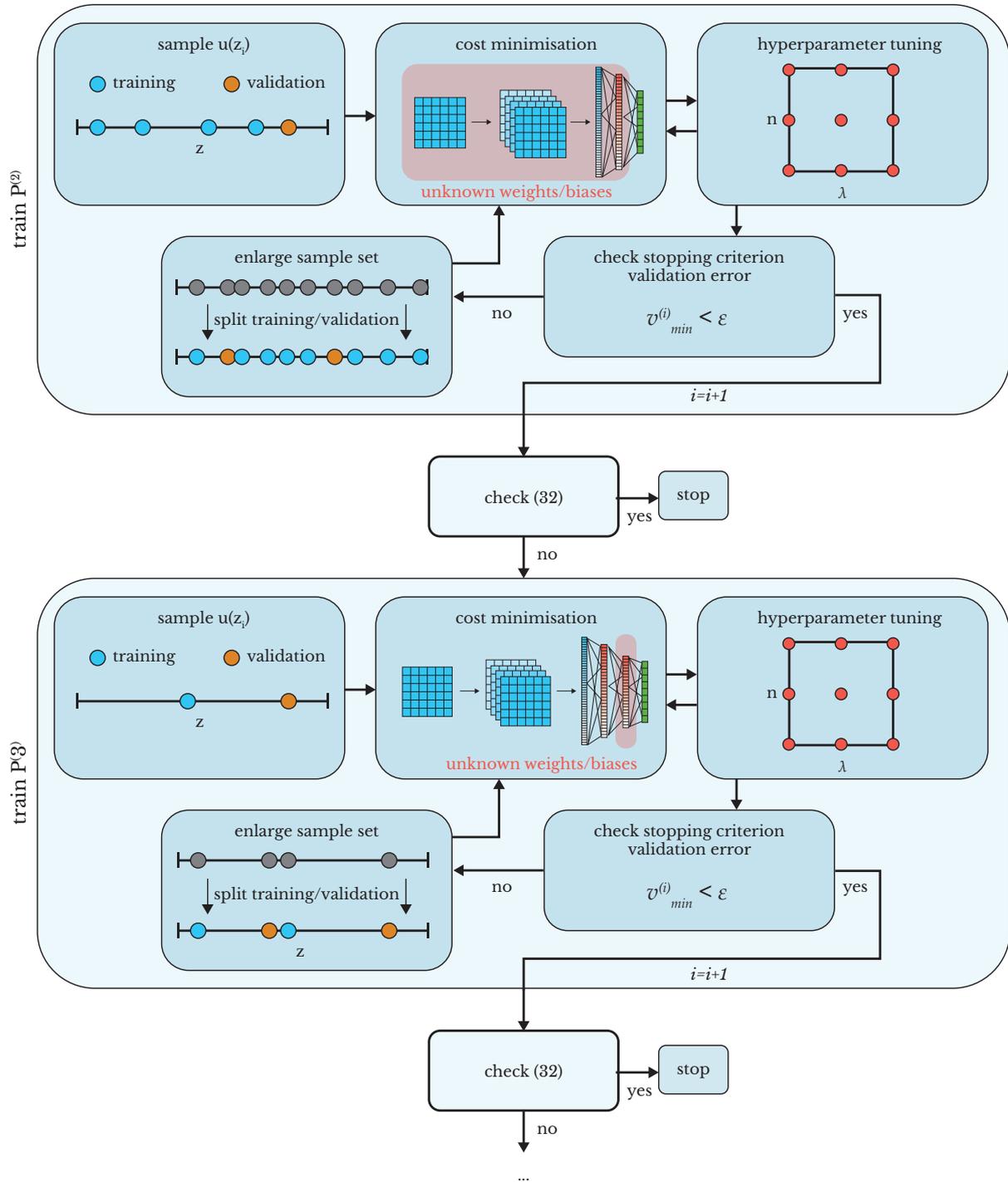


Figure 4: Schematic overview of the MLNN method.

The number of levels that should be picked, depends on the decay of $\mathbf{e}^{(i)}$ with increasing i . As a result, an intuitive criterion for adding new levels is:

$$\text{if } \exists \mathbf{u}^{(N_L-1)} \in T^{(N_L)} : \|P^{(N_L)}(\mathbf{u}^{(N_L-1)})\|_2 > \varepsilon_{\text{acc}} \rightarrow N_L = N_L + 1, \quad (29)$$

where ε_{acc} is the accuracy tolerance value. To clarify, two thresholds need to be specified, i.e., ε in (28) and ε_{acc} in (29). As mentioned before, the threshold ε indicates how well the trained neural network generalises to data not contained in the training set. The threshold ε_{acc} corresponds to the accuracy of the approximated high-fidelity solution when using (14). In general, ε_{acc} is set to the required accuracy of the approximation, and ε is set to a value which is one or two orders of magnitude smaller than ε_{acc} to ensure that the accuracy of the approximate high-fidelity solution is not influenced by possibly poor training of the neural networks.

The MLNN method is flexible and robust and works for a wide variety of problems, some of which are shown in the next section. We note that the assumption that $\mathbf{e}^{(i-1)}(\mathbf{z})$ and $\mathbf{e}^{(i)}(\mathbf{z})$ should possess similar behavior is a key assumption in the MLNN method. If this assumption does not hold, the transfer learning approach is not optimal and one might still require many samples for approximating $\mathbf{e}^{(i)}(\mathbf{z})$. The extension of transfer learning to problems where the similarity assumption does not hold will require further research in transfer learning; this is an open topic of research in the field of machine learning and it is therefore difficult to estimate how much $\mathbf{e}^{(i-1)}(\mathbf{z})$ and $\mathbf{e}^{(i)}(\mathbf{z})$ may deviate and which features are transferable [37].

6. Results

In this section we use the MLNN method to do surrogate construction for test cases with ranging complexity. These surrogates may be used to extract statistical quantities, e.g., mean and variance. As the purpose of this paper is to construct accurate surrogates, we assume that all the uncertain parameters are uniformly distributed, which does not put any emphasis on certain parameter configurations. The neural networks are constructed and trained using Tensorflow [38], which is a highly optimised library for machine learning.

I. Steady-State Advection Diffusion Equation

In this section we study the following:

- Construction of a parametric solution for a linear 1D differential equation.
- Behaviour of $P^{(i)}$ for a linear differential equation.
- Extrapolation outside the training domain.
- Comparison with MLSC [18].

I.1. Parametric Solution

We employ the MLNN method for constructing a parametric solution for the 1D steady-state advection diffusion equation:

$$\frac{du}{dx} - \frac{1}{\text{Re}} \frac{d^2u}{dx^2} = 0, \quad u(0) = 0, \quad u(1) = 1, \quad x \in [0, 1], \quad (30)$$

where Re is the Reynolds number and is uncertain, i.e., $\mathbf{z} = \text{Re}$. The exact solution is given by:

$$u_{\text{exc}}(x, \text{Re}) = \frac{\exp(x\text{Re}) - 1}{\exp(\text{Re}) - 1}. \quad (31)$$

We aim to construct a parametric solution for u as a function of $\text{Re} \in [1, 100]$.

The equations are discretised using a finite-difference approach on an equidistant grid with a resolution of Δx with $N + 1$ grid points. The solution vector on the computational grid $\mathbf{u} = (u_i)_{i=0}^N \approx (u(x_i))_{i=0}^N$, with $x_i = i\Delta x$ where $\Delta x = 1/N$, is obtained by solving the following linear system:

$$L_1 \mathbf{u} - \frac{1}{\text{Re}} L_2 \mathbf{u} = \mathbf{S}(\text{Re}), \quad (32)$$

where

$$L_1 = \frac{1}{2\Delta x} \begin{pmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & 0 & -1 & 0 \end{pmatrix}, \quad L_2 = \frac{1}{\Delta x^2} \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & 0 & 1 & -2 \end{pmatrix}, \quad (33a)$$

$$\mathbf{S}(\text{Re}) = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ -\frac{1}{2\Delta x} + \frac{1}{\text{Re}\Delta x^2} \end{pmatrix}, \quad (33b)$$

where the boundary conditions enter the discretised equation via the vector \mathbf{S} . The discretisation scheme is second order accurate and therefore the error in the solution converges with $O(\Delta x^2)$. The exact parametric solution for $\text{Re} \in [1, 100]$ is shown in figure 5. The fidelity of the solution increases when N increases, and we choose to increase

