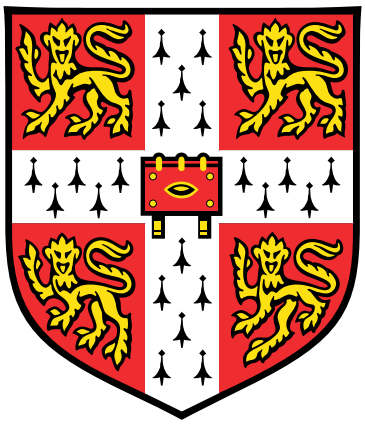
Exascale computing and deep learning-based reduced order modelling of parametric partial differential equations for engineering design

Final Report



Jan Jakub Derlatka

Supervisor: Prof. Garth N. Wells
Advisor: Dr Nirav V. Shah

St John's College

May 2024

Declaration

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work.

Jan Jakub Derlatka
May 2024

Exascale computing and deep learning-based reduced order modelling of parametric partial differential equations for engineering design

Final Report

Jan Jakub Derlatka
St. John's College

Supervisor: Prof. Garth N. Wells
Advisor: Dr Nirav V. Shah

Abstract

Dictating the tumultuous shape of the rough sea in Hokusai's "The Great Wave off Kanagawa", and efficient options prices in financial markets, partial differential equations (PDEs) are both ubiquitous and of great importance. They enable mathematical modelling of physical, or non-physical systems. Through parametric PDEs, we can formulate problems with variable configurations, where material properties, boundary conditions, and geometry can be defined in terms of parameters. Numerical schemes solving these problems remain computationally expensive, making them infeasible for real-time or multi-query applications.

To address this problem, model order reduction (MOR) methods seek to find a lower dimension approximation of a given PDE system, still capturing the key characteristics. Reduced basis (RB) methods are the MOR technique used to explore the solution manifold constructed by large-scale parametric PDEs under the variation of parameters. In this project, we consider Proper Orthogonal Decomposition (POD) for the construction of reduced basis space and Artificial Neural Network (ANN) for the reduced degrees of freedom computation at any given parameter (POD-ANN).

Given a set of solutions, POD identifies functions capturing the most variance in the solutions, and uses them to construct a RB for the solutions manifold. It can be thought of as a functional space equivalent of Principal Component Analysis.

ANNs are computational models consisting of interconnected nodes, which process information analogously to biological neural networks. In supervised learning, such a network can be trained by adjusting connections based on a large set of expected input – output pairs, in this case: parameters set – true solution's RB coefficients. This enables ANNs to predict the output for unseen inputs.

POD-ANN method works in two phases: offline, and online. During the offline phase, computationally expensive operations are performed, with access to high-performance computing resources, if required. The full-order model is used to calculate "snapshot" full-order model solutions, corresponding to a sampled set of parameter values. They are treated as "ground truth", used to construct the reduced basis. Next, the ANN is trained to map parameters to reduced basis coefficients of corresponding solutions. In the online phase, a solution for a new choice of parameters is obtained by querying the ANN.

The online stage is intended to be performed repeatedly for each new parameter with limited computational resources and in real time. On the other hand, the offline stage requires large computational effort both to generate the full-order model snapshots and to train the ANN. Usually, such computations would be carried out with large-scale parallel high-performance computing (HPC) systems.

Throughout this project the POD-ANN method was successfully implemented, and investigated on a lid driven cavity problem. In the course of doing that, harmonic mesh deformation was used to find FEM solutions to steady Navier-Stokes equations with geometric parameters. The accuracy of solutions obtained by our POD-ANN implementation approached the levels documented in the original paper [30]. Additionally, the implementation was successfully parallelised enabling it to work across any number of distributed processes.

Each of these achievements was made possible through the integration of various open-source libraries. The implementation is portable and self-contained, allowing anyone to experiment with this method. Access details are provided in appendix D. It is hoped that this work will serve as a useful reference in future related research.

We found that reusing the same dataset for POD and ANN training has a detrimental effect on overall performance. However, this drawback is offset by the positive effect of being able to calculate twice as many samples with same resources, when the dataset is reused. A comparison between a single network and two networks model revealed that two networks achieve better performance for the same number of trainable model parameters. The assumption that two hidden layers are sufficient to model the problem turned out to be correct. We confirmed that the POD-ANN offline phase can be efficiently distributed, and showed results of it, although it is hoped that these experiments ran on a more appropriate setup would substantiate the claim more. Finally, we implemented a full order solver for the time dependent DFG 2D-3 benchmark. In doing that, we showed the mesh degeneration problem of harmonic deformation, and implemented staged mesh deformation successfully addressing it.

The report begins by formally defining parametrised PDEs and introducing the challenges of geometric parametrisation in Chapter 1. Chapter 2 briefly summarises FEM, which is used to generate the dataset of solutions necessary to train our model. We then introduce Navier-Stokes equations in Chapter 3, which are a problem commonly addressed with numerical methods such as FEM, and used as both case studies of this project. Chapter 4 provides a more formal statement of model order reduction, and introduces POD, key element of the method discussed. The theoretical section is concluded in Chapter 5, by the a summary of the other element instrumental to POD-ANN – neural networks. Major part of the report is devoted to numerical results, all gathered in Chapter 7. These include a high level overview of the implementation, lists tools used, presents decisions that were faced during the project, and presents results of experiments. Chapter 8 concludes the work.

Introduction

Dictating the tumultuous shape of the rough sea in Hokusai's "The Great Wave off Kanagawa", and efficient options prices in financial markets, partial differential equations (PDEs) are both ubiquitous and of great importance. They enable mathematical modelling of physical, or non-physical systems. Through parametric PDEs, we can formulate problems with variable configurations, where material properties, boundary conditions, and geometry can be defined in terms of parameters.

In most practical scenarios, no analytical solutions are known, and only numerical approximations of the solution are feasible through methods such as finite difference, finite volume, finite element (FEM), or spectral [42]. However, these schemes remain computationally expensive, requiring days to complete for large problems, and whenever a parameter value is changed, the entire solution needs to be recalculated. This makes them infeasible for real-time or multi-query applications, such as engineering design optimisation problems, where the objective function or constraints are defined in terms of a PDE [30].

To address this problem, model order reduction (MOR) methods seek to find a lower dimension approximation of a given PDE system, still capturing the key characteristics. Reduced basis (RB) methods are the MOR technique used to explore the solution manifold constructed by large-scale parametric PDEs under the variation of parameters. In this project, we consider Proper Orthogonal Decomposition (POD) for the construction of reduced basis space and Artificial Neural Network (ANN) for the reduced degrees of freedom computation at any given parameter (POD-ANN) [7, 9, 30, 50].

Given a set of solutions, POD identifies functions capturing the most variance in the solutions, and uses them to construct a RB for the solutions manifold. It is possible to show, that approximation made by truncating the basis to d_{rb} first such functions results in the smallest squared reconstruction error [9]. It can be thought of as a functional space equivalent of Principal Component Analysis.

ANNs are computational models consisting of interconnected nodes, which process information analogously to biological neural networks. In supervised learning, such a network can be trained by adjusting connections based on a large set of expected input – output pairs, in this case: parameters set – true solution's RB coefficients. This enables ANNs to predict the output for unseen inputs [33].

POD-ANN method works in two phases: offline, and online. During the offline phase, computationally expensive operations are performed, with access to high-performance computing resources, if required. The full-order model is used to calculate "snapshot" full-order model solutions, corresponding to a sampled set of parameter values. They are treated as "ground truth", used to construct the reduced basis. Next, the ANN is trained to map parameters to reduced basis coefficients of corresponding solutions. In the online phase, a solution for a new choice of parameters is obtained by querying the ANN.

The online stage is intended to be performed repeatedly for each new parameter with limited computational resources and in real time. On the other hand, the offline stage requires large computational effort both to generate the full-order model snapshots and to train the ANN. Usually, such computations would be carried out with large-scale parallel high-performance computing (HPC) systems. These systems are continually advancing, with the most powerful at the time of writing being the USA's Frontier, the first exascale

system – capable of performing at least 10^{18} floating-point operations per second, or 1 exaFLOP [39]. Efficient use of such systems requires careful implementation, particularly for the POD-ANN MOR, as traditionally FEM solvers have been CPU-centric, in contrast to the GPU-favoured ANN training processes.

This project investigates the POD-ANN method through a geometrically parametrised incompressible Navier-Stokes problem. We experiment with using harmonic mesh deformation to handle geometric parameters, and explore the performance of various configurations of the neural network. Implementation of the problem enables distribution of computations involved in the offline stage, and the effects of that distribution are studied.

The report begins by formally defining parametrised PDEs and introducing the challenges of geometric parametrisation in Chapter 1. Chapter 2 briefly summarises FEM, which is used to generate the dataset of solutions necessary to train our model. We then introduce Navier-Stokes equations in Chapter 3, which are a problem commonly addressed with numerical methods such as FEM, and used as both case studies of this project. Chapter 4 provides a more formal statement of model order reduction, and introduces POD, key element of the method discussed. The theoretical section is concluded in Chapter 5, by the a summary of the other element instrumental to POD-ANN – neural networks. Major part of the report is devoted to numerical results, all gathered in Chapter 7. These include a high level overview of the implementation, lists tools used, presents decisions that were faced during the project, and presents results of experiments. Chapter 8 concludes the work.

1 Parametric partial differential equations

A parametric PDE can be used to describe a range of setups for a particular problem. Let us take the flow of a fluid in a bath tub as an example. The computational domain (shape of the bath tub) and the source terms (inflow and outflow rates) can both be changed. With some creativity, one could imagine boundary conditions being varied (wind blowing over the fluid's surface). Finally, the material parameters (density, viscosity) would change if, instead of water, the study investigates ancient Egypt's milk baths, or famous Czech beer baths.

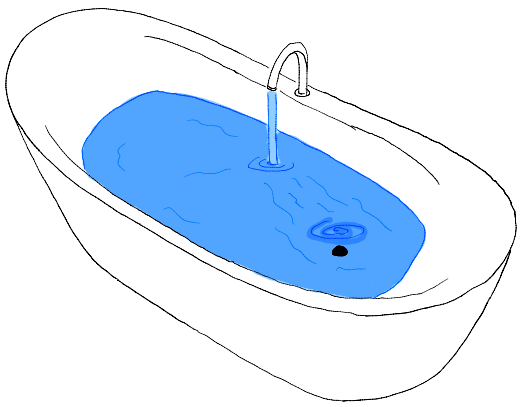


Figure 1: Fluid flow in a bath tub can be an example of a parametric PDE.

Parameters can come from different sets, continuous, or discrete. Denoting the set of values which the i-th parameter can take as $\mathbb{P}_i$, the set of all possible parameters

configurations is defined as $\mathbb{P} = \mathbb{P}_1 \times \mathbb{P}_2 \times ... \times \mathbb{P}_{d_\mu}$, where d_μ is the number of parameters. A point $\mu \in \mathbb{P}$ is a vector of values for all parameters, fully characterising the PDE.

Let us denote a general PDE parameterised by μ, as a differential operator $G(\mu)$. The d-dimensional domain of the PDE, is also a function of parameters, denoted by $\Omega(\mu) \subset \mathbb{R}^d$, with boundary $\partial\Omega(\mu)$. For a unique solution, we require boundary conditions, which can be imposed either on function values u, called "Dirichlet" boundary conditions, on $\Gamma_D(\mu)$, or its derivative normal to the boundary $\frac{\partial u}{\partial n}$, called "Neumann" boundary conditions, on $\Gamma_N(\mu)$,

$$\Gamma_D(\mu) \cup \Gamma_N(\mu) = \partial\Omega(\mu), \quad \Gamma_D(\mu) \cap \Gamma_N(\mu) = \emptyset$$

If G is of order s, we say that a function $u(\mu)$ belonging to a problem specific, sufficiently well behaved space U, is a solution of the PDE for parameters μ if it satisfies

$$\begin{aligned} G(\mu)(u(\mu)) &= f(\mu) && \text{in } \Omega(\mu) \\ u(\mu) &= u_0(\mu) && \text{on } \Gamma_D(\mu) \\ \frac{\partial u}{\partial n}(\mu) &= h(\mu) && \text{on } \Gamma_N(\mu) \end{aligned}$$

where $f(\mu)$, $u_0(\mu)$ and $h(\mu)$ are the source term, Dirichlet boundary condition and Neumann boundary condition respectively, all belonging to suitable problem dependent function spaces. A more in depth discussion of function spaces involved can be found in [18]. We call this the "strong formulation" of the PDE.

1.1 Geometric parameters

We distinguish parameters influencing the computational domain as "geometrical" parameters μ_{g}. Remaining parameters are referred to as "physical" ones μ_{ph}, so the parameters vector can be expressed as $\mu = [\mu_{\text{ph}}, \mu_{\text{g}}]$.

Physical parameters can often be turn into variables, and solved for. The distinguishment is put in place, because treatment of geometrical parameters turns out to be much more involved. They affect not only the solution, but also the function space it belongs to.

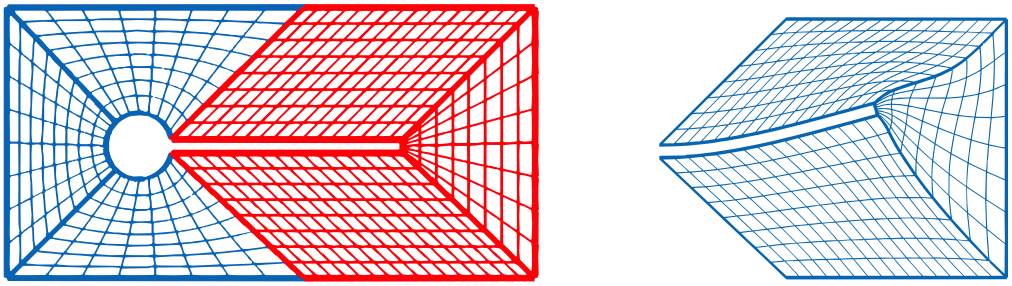
Having to work with different function spaces introduces additional challenges. Notably, some methods for solving PDEs, including the one discussed in this report, require a notion of similarity between two functions, quantified by an inner product, formally defined in appendix A.1. For two functions $f : X \to Y$, $g : X \to Y$ this is commonly achieved with L^2 norm: $\langle f, g \rangle_{L^2} = \int_X fg\mathrm{d}x$, which relies on the functions having same domain X.

One solution is to have a "reference" domain, Ω_{ref}, and map any parameterised domain, to this reference domain with a mapping $\Psi(\mu) : \Omega(\mu) \to \Omega_{\text{ref}}$. This project uses "harmonic extension" mapping [51]. This method can be interpreted by imagining having a rubber sheet, which represents the reference domain – a simple, flat surface like a rectangle or a square. We want to transform this rubber sheet into different shapes, which represent the deformed domains. If we pin the edges of this rubber sheet to a board at the original boundary points, and then re-pin them where we want the new boundaries to be, the surface of the rubber sheet between these pins stretches or compresses to fill the space between the new boundary positions. If a grid is drawn on the rubber sheet before transformation, the distortion of the grid would illustrate the deformation of domain interior. We find the unknown displacement field u_a, given the interface displacement u_s

on boundary Γ_D^{g} by solving

$$\begin{aligned} \Delta u_a &= 0 && \text{in } \Omega_{\mathrm{ref}} \\ u_a &= u_s && \text{on } \Gamma_D^{\mathrm{g}} \\ \Delta u_a &= 0 && \text{on } \partial\Omega_{\mathrm{ref}} \setminus \Gamma_D^{\mathrm{g}} \end{aligned}$$

Note, that the Dirichlet boundary Γ_D^{g} used to define this transformation is, in general, different from the problem's Γ_D.



(a) Reference mesh with patches subject to deformation indicated in red.

(b) Deformed mesh

Figure 2: Mesh deformation illustration from [51].

This transformation has to be smooth, and imposes limitations on what shapes can be achieved through the deformation. Because finding it involves solving a separate, deformation PDE, it has a computational cost associated with it, which could make up a non-negligible part of the time taken to solve the original PDE.

Other choices of mappings include bi-harmonic extension, linear elasticity and non-linear elasticity, which are capable of mapping between shapes of increasing complexity, but are also increasingly computationally expensive [51].

2 Finite Elements Method

2.1 Weak form

The weak formulation of a PDE allows it to be solved when the solution lacks sufficient smoothness for classical derivative to be defined, and forms a basis for the discretization in FEM.

The weak formulation of a PDE is obtained by multiplying the strong form by a test function v from a separate function space V, integrating the resulting expression over the domain, and applying integration by parts to move derivatives from the solution u to the test function v, relaxing the differentiability requirements on the solution u. This process can also involve incorporating boundary conditions into the formulation. We often require v to be zero outside of the interior of $\Omega(\mu)$ (to have "compact support"), to make the boundary terms vanish. This leads to a weak form of the PDE, denoted as

$$g(\mu)(u(\mu); v) = F(\mu)(v), \quad \forall\ v \in V \tag{1}$$

potentially nonlinear in u, but linear in v. $u(\mu)$ can now belong to a wider space U', such that $U \subset U'$. However, the less constrained U' is, the more constrained the test functions

space V becomes. If V is too narrow, it may restrict the types of numerical methods or the flexibility of the discretization schemes available. If $u(\mu)$ is a solution to the strong form of the PDE, then it also satisfies its weak form. We will refer to $u(\mu)$ as the “exact solution".

