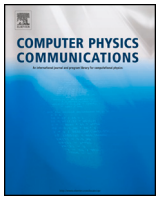
Computer Physics Communications 307 (2025) 109391

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc



Neural-Parareal: Self-improving acceleration of fusion MHD simulations using time-parallelisation and neural operators ☆

S.J.P. Pamela [a,*], N. Carey [a], J. Brandstetter [b,c], R. Akers [a], L. Zanisi [a], J. Buchanan [a], V. Gopakumar [a], M. Hoelzl [d], G. Huijsmans [e,f], K. Pentland [a], T. James [a], G. Antonucci [a] and the JOREK Team [g,1]

[a] CCFE, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK
[b] ELLIS Unit, Linz, LIT AI Lab, Institute for Machine Learning, Johannes Kepler University, Linz, Austria
[c] NXAI GmbH, Austria
[d] Max-Planck Institute for Plasma Physics, 85748 Garching, Germany
[e] CEA, IRFM, F-13108 Saint-Paul-lez-Durance, France
[f] Eindhoven University of Technology, 5612 AZ Eindhoven, the Netherlands
[g] EUROfusion Consortium, JET, Culham Science Centre, Abingdon, OX14 3DB, UK

ABSTRACT

The fusion research facility ITER is currently being assembled to demonstrate that fusion can be used for industrial energy production, while several other programmes across the world are also moving forward, such as EU-DEMO, CFETR, SPARC and STEP. The high engineering complexity of a tokamak makes it an extremely challenging device to optimise, and test-based optimisation would be too slow and too costly. Instead, digital design and optimisation must be favoured, which requires strongly-coupled suites of multi-physics, multi-scale High-Performance Computing calculations. Safety regulation, uncertainty quantification, and optimisation of fusion digital twins is undoubtedly an Exascale grand challenge. In this context, having surrogate models to provide quick estimates with uncertainty quantification is essential to explore and optimise new design options. But there lies the dilemma: accurate surrogate training first requires simulation data. Extensive work has explored solver-in-the-loop solutions to maximise the training of such surrogates. Likewise, innovative methods have been proposed to accelerate conventional HPC solvers using surrogates. Here, a novel approach is designed to do both. By bootstrapping neural operators and HPC methods together, a self-improving framework is achieved. As more simulations are being run within the framework, the surrogate improves, while the HPC simulations get accelerated. This idea is demonstrated on fusion-relevant MHD simulations, where Fourier Neural Operator based surrogates are used to create neural coarse-solver for the Parareal (time-parallelisation) method. Parareal is particularly relevant for large HPC simulations where conventional spatial parallelisation has saturated, and the temporal dimension is thus parallelised as well. This Neural-Parareal framework is a step towards exploiting the convergence of HPC and AI, where scientists and engineers can benefit from automated, self-improving, ever faster simulations. All data/codes developed here are made available to the community.

1. Introduction

1.1. Motivation

Solving non-linear systems of partial differential equations is a field of research that has applications in a wide range of scientific and engineering problems. In the aerospace and automotive industries, in weather and climate predictions, in fusion energy research, countless numerical solvers are being used routinely to predict the evolution of complex physical systems. Conventional PDE solvers are constantly being developed as part of scientific research (MOOSE, MFEM, Firedrake, OpenFoam, JOREK) [1–8] as well as industrial tools (ANSYS, ABAQUS, SIEMENS) [9–11]. Modern PDE solvers are typically parallelised on the spatial domain they address, but in the case of fully implicit solvers, which have strong numerical stability advantages, this typically results in large matrix inversions with preconditioners. These typically do not scale well on large High-Performance Computing (HPC) systems with GPU accelerators due to memory limits and bandwidth. Methods to further parallelise conventional solvers have been explored in recent decades including, among others, parallel-in-time methods such as Parareal [12] and deep learning methods such as neural operators [13–15]. While these emerging methods often lack the precision of the

☆ The review of this paper was arranged by Prof. Andrew Hazel.

* Corresponding author.
E-mail address: stanislas.pamela@ukaea.uk (S.J.P. Pamela).