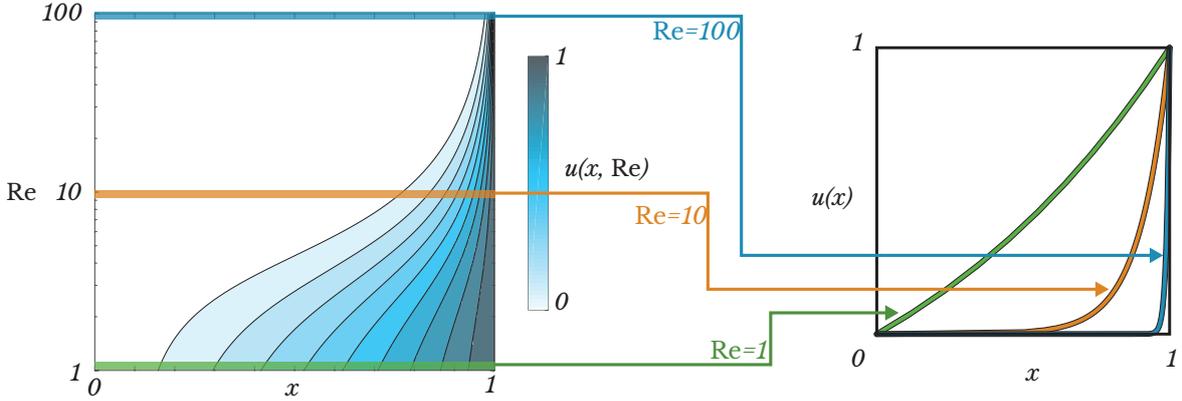


Figure 5: Advection diffusion equation. The exact parametric solution for the advection diffusion equation.

the fidelity by increasing the number of grid points with a factor 2 for consecutive levels, i.e., $N^{(i)} = 2N^{(i-1)}$. In order for the discretisation to produce stable results, the cell Reynolds number $\text{Re}_{\Delta x} = \text{Re}\Delta x$ should satisfy $\text{Re}_{\Delta x} < 2$, which is satisfied by picking Δx sufficiently small. The largest value for Re is 100, and therefore picking $N^{(1)} = 100$ satisfies the cell Reynolds condition.

The parametric solution is constructed using an accuracy tolerance $\epsilon_{\text{acc}} = 10^{-6}$, which corresponds to the desired accuracy of the surrogate. The training tolerance is set to a value which is two orders of magnitude smaller, $\epsilon = 10^{-8}$, following the rule explained in section 5. As mentioned in section II, we start with 10 samples on the coarsest level (8 training samples, 2 validation samples), and apply the MLNN method from there on. The error between our approximate solution (14) and the exact solution (31) for a varying number of levels is shown in figure 6. The error increases with increasing Reynolds number, due to the more difficult approximation of the thin boundary layer at high Reynolds numbers. As expected, we see a decrease in required number of samples with increasing level. Furthermore, the spacing between the errors for different levels at a specific Reynolds number (left figure) indicates that the method is indeed second order accurate.

1.2. Neural Network Mappings $P^{(i)}$

The neural network mappings $P^{(i)}$, $i = 2, 3, 4, 5$ are shown in figure 7. Important to note is that $P^{(i)}$, $i = 2, 3, 4, 5$ are approximately equal, apart from a scaling factor, which is the key insight utilised by our transfer-learning approach. This is as expected, because it can be shown that discrete solutions of this particular steady-state advection-diffusion

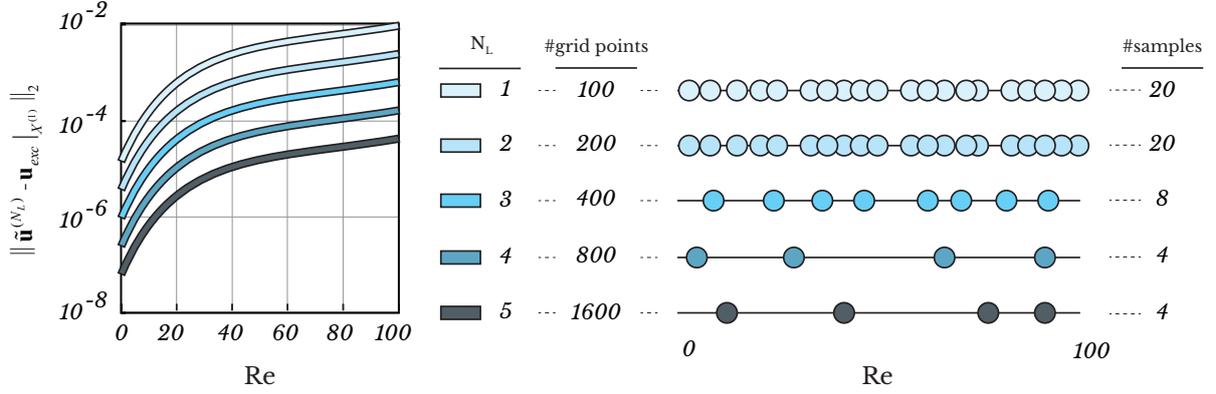


Figure 6: Advection diffusion equation. (left) Error behaviour for different number of total levels. (right) Sample placement on each level.

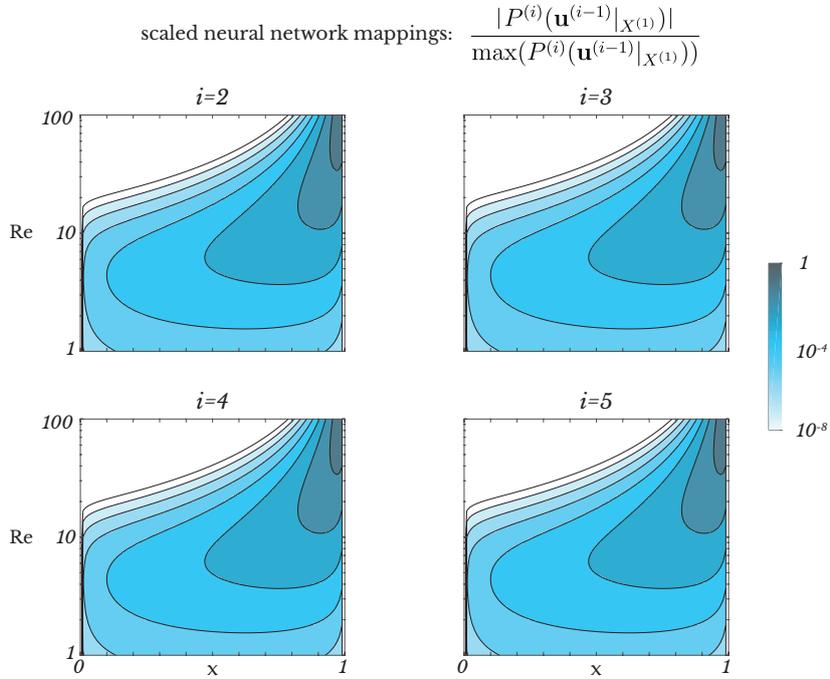


Figure 7: Advection diffusion equation. The neural networks mappings for the advection-diffusion test case in (x, Re) -space.

boundary value problem satisfy the conditions outlined in [29] (Theorem 2.1) so that expansion (18) holds [39, 40], and therefore theorem 1 holds for this case. As mentioned before, when the level increases, the error between consecutive levels decreases. As a result, less samples are required to approximate these high-level mappings, which decreases the required number of high-fidelity samples.

1.3. Extrapolation Capabilities

Figure 6 shows that the MLNN method is suitable for constructing parametric solutions inside the training domain $Re \in [1, 100]$ efficiently. However, extrapolation outside the domain where the neural networks are trained is difficult in general. Figure 8 shows the errors between our approximate solution, based on $P^{(i)}$, $i = 2, 3, 4, 5$ and computed with (14), and the exact solution (31) for $Re > 100$. The errors in the approximate solution increase rather drastically when

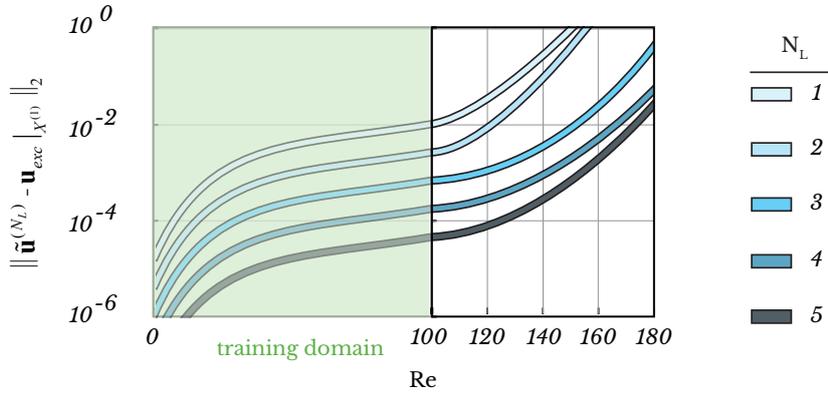


Figure 8: Advection diffusion equation. Extrapolation errors outside the training domain $Re \in [1, 100]$.

using the trained neural networks outside the training domain. As a result, the trained neural networks are not suitable for constructing accurate solutions outside the training domain, when deviating too far from the domain boundaries [31]. This is a common problem in machine learning. Fixing this problem is not the scope of this manuscript.