2.2 Discretisation

Let us consider V_h, a closed finite-dimensional subset of the continuous space V. Rather than attempting to find solutions in the infinite-dimensional space V, we focus on finding approximations within V_h, a more practical, finite-dimensional space. As we incrementally expand V_h, the accuracy of our approximations is expected to improve. Given $V_h \subset V$ the “Galerkin approximation" of (1) is to find $u_h \in V_h$ such that

$$g(\mu)(u_h(\mu); v_h) = F(\mu)(v_h), \quad \forall\ v_h \in V_h \tag{2}$$

with operators remaining unchanged. Following the convention of [29], we will refer to $u_h(\mu)$ as “truth".

It is also possible to choose a finite-dimensional space V_h which is not a subset of the space for continuous problem V. Such methods are called nonconforming and are not discussed in this report.

A Galerkin approximation in which the discrete space V_h is constructed by discretizing the domain Ω and employing piecewise polynomials as basis functions, is called finite elements method. [18]

2.3 Meshing

For next steps, a discretization of the computational domain is required. We choose elementary geometric shapes like triangles and quadrilaterals in 2D, or tetrahedra in 3D. The domain divided into n_{el} “cells" is called “grid" or “mesh", denoted by

$$\mathcal{T}(\mu) = \bigcup_{k=1}^{n_{\text{el}}} \tau_k, \qquad \forall\ i \neq j : \text{int}(\tau_i) \cap \text{int}(\tau_j) = \emptyset$$

This process, often called meshing, has great impact on how good the FEM solution will

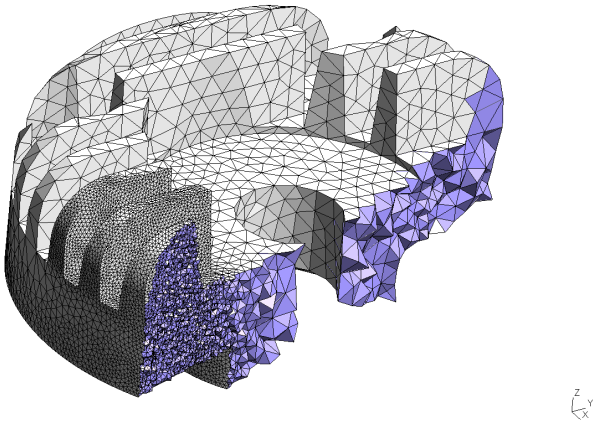


Figure 3: Example of a mesh obtained with Gmsh [26]

be. A finer mesh brings V_h closer to V, resulting in more accurate solution, but requiring more time and memory to solve. Therefore, to make optimal use of computational resources, instead of creating a mesh of uniform cells, one may try to create a finer mesh only in more sensitive areas. These may be areas near geometric irregularities, or regions where the function changes rapidly. The granularity can be linked to the geometry relatively simply, as the geometry is known when discretization begins. The behaviour of solution, however, is not known a priori. This can be addressed by updating the mesh as the solution is being calculated, in approach called adaptive meshing. [40] offers a more extensive discussion of meshing.

In the approach presented in this report, we create a mesh for the reference domain, and all other domains are obtained by applying geometrical deformation $\Psi(\mu)$ to the reference mesh

$$\mathcal{T}(\mu) = \Psi\left(\mathcal{T}(\mu_{\text{ref}})\right)$$

ensuring correspondence between cells.

2.4 Elements

We now construct the discrete function space by defining its basis functions $V_h = \text{span}\{\phi_1, ..., \phi_N\}$. Having discretised the domain into cells, we can approximate the solution over each cell τ with a function in a space $\mathcal{V}(K)$, where $K \subset \mathbb{R}^d$ is a reference cell that every τ can be mapped to. The global solution is then constructed from all the local ones. For computational practicality reasons, $\mathcal{V}$ is usually chosen to be a space of polynomials of a chosen degree, q, over the reference cell K

$$\mathcal{V}(K) = \mathcal{P}_q(K) = \text{span}\left\{ x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n} \middle| \sum_{i=1}^{n} \alpha_i \leq q, \quad \forall\ i = 1, \ldots, n : \alpha_i \geq 0 \right\} \tag{3}$$

We call a set $\mathcal{L} = \{\ell_1, ..., \ell_d\}$ "degrees of freedom" if it forms a basis for the dual space of $\mathcal{V}$. The dual space of a vector space V is the set of all linear functionals from V to the field over which it is defined. In commonly used Lagrange elements, the functionals ℓ_i are defined as point evaluations that uniquely specify a function in $\mathcal{V}$. For example, in a case of a piecewise linear basis, $\mathcal{P}_1$, a function is uniquely determined by its values at the vertices of the mesh. The number of points required increases with the order of polynomials.

Similarly to increasing the granularity of domain discretisation, increasing the polynomial order increases the number of degrees of freedom, and, consequently, the dimension of the discretised solution space V_h. That makes it a better approximation of the infinite dimensional solution space V, but comes at a higher computational cost. Additionally, increasing the accuracy by increasing the polynomials degree is only suitable for problems with sufficient solution smoothness. Conversely, we must be careful, to choose a basis that enables to represent functions of required smoothness. For example, if the solution belongs to H^2, we want to be able to represent any function in that space as a linear combination of the basis functions, so they must have non-zero second derivatives. In that case, first order polynomials would be a bad choice.

The triple $(K, \mathcal{V}, \mathcal{L})$ consisting of the refrence cell, local function space, and the set of its degrees of freedom, is the Ciarlet's definition of a "finite element" [12]. The three are problem dependent, and interconnected. For example, choosing triangular cells has the advantage of being computationally easier, as each cell defines a plane, and meshing can

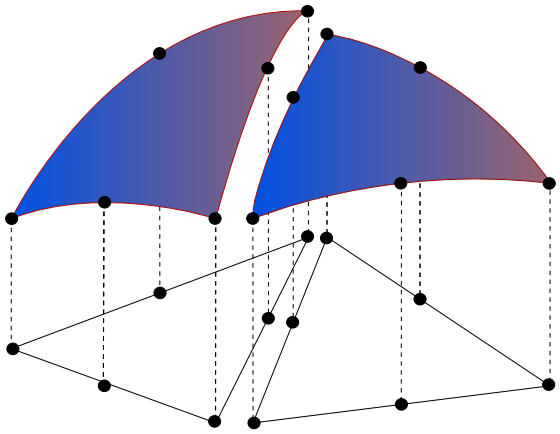


Figure 4: Two discontinuous quadratic finite elements. Figure from [35]

be achieved with established triangulation techniques. However, quadrilateral cells can provide more degrees of freedom than triangles when using the same polynomial basis. They are also generally less computationally demanding than triangles with higher-order bases, making them often a good middle ground. [47] provides an excellent overview of many finite element configurations.

For parametric equations, domain of the function space V_h varies with geometrical parameters, which must be reflected by its basis

$$V_h(\mu) = \text{span}\{\phi_1(\mu), ..., \phi_N(\mu)\}$$

Throughout the discussion of POD-ANN method we assume that $\|u(\mu) - u_h(\mu)\|$ can be made arbitrarily small for any given parameter value, $\mu \in \mathbb{P}$. In practice this is achieved by increasing the number of degrees of freedom, as described earlier.

2.5 Assembly in linear case

Once we have basis functions, the test function and solution approximation can both be represented as

$$u \approx u_h = \sum_{i=1}^{N} U_i \phi_i, \qquad v_h = \sum_{i=1}^{N} V_i \phi_i$$

From Galerkin approximation, (2), it follows that for each basis function ϕ_i

$$g(\mu)\left(\sum_{j=1}^{N} U_j \phi_j; \phi_i\right) = F(\mu)(\phi_i), \quad \forall\ \phi_i \in \{\phi_1, ..., \phi_N\} \subset \mathbb{V}_h \tag{4}$$

If $g(\mu)$ is linear, we can also write

$$\sum_{j=1}^{N} U_j g(\mu)\left(\phi_j; \phi_i\right) = F(\mu)(\phi_i), \quad \forall\ \phi_i \in \{\phi_1, ..., \phi_N\} \subset \mathbb{V}_h \tag{5}$$

Defining a “stiffness" matrix A as $A_{ij} = g(\mu)(\phi_j; \phi_i)$, collecting the degrees of freedom values in a vector U, and constructing a vector F where each component is defined as

$F_i = F(\mu)(\phi_i)$, the problem can be expressed as a matrix equation,

$$AU = F$$

Constructing these matrices is known as assembly, and enables solving the discretised PDE problem as a linear algebraic problem. Due to the nature of basis functions, which we defined to be piecewise polynomials nonzero only on neighbouring elements, the stiffness matrix A, will be a sparse one, heavily influencing the solver choice.

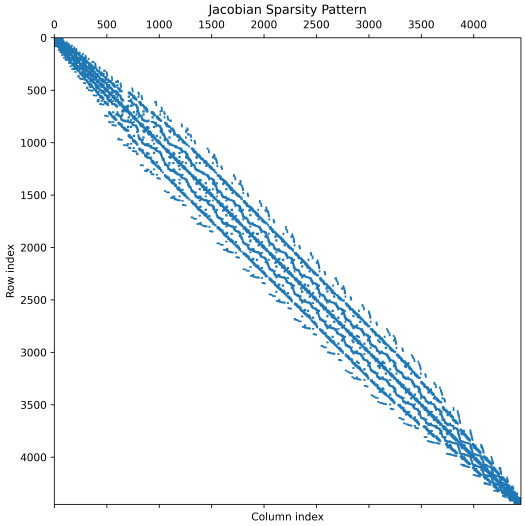


Figure 5: Sparsity pattern of Jacobian matrix for the lid driven cavity problem.

2.6 Linear solvers

The solution to a system $Ax = b$ can be approached through direct or iterative techniques. Direct techniques determine an exact solution in a predetermined number of operations, as seen with methods like LU decomposition. Iterative techniques refine an approximate solution through repeated iterations, stopping when the error becomes acceptably low. Such techniques, which include stationary and Krylov subspace methods, often require less memory and can be quicker, though their efficiency is, in general, not guaranteed. Another advantage is the option to stop calculations early, when lower accuracy is required. [54]

Solving large linear systems is done with computers, introducing rounding errors. The key concern is whether these errors significantly affect the solution's accuracy. This is examined by considering the equation $A(x + \delta x) = b + \delta b$, where δb is the error in the input and δx is the resulting error in the solution. The error's impact can be bounded using the matrix's condition number, $\kappa(A) = \|A\|\|A^{-1}\|$:

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta b\|}{\|b\|}$$

A high condition number indicates that the matrix is "ill-conditioned" and small input errors can lead to large error in the result. [54]

The condition number of a matrix can be lowered by multiplying it with a "preconditioner" – a matrix $P^{-1} \approx A^{-1}$ such that $P^{-1}A$ has a smaller condition number than A.

Preconditioners are useful in iterative methods, since the rate of convergence for most iterative linear solvers is inversely proportional to the condition number. The construction of preconditioners is a large research area.

2.7 Assembly in nonlinear case

In nonlinear problems we can't make the step from equation (4) to (5). The problem, in that case, can be represented as finding roots of a nonlinear function $\mathcal{F}(x)$. The most common way of tackling such problems is the iterative Newton's method, named after Isaac Newton, who studied it while an undergraduate at Cambridge.

Consider the Taylor expansion of $\mathcal{F} : \mathbb{R}^a \to \mathbb{R}^b$ around the point x_n corresponding to the n-th iteration

$$\mathcal{F}(x_n + \delta x) = \mathcal{F}(x_n) + J(x_n)\delta x + \delta x^T H(x_n)\delta x + ...$$

where $J \in \mathbb{R}^{b\times a}$ is the Jacobian of F and $H \in \mathbb{R}^{b\times a\times a}$ its Hessian. The expression for δx can be linearised by truncating the expansion to first order term

$$\mathcal{F}(x_n + \delta x) \approx \mathcal{F}(x_n) + J(x_n)\delta x$$

As the aim is for δx to move us to the root, $\mathcal{F}(x + \delta x) = 0$, so we have

$$J(x_n)\delta x = -\mathcal{F}(x_n) \tag{6}$$

which is a linear problem, that can be solved with previously described linear solvers. These steps are iteratively repeated, with updated values of $x_{n+1} = x_n + \delta x$, until sufficient accuracy is achieved.

Newton's method is applied to the problem in (4) by letting

$$\mathcal{F}_i(x) = g(\mu)\left(\sum_{j=1}^{N} x_j\phi_j; \phi_i\right) - F(\mu)(\phi_i), \qquad \mathcal{F} = [\mathcal{F}_1, ..., \mathcal{F}_N]^T, \qquad x = U$$

An FEM package is a software which implements steps described in this chapter, and usually abstracts them away from the user. Due to widespread application in engineering, most such packages are commercial and closed source, e.g. ANSYS. This usually means that the lower level structures, such as assembled matrices, can't be accessed directly. Methods which rely on access to assembled matrices of a problem are referred to as "intrusive". A big advantage of the POD-ANN method is its non-intrusiveness – it only requires full order solutions $u_h(\mu)$ to work. This flexibility means that it can be combined with established, highly optimised commercial software.

3 Navier-Stokes equations

Partial differential equations explaining the motion of a fluid are among the most famous ones. They were first formulated by Claude-Louis Navier, and completed in Cambridge, by Lucasian Professor of Mathematics, George Stokes [37, 52]. The Navier-Stokes equations encapsulate the balance of momentum, by simplifying the molecular dynamics of a fluid into continuum mechanics approximations. A further approximation often made, and used in this project, is the incompressible form of Navier-Stokes equation, where the density

of the fluid is assumed constant throughout, giving following, strong, formulation of the problem

$$\begin{cases} \rho(\mu)\dfrac{\partial u}{\partial t} - \nu(\mu)\Delta u + \rho(\mu)(u \cdot \nabla)u + \nabla p = f \\ \nabla \cdot u = 0 \end{cases} \tag{7}$$

where u is the velocity field, and p the pressure field. They form the foundation of fluid mechanics, and, as such, help in designing aircraft and cars, study blood flow, forecast weather, and can even be applied to model crowd dynamics.

However, the Navier-Stokes equations involve two physically distinct quantities, pressure and velocity, which are coupled, and must be solved for simultaneously. Furthermore, the equations are nonlinear in velocity, making them possible to solve analytically only in the simplest of cases. There is not even a proof that a smooth solutions always exists, or that is is unique. This problem, is one of seven Millennium Prize problems, identified by Clay Mathematics Institute in 2000 as a representation of the most difficult mathematical problems at the turn of the second millennium. [1]

Due to these difficulties, and the practical importance of the problem, numerical methods of Computational Fluid Dynamics (CFD) are predominantly used to solve Navier-Stokes equations.

3.1 Reynolds number

Reynolds number is defined as

$$\mathrm{Re} = \frac{uL}{\nu}$$

where u is the velocity, L a characteristic length, and ν the kinematic viscosity of the fluid. It reflects the ratio between inertial and viscous forces, helping to characterise the fluid behaviour. At low Reynolds numbers, flows tend to be dominated by “laminar" flow, while at high Reynolds numbers, flows tend to be “turbulent".

Laminar flow is characterized by a highly ordered, smooth fluid motion, with layers that do not mix. The non-linear term, accounting for convective accelerations, becomes negligible, significantly simplifying Navier-Stokes equations. This simplification often leads to a linear system that can be solved more easily using numerical methods.

In contrast, turbulent flow is marked by rapid variations in pressure and velocity in both time and space, resulting in chaotic fluid motion and intensive mixing between layers. Solving the Navier-Stokes equations for turbulent flow requires handling the non-linear terms that dominate the dynamics, making the equations significantly more complex to solve numerically.

The transition from laminar flow regime to turbulent flow regime occurs usually within $2000 < \mathrm{Re} < 4000$ [55].

3.2 Finite Elements Method setup

Introducing test function spaces for velocity and pressure respectively, as

$$V = \left\{ v \in [H^1(\Omega(\mu))]^d : v = 0 \text{ on } \Gamma_D(\mu) \right\}, \qquad Q = L^2(\Omega(\mu))$$

we can express the problem in a weak formulation

$$\begin{cases} \int\limits_{\Omega(\mu)} \rho(\mu)\frac{\partial u}{\partial t}\cdot v + \int\limits_{\Omega(\mu)} \nu(\mu)\nabla u\cdot\nabla v + \int\limits_{\Omega(\mu)} \rho(\mu)(u\cdot\nabla)u\cdot v - \int\limits_{\Omega(\mu)} p\nabla\cdot v \\ \qquad\qquad = \int\limits_{\Omega(\mu)} f\cdot v + \int\limits_{\Gamma_N(\mu)} h\cdot v \qquad \forall\, v\in V \\ \int\limits_{\Omega(\mu)} \nabla u\cdot q = 0 \quad \forall\, q\in Q \end{cases} \tag{8}$$

The full derivation can be found in appendix B, and definition of spaces H and L in appendix A. Pressure is only specified up to a constant, because it appears in the equation only through its gradient. We usually fix it by specifying its value at a certain point, or adding a constraint $\int_\Omega p\mathrm{d}\Omega = 0$.

In steady state $\frac{\partial u}{\partial t} = 0$, so the first term vanishes. Other terms can be represented as

$$\begin{cases} a(\mu)(u,v) + c(\mu)(u;u,v) + b(\mu)(p,v) = s(f,v) \\ b(\mu)(q,u) = 0 \end{cases} \tag{9}$$

where c is nonlinear. After discretisation (9) can be represented as

$$\begin{cases} AU + C(U)U + BP = S \\ B^T U = 0 \end{cases}$$

or in matrix form

$$\underset{\text{Stiffness matrix}}{\begin{bmatrix} A+C(U) & B \\ B^T & 0 \end{bmatrix}} \underset{\text{Solution vector}}{\begin{bmatrix} U \\ P \end{bmatrix}} = \underset{\text{Right hand side}}{\begin{bmatrix} S \\ 0 \end{bmatrix}} \tag{10}$$

where A represents the discretised vector Laplacian, B^T represents the gradient (mapping from scalar-valued Q to vector-valued V), and B represents the divergence (mapping from vector-valued V to scalar-valued Q) [18]. $C(U)$ represents the advection operator, dependent on solution vector U, which makes the system of equations non linear.