[1] See author list of M. Hoelzl et al., Nuclear Fusion 61 (2021) 065001.

https://doi.org/10.1016/j.cpc.2024.109391
Received 1 May 2024; Received in revised form 13 August 2024; Accepted 6 October 2024
Available online 15 October 2024

underlying conventional solvers they exploit, their efficacy and practical relevance is highly dependent on the use-case of interest. Their value lies in providing fast approximations of the detailed computation, which is of interest for wider integrated digital engineering tools and digital twins [16].

In fusion research, the design (and design optimisation) of new tokamak and stellerator devices requires a wide range of HPC calculations, to be integrated in a coupled workflow, that may require several steps to converge to a final solution. Some of these individual components needed for fusion power plant designs are themselves integrated workflows comprising of several HPC codes. For example, the blankets around a tokamak plasma, which will be used to breed tritium from the fusion-born neutrons and extract their energy into a cooling system, require neutronics calculations with codes like OpenMC or MCNP [17–19], coupled to fluid mechanics [20] or even liquid-metal Magnetohydrodynamics (MHD) [21]. However, the material and mechanical properties of these blankets are also strongly dependent on the heat-fluxes that result from plasma turbulence at the plasma edge [22,23], which eventually requires integrated simulations from a plasma flight simulator like JINTRAC [24], which couples key characteristics of the plasma dynamics, such as the Grad-Shafranov equilibrium, turbulent pressure transport fluxes, deposition of various heating and fueling systems, MHD stability limits, and Scrape-Off Layer kinetic simulations. Breeding blankets and plasma dynamics are just two examples of the complex coupled system required to obtain a fully consistent digital design or digital twin of tokamak devices. Having the ability to accelerate some of these components, even to obtain an initial guess, can enable engineers to quickly explore a wider range of configurations in order to optimise the design of future machines.

The work proposed here combines two independent methods to provide a novel approach of accelerating conventional solver approximations. Namely, using neural operators as the coarse solvers required by the Parareal method [12]. The convergence of High-Performance-Computing and Artificial Intelligence is illustrated by this approach, where the training of the neural operator is bootstrapped into the large data-production feature of the Parareal method. As more simulations are produced and more data becomes available, the neural coarse solver gains accuracy, while the Parareal time-parallelisation gains speed-up efficiency. To demonstrate this self-improving approach, a generic fusion application is chosen, using a set of MHD equations in toroidal geometry, where filamentary blob structures are evolved inside a 2D slab domain. The outcome of this framework is threefold: 1. the speed-up provided by the Parareal algorithm is increased by improving the accuracy of the coarse-solver, 2. a fast, accurate coarse-solver is obtained which can be used as surrogate inside other workflows to accelerate estimates of calculations, and 3. the accuracy of the coarse-solver can be defined by how fast the Parareal simulations converge, providing an uncertainty quantification of the surrogate.

1.2. Current research on parareal and neural operators

The current research on parallel-in-time methods is a wide field of science and the Parareal method [12] is only one of its branches. The particularity of the Parareal method is that it is relatively straightforward to understand and implement, with a non-intrusive implementation for the numerical tool, although in practice it requires the developer to understand in detail the i/o of the code in question, while adding a non-negligible layer of orchestration scripts for end users to deal with. In this work, the code chosen for the demonstration uses Finite-Element Methods (FEM), which makes this aspect of the Parareal implementation more intrusive, as will be explained in detail in further sections. Applications of Parareal methods have been achieved in various fields of numerical studies, including Molecular Dynamics [25], fluid dynamics [26], geodynamics [27], as well as fusion [28].

The Parareal approach, although abstract, is relatively simple. A good introduction to Parareal can be found in [29]. It consists of splitting a time-domain into multiple *time-windows*, and evolving each window concurrently. A first rapid estimate is done across all time-windows with a so-called *coarse-solver*, which should have a negligible execution time compared to the full simulation code (often called the *fine-solver*), which is run for each time-window in parallel, starting from the estimate provided by the coarse solver. The same procedure is repeated at each Parareal *cycle*, where the initial-value for each time-window is calculated using a *predictor-corrector* scheme, which combines the result of the previous time-window's coarse solution and the previous time-window's coarse and fine solutions from the previous iteration cycle. The Parareal algorithm is illustrated in further details in Appendix A and in [30].