1.4. Comparison with MLSC

We compare the computational cost of MLNN and MLSC (as proposed in [18]) for the case of constructing a surrogate with a specified accuracy. The computational cost comprises both solving the underlying differential equation and sampling procedure, i.e., training the neural networks (MLNN) or constructing the Clenshaw-Curtis grids (MLSC). The dependence of the computational cost on the implementation is negligible, because MLSC requires a minimum amount of implementation whereas training and constructing the neural networks for the MLNN method are performed using Tensorflow, which is a highly optimised library for machine learning. Both approaches use a sampling threshold on each level of $\varepsilon = 10^{-10}$. The levels are determined by the grid resolution which is refined with a factor 2 for subsequent levels. As both approaches use the same solver with the same grid resolution on each level, we are able to compare both approaches directly. The computational costs are scaled with respect to the maximum computational cost of the MLSC approach and the results are shown in figure 9. Notice that the error on the horizontal axis is directly

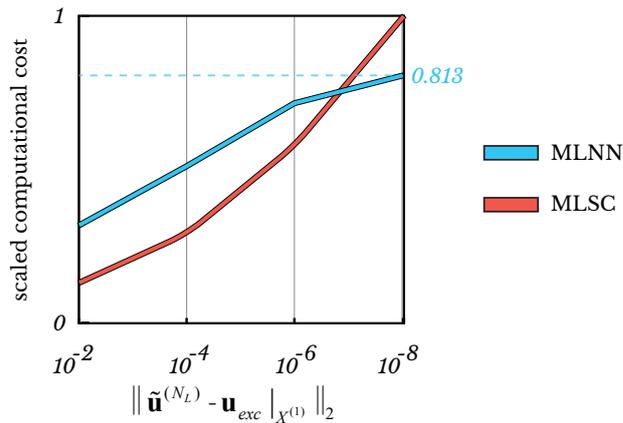


Figure 9: Advection diffusion equation. The scaled computational cost when constructing a surrogate for a range of accuracies.

related to the number of levels that are used in both approaches. The MLNN method requires more computational

time for the approximation on the first level when compared to the MLSC. However, the transfer learning approach significantly reduces the required number of samples on subsequent levels. As a result, MLNN appears to be more computationally efficient with increasing surrogate accuracy.

II. Steady-State Burgers Equation

In this section we increase the complexity of the previous test case by introducing a non-linearity in the underlying differential equation. We study the following:

- Construction of a parametric solution for a non-linear 1D differential equation.
- Behaviour of $P^{(l)}$ for a non-linear differential equation.
- Comparison with MLSC [18].

II.1. Parametric Solution

In order to study how the MLNN method performs in the presence of non-linearities in the underlying equation, we consider the non-linear steady-state Burgers equation:

$$\frac{1}{2} \frac{du^2}{dx} - \frac{1}{\text{Re}} \frac{d^2u}{dx^2} = 0, \quad u(0) = 0, \quad u(1) = 1, \quad x \in [0, 1], \quad (34)$$

where Re is again the Reynolds number and is assumed to be uncertain, i.e., $\mathbf{z} = \text{Re}$. The effect of the non-linearity increases for increasing Reynolds number. In the linear advection-diffusion case we considered $\text{Re} \in [1, 100]$, but this range is increased to have a more pronounced non-linear effect in the solution at higher Reynolds numbers. As a result, we aim to construct a parametric solution for u as a function of $\text{Re} \in [1, 1000]$.

The equations are discretised using a finite-difference approach on an equidistant grid with a resolution of Δx with $N + 1$ grid-points. The solution vector on the computational grid $\mathbf{u} = (u_i)_{i=0}^N \approx (u(x_i))_{i=0}^N$, with $x_i = i\Delta x$ where $\Delta x = 1/N$, is obtained by solving the following non-linear system:

$$\mathbf{F}(\mathbf{u}) = 0, \quad (35)$$

where

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} \frac{1}{4\Delta x} (u_2^2 - u_0^2) - \frac{u_0}{\text{Re}\Delta x^2} (u_2 - 2u_1 + u_0) \\ \vdots \\ \frac{1}{4\Delta x} (u_N^2 - u_{N-2}^2) - \frac{1}{\text{Re}\Delta x^2} (u_N - 2u_{N-1} + u_{N-2}) \\ u_N - 1 \end{pmatrix}. \quad (36)$$

This non-linear system is solved using Newton iteration and the complete discretisation scheme is second order accurate. We choose to increase the fidelity by increasing the number of grid-points with a factor 2 for consecutive levels, i.e., $N^{(i)} = 2N^{(i-1)}$, and picking $N^{(1)} = 300$. The grid-resolution of the first level is picked such that we produce stable results for $\text{Re} \in [1, 1000]$. The parametric solution for $\text{Re} \in [1, 1000]$ is shown alongside the solution error convergence for $\text{Re} = 1000$ in figure 10. As for the linear case, the tolerance for the training procedure is set to $\varepsilon = 10^{-8}$ and the accuracy tolerance is set to $\varepsilon_{\text{acc}} = 10^{-6}$. As mentioned in section II, we start with 10 samples on the coarsest level (8 training samples, 2 validation samples), and apply the MLNN method from there on. The error between our approximate solution (14) and the converged solution, with a varying number of levels, is shown in figure 11. Again, we see a decrease in required number of samples with increasing level. The error increases with increasing Reynolds number, due to the more difficult approximation of the thin boundary layer at high Reynolds numbers and possibly the increasing effect of the non-linearity in the underlying equation.

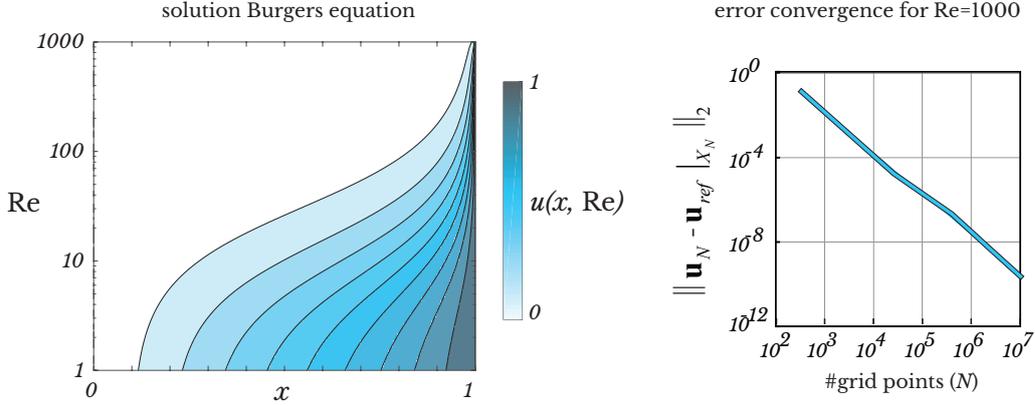


Figure 10: Burgers equation. (left) The parametric solution for the Burgers equation computed with $N = 10^5$ grid points. (right) The convergence of the error in the solution for $\text{Re} = 1000$ as a function of the number of grid points. The reference solution is computed with $N = 10^8$.

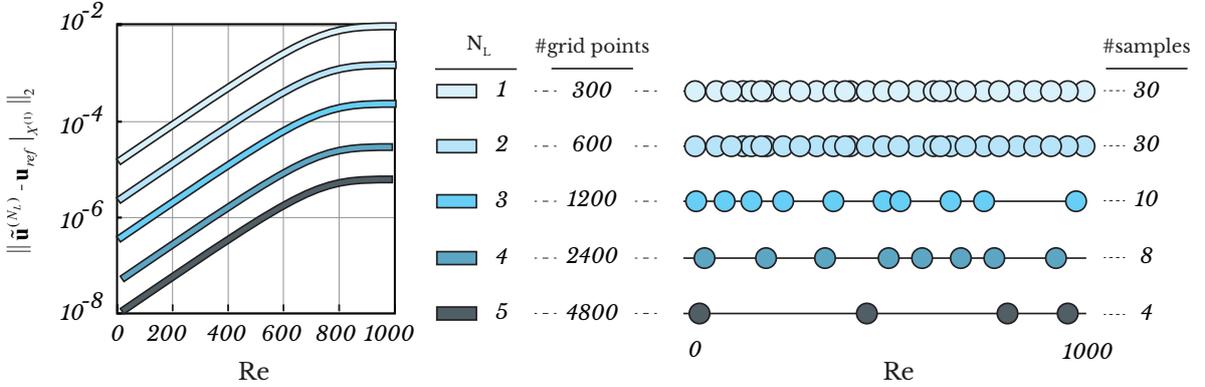


Figure 11: Burgers equation. (left) Error convergence for different number of total levels. (right) Sample placement on each level.

II.2. Neural Network Mappings $P^{(i)}$

The neural network mappings $P^{(i)}$, $i = 2, 3, 4, 5$ are shown in figure 12. The approximated mappings $P^{(i)}$ differ slightly for consecutive levels, which is caused by the increasing effect of the non-linearity in the underlying PDE for higher Reynolds numbers. This is in contrast to the previous linear test case, where the scaled mappings remained basically indistinguishable for increasing levels. Note that for $\text{Re} \in [1, 100]$ the mappings for the consecutive levels are very similar, just as with the linear advection-diffusion equation.

To summarise, the MLNN method shows similar result when comparing it to the linear advection-diffusion test case. The number of samples required on each level increased, due to the more complex error responses $\mathbf{e}^{(i)}(\mathbf{z})$, which is caused by the increasing non-linearity in the underlying PDE with increasing Reynolds number. However, as the error responses are still similar, our proposed transfer learning approach is a very efficient way to learn the neural network mappings with increasing i .