A problem of the form (10) is an example of a "saddle point problem" [8], with both negative and positive eigenvalues, meaning that the matrix is not positive semidefinite, which has implications on the choice of method used to solve it.

The weak formulation of Navier-Stokes requires velocity to be once differentiable, and puts no differentiability requirements on pressure. Therefore, we can use quadratic polynomial velocity function space, and linear polynomial pressure function space. Degrees of freedom for both spaces over a cell can be combined to form a single, mixed finite element, called Taylor-Hood. Such elements can be shown to satisfy the inf-sup condition [53], ensuring stability of the numerical solution. Taylor-Hood elements are capable of satisfying the divergence-free condition in an integral sense over the domain, as required by the Navier-Stokes equation. However, they do not guarantee local incompressiblity – divergence of the velocity field may not be zero at every single point within each element.

4 Model Order Reduction

MOR has significantly advanced the simulation and design of integrated circuits with complex three-dimensional structures. In late 1990s, integrated circuits were mostly limited to one or two layers of metal, which had negligible effects on circuit performance. By

2009, the norm was already 8-10 metal layers, which introduce signal delays and parasitic effects at high frequencies. This complexity is managed by accurately simulating the electromagnetic behavior through 3D solutions of Maxwell's equations. Simulating a single MOS transistor involves solving a system of three partial differential equations, which with FEM translates into a discrete system of the order of 30,000 unknowns. Given that an electronic circuit may contain between 10,000 and 1,000,000 such MOS devices, the scalability of simulations becomes feasible only through MOR techniques like balanced truncation and moment matching [44]. The indispensable role of MOR in modern circuit design suggests its potential to revolutionize other fields, such as electric aviation or fusion energy, by enabling similarly complex simulations and designs.

Imagine someone who decides to explore how an incompressible Newtonian fluid flows through a pipe with a constant cross-section at low Reynolds numbers. Although they haven't taken a course in fluid mechanics, they excelled in a course on finite element method (FEM) and are familiar with the Navier-Stokes equations. So, for a number of pipe and fluid configurations, they set up the FEM problem and solve it, getting numerical velocity field values for each case. Over time, they start to notice a recurring pattern: the velocity consistently follows a quadratic profile. It turns out that the entire velocity field has always a parabolic profile, and for every parameters combination, the solution can be exhaustively described by scaling factor of that profile, instead of hundreds of values for all FEM's degrees of freedom. This scenario is called Poiseuille flow, after J.L.M. Poiseuille who made that discovery in 1839, and there is an analytic solution for this case.

This example, while idealized, illustrates the idea of model order reduction: complex solutions can often be represented as a combination of simpler subsolutions. We refer to these foundational subsolutions as the "reduced basis" of the problem. This concept allows us to represent the problem in a much lower-dimensional space, simplifying the analysis and solution of parameterized partial differential equations.

This is captured formally by introducing a manifold, a topological space that locally resembles Euclidean space near each point, comprising of all solutions of the parametric problem under variation of the parameters

$$\mathcal{M}_h = \{u_h(\mu) | \mu \in \mathbb{P}\} \subset V_h$$

where each $u_h(\mu) \in V_h$ corresponds to the solution of the parametric truth problem (2).

MOR relies on the assumption that the solution manifold is of low dimension, i.e., that the span of a low number of appropriately chosen reduced basis functions represents the solution manifold with a small error. Given an L-dimensional reduced basis, denoted as $\{\xi_n\}_{n=1}^{L} \subset V_h$, the associated reduced basis space is given by

$$V_{\text{rb}} = \text{span}\{\xi_1, ..., \xi_L\} \subset V_h$$

We call projection of the truth on this reduced space as "reduced basis solution", and denote by $u_{\text{rb}}(\mu)$,

$$u_{\text{rb}}(\mu) = \sum_{i=1}^{L} U_i^{\text{rb}} \xi_i$$

where U^{rb} is the vector of coefficients of the reduced basis approximation [29].

In reality, we won't, in general, be able to calculate all solutions in the manifold. Denoting the discrete set of samples evaluated as $\mathbb{P}_h \subset \mathbb{P}$, we can introduce the following set

$$\mathcal{M}_h(\mathbb{P}_h) = \{u_h(\mu) | \mu \in \mathbb{P}_h\} \subset \mathcal{M}_h$$

of cardinality $M = |\mathbb{P}_h|$. In this context, $u_h(\mu)$ are often called "snapshots". If sampling is done well, $\mathcal{M}_h(\mathbb{P}_h)$ can be a good representation of $\mathcal{M}_h$. These concepts and notation are adapted from [29].

4.1 Proper Orthogonal Decomposition

Previously we assumed that the reduced basis $\{\xi_n\}_{n=1}^{L}$ was given. Now, we explore how we can find it from a discrete set of samples from the manifold, $\mathcal{M}_h(\mathbb{P}_h)$. We do this using POD, which identifies the the most significant modes of variation in the data set, and creates the reduced basis out of them. As such, in the discretised form, it is analogous to principal component analysis [56].

First step of the method is to build the correlation matrix $C \in \mathbb{R}^{M\times M}$

$$C_{ij} = \frac{1}{M}\langle u_h(\mu_i), u_h(\mu_j)\rangle$$

Here, we encounter the challenge mentioned in section 1.1, where we need to be able to compare between functions from potentially different geometrical domains. We choose to address this by projecting both solutions back onto the reference domain, and taking the L^2 inner product over reference domain

$$\langle\cdot,\cdot\rangle = \langle\cdot,\cdot\rangle_{L^2(\Omega_{\text{ref}})}$$

Next, we determine the L eigenvalue-eigenvector pairs (λ_n, v_n) corresponding to the largest eigenvalues, ensuring that each eigenvector v_n is normalized. This relationship is expressed as:

$$Cv_n = \lambda_n v_n, \quad \|v_n\|_{\ell^2} = 1, \quad \forall\ n = 1,\dots,L.$$

Since we require only the L largest eigenvalues of the dense matrix C, iterative eigenvalue algorithms, which are the current state of the art, are particularly suitable for this application.

Because we don't know the manifold's dimension L in advance, we don't know how many reduced basis functions are required, and therefore how many eigenvectors to find. A common approach is to use the minimum ratio of the final, smallest, eigenvalue to the initial, largest one, or to predefine the number of reduced basis functions, L. The algorithm terminates once either criterion is satisfied.

In any case, we are making an approximation. Any N dimensional FEM function is truncated to an $L < N$ dimensional representation. The error introduced as an effect of that is called "projection error"

$$\varepsilon_{\text{proj}} = \|u_h(\mu) - u_{\text{rb}}(\mu)\|$$

If we apply the projection to all elements in $\mathcal{M}_h(\mathbb{P}_h)$, the average error can be written as

$$\sqrt{\frac{1}{M}\sum_{i=1}^{M}\|u_h(\mu_i) - u_{\text{rb}}\|^2} = \sqrt{\sum_{i=L+1}^{M}\lambda_i}$$

for a given dataset $\mathcal{M}_h(\mathbb{P}_h)$, POD determines the base of length L, which minimizes this error metric [17]. The projection error can be decreased by increasing L.

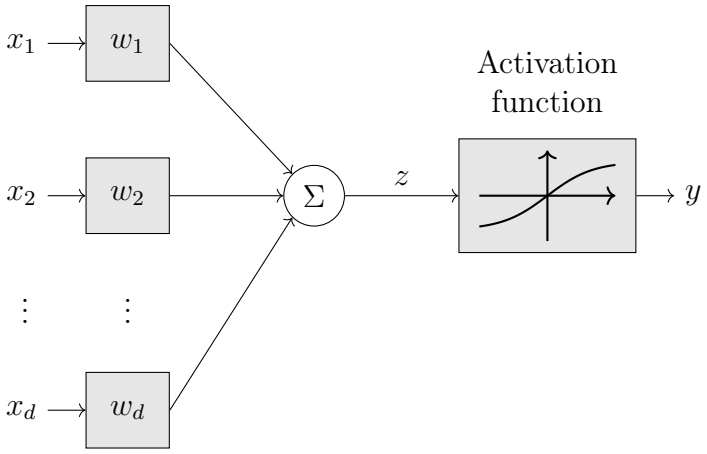


Figure 6: Artificial neuron

Because $V_{\text{rb}} \subset V_h$, we can express reduced basis in terms of FEM basis functions $\{\phi_n\}_{n=1}^N$ as

$$\xi_j = \sum_{i=1}^{N} B_{i,j}\phi_i, \quad B \in \mathbb{R}^{L\times N}$$

With that representation, coefficients of any function from V_h can be projected to V_{rb} with the matrix B as

$$U_{\text{rb}} = BU_h$$

Note, that for a geometrically parameterised problem, the reduced basis $\{\xi_n(\mu)\}_{n=1}^L$ deforms following FEM basis functions $\{\phi_n(\mu)\}_{n=1}^N$, which deform as the domain varies.

5 Artificial Neural Networks

Let us return to the example of fluid dynamics oblivious person studying pipe flow. Through POD, they found that the solution is always a scaled paraboloid, which means the solution manifold is one-dimensional and they have identified the only reduced basis vector. Now they want determine the scaling of that vector, based on parameters. We know there is an analytic solution to that, but how could it be found automatically?

In this section we introduce ANNs, which are biology-inspired combinations of multiple simple mathematical functions that implement more complicated functions, typically, from real-valued vectors to real-valued vectors. Crucially, they learn these functions through patterns observed in training data, rather than through direct programming.

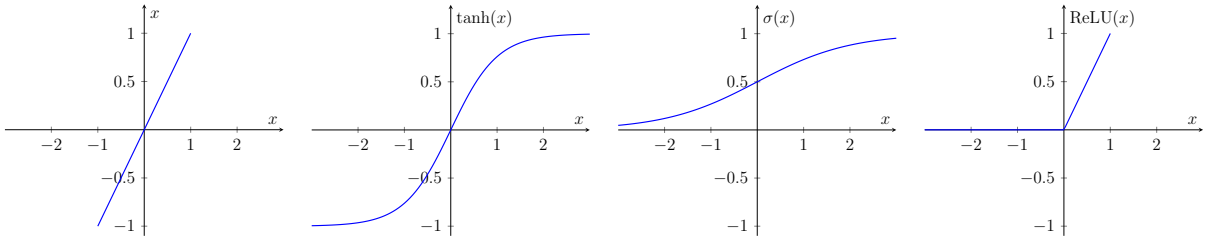
5.1 Multilayer perceptron

We focus on the most basic type of currently used ANNs - multilayer feed-forward networks. Their building blocks are “neurons".

A single neuron takes several inputs, $x_1, x_2, \ldots$ and produces a single output y. Weights, $w_1, w_2, \ldots$ are real numbers expressing the importance of the respective inputs to the output. The output is produced as a weighted sum of inputs, passed through an “activation function" $\phi(\cdot)$, which decides the level of “neural activation" [21].

$$z = w^T x, \quad y = \phi(z)$$

The only constraint on the activation functions is that they need to be differentiable, for reasons that will be discussed shortly. For “output" neurons that directly represent the modeled distribution, the activation function choice is additionally restricted by that distribution. For example, if the node represents a probability, its value should be between 0 and 1. While even a simple linear function can be used, nonlinear functions significantly expands the space of functions that the network can approximate. Common choices are hyperbolic tangent, sigmoid, and ReLU, all shown in figure 7. Strictly speaking, the derivative of the ReLU function is not defined when $x = 0$, but in practice this can be safely ignored, and ReLU is one of the best-performing activation functions, in widespread use. [10]



(a) Linear (b) tanh (c) Sigmoid (d) ReLU

Figure 7: Various activation functions

We can combine a number of such neurons, each with its own weights, in a “layer". In a classification case, each neuron of that layer could correspond to probability assigned to the input data corresponding to a given class.

In multilayer feed-forward networks, outputs of one layer, $y^(k)$, serve as inputs to the next one, $x^{(k+1)}$,

$$x^{(1)} = x, \quad x^{(k+1)} = y^{(k)}, \quad z^{(k)} = W^{(k)}x^{(k)}, \quad y^{(k)} = \phi(z^{(k)}), y = y^{(L+1)}$$

The input and output layers are typically determined by the problem being solved. The intermediate layers, known as “hidden layers" can be adjusted in terms of their number and the number of neurons each of them contains. Defining the network’s topology, these elements are examples of hyperparameters.

The space of multivariate functions a network can implement is determined by its topology, activation functions, and multiplicative parameters. [31] showed that multilayer feedforward networks with non-linear activation function are a class of universal approximators, meaning that they are capable of approximating any reasonably well-behaved (Borel measurable) function between finite-dimensional spaces to any level of precision, as long as there are enough neurons in the hidden layer.

5.2 Training

At the heart of the model are its weights, which determine how input data is transformed into outputs through algebraic operations. How, then, do we select these weights to ensure the model performs as intended? Here, we only consider supervised learning. In that learning paradigm we have a set of inputs and corresponding desired outputs. For our task, the inputs are PDE’s parameters values $\mu \in \mathbb{P}_h^{\text{tr}} \subset \mathbb{P}_h$, where $\mathbb{P}_h^{\text{tr}}$ represents the training dataset of size M^{tr}. Note, that the ANN doesn’t discriminate between geometrical

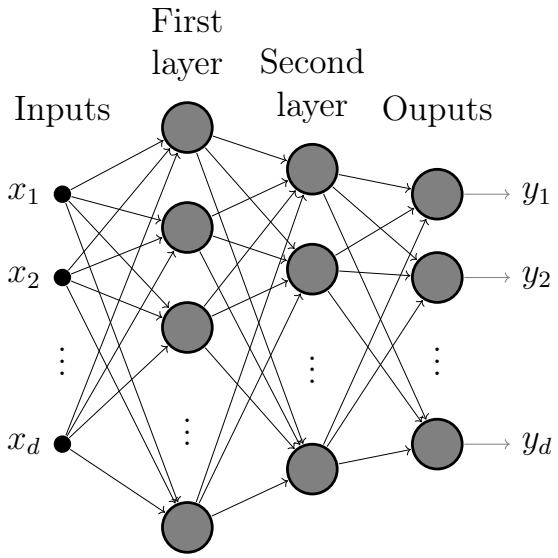


Figure 8: Two layer feed-forward network

and physical parameters. For each input μ, the desired output are coefficients $U_{\text{rb}}(\mu)$ of the FEM solution projected on the reduced basis space

$$U_{\text{rb}}(\mu) = BU_h(\mu), \quad U_h(\mu) : u_h(\mu) \in \mathcal{M}_h(\mathbb{P}_h)$$

The weights are chosen in a way that aligns model's outputs with the desired ones. To do this, we need to define a loss function, which quantifies how far from the target model's outputs are. Typical choices for the loss function include mean squared error (MSE) in regression tasks, or cross-entropy in classification. For this project we choose MSE. Denoting the prediction of an ANN with weights w for an input μ by $U_{\text{pred}}(\mu)$ the error can be calculated as

$$\mathcal{L}(w) = \frac{1}{2M^{\text{tr}}} \sum_{\mu \in \mathbb{P}_h^{\text{tr}}} \|U_{\text{pred}}(\mu, w) - U_{\text{rb}}(\mu)\|_2^2$$

Once we have the loss function, we can use optimisation algorithms to minimise the loss with respect to weights vector, aligning model's outputs with the desired ones. A very common and simple one is "gradient descent". It is an iterative algorithm for finding a local minimum of a differentiable multivariate function, which in every step moves in the direction of steepest descent. Size of the step is determined by a "learning rate" hyperparameter η,

$$w[\tau + 1] = w[\tau] + \eta \, \nabla_w \mathcal{L}|_\tau$$

In the case of MSE loss function, the gradient can be expressed as

$$\nabla_w \mathcal{L} = \frac{1}{2M^{\text{tr}}} \sum_{\mu \in \mathbb{P}_h^{\text{tr}}} \nabla_w \|U_{\text{pred}}(\mu, w) - U_{\text{rb}}(\mu)\|_2^2$$

That means, that every step requires calculating a gradient that is based on all training samples, which for large data set is very expensive, and would result in slow updates. Instead, in an approach called stochastic gradient descent (SGD) we divide the training dataset into random subsets, "mini-batches", $\mathbb{P}_h^{\text{tr}^*}$, and update weights based on the gradient estimate calculated from these subsets. Size of the mini-batch, M^{tr^*}, is another

hyperparameter that can be varied. We hope that each mini-batch is a good representation of the entire data set, and the error in gradient, resulting from this approximation is small.

$$\nabla_w \mathcal{L} \approx \frac{1}{2M^{\mathrm{tr}^*}} \sum_{\mu \in \mathbb{P}_h^{\mathrm{tr}^*} \subset \mathbb{P}_h^{\mathrm{tr}}} \nabla_w \| U_{\mathrm{pred}}(\mu, w) - U_{\mathrm{rb}}(\mu) \|_2^2$$

An “epoch" is completed when enough mini-batches have been processed to include every sample in the training dataset. Typical training process involves many such epochs to converge on the optimal model.