The closest work that the authors are aware of, and similar to the study presented here, is that of Gorynina et al. [31], where a Machine Learning surrogate is used as coarse solver for molecular dynamics, and more recently, of notable interest, the work by Qadir-Ibrahim et al. [32], where a Physics-Informed version of the FNO [13] algorithm (PINO [33]) is used as the coarse solver. Another relevant study is that of Pentland et al. [34], where a Gaussian Process is used to learn the difference (i.e. the predictor-corrector) between a given coarse-solver and the fine-solver. The way the work described here differs from previous achievements is both the application, which is fusion-specific with a set of highly non-linear PDEs (MHD), and most importantly the demonstration of how the training of the Machine Learning coarse solver can be integrated into the Parareal workflow to produce a more accurate coarse solver as the number of simulations increases. This framework integration can be compared to Solver-in-the-Loop methods [35,36], where the simulation code is integrated inside the neural operator training, except that it does the reverse: here the training of neural operators is integrated inside the HPC simulation algorithm. This 'AI-in-the-Loop' concept is illustrated in Fig. 1. It may be compared to methods where AI acceleration is used for preconditioning of simulations, such as [37]. This convergence of HPC and AI methods is most relevant to the development of Digital Twins and Digital Models for fusion research, where the need for rapid designs of future fusion power plants requires quick yet accurate estimations of costly and slow HPC simulations using Machine Learning surrogates.

1.3. Fusion application

In the current alarming climate change situation [38], nuclear fusion could provide an abundant energy source with a minimal level of greenhouse gas emissions and no long-lived radioactive nuclear waste. Together with renewable energies, fusion could contribute to the electricity of future societies, without the limit of exhaustible natural resources. Currently, the most promising candidate for industrial fusion reactors is the tokamak device [39], which uses a magnetic field to confine a hot plasma of ionised hydrogen isotopes. The toroidal, periodic nature of the tokamak ensures that the hydrogen ions and the electrons, which approximately follow the magnetic field lines, are not lost at the end of open field lines, like in linear plasma devices. However, this periodicity can lead to resonance and instability. Resonant and unstable modes typically involve the plasma and the magnetic field, and are commonly studied using MHD models [40–42], combining the Navier–Stokes equations with Maxwell's equations.

Theoretical analysis of the MHD equations can provide some limited insight into the properties of various waves and unstable modes in a tokamak [40], however to obtain a more detailed understanding of tokamak MHD instabilities, numerical simulations are required. Some of the main tokamak MHD instabilities include Edge-Localised-Modes (ELMs), Toroidal Alfven Eigenmodes (TAEs) and global instabilities (Disruptions). ELMs eject plasma filaments from the edge region onto the first wall of the machine, leading to large heat-fluxes on surface materials [43–46]. TAEs, which are excited by the 3.5 MeV alpha-particles born of fusion reactions, can limit the performance of plasma operations [47–50]. Global MHD instabilities which affect the entire plasma can lead to the total loss of plasma control, these are called disruptions.