II.3. Comparison with MLSC

We compare the computational cost of MLNN and MLSC for the case of constructing a surrogate with a certain accuracy. The computational cost is computed in the same way as described in section 6.I.4 and the results are shown in figure 13. The MLNN method requires significantly more computational time for the approximation on the first level when compared to the MLSC. Again we benefit using our approach for higher accuracy levels.

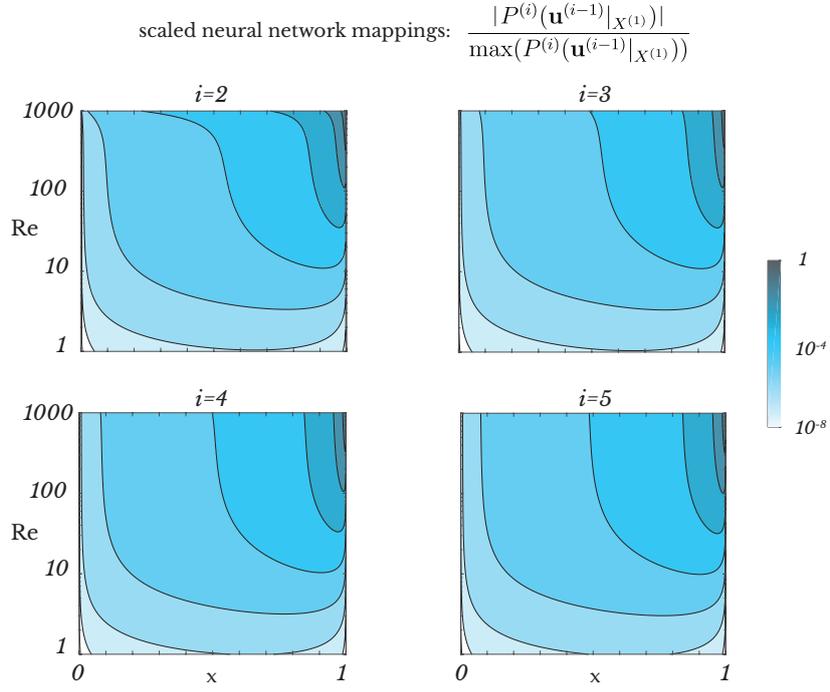


Figure 12: Burgers equation. The neural networks mappings for the Burgers test case in (x, Re) -space.

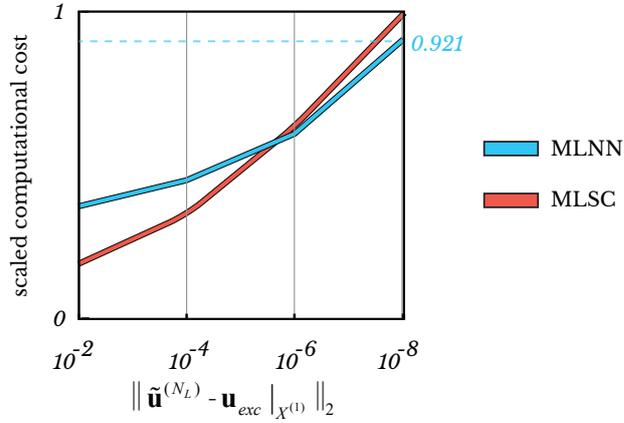


Figure 13: Burgers equation. The scaled computational cost when constructing a surrogate for a range of accuracies.

III. Steady-State Incompressible Navier-Stokes Equations

In this section we study the following:

- Construction of a parametric solution for the 2D steady-state incompressible Navier-Stokes equations.
- Behaviour of $P^{(i)}$ for the steady-state incompressible Navier-Stokes equations.
- Comparison with MLSC [18].

III.1. Parametric Solution

This test case discusses the steady-state flow over a backward-facing step, which is a common fluid mechanics problem. The governing equations are the steady-state incompressible Navier-Stokes equations in dimensionless form:

$$\nabla \cdot \mathbf{u} = 0, \quad (37a)$$

$$(\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \frac{1}{\text{Re}}\nabla^2\mathbf{u}. \quad (37b)$$

where $\mathbf{u} = (u, v)$ is the velocity field, p the modified pressure, and Re the Reynolds number which is assumed to be uncertain, i.e., $\mathbf{z} = \text{Re}$. The incompressible Navier-Stokes equations are hard to solve due to the non-linearity and the implicit coupling between mass conservation (37a) and momentum conservation (37b) by means of the pressure.

The set of steady state Navier-Stokes equations (37a)-(37b) are accompanied with a proper set of boundary conditions on a specified domain. In this section we consider the boundary conditions and domain that correspond to the backward-facing step problem, of which a schematic representation is shown in figure 14. The domain comprises a

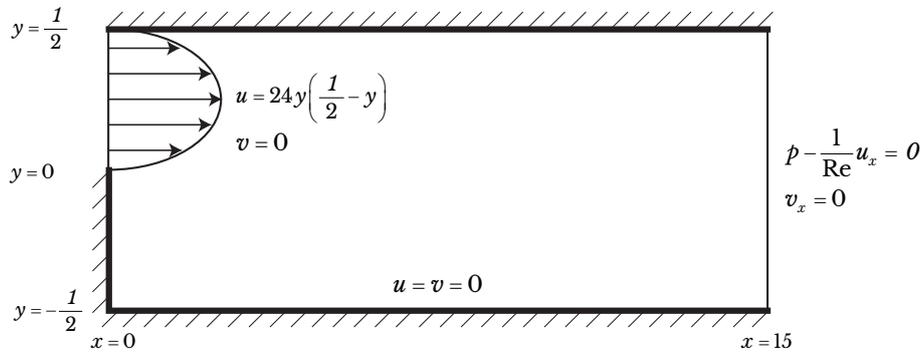


Figure 14: Schematic of the backward-facing step problem.

rectangle with length 15 and height 1 with solid boundaries, except for the upper part of the inlet ($x = 0$) and the full outlet ($x = 15$). A Poiseuille flow is imposed at the upper part of the inlet, while a pressure outlet condition is enforced at the outlet. The solver [41] is verified by comparing solutions for different mesh-sizes with a reference solution, which is believed to be an accurate benchmark for $\text{Re} = 800$ [42]. The flow is highly dependent on the Reynolds number, and the solutions for $\text{Re} = 100$ and 800 are shown in figure 15. The Reynolds number determines the size

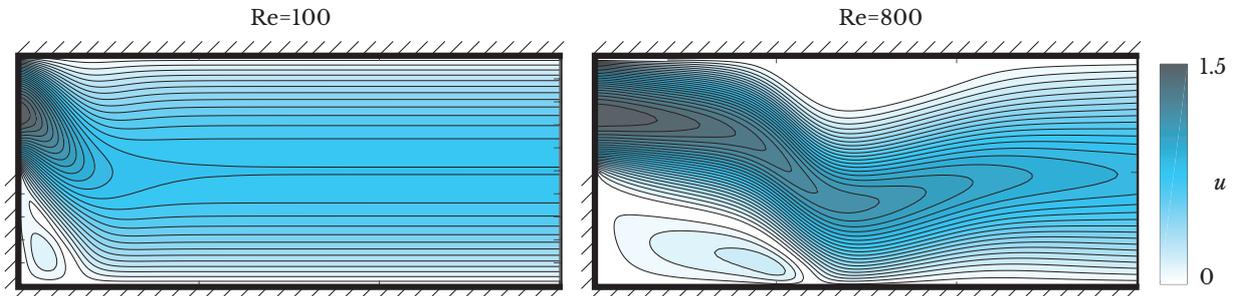


Figure 15: Examples of flows over a backward facing step at different Reynolds numbers.

and location of the recirculating flow region right after the step and near the top boundary. The flow develops to a steady Poiseuille flow after a distance which is determined by the Reynolds number. However, in this case the flow

is not yet fully developed and that is why we impose no Poiseuille boundary conditions at the outlet. The u -velocity profiles at $x = 7$ and $x = 15$ are shown in figure 16.

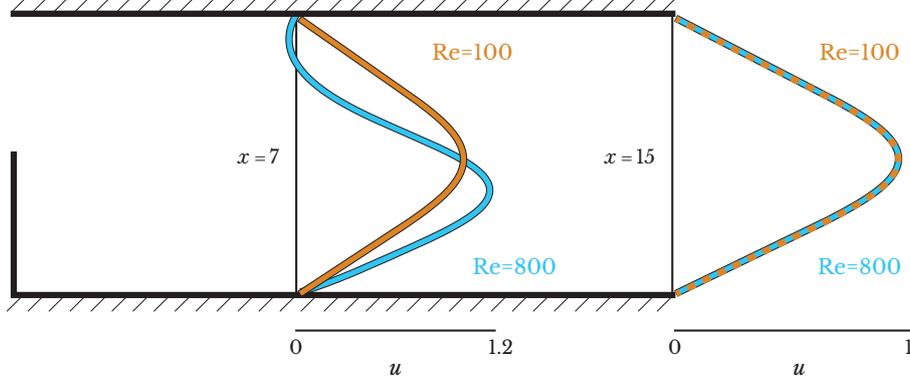


Figure 16: The u -velocity profiles for backward facing step at $x = 7$ and $x = 15$ for two different Reynolds numbers.