An issue common to both gradient descent, and its stochastic version, is dependence on the fixed learning rate η. High learning rate can lead to oscillations and a failure to converge. Low learning rate, on the other hand, makes the convergence wastefully slow. Extensions of gradient descent, partially addressing this problem through incorporation of momentum and adaptive learning rates for each parameter, are the current state of the art in neural networks training. ADAM [32] computes an exponential moving average of the gradients, which helps in accelerating the gradients vectors in the right directions.

Another issue in gradient-based optimisation methods is slow progress in regions with small gradient values. This is known as “vanishing gradient" problem, and can happen with sigmoid and hyperbolic tangent activation functions, when operating in their saturated regions. To promote operation in the linear region of the activation function, we normalize the data by subtracting the mean and dividing by the standard deviation.

In batch normalisation, we derive the normalisation statistics from a batch of data. In inference time, the single sample is normalised with statistics found during training. Layer normalisation, in contrast, computes the normalisation statistics from all the features within a single layer, for a single data sample. In that case, as we don't require a batch to derive the normalisation statistics, normalisation in inference in no different to training. The project uses batch normalisation.

Another aspect that needs careful consideration to avoid vanishing gradients, is weights initialisation. We must be careful to not start the gradient descent from a point where the weights configuration saturates activation functions. This can be addressed with Xavier initialisation [25].

Our aim is to train a model that captures the underlying, general, relationships in the data, rather than fits to the particular dataset, which is a problem known as overfitting. To check for this, we divide the dataset into two subsets: a training set $\mathcal{M}_h^{\mathrm{tr}}$, used to update the model's weights, and a validation set $\mathcal{M}_h^{\mathrm{val}}$, used exclusively for evaluating performance without influencing weight adjustments.

$$\mathcal{M}_h^{\mathrm{tr}}, \mathcal{M}_h^{\mathrm{val}} \subset \mathcal{M}_h(\mathbb{P}_h), \quad \mathcal{M}_h^{\mathrm{tr}} \cap \mathcal{M}_h^{\mathrm{val}} = \emptyset$$

When updates to the model's weights no longer improve performance on the validation set, this may indicate overfitting. In a method known as “early stopping", used in the project, this condition is used as a training termination criterion.

6 Computation distribution

As improvements in processors' clock speeds are becoming increasingly difficult due to physical limitations [3], interest grows in computing parallelism – increasing the number of instructions in unit time by executing multiple of them at once, rather than executing

each of them faster. "To pull a bigger wagon, it is easier to add more oxen, than to grow a gigantic ox." [27]

Parallel architectures can be categorized based on the types of instruction and data streams they utilize. [19] introduced a taxonomy that includes single instruction stream, multiple data streams (SIMD) as seen in GPUs, and multiple instruction streams, single data stream (MISD), which is less common and not the focus here. The most versatile category, multiple instruction streams, multiple data streams (MIMD), can be implemented within a single processor, known as uniprocessor parallelism, or across multiple processors, referred to as multiprocessor parallelism [23]. This report concentrates on multiprocessor parallelism, with each process operating on its own core.

This focus stems from the architecture of modern computing systems, in particular those relevant to this project. High-performance computing centers, typically used in FEM calculations, like Cambridge's CSD3 and Frontier, the first exascale center, are large clusters [11, 38]. A cluster consists of many standard computing units connected in an efficient network, enabling them to work on a single task simultaneously. Even modern laptops are MIMD systems, equipped with CPUs that have multiple cores, each capable of processing multiple instruction streams in parallel due to their "superscalar" design [23].

With MIMD architecture, multiple processes can run at the same time. A process in computing is an active instance of a program, including its code, execution state, and allocated resources. This is in contrast to threads, which exist as subsets of a process, and all share the parent process's state and resources.

When multiple processes work on a single task in parallel, they usually need to exchange data relevant to their computations. Within a single machine, these processes can utilize shared memory for data exchange. However, processors primarily achieve high speeds by limiting their memory accesses to low-level caches, which are typically separate for different cores. therefore, accessing shared, higher-level caches, or worse, the main memory, can significantly slow them down. With processes running across separate physical machines, shared memory communication is also possible, but usually message passing is used instead. The message-passing model assumes a set of processes that have only local memory but are able to communicate with other processes by sending and receiving messages. Such message exchange involves slow I/O mechanisms, like example network sockets, which take thousands CPU clock cycles to complete. Therefore, regardless of the method, minimizing interprocess communication is crucial, especially in distributed clusters.

We can split the offline phase into two sections - generating the data set for neural network, and training the neural network on that data set. We look at distribution of each of these processes in turn.

6.1 Dataset generation distribution

Let us first take a look at the ways in which the dataset generation can be parallelised. For reference, a non-parallel approach, with a single process, is summarised in figure 9.

Suppose we have P processes available. Given a set of M parameters values that we seek solutions for, $\mathbb{P}_h = \{\mu_1, \ldots, \mu_M\}$, we can split it into P subsets

$$\mathbb{P}_{h,1} = \{\mu_1, \ldots, \mu_{\lfloor M/P \rfloor}\}, \ \ldots, \ \mathbb{P}_{h,P} = \{\mu_{(P-1)\lfloor M/P \rfloor + 1}, \ldots, \mu_M\},$$

and calculate all solutions in a given subset within one processes. This approach is presented in figure 10. By allocating similar problems to one process, we may be able

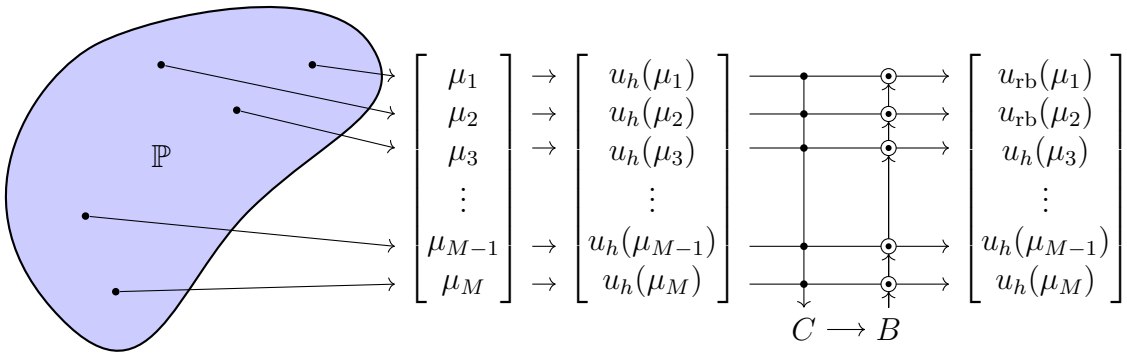


Figure 9: A diagram of POD-ANN data set generation with a single process.

to take advantage of the similarity, for example by using solution of one problem as a preconditioner to another one.

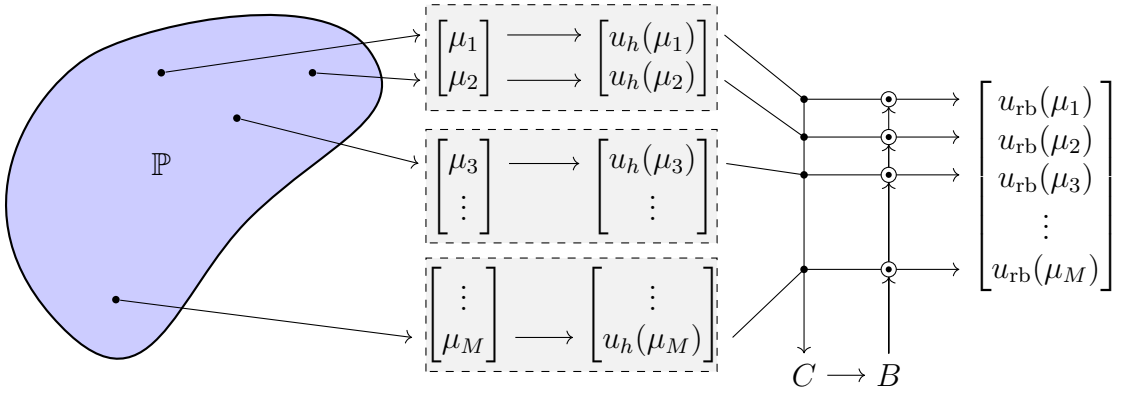


Figure 10: A diagram of POD-ANN data set generation with multiple processes. Each shaded box is used to indicate a separate process.

Another way in which parallel resources can be used in FEM solutions calculations, is to solve a single FEM problem using multiple processes in parallel. This is achieved by dividing the computational domain into subdomains, each assigned to a different process. Each processes assembles a part of the global stiffness matrix, corresponding to its subdomain. It resembles solving a puzzle, where the entire image can be split into sections, each section can be solved on its own, and at the end they are put together to form the big picture. A key difference is that in distributed FEM, the problem is solved without ever combining the submatrices into the full, global stiffness matrix. Iterative solvers, which are more efficient in parallel settings, are typically used for this purpose instead of direct solvers that operate serially [54].

This approach could be employed in our method by solving parameters sequentially, one at a time, using all available processes in parallel. As this requires communication between processes to exchange information about the boundaries of their subdomains, it is not our approach of choice. Not uncommonly, however, a single model is so large, that its solution in a single process in infeasible due to memory or time constraints. In that case, we would typically use a mixture of the two approaches by determining the

smallest group that can solve a single problem, and dividing the parameters set between such processor groups, illustrated in figure 11.

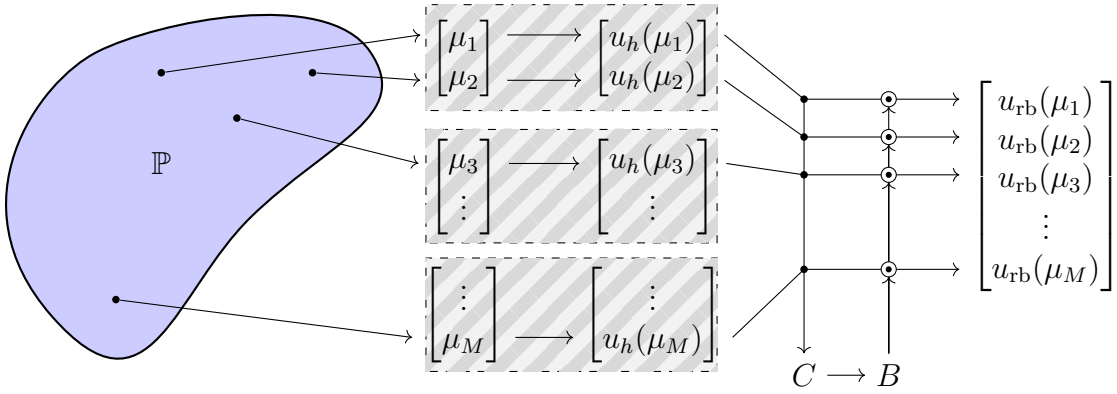


Figure 11: A diagram of POD-ANN data set generation with multiple processes. Each box shaded with multiple colours is used to indicate multiple processes solving one FEM problem in parallel.

6.2 Neural network training distribution

Training neural networks is highly parallelizable because it involves performing the same operations across all elements of a large dataset. The gradient update for each minibatch is independent of other batches. This independence allows for 'data parallelism,' where the dataset is divided into subsets of minibatches, each assigned to a different processor. Each processor calculates its gradient updates and then synchronizes these updates with other processors. Additionally, the processing of each minibatch can itself be parallelized.

Analogously to solving a single FEM problem across multiple processes, neural network models also are sometimes too large for a single process to handle. In an approach known as 'model parallelism,' the model is divided among different processes, which might involve distributing layers across these processes, and allowing outputs to be sequentially transferred from one layers group to the next. As in the FEM case, here also we can combine model parallelism with data parallelism, and divide our computational resources into groups, with each of them capable of handling the entire model, and each scuh group processing a subset of the entire dataset.

7 Numerical results

7.1 Lid driven cavity problem

We now discuss an implementation of the POD-ANN method, applied to a lid driven cavity problem, which is a popular choice within the field of computational fluid dynamics (CFD) for validating computational methods, also used in [30]. While the setting is relatively simple, the flow features created can be complex [5].

The lid-driven cavity flow problem is concerned with solving the steady-state, incompressible Navier-Stokes equations, (7), within a parallelogram-shaped cavity. The motion is instigated by the movement of one wall, referred to as the "lid" as seen in figure 12. The

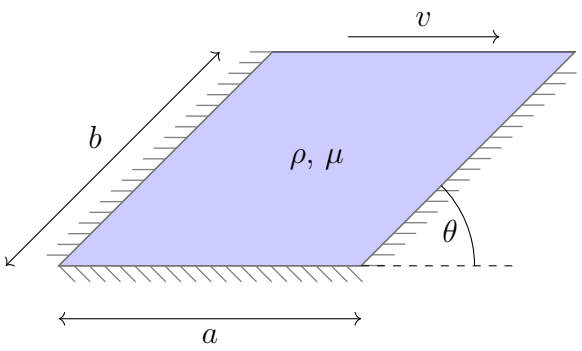


Figure 12: Lid driven cavity flow problem. The shaded region represents a fluid, enclosed by three fixed walls indicated with dashed lines, and a lid moving horizontally with speed v.

problem has three geometric parameters: a, b, θ, and two physical parameters: density, ρ, and viscosity, ν.

In our experiments we keep rho and mu fixed at 1, and vary geometrical parameters in ranges

$$\begin{aligned} a &\in \mathbb{P}_a = [0.5, 2.5) \\ b &\in \mathbb{P}_b = [0.5, 2.5) \\ \theta &\in \mathbb{P}_\theta = [\pi/10, 9\pi/10) \\ \mathbb{P} &= \mathbb{P}_a \times \mathbb{P}_b \times \mathbb{P}_\theta \end{aligned}$$

leading to a Reynolds number of the order of 1, well within laminar flow region. Note that the range of deformation is higher than the one demonstrated in [30].

7.2 Hardware

The computational experiments presented in this section were implemented in Python and executed in an environment encapsulated within a Docker container. This was done to ensure portability and reproduciblity of experiments, but imposed limitations on resource allocation. The experiments were conducted within macOS Sonoma 14.2.1 (23C71) operating system, running on a computer with a 1,4 GHz Quad-Core Intel Core i5 CPU and 8 GB of 2133 MHz LPDDR3 RAM. All four physical cores of the CPU feature hyper-threading, giving 8 logical cores (threads).

7.3 Message Passing Interface

We adopt a computational model of a set of processes which have only local memory, but are able to communicate with other processes by sending and receiving messages. Such a model is known as message passing, and while not necessary on a single multicore CPU with shared memory, is well suited to modern distributed computing infrastructure. A specific realisation of the message-passing model, used in this project, is an open-source standard “Message Passing Interface" (MPI) [20]. It’s portability and wide adoption, performing efficiently on everything from a single multicore CPU, to large supercomputers, makes it perfectly suited for prototyping, and subsequent diverse testing. In this project, it is used through Python bindings, provided by mpi4py [15].

In practice, it works by executing a chosen number of identical instances of the program within an “mpiexec" environment. Each instance of a program running under MPI is known as a “process", and each process is assigned a unique identifier called a “rank". We can write the program in a way that each process takes a different path through the program, depending on its rank. Processes can exchange data between each other with methods such as one to one messages, or broadcasting data from one process to all others and gathering data from all to one. Primitive operations on data across different ranks, such as summing a variable from all processes and storing the result in one of them, are also possible. A notable limitation is that for processes to communicate objects, these must support serialisation and deserialisation through Python's special `__setstate__` and `__getstate__` methods. Many objects from libraries used in the project lack this feature, making it labour-intensive to implement their transfers.

7.4 Parameters sampling

The first step of the POD-ANN method, and the first step of our program, is to sample the parameters set $\mathbb{P}$, to generate the discrete set $\mathbb{P}_h$. We choose to do that with the randomised Latin hypercube sampling, [36]. Parameters set, and a particular draw of $M = 100$ samples are presented in figure 13. 2D projections are presented in figure 14.

To ensure that all processes have the same set $\mathbb{P}_h$, we generate it entirely within one process, and broadcast to others. Next, each of P processes is assigned to a subset $\mathbb{P}_{h,i}$ of these parameters, as per diagram 10.

Sampling could be distributed if running it in one process only becomes longer than the inter-process communication penalty. This could be the case if a more computationally demanding sampling scheme is used, or sufficiently many samples are required. The cost of communication at this stage could be avoided altogether, while ensuring each process is solving similar problems, by subdividing the parameters set and allowing each process to sample from its specific subset.

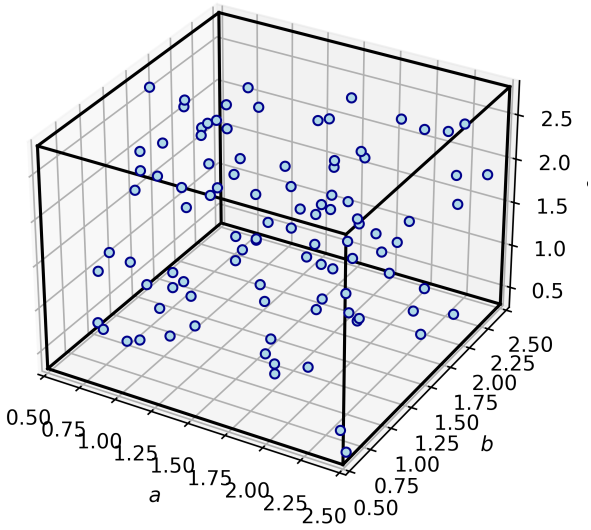


Figure 13: Parameters domain $\mathbb{P}$ and a discrete set $\mathbb{P}_h$ of $M = 100$ samples drawn from $\mathbb{P}$ with Latin hypercube sampling.