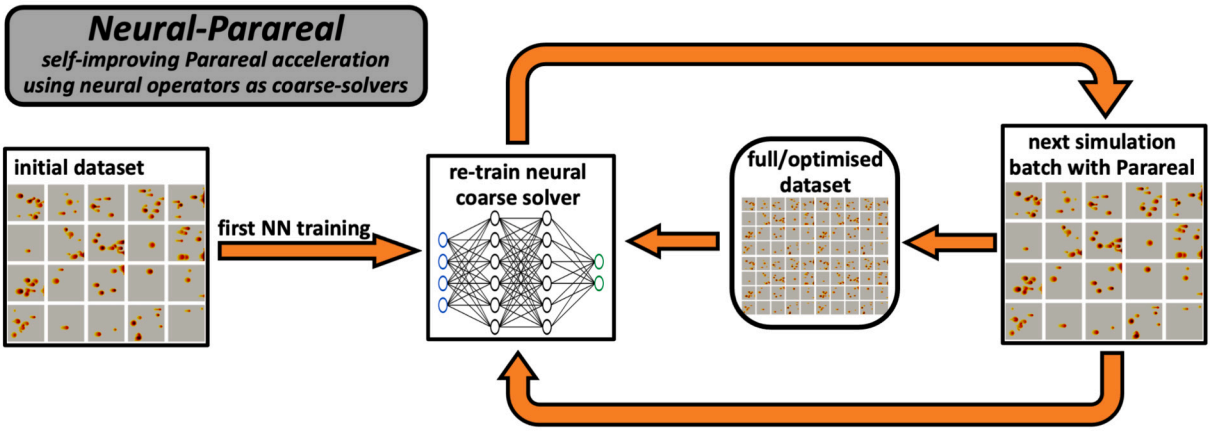


Fig. 1. The self-improving Neural-Parareal framework, where neural operators are trained live as more simulation data becomes available, providing a progressively more accurate coarse-solver, leading to higher speedup efficiency as simulations are produced.

During disruption events, the kinetic and magnetic energy of the plasma can be transferred to the wall, leading to unsustainable material heat-fluxes and/or wall-currents that can damage the structural components of the machine [51–56]. In order to study, understand and predict these MHD instabilities, numerical simulations are performed using codes like JOREK [7,8,57,58], M3D-C1 [59,60], NIMROD [61,62], XTOR [63], BOUT++ [64,65], MEGA [66–68], HALO [49] (and many others).

In this study, we use the Reduced-MHD equations [69,70] in a 2D slab geometry with toroidal curvature (toroidally axisymmetric domain). The simulations are evolving filamentary blobs similar to ELM filaments at the plasma edge. Although the geometry is simplified, the physics model and type of dynamics is similar to state-of-the-art applications of the JOREK code [8,71], and therefore represents a practical demonstration from which future extensions of the framework could be developed to address realistic tokamak use cases. Note that blob convection in tokamaks is also extensively studied in the context of electrostatic turbulence, such as in [72,73].

1.4. Overview of the work

In this paper, we present an integrated framework that combines Parareal simulations of the JOREK code [7] with the training of neural operators in PDEarena [74,75], bootstrapped into the workflow to benefit from the large data-generation of the Parareal algorithm in real time. This integration results in progressively more and more accurate coarse solvers as the input-parameter domain is explored with new simulations, and thus higher speed-up efficiency. Section 2 introduces the MHD simulation use cases that were used for the development of the Parareal framework and the initial development of the Neural Operator surrogates. Section 3 presents the work done with PDEarena [74,75] to create surrogate models of the simulations, which are used as the coarse solvers of the Parareal framework. Section 4 describes the full implementation of the Parareal framework which accommodates the FEM discretisation of JOREK, the neural coarse solvers from PDEarena, and the non-negligible parallel i/o processing required for an efficient framework. Finally, Section 5 presents the main results of the framework while Section 6 summarises the work and lays out the further improvements desirable for future studies and extensions.

2. The MHD simulation use cases

In this study, two use cases are addressed, both simulating blob convection in a 2D slab domain with toroidal axisymmetric geometry. The first use case employs a simplified electrostatic model, which already has a large dataset published in several studies [76,77]. This use case was employed for the development of the Neural-Parareal framework and initial tests that the performance was reasonable.

The second use case is a new dataset based on similar blob simulations but with a more complex MHD model, the so-called Reduced-MHD model, which has been used extensively in literature for the study of tokamak instabilities [8,71]. This use case was employed to demonstrate the integrated framework with the dynamic training of the neural coarse-solver, bootstrapped inside the workflow to exploit the data generated by new simulations.

In both use cases, the simulations are run with a bi-cubic (high-order) C1-continuous Bezier finite-element grid with uniform resolution of 200 by 200 elements. The poloidal 2D slab is centred at a toroidal major radius of $10m$ with height and width of $1m$. The time-step size is approximately 0.15 μs. Both spatial and temporal resolutions are voluntarily chosen to be conservatively high (fine) to ensure numerical stability across the entire input-parameter domain. For all simulations, 2000 time-steps are run. The boundary conditions around the domain are Dirichlet for all variables.