The solutions on the first level are computed on a 240×16 grid, and consist of the three quantities $(\mathbf{u}, \mathbf{v}, \mathbf{p})$. We choose to increase the fidelity by increasing the number of grid points with a factor 2 for consecutive levels, i.e., $(N_x^{(i)}, N_y^{(i)}) = 2(N_x^{(i-1)}, N_y^{(i-1)})$. All three quantities $(\mathbf{u}, \mathbf{v}, \mathbf{p})$ are given as input to the neural network as a 2D 3-channel convolutional input. As in the previous Burgers test case, the tolerance for the training procedure is set to $\varepsilon = 10^{-6}$ and an accuracy tolerance $\varepsilon_{\text{acc}} = 10^{-4}$ is used. We start with 10 samples on the coarsest level (8 training samples, 2 validation samples), and apply the MLNN method from there on. The normed error difference for different numbers of levels, between our approximate solution and the solution computed on a fine 3840×256 grid (corresponding to level 5), is shown in figure 17. The number of samples required for approximating the mappings accurately, increases

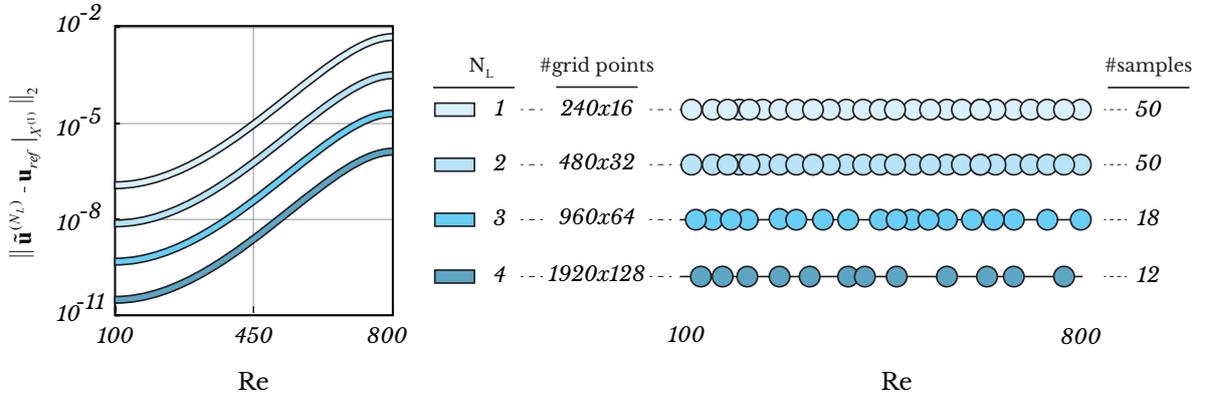


Figure 17: Navier-Stokes equations. (left) Error convergence for different number of total levels. (right) Sample placement on each level.

when compared to the previous test cases. This is due to the increase in degrees of freedom in the neural networks, as the 2D multi-channel inputs require 2D convolutional layers, instead of the 1D convolutional layers that were used in the previous test cases. Furthermore, the mappings are more complex when compared to the previous two test cases.

III.2. Neural Network Mappings $P^{(i)}$

The neural network mappings $P^{(i)}$, $i = 2, 3, 4$ are shown in figure 18. It is striking that, even for this complex non-linear test case, the neural network approximations still show very similar behaviour for consecutive levels. We stress once again that this is the property that is utilised by our proposed transfer-learning approach to reduce the total number of samples required.

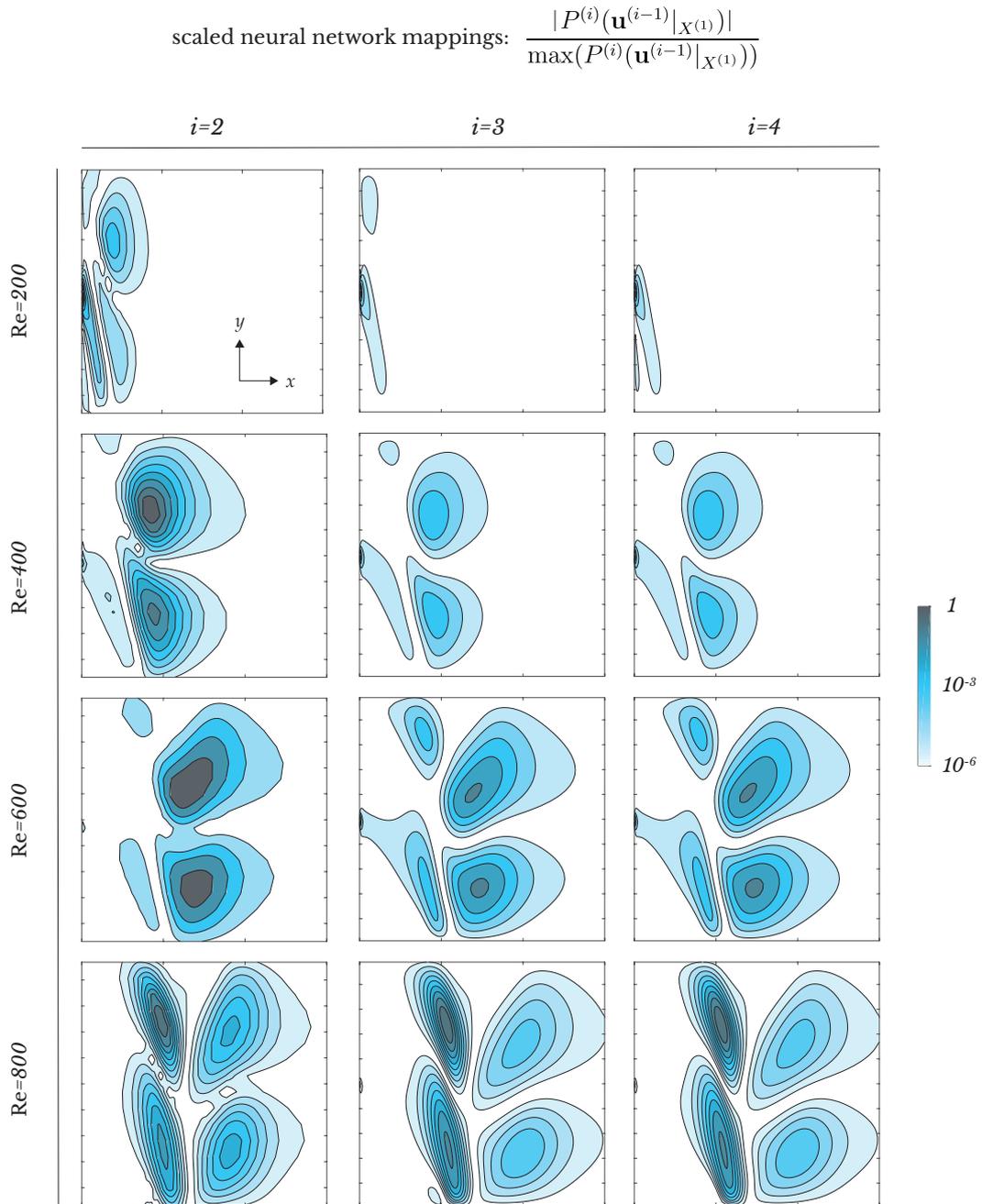


Figure 18: Navier-Stokes equations. The neural networks mappings for the Navier-Stokes test case in (x, y, Re) -space.

III.3. Comparison with MLSC

We compare the computational cost for MLNN and MLSC when constructing a surrogate with a certain accuracy. The computational cost is computed in the same way as described in section 6.I.4 and the results are shown in figure 19. Again, the MLNN method requires more computational time for the approximation on the first level when compared

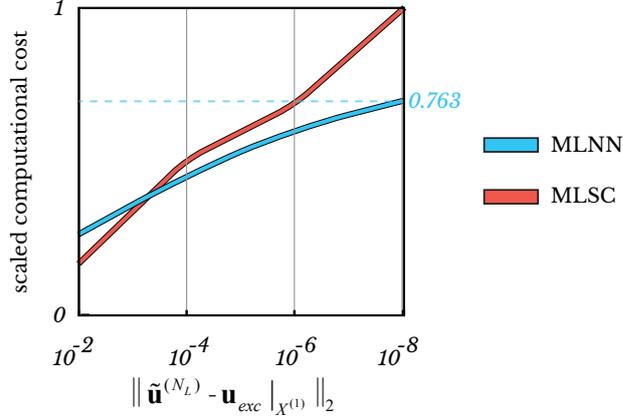


Figure 19: Navier-Stokes equations. The scaled computational cost when constructing a surrogate for a range of accuracies.

to MLSC. However, for this non-linear test case, the MLNN method shows an advantage over the MLSC approach, which is more pronounced than for the Burgers case.