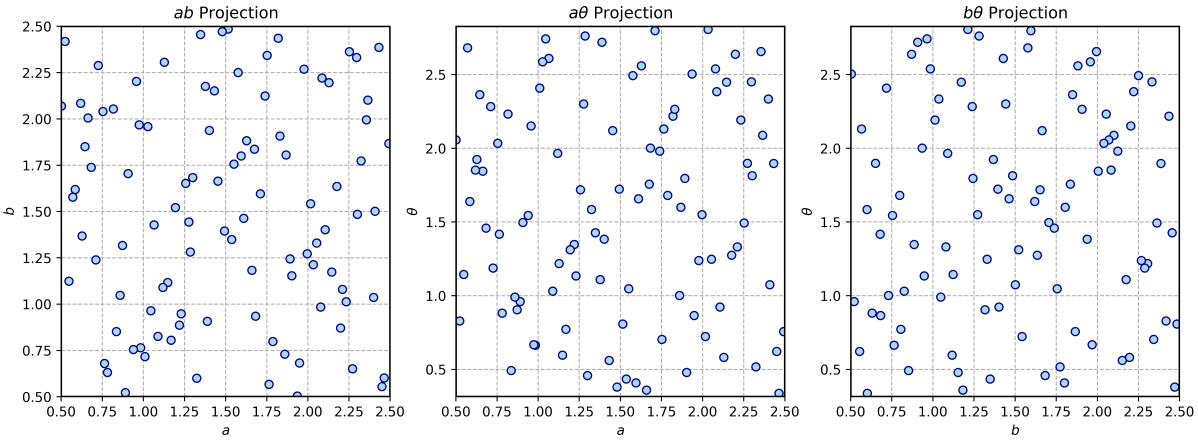


Figure 14: Projections of figure 13.

7.5 Mesh

The mesh is only generated once, for the reference geometry defined as

$$\Omega_{\text{ref}} = \Omega(\mu_g = [a = 1, b = 1, \theta = \pi/2])$$

We use GMsh with granularity `mesh_size = 0.05` to generate a mesh with 944 triangular cells, presented in figure 15b. Gmsh is an open source finite element mesh generator [24], offering a Python API.

Each process reads the entire mesh, and then deforms it, to obtain the domain specified by geometrical parameters. We use harmonic deformation described in 1.1, implemented by MDFEniCSx, an open-source library in development [49]. For every point $[x, y]$ on the boundary of the reference mesh, MDFEniCSx needs to be provided with a corresponding point $[x, y](\mu)$ on the deformed domain. In the parallelogram case, this is achieved simply with the linear transformation

$$\begin{bmatrix} x \\ y \end{bmatrix}(\mu) = \begin{bmatrix} a(\mu) & b(\mu)\cos(\theta(\mu)) \\ 0 & b\sin(\theta(\mu)) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \quad \forall (x, y) \in \partial\Omega$$

Note that in this simple case, the above linear transformation for boundary points could be extended to the entire domain to arrive at the deformed mesh, instead of using the harmonic deformation.

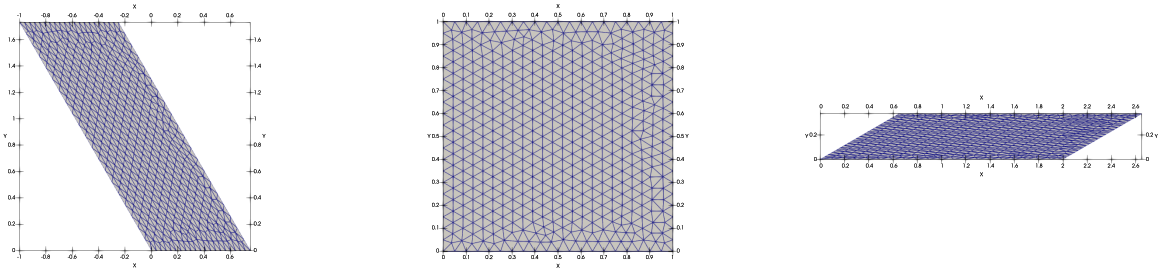
Figure 15 shows the reference mesh, and two deformation examples, for parameters

$$\mu_1 = [a = 0.75, b = 2, \theta = 2\pi/3, \rho = 1, \nu = 1]$$
$$\mu_2 = [a = 2, b = 0.75, \theta = \pi/6, \rho = 1, \nu = 1]$$

7.6 Finite elements method

Equipped with a discretised computational domain of the parameterised problem, we can define the finite elements space. We use Taylor-Hood elements defined in section 3.2, with quadratic space for velocity in both dimensions, V_h, and linear space for pressure, Q_h. This results in 3938 degrees of freedom for velocity, and 513 for pressure.

$$\dim(V_h) = 3938, \quad \dim(Q_h) = 513$$



(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 15: Mesh

To solve the FEM problem, we use open-source FEniCSx [4, 6, 45, 46], which offers a high-level symbolic API to define the weak forms of differential equations and automatically handles the assembly of stiffness matrices and load vectors. For nonlinear problems, FEniCSx employs various iterative methods, including Newton-type methods, used in the project, and efficiently manages the nonlinear system, handling the Jacobian assembly within each iteration. The resulting linear systems are solved with methods provided by PETSc [14], a framework for parallel solution of such systems, with Python bindings and supporting MPI.

In the default configuration, FEniCSx distributes each FEM problem across all available processes. However, as outlined in section 6.1, such approach may not be the optimal use of computational resources in the context of POD-ANN method. Therefore, we explicitly restrict FEniCSx from distributing each solution.

FEM results for μ_{ref}, μ_1, and μ_2 are presented in figures 25 and 29.

7.7 Solvers

FEniCSx gives user the freedom to choose the combination of a Krylov subspace method and a preconditioner from the broad range offered by PETSc, to solve the linear system involved in Newton's method, (6), which for this problem is visualised in figure 5. To make an informed choice, several such combinations were benchmarked. Preconditioners considered were Jacobi, Block Jacobi, Cholesku, SOR, Algebraic Multigrid, LU, and no preconditioning. Iterative methods tested were Richardson, Generalized Minimal Residual, Chebyshev, Conjugate Gradient, BiCGSTAB, and no iterative method, denoted as `preonly`. Krylov subspace method `preonly` combined with the `lu` preconditioner is therefore a direct LU solver. The benchmark set was identical for each configuration, and consisted of 100 configurations of the lid driven cavity problem, shown earlier in figures 13, 14. Each benchmark was run in parallel on four process, representative of conditions of later experiments. The results are presented in figure 16.

The emerging pattern indicates that the LU preconditioning is particularly effective in this problem. Therefore, the solver configuration for obtaining FEM solutions was chosen as the direct LU solver,

```
"ksp_type" = "preonly",  "pc_type" = "lu"
```

Conjugate gradient method is suited to positive definite matrices, which is not the case in this problem. Therefore, as expected, it fails to converge in most cases. Interestingly,

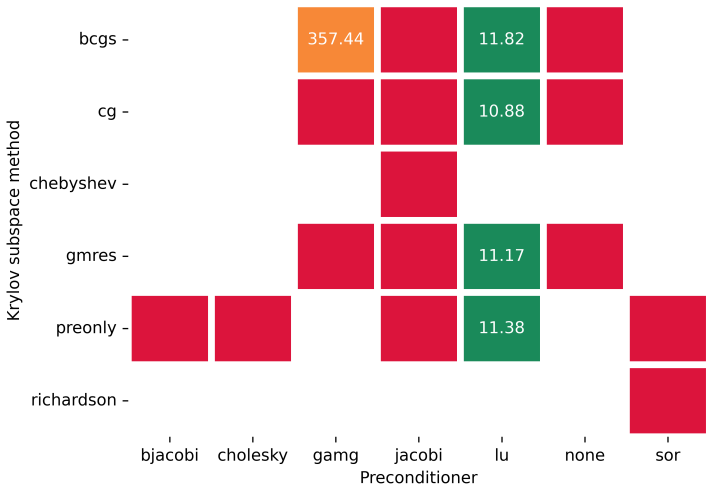


Figure 16: Benchmark of solver configurations. Green and orange cells indicate successful convergence, annotated with time taken, in seconds. Red cells represent failure to converge. Uncolored cells denote untested configurations.

however, it does converge in one case, corresponding to LU preconditioning. Presumably, the LU preconditioner nearly solves the problem and in the process, transforms the matrix to a positive definite form, suitable for the Conjugate Gradient method.

7.8 Proper orthogonal decomposition

Previous steps enable us to generate a set of velocity and pressure solutions in each process, corresponding to the set of parameters assigned to that process. Once all solutions are evaluated, we gather them in a single process, to construct velocity and pressure solutions matrices – the training data set for POD, $\mathcal{M}_h(\mathbb{P}_h)$.

Calculating the POD reduced basis is easily accomplished with RBniCSx [2]. It builds the correlation matrix from the provided solution matrix U, and an inner product definition. Next, it builds the reduced basis from highest eigenvalues of the correlation matrix, until the ratio of the smallest eigenvalue to the highest eigenvalue exceeds specified tolerance `tol`, or until the limit on the number of eigenvalues `Nmax` is exceeded. We choose

$$\langle \cdot, \cdot \rangle = \langle \cdot, \cdot \rangle_{L^2(\Omega_{\text{ref}})}, \quad \texttt{tol = 1.e-6}, \quad \texttt{Nmax = 100}$$

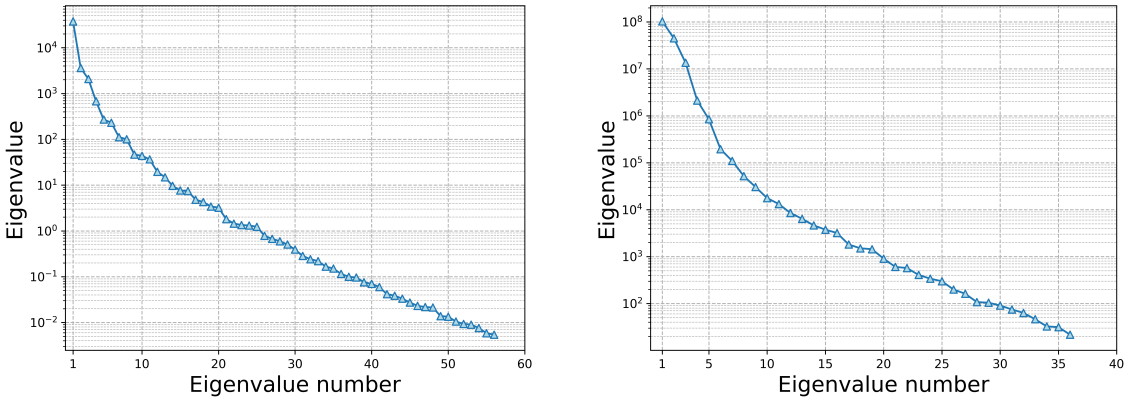
RBniCSx internally uses SLEPc, a software package for the solution of large eigenproblems on parallel computers [28]. However, RBniCSx's POD implementation doesn't take advantage of SLEPc's parallel capabilities, forcing single process execution. This is not an issue for this problem, however, for larger problems, distributing this step would be preferred due to both time and memory constraints.

The POD is done separately on velocity and pressure solutions, to construct separate, independent reduced bases for both. In figure 17 we show the evolution of most significant eigenvalues for velocity and pressure correlation matrices constructed from $M = 200$ solutions. Figures 18 and 20 show the two most significant modes for pressure and velocity respectively.

$$\dim(V_{\text{rb}}) = 51, \quad \dim(Q_{\text{rb}}) = 33$$

As the training dataset is random, the solutions matrix will vary, leading to minor variations in the reduced basis across executions. In the experiments conducted, $50 \leq$

$\dim(V_{\text{rb}}) \leq 55$, and $30 \leq \dim(Q_{\text{rb}}) \leq 35$ in most cases. Examples of FEM solutions projected on the reduced basis and reconstructed back to the full order space are given in figures 26 and 30.



(a) Velocity (b) Pressure

Figure 17: Eigenvalues decay

It is also interesting to see how changing the parameters set changes the reduced basis. We show it by briefly changing the range of angle θ, so that the POD training set only includes parallelograms tilted to the right

$$\theta \in [\pi/10, \pi/2)$$

That reduction in the size of parameters domain is reflected in reduction of the reduced bases dimensions

$$\dim(V_{\text{rb}}) = 36, \quad \dim(Q_{\text{rb}}) = 23$$

and in the shape of most significant modes, see figures 19 and 21.

7.9 Evaluating performance

Experiments introduced in the following steps require us to be able to quantify the performance of the network. We define the error made by the neural network for a parameter μ as

$$\varepsilon_{\text{ANN},u} = \frac{\|u_{\text{rb}}(\mu) - u_{\text{pred}}(\mu)\|}{\|u_{\text{rb}}(\mu)\|}$$

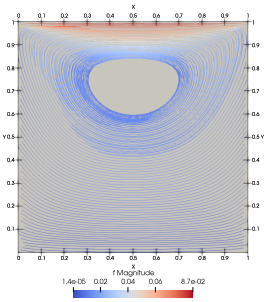
where $u_{\text{pred}}(\mu) \in V_{\text{rb}}$ is the reduced basis function corresponding to coefficients U_{pred} returned by the neural network. We use an inner product induced norm

$$\|\cdot\| = \langle \cdot, \cdot \rangle_{L^2(\Omega(\mu))}$$

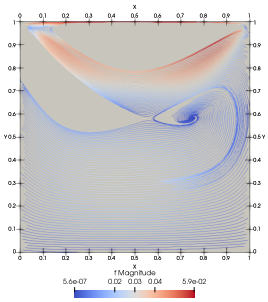
Unlike before, in this case the inner product doesn't need to be evaluated between different function spaces, so we can directly use the L^2 inner product on the deformed domain. Note, that the "truth" for this parameter is $u_h(\mu)$, but the network operates in reduced basis space, so we compare it to the reduced basis projection.

To evaluate performance of the network, we introduce a discrete set of E parameters configurations, entirely separate from sets used for training.

$$\mathbb{P}_h^{\text{perf}} \subset \mathbb{P}, \quad \mathbb{P}_h^{\text{perf}} \cap \mathbb{P}_h = \emptyset$$

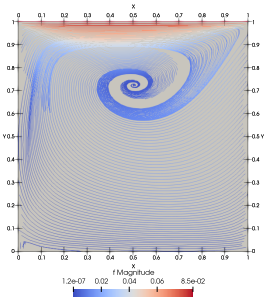


(a) First mode

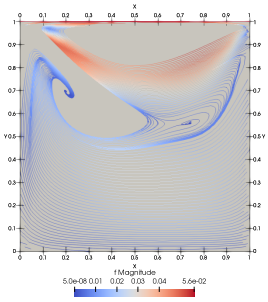


(b) Second mode

Figure 18: Velocity, symmetrical data set

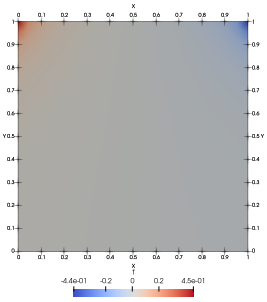


(a) First mode

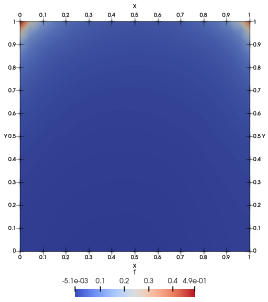


(b) Second mode

Figure 19: Velocity, unsymmetrical data set

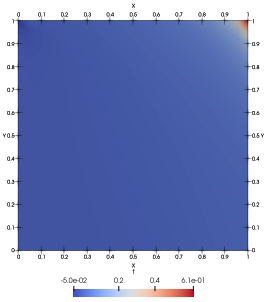


(a) First mode



(b) Second mode

Figure 20: Pressure, symmetrical data set



(a) First mode

(b) Second mode

Figure 21: Pressure, unsymmetrical data set

This set is sampled from $\mathbb{P}$ using Latin hypercube sampling, and we chose a small size of $E = 27$. We find the solution for each parameter, and use the average error ε_{ANN} across this set as a performance metric for the neural network.

$$\mathcal{E}_{\text{ANN},u} = \frac{1}{E} \sum_{\mu \in \mathbb{P}_h^{\text{perf}}} \frac{\|u_{\text{rb}}(\mu) - u_{\text{pred}}(\mu)\|}{\|u_{\text{rb}}(\mu)\|}$$

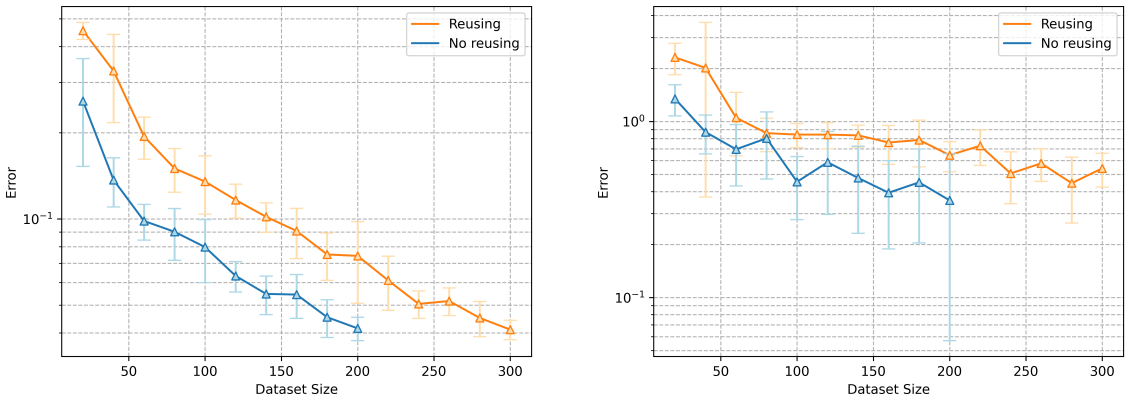
Error made in pressure $\varepsilon_{\text{ANN},p}$, and corresponding neural network performance metric $\mathcal{E}_{\text{ANN},p}$ are separate from those for velocity, defined analogously.