2.1. Electrostatic blob simulations

The first use case employs an electrostatic model, which is equivalent to the Reduced-MHD model as routinely used in JOREK [8], but without the magnetic potential and current. There are a total of 4 variables in the model: 3 physical variables, plus an auxiliary variable used for numerical stability (see [8]). These physical variables are the fluid density ρ, the fluid temperature T and the electric potential Φ. The auxiliary variable is the toroidal vorticity, defined as $\omega = \nabla^2\Phi$. Note that the Laplacian here is in toroidal coordinates. This model is very similar to the Navier-Stokes equations, where Φ can be associated to the stream function of the fluid velocity. The exact formulation of the velocity is given, as in [8], with $\vec{v} = R^2\nabla\phi \times \nabla\Phi$, where R is the major radius, and ϕ is the toroidal coordinate. One can easily derive that the toroidal vorticity is in fact simply $\omega = \nabla\phi \cdot (\nabla \times \vec{v})$.

The simulations are initialised with multiple blobs inside the slab, varying randomly the number of blobs, their positions, their (2D-Gaussian) width, their density amplitude and their temperature amplitude. The range of these input parameters is as follows:

input parameter	min/max	type & unit
number of blobs	[1 : 10]	discrete
R-position of blobs	[9.6 : 10.4]	continuous $[m]$
Z-position of blobs	[-0.4 : 0.4]	continuous $[m]$
width of blobs	[0.02 : 0.1]	continuous $[m]$
density amplitude of blobs	[0.1 : 0.4]	continuous $[10^{20}m^{-3}]$
temperature amplitude of blobs	[12 : 72]	continuous $[eV]$

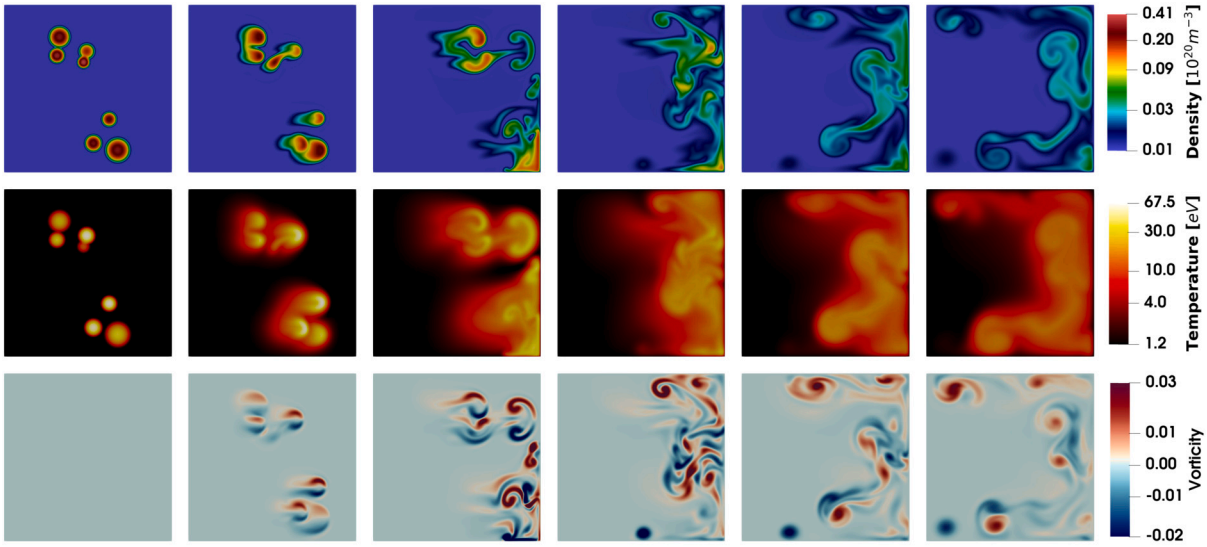


Fig. 2. The first use case with an electrostatic Reduced-MHD model, showing one of the simulations with blobs initialised in a poloidal 2D slab domain. The first 1000 time-steps are illustrated here (half of the full simulation), with each frame corresponding to approximately 30 μs, i.e. [0,30,60,90,120,150] μs.