IV. Surrogate Modelling for 3D Fluid Sloshing

In this section we investigate the MLNN method to construct a surrogate for unsteady fluid sloshing in a 3D rectangular tank with 2 uncertainties. These surrogates are perfectly suited for accurate uncertainty propagation. The governing equations are the 3D unsteady incompressible Navier-Stokes equations:

$$\nabla \cdot \mathbf{u} = 0, \quad (38a)$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\frac{\nabla p}{\rho} + \mathbf{g} + \nu \nabla^2 \mathbf{u}, \quad (38b)$$

where \mathbf{u} is again the velocity field, p the pressure, ρ the density, \mathbf{g} the gravitational body force, and ν the kinematic viscosity. In contrast to previous test cases, the solver is not implemented in dimensionless form as we followed the implementation described in [43].

The Solver

In this work we use a PIC/FLIP solver [44, 43] for predicting single-phase free-surface flows. PIC/FLIP uses elements of a particle-based method such as Smoothed Particle Hydrodynamics (SPH) [45] (kernel approximation of velocities) and a grid-based finite-volume method [46]. The main difference between pure particle-based methods such as SPH is that the particles in PIC/FLIP are passive and interactions take place on the grid. The positions of the particles that represent the fluid are evolved over time by using velocity values that are computed on a staggered grid.

Solver Initialisation

In our simulations we consider a rectangular shaped tank of dimensions $\mathbf{x} = (x, y, z) \in [0, 10] \times [0, 5] \times [0, 5]$. The computational cost of a PIC/FLIP solver is mainly due to the operations on the grid, as the operations performed for the particles are highly parallelisable. As a result, we can use a large amount of particles $N_p = 10^6$ in all simulations and randomly place these particles in the rectangular tank below $z = 2$ according to a uniform distribution. The grid resolution differs depending on the fidelity of the simulation, which is discussed hereafter. An example of a particle/grid initialisation is shown in figure 20.

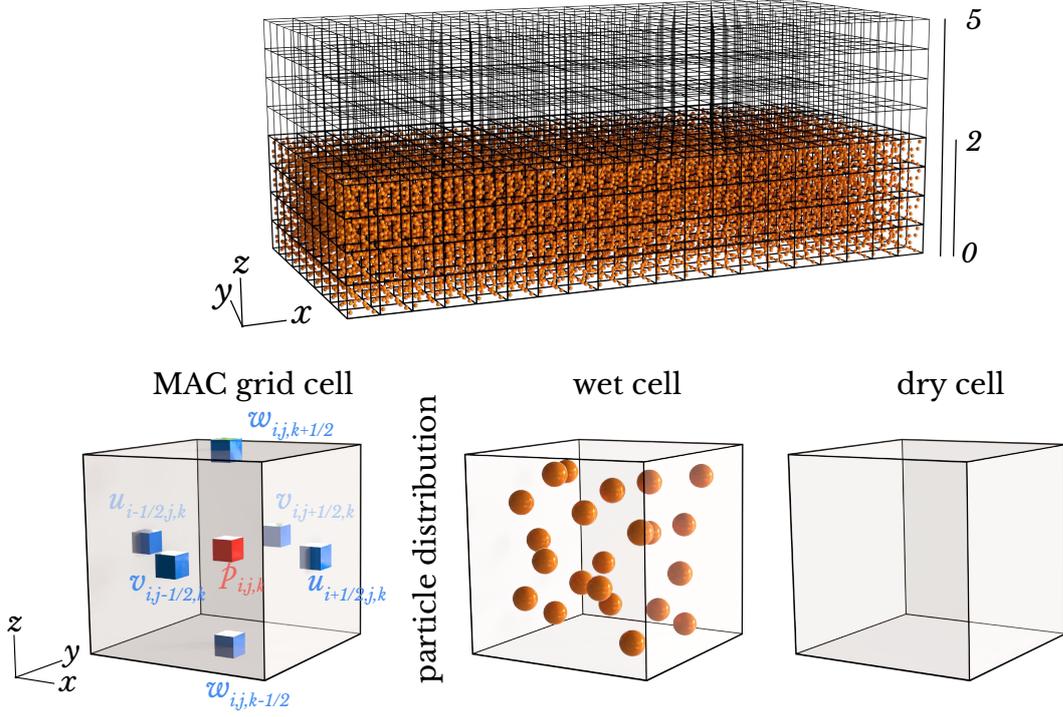


Figure 20: Example of solver initialisation. A 3D staggered (128, 64, 64) grid, which is partially filled with particles.

Uncertainties

To excite sloshing, the fluid in the partially filled tank needs forcing. In this test case we excite the fluid motion by rotating the gravitational body force. The gravity vector is defined as

$$\mathbf{g} = 9.81(\sin(\psi)\cos(\phi), \sin(\phi), -\cos(\psi)\cos(\phi)), \quad (39)$$

where the two angles ψ and ϕ represent rotation in the x, y -plane and y, z -plane, respectively. These angles are assumed to change during the simulation as a function of time, given by

$$\psi(t) = \begin{cases} A \sin(2\lambda t), & t < \frac{2\pi}{\lambda}, \\ 0, & t \geq \frac{2\pi}{\lambda}, \end{cases} \quad \phi(t) = \begin{cases} A \sin(\lambda t), & t < \frac{2\pi}{\lambda}, \\ 0, & t \geq \frac{2\pi}{\lambda}, \end{cases} \quad (40)$$

where the amplitude A and period λ of the oscillation are assumed to be uncertain, i.e., $\mathbf{z} = (A, \lambda)$ and are assumed to be uniformly distributed on the intervals $I_A = [\frac{\pi}{8}, \frac{3\pi}{8}]$ and $I_\lambda = [\frac{1}{2}, \frac{3}{2}]$, respectively. This particular motion leads to heavy sloshing inside the tank and is suitable for testing the applicability of the MLNN method for a highly irregular fluid motion.

The QoI

When performing PIC/FLIP simulations, the shape of the fluid surface is our main interest, which follows directly from the particle positions. As a result, the QoI is defined as the number of particles contained within each grid cell at a given time level, which is chosen to be $t = 3\pi$. At this time instant, the tank has the same orientation for all considered motion parameters, which allows us to compare the QoI.

Fidelities

As most of the computational expense in a PIC/FLIP simulation comes from the grid based computations, we define the fidelity as the grid resolution. The time-step is tuned accordingly to satisfy the stability condition with safety factor

0.8 [43]. The solutions on the first level are computed on a fixed grid of $32 \times 16 \times 16$ cells. We choose to increase the fidelity by increasing the number of grid cells in every spatial direction with a factor 2 for consecutive levels, i.e., $(N_x^{(i)}, N_y^{(i)}, N_z^{(i)}) = 2(N_x^{(i-1)}, N_y^{(i-1)}, N_z^{(i-1)})$. Obtaining a fully converged solution for this test case is difficult and we therefore consider as reference solution a solution that is computed on a fine $256 \times 128 \times 128$ grid, which corresponds to the 4th level. The QoI defined on the low-fidelity grid is given as input to the neural network as a 3D 1-channel convolutional input. An example of a low and a high-fidelity simulation result for $A = \frac{\pi}{4}$ and $\lambda = 1$ is shown in figure 21.

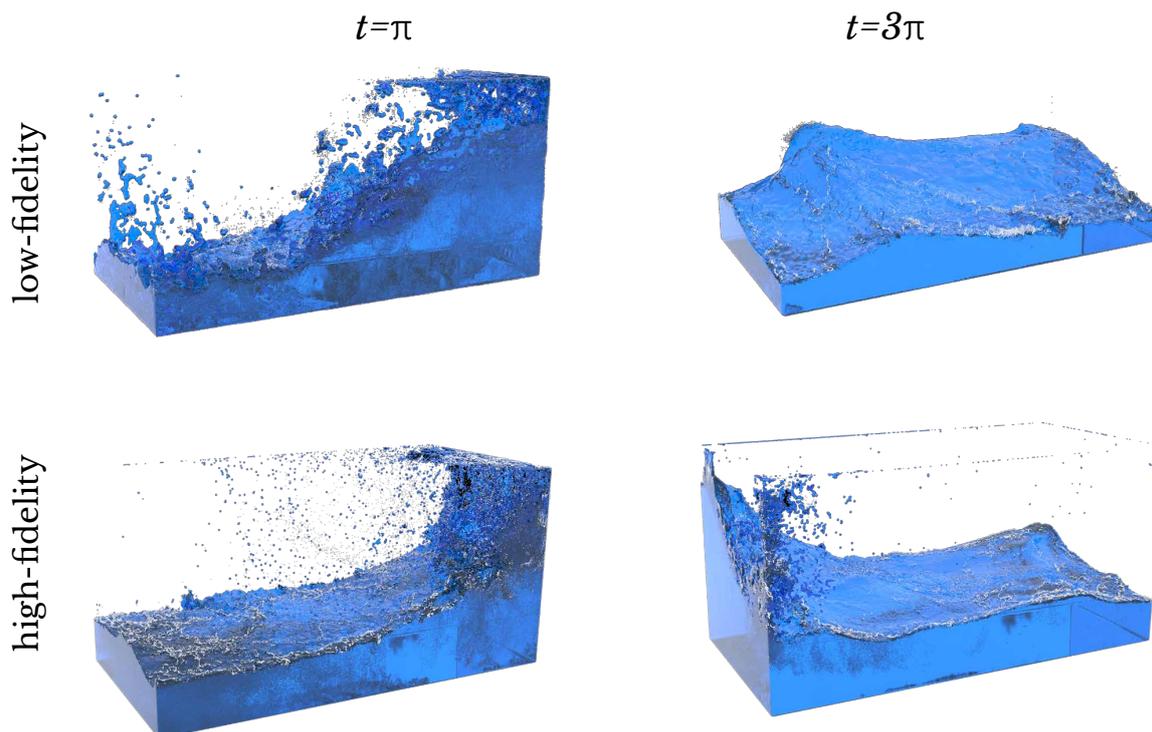


Figure 21: 3D Sloshing. Example simulation for $A = \frac{\pi}{4}$ and $\lambda = 1$. Isosurface generation is used to convert the particle positions to a fluid surface, which can then be rendered [45].