7.10 Neural network dataset

Having determined the reduced bases of both solution spaces, we are ready to build the data set to train neural network. The data set will consist of parameters, which are inputs to the neural network, and corresponding solutions projected on their reduced basis, which are desired outputs. As we construct this data set, we record means and variances which are used for data normalisation, to accelerate the training. We use randomly selected 70% of that dataset as training set, and the remaining part as validation set.

We can reuse the dataset already built for POD training $\mathbb{P}_h^{\text{POD}} = \mathbb{P}_h^{\text{ANN}}$, or generate a new one, at a significant computational cost. The motivation behind choosing a new one would be that training the neural network to determine reduced basis coefficients, using the exact samples that were used to construct the reduced basis, may introduce undesired effects.

To investigate the two options, we compare the performance metrics $\mathcal{E}_{\text{ANN},u}$, $\mathcal{E}_{\text{ANN},p}$ for a range of dataset sizes between the two models: one trained using the POD dataset and the other trained on a dataset separate from POD. To minimize the impact of randomness from parameters sampling and network training, each measurement is repeated five times. The results are reported in figure 22.



(a) Velocity

(b) Pressure

Figure 22: Effect of reusing the POD training set as the ANN training set.

We see that the suspicion of detrimental effect of reusing samples is confirmed, as the model trained on two separate datasets consistently outperforms the one trained on reused data, for both variables.

However, having to only generate one dataset, we can make it twice as big as the two separate datasets. For example, having two separate sets of size $M = 100$ results in a

method with $\mathcal{E}_{\text{ANN},u} \approx 0.08$, according to figure 22. If we decided to reuse a single dataset, it could have size $M = 200$, resulting in $\mathcal{E}_{\text{ANN},u} \approx 0.075$. So the negative effect of using identical samples is offset by the positive impact of having a bigger dataset. Therefore, in following experiments, the POD data set is reused for ANN training.

Both the training and validation data sets are are distributed across processes. These datasets, and other neural network related features, were first implemented directly with PyTorch, an open-source machine learning library used for developing and training neural network models [41]. These efforts gradually led to a solution structured similarly to DLRBniCSx, an open source library for deep learning based reduced order modelling in development [48]. Therefore, after confirming the authorial implementation worked, it was replaced with equivalent DLRBniCSx functions.

7.11 Neural network topology

Equipped with a dataset, we can finally construct a model to capture the relationship between parameters and solution's reduced basis coefficients. We are seeking two vectors of coefficients – one for velocity, U_{pred}, and one for pressure, P_{pred}. Therefore, the first step in constructing the model, is deciding whether it should consist of two separate networks, or a single combined one

$$\begin{cases} \mu \to U_{\text{pred}} \\ \mu \to P_{\text{pred}} \end{cases} \qquad \text{or} \qquad \mu \to [U_{\text{pred}}, P_{\text{pred}}]$$

Regardless of the choice made, next, we have to decide on topology of the network, or networks. Following [30], we use the result of [13] to limit ourselves to two hidden layers.

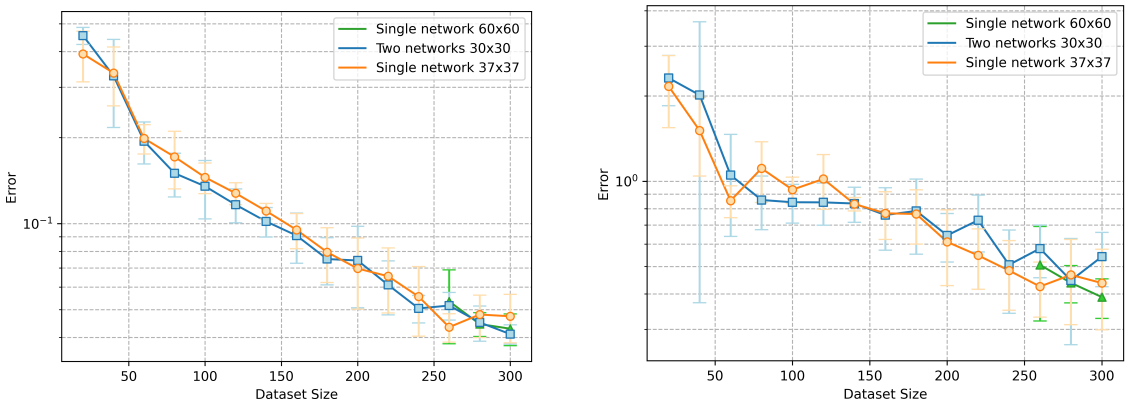
For the two separate networks model, we use identical topologies for both, each featuring 30 units in every hidden layer and a hyperbolic tangent activation function. This configuration results in approximately 2700 trainable parameters in the velocity neural network and 2135 in the pressure neural network. These numbers vary slightly as the size of the output layer changes, following the reduced basis size fluctuations. Nevertheless, they make it a very modest neural network. We owe that to the POD dimensionality reduction, without which, only the output layer would have to be over 50 times larger.

For a single model, we also use two hidden layers with same number of units, and a hyperbolic activation function. We choose the number of units in both layers as 37, to match the number of trainable parameters of two networks for a fair comparison.

We compare the two models across a number of dataset sizes using performance metrics $\mathcal{E}_{\text{ANN},u}$, $\mathcal{E}_{\text{ANN},p}$, again repeating each measurement 5 times, and present results in figure 23.

The first observation is that despite repeated measurements, results are rather noisy, particularly for pressure. Similar difficulties were reported by [30]. Nevertheless, we draw a conclusion that two separate networks performed better across the experiment. This was a surprising finding. One could expect that two separate networks are a special case of the single, combined one, with weights set such that there is effectively no connection between the two parts. Then, the performance of the broader class of neural network should be no worse than that of its subclass. It is hypothesized that the reduction in the total number of units, from 120 to 74, narrowed down the solution space by limiting the number of nonlinear functions available, negatively impacting performance. Therefore, we repeat the experiment with the number of units in each hidden layer matching the two network case, 60, but leading to significantly more model parameters to be trained. Results are better than for the combined network with 37x37 units, but not visibly better

than for the two network setup, which requires less training and less memory to store. Therefore, we proceed with two, separate neural networks.



(a) Velocity

(b) Pressure

Figure 23: $\mathcal{E}_{\text{ANN}}$ across different dataset sizes for three networks topologies: separate neural networks with 30 units in hidden layers, and combined neural network with 37 or 60 units in hidden layers.

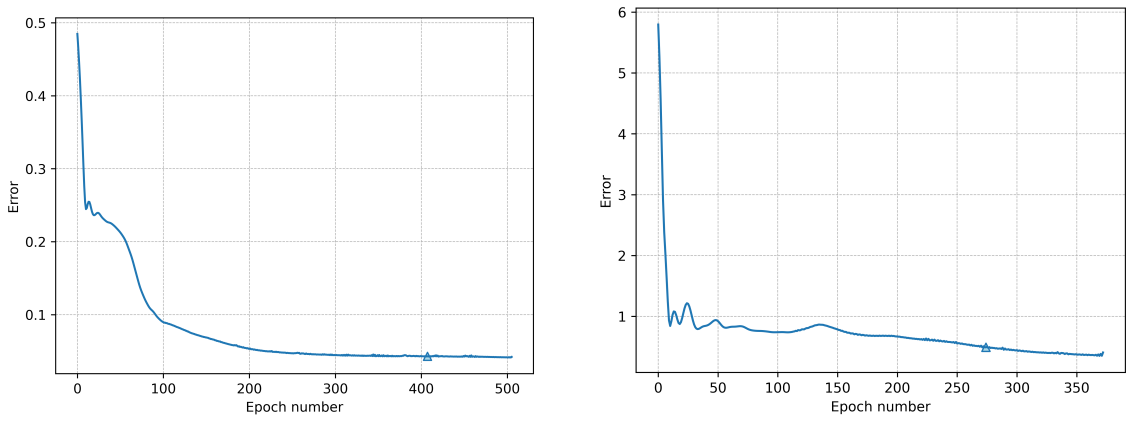
7.12 Neural network training

The loss function used in training is MSE, and we use ADAM [32] to minimise it with respect to model parameters. We distribute the network training with data parallelism – each process is assigned a subset of the training dataset, based on which it calculates the parameters update, and the update is synchronised between all processors after every batch. Batch size of 64 is used. Model parallelism was not explored in this project, as the problem is far too small to make this applicable.

We terminate the training after 4000 epochs, or after 100 epochs of no improvement in performance over the validation dataset. After termination, we set the model parameters to the configuration that performed best on the validation set.

Error evolution with training progress is presented in figure 24. Note that the plot presents the evolution of our performance metrics $\mathcal{E}_{\text{ANN}}$ rather than the validation set loss. This enables us to see the discrepancy between our metric and MSE in coefficients. For pressure, we clearly see that $\mathcal{E}_{\text{ANN},p}$ was constantly decreasing for 100 epochs after the lowest MSE was reached. Using $\mathcal{E}_{\text{ANN}}$ as loss function would have been more suitable, but would require implementing it as a custom loss function in pytorch. These figures also highlight a risk that we run into with early stopping training termination. Assuming the analogous MSE evolution is similar, if we terminated training at the first validation loss increase, we would have stopped at the first local minimum of both curves, very far from best achievable configuration. That's why in the implementation a relatively large early stopping window of a 100 is used.

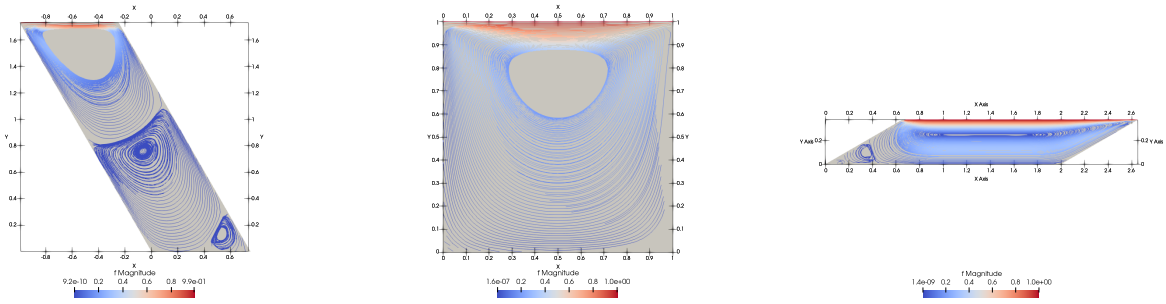
Full order solutions reconstructed from the reduced basis results of the neural networks for parameters μ_{ref}, μ_1 and μ_2 are presented in figures 27 and 31.



(a) Velocity

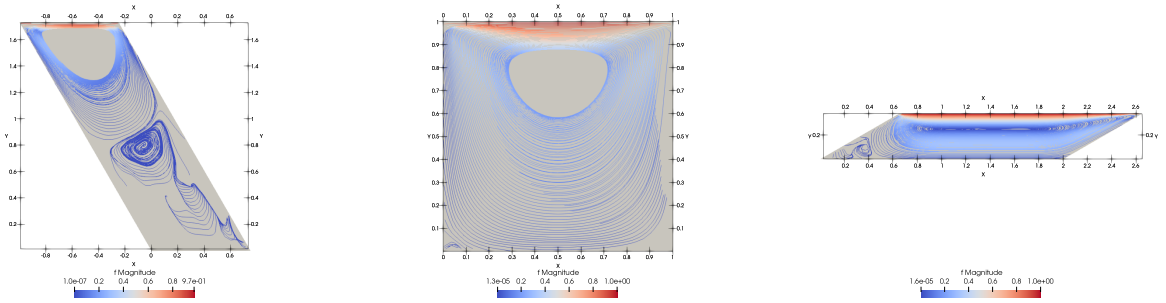
(b) Pressure

Figure 24: Evolution of $\mathcal{E}_{\text{ANN}}$ for $M = 300$ samples. The point highlighted corresponds to lowest error on the validation set.



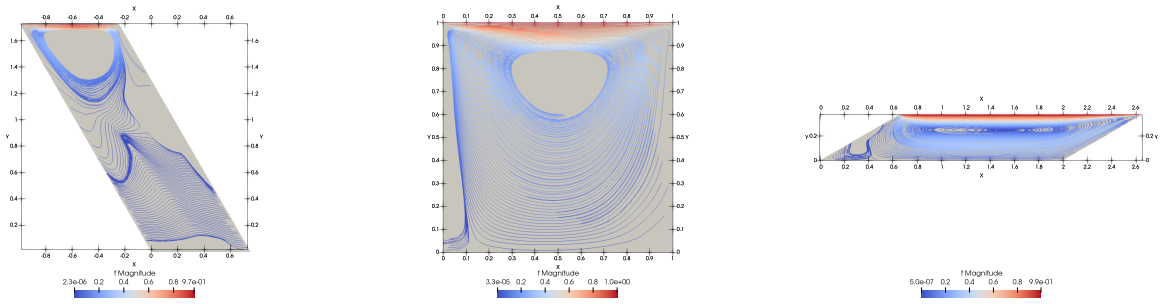
(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 25: Finite elements method velocity solution.



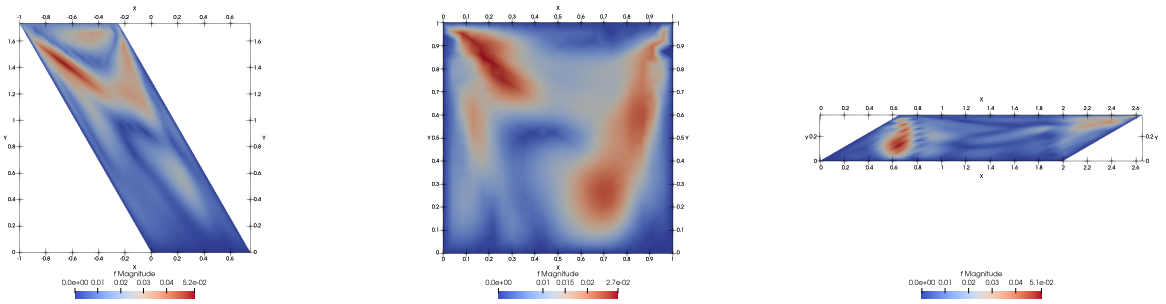
(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 26: Reconstructed reduced basis projection of FEM velocity solution.



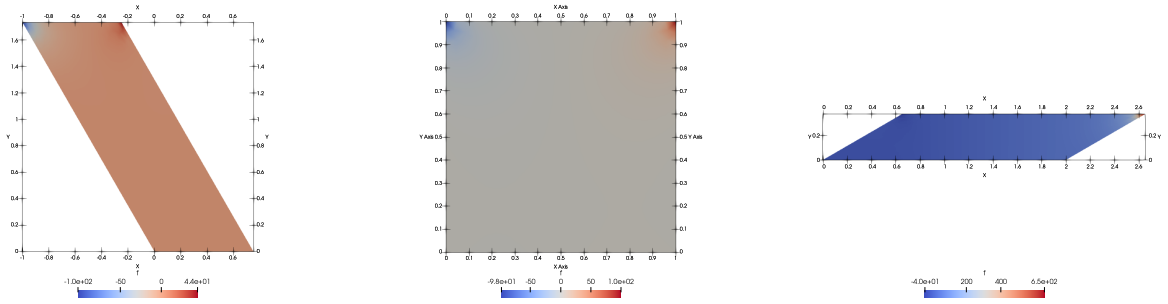
(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 27: POD-ANN method velocity output



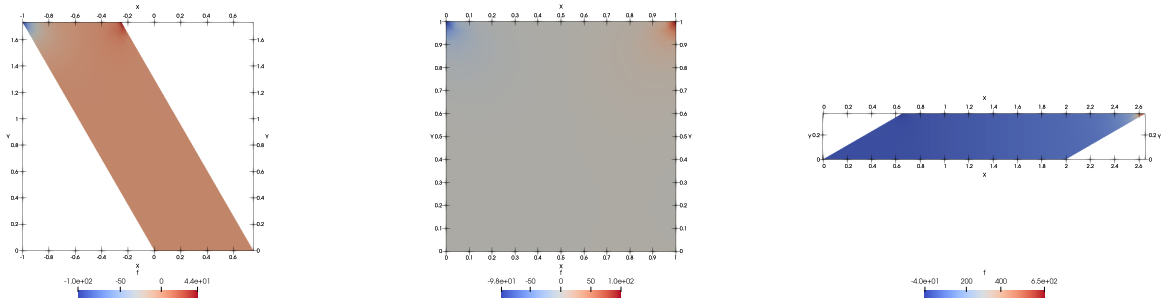
(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 28: Magnitude of velocity difference between the POD-ANN output and reconstructed reduced basis projection of FEM solution.