These quantities are representative of small filamentary blobs in the Scrape-Off Layer of a tokamak plasma, i.e. just outside the hot, confined plasma region. In a tokamak, such filamentary structures are expelled from the confined region due to turbulence and/or MHD instabilities. As a reference, the electron density and temperature in the JET-ILW tokamak, just at the edge of the confined plasma region, is typically of the order of $5.10^{19}m^{-3}$ and $1keV$.

The electric potential Φ is initialised as zero. As the simulation starts, the toroidal curvature combined with the pressure gradient of the blobs generates an electric field that leads the blobs to move radially outwards (away from the centre of the torus). The hotter the blob, the faster its motion. The Dirichlet boundary conditions cause the blob material to mix inside the slab until they dissipate through diffusion.

The poloidal diffusion parameters are the density diffusion $D = 3.5m^2.s^{-1}$, the temperature diffusion $\kappa = 2.10^{-7}kg.m^{-1}.s^{-1}$, and the viscosity $\mu = 2.10^{-6}kg.m^{-1}.s^{-1}$. For more details on the Reduced-MHD model and its parameters, see [8]. Fig. 2 shows an example of a simulation with 7 blobs moving radially towards the outer boundary of the domain. Note that only half the simulation (1000 time-steps) is shown in Fig. 2 for illustration purposes. All datasets can be downloaded from Zenodo, including GIFs [78–82]. The simulations (2000 time-steps) take just above 6 hours to run on a 2x24-cores Intel-Xeon-8160 (SkyLake) node. More information about the existing (spatial) parallelisation of the JOREK code can be found in [8].

2.2. Electromagnetic blob simulations

The second use case is exactly the same as above but with the more complex Reduced-MHD model, as implemented in routine JOREK studies, which includes a magnetic field but without the (optional) parallel velocity [8]. This model has the same variables ρ, T, Φ and ω as before, with an extra physical variable for the poloidal magnetic potential ψ, and an additional auxiliary variable for the toroidal plasma current density, defined as $j = R\nabla \cdot (R^{-1}\nabla\psi)$. The magnetic field is given, as in [8], by $\vec{B} = \vec{B}_\phi + \vec{B}_{pol}$, where the toroidal magnetic field $\vec{B}_\phi = F_0\nabla\phi$ is constant in time in the Reduced-MHD model, and only the poloidal magnetic field $\vec{B}_{pol} = \nabla\psi \times \nabla\phi$ is evolved through the scalar potential variable ψ. Note that this is why the model is called *Reduced*-MHD, as opposed to *full*-MHD where the toroidal field also evolves [71]. In this set-up, one can easily derive that the toroidal current is in fact simply $j = R^2\nabla\phi \cdot (\nabla \times \vec{B})$.

In the simulations presented here, the toroidal magnetic field is set to $1T$, and the poloidal magnetic field is set to a background of $10^{-3}T$ to represent the Scrape-Off Layer of a tokamak, just outside the confined plasma, where there the poloidal magnetic field is much lower than inside the confined plasma region. The poloidal magnetic field actually vanishes in the so-called X-point (saddle point) region of the plasma.

No current is initialised inside the blobs, but as can be seen in Fig. 3, as the blobs start evolving, they generate their own current, which affects their dynamics and bends the magnetic field. This is illustrated by the bottom row of the figure, which shows the current scalar together with contour lines of ψ (which can be considered as the stream function of the poloidal magnetic field). The resistivity in these simulations is relatively high, at $4.10^{-5}\Omega.m$, such that the plasma fluid is not 'frozen' to the magnetic field and can travel through flux surfaces. Still, the field-line bending and current have a significant effect on the dynamics of the blobs, which can be clearly seen when comparing Fig. 3 to Fig. 2 which has exactly the same initial conditions. In particular, with the Reduced-MHD model, the blobs can be observed to travel slower and deviate up/down-wards compared to the simpler model where they just travel radially outward until reaching the outer boundary.