Parametric Solution

The number of particles in each of the low-fidelity grid cells, i.e., the QoI, is a function of the uncertain parameters A and λ . The MLNN method is used to construct a surrogate model of the QoI in the random space spanned by the uncertain parameters. The goal is to approximate the solution that is computed on the high-fidelity $256 \times 128 \times 128$ reference grid, which corresponds to the 4th level in the MLNN approach. Minimising the samples on the 4th level required for accurate surrogate construction is paramount for the feasibility of the MLNN approach. As before, the tolerance for the training procedure is set to $\varepsilon = 10^{-6}$ and an accuracy tolerance $\varepsilon_{\text{acc}} = 10^{-4}$ is used. Errors are computed using a reference surrogate that is constructed using high-fidelity simulations (on level 4) with a 25×25 Gauss-Legendre grid [9] and the L_2 -norm is taken over the solutions in the entire parameter space. The resulting error convergence for the QoI is shown in figure 22.

For these highly non-linear fluid motions, the surrogate construction is challenging, which can be noticed in the required number of samples on each level. The figure shows that constructing the surrogate requires significantly more samples on each level when compared to previous test cases, which is caused by having two uncertainties and

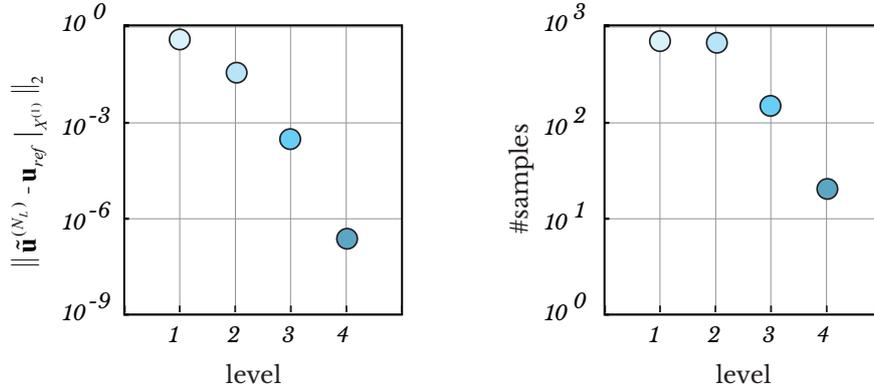


Figure 22: 3D Sloshing. Error convergence of the surrogate for the QoI constructed with the MLNN method.

the complexity of the test case. However, there is still a significant decrease in the required number of samples when increasing the level, which shows the applicability of our transfer learning approach also for this complex test case.

Comparison with MLSC

The MLNN method is again compared with the MLSC method and the results are summarised in figure 23.

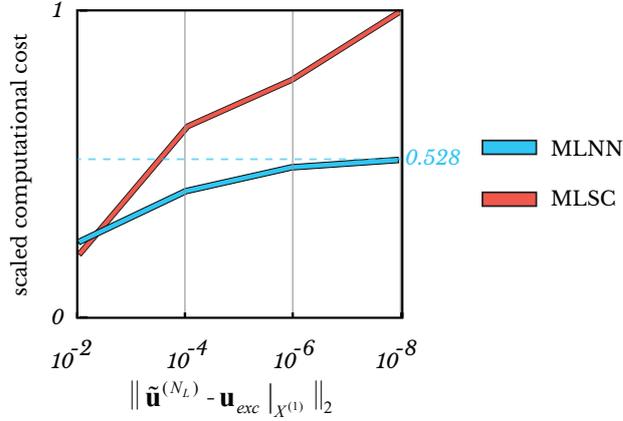


Figure 23: 3D Sloshing. Comparison with MLSC.

Our approach requires slightly more computational time for the approximation on the first level when compared to MLSC. On second and higher levels, our method shows a significant gain in computational efficiency when compared to the MLSC method, which is more pronounced than in previous test cases. In this case, the MLNN method has a clear advantage when used for surrogate construction with medium to high level of accuracy.

7. Conclusion

In this paper we have presented a novel method for surrogate construction. The method is based on a multi-level expansion of the solution and constructs approximations of the relative global discretisation error on different levels to enhance low-fidelity solutions. Inspired by the idea that these errors can be expressed in terms of the solution, the approximations of the error on each level are constructed using convolutional neural networks that apply a non-linear

mapping of the solution values to the difference between the solutions computed on the corresponding level and the subsequent level. Transfer learning reduces the amount of training samples that are required to properly train the neural networks.

The MLNN method has been employed for surrogate construction for several parametric partial differential equations: steady 1D advection diffusion equation, steady 1D Burgers equation, steady 2D incompressible Navier-Stokes equations, and unsteady 3D incompressible Navier-Stokes. We justified the use of transfer learning for these specific test cases by studying the neural network mappings. We expect the applicability of transfer learning to generalise to other types of differential equations, provided that they possess similar smoothness properties, which is necessary for the error behaviour to be similar at different levels. In all test cases, fast convergence is obtained, leading to an accurate surrogate model already at a relatively low number of model runs. We compared our approach to MLSC. The transfer learning reduces the number of samples on the higher levels, and therefore significantly decreases the computational cost when medium/high fidelity samples are expensive to sample or when a surrogate is wanted with a high accuracy, which effectively increases the number of required levels. The conclusion is that our method outperforms MLSC when either the PDEs are sufficiently complex, or when a highly accurate surrogate model is required. For example, for the complex test case of a liquid sloshing in a tank, our approach leads to a computational cost savings of a factor of 2 compared to MLSC, when requiring medium to high accuracy. The resulting surrogate model can be directly used as a computationally inexpensive tool for uncertainty quantification.

Furthermore, the method can be applied to unsteady problems, but is not optimal for it in its current form. Either the neural networks have to be retrained for different time instances, or the temporal component should be added as an extra dimension to the convolutional layers and the output of the neural networks. Furthermore, it is not shown how the method scales for high-dimensional random spaces. Optimising the method proposed here for unsteady problems and high-dimensional random spaces is scheduled for future work.

Acknowledgements

This work is part of the research programme "SLING" (Sloshing of Liquefied Natural Gas), which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

References

- [1] H. Xiao, J. L. Wu, J. X. Wang, R. Sun, C. J. Roy, Quantifying and reducing model-form uncertainties in Reynolds-averaged Navier-Stokes simulations: a data-driven, physics-informed Bayesian approach, *Journal of Computational Physics* 324 (2016) 115–136. doi:10.1016/j.jcp.2016.07.038.
URL <http://www.sciencedirect.com/science/article/pii/S0021999116303394>
- [2] W. N. Edeling, R. P. Dwight, P. Cinnella, Simplex-stochastic collocation method with improved scalability, *Journal of Computational Physics* 310 (2016) 301–328. doi:10.1016/j.jcp.2015.12.034.
URL <http://www.sciencedirect.com/science/article/pii/S0021999115008530>
- [3] F. Simon, P. Guillen, P. Sagaut, D. Lucor, A gPC-based approach to uncertain transonic aerodynamics, *Computer Methods in Applied Mechanics and Engineering* 199 (2010) 1091–1099. doi:10.1016/j.cma.2009.11.021.
URL <http://www.sciencedirect.com/science/article/pii/S004578250900396X>
- [4] K.-H. Cho, S.-Y. Shin, W. Kolch, O. Wolkenhauer, Experimental design in systems biology, based on parameter sensitivity analysis using a Monte Carlo method: a case study for the TNF-mediated NF- κ B signal transduction pathway, *Simulation* 79 (2003) 726–739. doi:10.1177/0037549703040943.
URL <http://journals.sagepub.com/doi/abs/10.1177/0037549703040943>
- [5] R. Abagyan, M. Totrov, Biased probability Monte Carlo conformational searches and electrostatic calculations for peptides and proteins, *Journal of Molecular Biology* 235 (3) (1994) 983–1002. doi:10.1006/jmbi.1994.1052.
URL <http://www.sciencedirect.com/science/article/pii/S0022283684710527>
- [6] E. M. Constantinescu, V. M. Zavala, M. Rocklin, S. Lee, M. Anitescu, A computational framework for uncertainty quantification and stochastic optimization in unit commitment with wind power generation, *IEEE Transactions on Power Systems* 26 (1) (2011) 431–441. doi:10.1109/TPWRS.2010.2048133.
- [7] J. Mateos, T. Gonzalez, D. Pardo, V. Hoel, A. Cappy, Monte Carlo simulator for the design optimization of low-noise HEMTs, *IEEE Transactions on Electron Devices* 47 (10) (2000) 1950–1956. doi:10.1109/16.870579.
- [8] M. Papadrakakis, N. D. Lagaros, Reliability-based structural optimization using neural networks and Monte Carlo simulation, *Computer Methods in Applied Mechanics and Engineering* 191 (32) (2002) 3491–3507. doi:10.1016/S0045-7825(02)00287-6.
URL <http://www.sciencedirect.com/science/article/pii/S0045782502002876>