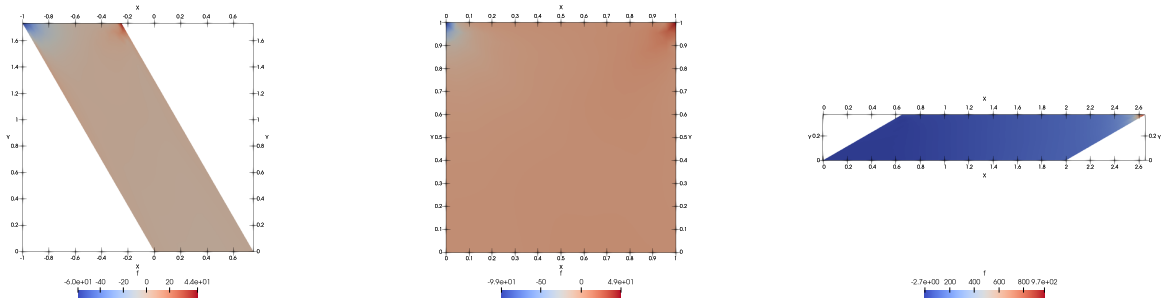
(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 29: Finite elements method pressure solutions.



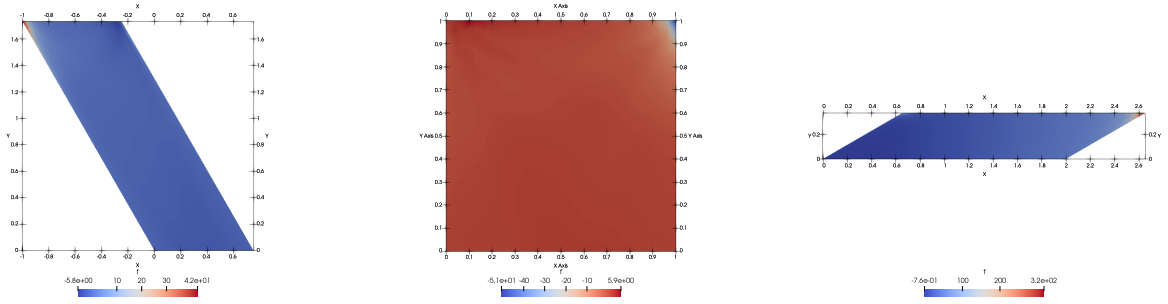
(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 30: Reconstructed reduced basis projection of FEM pressure solutions.



(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 31: POD-ANN method pressure output



(a) μ_1 (b) μ_{ref} (c) μ_2

Figure 32: Pressure difference between the POD-ANN output and reconstructed reduced basis projection of FEM solution.

7.13 Discussion

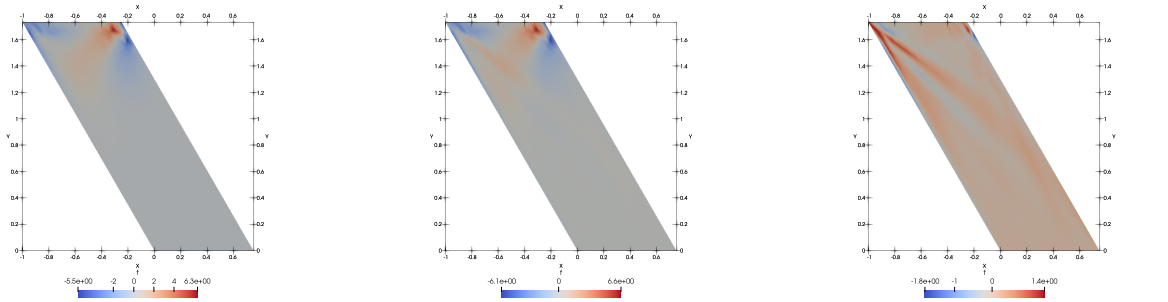
Let us take a qualitative look at figures of solutions generated. In all three cases, the method accurately identified the primary vortex, but it was less effective at capturing secondary vortices. However, looking at colour scales, we notice that the difference between the primary and secondary vortices can be as high as six orders of magnitude. The fluid is practically stationary there. Therefore, while the inaccuracies in modeling the secondary vortices can significantly affect the visual representation of the solution, they do not substantially impact the method's quantitative performance, which is summarised below,

M	$\mathcal{E}_{\mathrm{ANN},u}$	$\mathcal{E}_{\mathrm{ANN},p}$
300	0.036	0.142

Pointwise difference between neural network results and reduced basis projection of corresponding FEM solutions are shown in figures 28 and 32. We note that the maximum pointwise error for velocity is of the order of only 5%. It is acknowledged, however, that analogous results for the pressure field are much less satisfactory.

For similar hyperparameters, [30] achieves errors about an order of magnitude lower. As examples presented in this report are more extreme than those in original paper, we may hypothesise that the range of parameters sampled here is wider than that in orignal paper. For the same number of samples, that would lead to a lower samples density in our work, and, subsequently, higher error.

It was mentioned before that local divergence conformity is desired, but not easy to achieve with FEM methods. In figure 33a we show pointwise divergence for the FEM method implemented by this project. We then compare it to the pointwise divergence of our method's output, figure 33b, revealing that the result is not significantly worse than the original.



(a) $\nabla \cdot u_{\mathrm{rb}}(\mu_1)$ (b) $\nabla \cdot u_{\mathrm{pred}}(\mu_1)$ (c) $\nabla \cdot u_{\mathrm{pred}}(\mu_1) - \nabla \cdot u_{\mathrm{rb}}(\mu_1)$

Figure 33: Pointwise divergence comparison for μ_1

7.14 Timing

After verifying the accuracy of the solutions obtained through this method, we now turn our attention to the time efficiency of generating them. A key premise of the method was that querying our model for solutions during the online stage should be faster than using traditional methods. To demonstrate this, we compare the time required to obtain

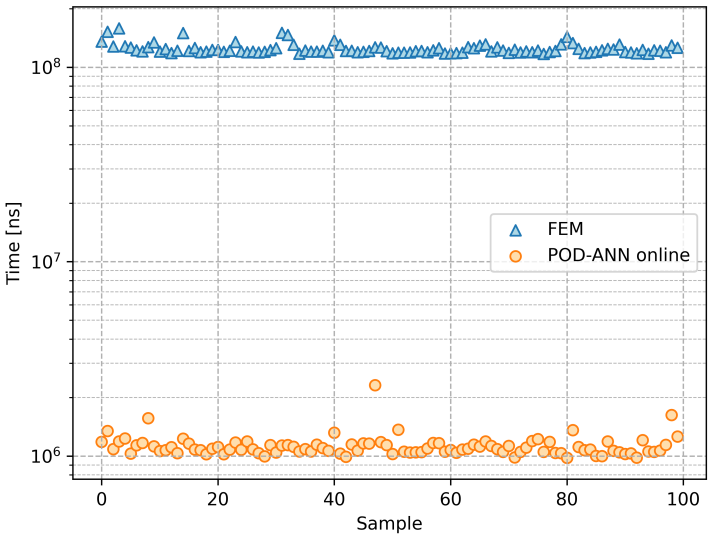


Figure 34

a solution using the POD-ANN method against that using the FEM for 100 samples, as illustrated in figure 34. The results show that our method can generate approximately 100 solutions in the same amount of time it takes for FEM to produce a single one.

While the online stage of our method delivers rapid solutions, we must be aware that it needs to be preceded by the upfront time investment in the offline phase. For instance, when tasked with solving a parametric PDE for a specific set of parameter values, we have the choice of either addressing these sequentially using traditional methods like FEM or employing our method. Starting with no pre-existing model, the time from receiving the parameters to producing results may occasionally be longer with our method than if classical methods were applied sequentially to all parameters of interest. We do our best to ensure that is rarely the case, however, it is crucial to recognize that even in such scenarios, our method still offers substantial benefits. While the overall time may not always be reduced, we strategically shift most of the computational effort to an earlier, preparatory period. This shift is particularly significant when immediate solution speed is a priority and when high-performance computing resources are available to bear the load of the offline phase.

We now investigate the efficiency of our parallel implementation of the offline stage. Our goal is to measure how long each step in the offline phase takes and to explore how these times change when varying the number of processes involved. We do that by defining 5 checkpoints, and, in each process, recording the time when each checkpoint is reached. The results are measured in two ways: using process time, and wall clock time, and both are presented in figure 35. Process time represents the combined total of system and user CPU time for the current process and does not account for time elapsed during periods of inactivity or sleep, which typically occur when processes await I/O operations, such as MPI messages. Conversely, wall clock time captures the total elapsed time, including delays caused by the activity of other processes that may be using the CPU simultaneously.

Usually timed experiments would be conducted in a much more tightly controlled environment. Here, the experiment was conducted with every effort made to minimize the influence of external factors on the results with the setup available. Other processes running on the computer were reduced to the minimum, and didn't change throughout.

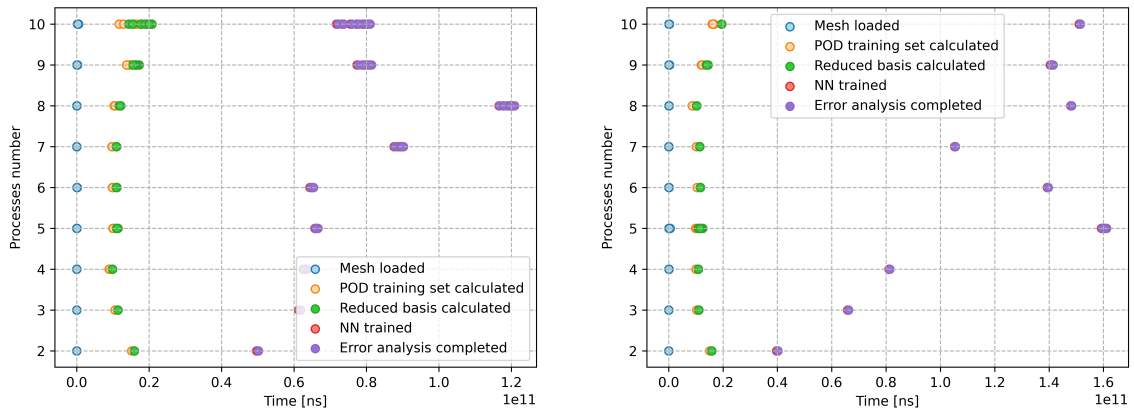
A long break was allowed between every measurement to limit influence of experiments on one another.

In our implementation, process time is measured using Python's `time.process_time_ns()`, and the wall clock time using `time.perf_counter_ns()`.

There is no clear trend in the overall length taken by the method for each case. This is mostly influenced by randomness of the neural network training, which can vary in length as much as 1000 epochs, depending on the random initialisation and the dataset. In previous experiments that effect was reduced by averaging measurements. Here, it was deliberately skipped, to show differences between individual processes within each run, and to show the extent of this random behaviour, as it is a part of the method.

Concentrating on individual parts of the method, we see that despite the noisy in neural network training time, there is a clear increasing trend with the number of processes. This indicates that the cost of synchronising the network after every batch exceeds the benefit from parallelising gradient calculations. Normally, this could be addressed by increasing the batch size, but in this case it is already large compared to the entire, modest, dataset. This hypothesis is supported by the difference in process time and wall clock time for neural network training, which increases with the number of processes, suggesting more time is spent handling I/O operations.

We also observe that the dataset generation time decreases until 4, remains roughly constant until 8, and then visibly increases again. This could be related to the fact that the processor used has four physical cores. Therefore, up until four processes, the operating system, in theory, can assign each process to a separate core, and make the optimal use of it. Beyond that, new processes can't be assigned to separate physical cores, but can be assigned to the other four logical cores within the physical cores. The two processes now share resources, for example lowest level cache, negatively impacting each others performance, offsetting the performance improvement from an increased number of processes. Finally, beyond 8 processes, some must be assigned to the same logical core, with the operating system governing how the CPU time is shared between them. Still only 8 processes can do useful work at once, but now there is an added inefficiency of swapping processes in and out, increasing the time.



(a) Process time

(b) Wall clock time

Figure 35: Time efficiency across various distribution scenarios. Every dot corresponds to a single process reaching the corresponding checkpoint at the given point in time.

7.15 DFG 2D-3 benchmark problem

The second case study for the method involves 2D incompressible flow past a cylinder within a channel, as depicted in figure 36, commonly known as the "DFG 2D-3 benchmark". The boundary at $x = 0$ is designated as an inflow, the boundary at $x = L$ as an outflow, and all remaining boundaries are subject to a no-slip condition. The velocity profile at the inflow is defined as follows:

$$u(x, y, t) = \left(\frac{4Uy(0.41 - y)}{0.41^2}, 0 \right)$$

$$U = U(t) = 1.5 \sin \left(\frac{\pi t}{8} \right)$$

Like lid driven cavity, this problem commonly serves as a benchmark in CFD. However, it involves a more complex geometry and varies over time. It is parameterised by changing

$$\mu = [L, H, c_x, c_y, r, \nu, \rho]$$

We define the reference geometry as

$$\Omega_{\text{ref}} = \Omega(\mu_g = [L = 2.2, H = 0.41, c_x = 0.2, c_y = 0.2, r = 0.05])$$

The reference mesh, displayed in figure 37, is created using Gmsh. In this case, however, the mesh is non-uniform – it is made finer in areas where more rapid variations in flow behaviour are expected, near the cylinder.

Again, we deform the reference mesh to a given geometrical configuration using MDFEniCSx. Transformations of boundaries are more involved for this problem, but have been included in appendix C. With the more complex geometry, limitations of adapted mesh deformation approach are emerging. Even for a relatively modest deformation, to parameters

$$\mu_1 = [L = 2.2, H = 1, c_x = 0.4, c_y = 0.5, r = 0.1, \nu = 0.001, \rho = 1]$$

the mesh degenerates by overlapping with itself, as illustrated in figure 38a.

To address this issue, Prof. Wells recommended dividing the deformation into smaller steps. An extension to the MDFEniCSx library, adding this feature, was implemented, and verified to work as intended. Figure 38b shows deformation to the same parameters μ_1, but conducted in three steps. We can see that the outline of the shape is not correct, and mesh overlapping has been prevented. However, zooming in, we can see that internally some cells are pretty much degenerate, which will lead to difficulties in FEM solve process. The effect generally improves with the number of steps, but the improvements are diminishing.

The full order model for this problem was implemented using FEniCSx, following [16]. Using a combination of Crank-Nicolson method and semi-implicit Adams-Bashforth approximation, the time domain $t = [0, 8)$ is discretized into 12,800 steps. We continue to use Taylor-Hood finite elements with quadratic velocity space and linear pressure space. Figure 39 presents the velocity solution obtained for the reference domain at time $t = 3.75$.

At the time of writing, the subsequent steps of the POD-ANN method have not yet been implemented for this benchmark.

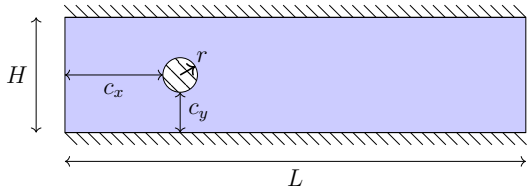


Figure 36: DFG 2D-3 problem diagram

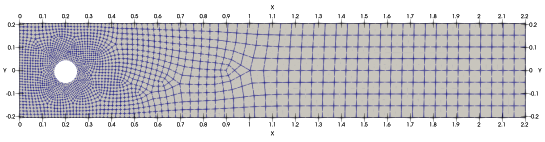
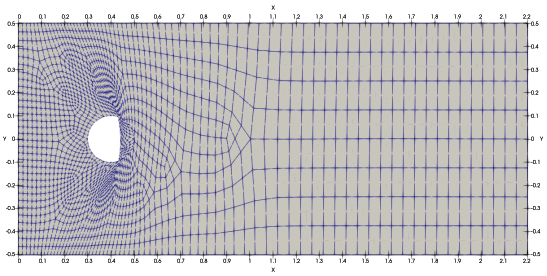
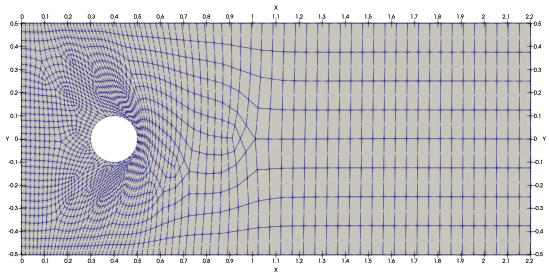


Figure 37: DFG 2D-3 problem reference mesh



(a) Mesh deformed in one step.



(b) Mesh deformation broken down into three steps.

Figure 38: DFG 2D-3 mesh deformation to parameters $\mu_1 = [L = 2.2, H = 1, c_x = 0.4, c_y = 0.5, r = 0.1, \nu = 0.001, \rho = 1]$

8 Conclusions

Throughout this project the POD-ANN method was successfully implemented, and investigated on a lid driven cavity problem. In the course of doing that, harmonic mesh deformation was used to find FEM solutions to steady Navier-Stokes equations with geometric parameters. The accuracy of solutions obtained by our POD-ANN implementation approached the levels documented in the original paper [30]. Additionally, the implementation was successfully parallelised enabling it to work across any number of distributed processes.

Each of these achievements was made possible through the integration of various open-source libraries. The implementation is portable and self-contained, allowing anyone to experiment with this method. Access details are provided in appendix D. It is hoped that this work will serve as a useful reference in future related research.

We found that reusing the same dataset for POD and ANN training has a detrimental effect on overall performance. However, this drawback is offset by the positive effect of being able to calculate twice as many samples with same resources, when the dataset is reused. A comparison between a single network and two networks model revealed that two networks achieve better performance for the same number of trainable model parameters. The assumption that two hidden layers are sufficient to model the problem turned out to be correct. We confirmed that the POD-ANN offline phase can be efficiently distributed, and showed results of it, although it is hoped that these experiments ran on

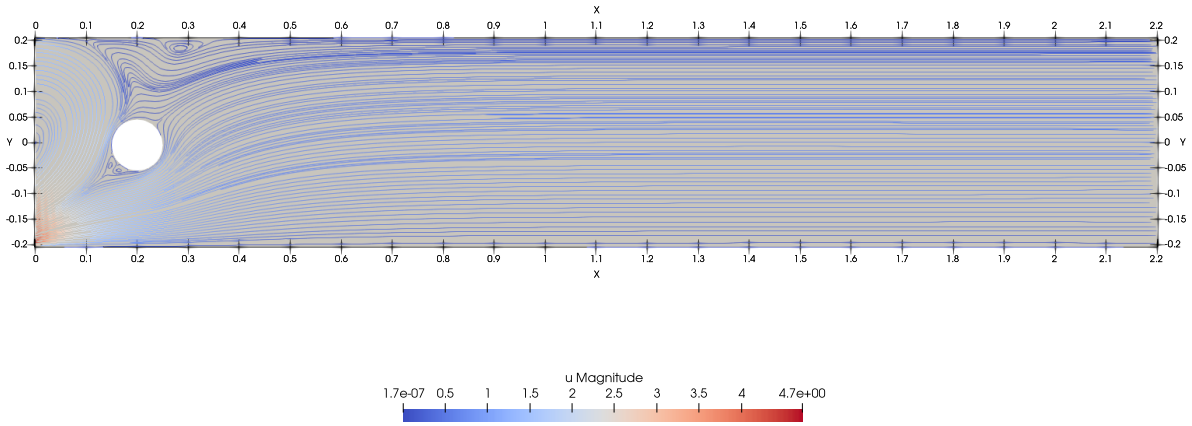


Figure 39: Velocity in DFG 2D-3 problem for reference parameters μ_{ref} at time $t = 3.75$.

a more appropriate setup would substantiate the claim more. Finally, we implemented a full order solver for the time dependent DFG 2D-3 benchmark. In doing that, we showed the mesh degeneration problem of harmonic deformation, and implemented staged mesh deformation successfully addressing it.

The investigation of this project, although simple, pictures POD-ANN as a promising method. Its capability to accurately model a manifold of solutions in a representation that can be quickly and easily queried can definitely find its applications in real time systems, which can justify the potentially large training cost. One could imagine it being trained to model the physical response of a drone, and using the compact neural network representation onboard the drone, as a part of its control system.

The efficiency of the offline stage, improved in this project by parallelisation, could be further enhanced by overlapping parameters generation with neural network training.

While a number of hyperparameters configurations were manually tested during the project, a more comprehensive exploration of the configuration space could be conducted with automated hyperparameters optimization methods to identify optimal values.

Finally, entirely different deep learning approaches to the problem can be employed. Bayesian Neural Networks can be used to provide estimates of uncertainty of results, which is particularly important in many engineering applications. [22] Another approach could involve recently introduce Kolmogorov–Arnold Networks, which instead of learning weights, learn the activation functions. They can be used to find analytic representations of the data relationships, making them easier to interpret than ANNs used in this work. [34]

Such improvements could help to move reduced order modelling forward, and hopefully contribute to its even more widespread adoption, leading to advancements like those seen in integrated circuits design in recent years.

References

[1] Navier-stokes equation. https://www.claymath.org/millennium/navier-stokes-equation/. Accessed: 2024-05-26.

[2] Rbnicsx. https://github.com/RBniCS/RBniCSx/tree/main, 2024. Accessed: 2024-05-24.

[3] S. V. Adve et al. Parallel computing research at illinois: The upcrc agenda, November 2008. Archived 2018-01-11 at the Wayback Machine (PDF).

[4] Martin S. Alnaes, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40, 2014.

[5] CSNOR AZWADI, MH Al-Mola, and S Agus. Numerical investigation on shear driven cavity flow by the constrained interpolated profile lattice boltzmann method. *WSEAS Trans Math*, 12(4):426–435, 2013.

[6] Igor A. Baratta, Joseph P. Dean, Jørgen S. Dokken, Michal Habera, Jack S. Hale, Chris N. Richardson, Marie E. Rognes, Matthew W. Scroggs, Nathan Sime, and Garth N. Wells. DOLFINx: the next generation FEniCS problem solving environment. preprint, 2023.

[7] Peter Benner, Wil Schilders, Stefano Grivet-Talocia, Alfio Quarteroni, Gianluigi Rozza, and Luís Miguel Silveira. *Model Order Reduction: Volume 2: Snapshot-Based Methods and Algorithms*. De Gruyter, 2020.

[8] Michele Benzi, Gene H Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta numerica*, 14:1–137, 2005.

[9] Gal Berkooz, Philip Holmes, and John L Lumley. The proper orthogonal decomposition in the analysis of turbulent flows. *Annual review of fluid mechanics*, 25(1):539–575, 1993.

[10] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Springer, 2024.

[11] Cambridge Service for Data-Driven Discovery. High performance computing, 2024. Accessed: 2024-05-26.

[12] Philippe G Ciarlet. *The finite element method for elliptic problems*. SIAM, 2002.

[13] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[14] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools.

[15] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[16] Jørgen S. Dokken. Dolfinx tutorial: Generating the mesh. Web page, 2023. Accessed: 2024-05-24.

[17] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[18] Patrick E. Farrell. Finite element methods for pdes. Online, 2021. Accessed on 2024-05-17. Available at https://people.maths.ox.ac.uk/farrellp/femvideos/notes.pdf.

[19] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[20] Message Passing Interface Forum. Mpi: A message-passing interface standard version 4.1. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf, November 2023.

[21] Mark Gales. Module 4f10: Deep learning & structured data. Lecture notes, 2017. Michaelmas Term.

[22] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. A survey of uncertainty in deep neural networks, 2022.

[23] Andrew Gee. Module 4f14: Computer systems. Lecture notes, 2024. Lent Term.

[24] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331, 2009.

[25] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[26] Gmsh. 3d mesh visualization of a cylinder head. https://gmsh.info/gallery/Zylkopf3D.png, 2024. Accessed: 2024-05-26.

[27] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[28] Vicente Hernandez, Jose E Roman, and Vicente Vidal. Slepc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):351–362, 2005.

[29] Jan S Hesthaven, Gianluigi Rozza, Benjamin Stamm, et al. *Certified reduced basis methods for parametrized partial differential equations*, volume 590. Springer, 2016.

[30] Jan S Hesthaven and Stefano Ubbiali. Non-intrusive reduced order modeling of nonlinear problems using neural networks. *Journal of Computational Physics*, 363:55–78, 2018.

[31] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[32] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[34] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y. Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks, 2024.

[35] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[36] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[37] Claude-Louis Navier. Mémoire sur les lois du mouvement des fluides. *Mémoires de l'Académie Royale des Sciences de l'Institut de France*, 6:389–440, 1822.

[38] Oak Ridge Leadership Computing Facility. Frontier user guide, 2024. Accessed: 2024-05-26.

[39] Oak Ridge National Laboratory. Frontier supercomputer debuts as world's fastest, breaking the exascale barrier. https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier, 2022. Accessed: 2024-01-18.

[40] Paul-Louis George Pascal Frey. *Mesh Generation*. Wiley-ISTE, 2 edition, 2008.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[42] Alfio Quarteroni, Andrea Manzoni, and Federico Negri. *Reduced basis methods for partial differential equations: an introduction*, volume 92. Springer, 2015.

[43] Alfio Quarteroni and Silvia Quarteroni. *Numerical models for differential problems*, volume 2. Springer, 2009.

[44] Wil Schilders. Model order reduction. Presented at the Mathematics Staff Colloquium, Utrecht University, 11 2009. NXP Semiconductors & TU Eindhoven.

[45] Matthew W. Scroggs, Igor A. Baratta, Chris N. Richardson, and Garth N. Wells. Basix: a runtime finite element basis evaluation library. *Journal of Open Source Software*, 7(73):3982, 2022.

[46] Matthew W. Scroggs, Jørgen S. Dokken, Chris N. Richardson, and Garth N. Wells. Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Transactions on Mathematical Software*, 48(2):18:1–18:23, 2022.

[47] Matthew W. Scroggs et al. DefElement: an encyclopedia of finite element definitions. https://defelement.com, 2024. [Online; accessed 27-May-2024].

[48] Nirav Shah. Dlrbnicsx. https://github.com/Wells-Group/dlrbnicsx/tree/main, 2024. Accessed: 2024-05-24.

[49] Nirav Shah. MDFEniCSx. github.com/niravshah241/MDFEniCSx, 2024. Accessed: 2024-01-18.

[50] Nirav Vasant Shah, Michele Girfoglio, Peregrina Quintela, Gianluigi Rozza, Alejandro Lengomin, Francesco Ballarin, and Patricia Barral. Finite element based model order reduction for parametrized one-way coupled steady state linear thermo-mechanical problems. *Finite Elements in Analysis and Design*, 212:103837, 2022.

[51] Alexander Shamanskiy and Bernd Simeon. Mesh moving techniques in fluid-structure interaction: robustness, accumulated distortion and computational efficiency. *Computational Mechanics*, 67(2):583–600, 2021.

[52] George Gabriel Stokes. On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastic solids. *Transactions*

of the Cambridge Philosophical Society, 8:287–319, 1845.

[53] Rüdiger Verfürth. Error estimates for a mixed finite element approximation of the stokes equations. *RAIRO. Analyse numérique*, 18(2):175–182, 1984.

[54] Garth N. Wells. Module 3m1: Mathematical methods - linear algebra. Lecture notes, 2023. Lent Term.

[55] Frank M. White. *Fluid Mechanics*. McGraw-Hill Series in Mechanical Engineering. McGraw-Hill Education, 8th edition in si units edition, 2017.

[56] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

A Spaces

A.1 Inner product

Inner product in a vector space V over a field F is a mapping $\langle \cdot, \cdot \rangle : V \times V \to F$ such that $\forall\, x, y, z \in V$, $\forall\, a, b \in F$

$$\begin{aligned} \langle x, y \rangle &= \overline{\langle y, x \rangle} \\ \langle ax + by, z \rangle &= a\langle x, z \rangle + b\langle y, z \rangle \\ \langle x, x \rangle &> 0 \;\forall\, x \neq 0 \end{aligned}$$

A.2 L^p spaces

For a measure space (S, Σ, μ) and a field F, L^p norm of $f : S \to F$ is defined as

$$\|f\|_p \stackrel{\text{def}}{=} \left(\int_S |f|^p \mathrm{d}\mu \right)^{\frac{1}{p}}$$

An $L^p(S)$ space is defined as all functions $f : S \to F$ with finite L^p norm

$$L^p(S) = \{ f : S \to F \quad | \quad \|f\|_p < \infty \}$$

A.3 Sobolev space

A Sobolev space $H_p^k(S)$ is defined as all functions $f : S \to F$ with finite L^p norm, and having derivatives up to k-th order with finite L^p norms

$$H_p^k(S) = \{ f : S \to F \quad | \quad \|f\|_p < \infty \quad \text{and} \quad \|D^s f\|_p < \infty, \quad \forall\; s = 1, ..., k \}$$

B Incompressible Navier-Stokes weak form derivation

The strong form of d-dimensional incompressible Navier-Stokes equations states

$$\begin{cases} \rho(\mu)\dfrac{\partial u}{\partial t} - \nu(\mu)\Delta u + \rho(\mu)(u \cdot \nabla)u + \nabla p = f \\ \nabla u = 0 \end{cases}$$

Let us introduce test function spaces for velocity and pressure respectively, as

$$V = \left\{ v \in [H^1(\Omega(\mu))]^d : v = 0 \text{ on } \Gamma_D(\mu) \right\}, \qquad Q = L^2(\Omega(\mu))$$

enabling us to multiply and integrate equations with relevant test functions

$$\begin{cases} \int\limits_{\Omega(\mu)} \rho(\mu) \dfrac{\partial u}{\partial t} \cdot v - \int\limits_{\Omega(\mu)} \nu(\mu) \Delta u \cdot v + \int\limits_{\Omega(\mu)} \rho(\mu)(u \cdot \nabla) u \cdot v + \int\limits_{\Omega(\mu)} \nabla p \cdot v = \int\limits_{\Omega(\mu)} f \cdot v \quad \forall\, v \in V \\ \int\limits_{\Omega(\mu)} \nabla u \cdot q = 0 \quad \forall\, q \in Q \end{cases}$$

We apply integration by parts to decrease the gradient degree on u and p

$$-\int\limits_{\Omega(\mu)} \nu(\mu) \Delta u \cdot v = \int\limits_{\Omega(\mu)} \nu(\mu) \nabla u \cdot \nabla v - \int\limits_{\Omega(\mu)} \nu(\mu) \nabla (\nabla u \cdot v)$$

$$\int\limits_{\Omega(\mu)} \nabla p \cdot v = -\int\limits_{\Omega(\mu)} p \nabla \cdot v + \int\limits_{\Omega(\mu)} \nabla (p \cdot v)$$

and the Gauss' divergence theorem

$$\int\limits_{\Omega(\mu)} \nu(\mu) \nabla (\nabla u \cdot v) = \int\limits_{\partial\Omega(\mu)} \nu(\mu) \frac{\partial u}{\partial \hat{n}} \cdot v$$

$$\int\limits_{\Omega(\mu)} \nabla (p \cdot v) = \int\limits_{\partial\Omega(\mu)} p v \cdot \hat{n}$$

leading to

$$\begin{cases} \displaystyle\int\limits_{\Omega(\mu)} \rho(\mu) \frac{\partial u}{\partial t} \cdot v + \int\limits_{\Omega(\mu)} \nu(\mu) \nabla u \cdot \nabla v + \int\limits_{\Omega(\mu)} \rho(\mu)(u \cdot \nabla) u \cdot v - \int\limits_{\Omega(\mu)} p \nabla \cdot v \\ \qquad\qquad = \displaystyle\int\limits_{\Omega(\mu)} f \cdot v + \int\limits_{\partial\Omega(\mu)} \left(\nu(\mu) \frac{\partial u}{\partial \hat{n}} - p \hat{n} \right) \cdot v \qquad \forall\, v \in V \\ \displaystyle\int\limits_{\Omega(\mu)} \nabla u \cdot q = 0 \quad \forall\, q \in Q \end{cases}$$

Because the test function v was chosen to vanish on Dirichlet boundaries, and considering the Neumann condition, we can write

$$\int\limits_{\partial\Omega(\mu)} \left(\nu(\mu) \frac{\partial u}{\partial \hat{n}} - p \hat{n} \right) \cdot v = \underbrace{\int\limits_{\Gamma_D(\mu)} \left(\nu(\mu) \frac{\partial u}{\partial \hat{n}} - p \hat{n} \right) \cdot v}_{v = 0 \text{ on } \Gamma_D(\mu)} + \int\limits_{\Gamma_N(\mu)} \underbrace{\left(\nu(\mu) \frac{\partial u}{\partial \hat{n}} - p \hat{n} \right)}_{= h \text{ on } \Gamma_N(\mu)} \cdot v = \int\limits_{\Gamma_N(\mu)} h \cdot v$$

arriving at

$$\begin{cases} \displaystyle\int_{\Omega(\mu)} \rho(\mu)\frac{\partial u}{\partial t}\cdot v + \int_{\Omega(\mu)} \nu(\mu)\nabla u\cdot\nabla v + \int_{\Omega(\mu)} \rho(\mu)(u\cdot\nabla)u\cdot v - \int_{\Omega(\mu)} p\nabla\cdot v \\ \qquad\qquad = \displaystyle\int_{\Omega(\mu)} f\cdot v + \int_{\Gamma_N(\mu)} h\cdot v \qquad \forall\, v\in V \\ \displaystyle\int_{\Omega(\mu)} \nabla u\cdot q = 0 \quad \forall\, q\in Q \end{cases}$$

which is the weak form of incompressible Navier-Stokes, required for FEM. [43]

C DFG 2D-3 benchmark problem boundary transformations

$$\begin{bmatrix} x \\ y \end{bmatrix}(\mu) = \begin{bmatrix} x_{\text{ref}} \\ y_{\text{ref}} \end{bmatrix} + \begin{bmatrix} 0 \\ y_{\text{ref}}\frac{H(\mu)}{H(\mu)-H_{\text{ref}}} \end{bmatrix}, \quad \forall\,(x,y)\in\partial\Omega_{\text{inlet}}$$

$$\begin{bmatrix} x \\ y \end{bmatrix}(\mu) = \begin{bmatrix} x_{\text{ref}} \\ y_{\text{ref}} \end{bmatrix} + \begin{bmatrix} x_{\text{ref}}\frac{L(\mu)}{L(\mu)-L_{\text{ref}}} \\ y_{\text{ref}}\frac{H(\mu)}{H(\mu)-H_{\text{ref}}} \end{bmatrix}, \quad \forall\,(x,y)\in\partial\Omega_{\text{outlet, wall}}$$

$$\begin{bmatrix} x \\ y \end{bmatrix}(\mu) = \begin{bmatrix} x_{\text{ref}} \\ y_{\text{ref}} \end{bmatrix} + \begin{bmatrix} c_x(\mu) - c_{x,\text{ref}} \\ c_y(\mu) - c_{y,\text{ref}} \end{bmatrix} + \left(\frac{r(\mu)}{r_{\text{ref}}} - 1\right)\begin{bmatrix} x_{\text{ref}} - c_{x,\text{ref}} \\ y_{\text{ref}} - c_{y,\text{ref}} \end{bmatrix}, \quad \forall\,(x,y)\in\partial\Omega_{\text{obstacle}}$$

D Code access

All source code developed for this project can be found in the GitHub repository linked below.
https://github.com/jjderlatka/iib-project

E Risk assessment retrospective

No new risks were identified during the project, and the risks outlined in the initial risk assessment were successfully mitigated.