- [9] D. Xiu, *Numerical Methods for Stochastic Computations: A Spectral Method Approach*, Princeton University Press, 2010.
- [10] B. Peherstorfer, K. Willcox, M. Gunzburger, Survey of multifidelity methods in uncertainty propagation, inference, and optimization, *SIAM Review* 60 (3) (2018) 550–591.
- [11] X. Zhu, E. M. Linebarger, D. Xiu, Multi-fidelity stochastic collocation method for computation of statistical moments, *Journal of Computational Physics* 341 (2017) 386–396. doi:10.1016/j.jcp.2017.04.022.
URL <http://www.sciencedirect.com/science/article/pii/S0021999117302930>
- [12] A. Forrester, A. Sobester, A. Keane, Multi-fidelity optimization via surrogate modelling, *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 463 (2088) (2007) 3251–3269. doi:10.1098/rspa.2007.1900.
URL <https://royalsocietypublishing.org/doi/full/10.1098/rspa.2007.1900>
- [13] A.-L. Haji-Ali, F. Nobile, L. Tamellini, R. Tempone, Multi-Index Stochastic Collocation for random PDEs, *Computer Methods in Applied Mechanics and Engineering* 306 (2016) 95–122. doi:10.1016/j.cma.2016.03.029.
URL <http://www.sciencedirect.com/science/article/pii/S0045782516301141>
- [14] A. Narayan, C. Gittelsohn, D. Xiu, A stochastic collocation algorithm with multifidelity models, *SIAM Journal on Scientific Computing* 36 (2) (2014) A495–A521.
- [15] X. Zhu, A. Narayan, D. Xiu, Computational aspects of stochastic collocation with multifidelity models, *SIAM/ASA Journal on Uncertainty Quantification* 2 (1) (2014) 444–463.
- [16] K. Cliffe, M. Giles, R. Scheichl, A. Teckentrup, Multilevel Monte Carlo methods and applications to elliptic PDEs with random coefficients, *Computing and Visualization in Science* 14 (3) (2010) 444–463.
- [17] A. Teckentrup, *Multilevel Monte Carlo Methods and Uncertainty Quantification*, PhD Thesis at University of Bath, 2013.
URL <https://books.google.nl/books?id=RunSnQEACAAJ>
- [18] A. Teckentrup, P. Jantsch, C. Webster, M. Gunzburger, A multilevel stochastic collocation method for partial differential equations with random input data, *SIAM/ASA Journal on Uncertainty Quantification* 3 (1) (2015) 1046–1074. doi:10.1137/140969002.
URL <https://epubs.siam.org/doi/abs/10.1137/140969002>
- [19] D. Kouri, A Multilevel Stochastic Collocation Algorithm for Optimization of PDEs with Uncertain Coefficients, *SIAM/ASA Journal on Uncertainty Quantification* 2 (1) (2014) 55–81. doi:10.1137/130915960.
URL <https://epubs.siam.org/doi/abs/10.1137/130915960>
- [20] J. Charrier, R. Scheichl, A. Teckentrup, Finite Element Error Analysis of Elliptic PDEs with Random Coefficients and Its Application to Multilevel Monte Carlo Methods, *SIAM Journal on Numerical Analysis* 51 (1) (2013) 322–352. doi:10.1137/110853054.
URL <https://epubs.siam.org/doi/abs/10.1137/110853054>
- [21] H. Harbrecht, M. Peters, M. Siebenmorgen, On Multilevel Quadrature for Elliptic Stochastic Partial Differential Equations, *Lecture Notes in Computational Science and Engineering* 88 (2013) 161–179. doi:10.1007/978-3-642-31703-3-8.
- [22] I.-G. Farcas, P. C. Sărbu, H.-J. Bungartz, T. Neckel, B. Uekermann, Multilevel adaptive stochastic collocation with dimensionality reduction, in: J. Garcke, D. Pflüger, C. G. Webster, G. Zhang (Eds.), *Sparse Grids and Applications - Miami 2016*, Springer International Publishing, Cham, 2018, pp. 43–68.
- [23] J. Lang, R. Scheichl, D. Silvester, A fully adaptive multilevel stochastic collocation strategy for solving elliptic PDEs with random data (2019). arXiv:1902.03409.
- [24] M. B. Giles, Multilevel Monte Carlo path simulation, *Operations Research* 56 (3) (2008) 607–617. arXiv:<https://doi.org/10.1287/opre.1070.0496>, doi:10.1287/opre.1070.0496.
URL <https://doi.org/10.1287/opre.1070.0496>
- [25] M. B. Giles, Multilevel Monte Carlo methods, *Acta Numerica* 24 (2015) 259–328.
- [26] K. Hornik, Approximation capabilities of multilayer feedforward networks, *Neural Networks* 4 (2) (1991) 251–257. doi:[https://doi.org/10.1016/0893-6080\(91\)90009-T](https://doi.org/10.1016/0893-6080(91)90009-T).
URL <http://www.sciencedirect.com/science/article/pii/089360809190009T>
- [27] G. Lewicki, G. Marino, Approximation by superpositions of a sigmoidal function, *Applied Mathematics Letters* 17 (2004) 1147–1152. doi:10.1016/j.aml.2003.11.006.
- [28] B. Hanin, M. Sellke, Approximating continuous functions by ReLU nets of minimal width (2017). arXiv:1710.11278.
- [29] G. Marchuk, V. Shaidurov, *Difference Methods and Their Extrapolations*, Stochastic Modelling and Applied Probability, Springer New York, 1983.
- [30] C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*, Springer, 2018.
- [31] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.
- [32] P. Robbe, D. Nuyens, S. Vandewalle, Recycling samples in the multigrid multilevel (quasi-)Monte Carlo method, *SIAM Journal on Scientific Computing* 41 (5) (2019) S37–S60.
- [33] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, arXiv:1711.10561 [cs, math, stat]ArXiv: 1711.10561.
URL <http://arxiv.org/abs/1711.10561>
- [34] B. Kim, V. C. Azevedo, N. Thuerey, T. Kim, M. Gross, B. Solenthaler, Deep fluids: A generative network for parameterized fluid simulations, arXiv:1806.02071 [physics, stat]ArXiv: 1806.02071.
URL <http://arxiv.org/abs/1806.02071>
- [35] L. Ladick, S. Jeong, N. Bartolovic, M. Pollefeys, M. Gross, Physicsforests: real-time fluid simulation using machine learning, in: *ACM SIGGRAPH 2017*, 2017, pp. 22–22. doi:10.1145/3098333.3098337.
- [36] J. Schmidhuber, Deep learning in neural networks: An overview, *Neural Networks* 61 (2015) 85–117.
- [37] J. Yosinski, J. Clune, Y. Bengio, H. Lipson, How transferable are features in deep neural networks?, in: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14*, MIT Press, Cambridge, MA, USA, 2014, pp. 3320–3328.
URL <http://dl.acm.org/citation.cfm?id=2969033.2969197>
- [38] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga,

- S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: A system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265–283.
URL <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>
- [39] E. Bertolazzi, G. Manzini, A second-order maximum principle preserving finite volume method for steady convection-diffusion problems, *SIAM Journal on Numerical Analysis* 43 (2005) 2172–2199. doi:10.1137/040607071.
- [40] H. Dret, B. Lucquin, *The Variational Formulation of Elliptic PDEs*, Partial Differential Equations: Modeling, Analysis and Numerical Approximation, Springer International Publishing, 2016, pp. 117–143.
- [41] B. Sanderse, ECNS: Energy-Conserving Navier-Stokes solver verification of steady laminar flows, ECN, 2011.
- [42] D. K. Gartling, A test problem for outflow boundary conditions flow over a backward-facing step, *International Journal for Numerical Methods in Fluids* 11 (7) (1990) 953–967. arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/flid.1650110704>, doi:10.1002/flid.1650110704.
URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/flid.1650110704>
- [43] R. Bridson, *Fluid Simulation for Computer Graphics*, CRC Press, 2015.
- [44] Y. Zhu, R. Bridson, Animating sand as a fluid, in: *ACM SIGGRAPH 2005, SIGGRAPH '05*, ACM, New York, NY, USA, 2005, pp. 965–972, event-place: Los Angeles, California. doi:10.1145/1186822.1073298.
URL <http://doi.acm.org/10.1145/1186822.1073298>
- [45] A. J. C. Crespo, J. M. Dominguez, B. D. Rogers, M. Gomez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro, O. Garcia-Feal, DualSPHysics: Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH), *Computer Physics Communications* 187 (Supplement C) (2015) 204–216. doi:10.1016/j.cpc.2014.10.004.
URL <http://www.sciencedirect.com/science/article/pii/S0010465514003397>
- [46] C. Hirsch, *Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics*, Elsevier, 2007.