With this Reduced-MHD model, the simulations (2000 time-steps) take just under 14 hours to run on a 2x24-cores Intel-Xeon-8160 (SkyLake) node. Note that in a fully implicit time-scheme, the problem (matrix) size scales as the number of variables squared, so the factor 2 in computation increase between the two models corresponds to $6^2/4^2 = 2.25$.

3. Neural operator surrogates

3.1. PDEarena using FNO method

The initial dataset using the electrostatic model from Section 2.1 was used for previously published studies in [76,77]. It comprises of 2000 simulations [79–82]. From these samples, 90% was used for training, and 10% for testing. In order to ensure numerical stability over the entire input-parameter domain, overly conservative resolution was used in the simulations. Therefore, the data is down-sampled both spatially and temporally before ingestion into the neural operator training. The spatial resolution is set to 100×100, while the temporal frequency is reduced by a factor 10, hence 200 time-frames per simulation. Note that this particular choice of down-sampling is arbitrary, and a more extensive study would require an additional investigation to identify the optimal down-sampling level. The main advantage of having higher resolution for the neural operator is precision, the disadvantage is longer training times (currently of the order of 72 hours for these settings) and

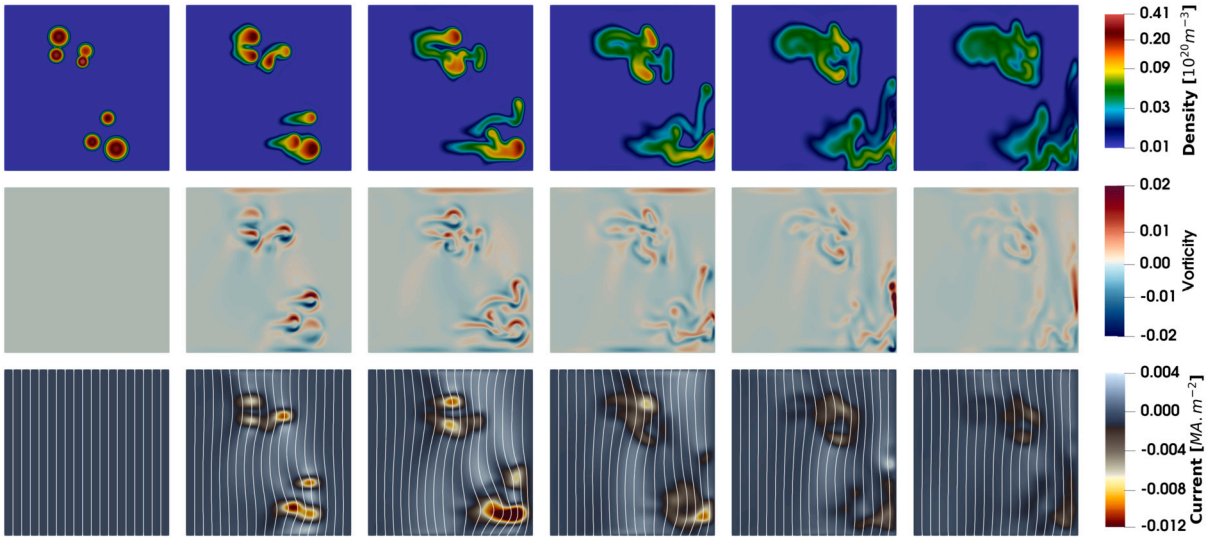


Fig. 3. The second use case with an electromagnetic Reduced-MHD model, showing a simulation with the same initialisation as in Fig. 2. The first 1000 time-steps are illustrated here as well, each frame corresponding to approximately 30 μs. It is worth noting the difference between the electromagnetic model and the electrostatic model: the blobs move slower and deviate up/down-wards, due to the interaction with the background magnetic field. In the bottom row, the colour shows the current generated by the filaments, as well as contours of the magnetic potential ψ, to illustrate the bending of the magnetic field by the blobs.

larger model files (currently 1.6 GB). For the demonstrative purpose of this study, such a down-sampling is appropriate. All variables are normalised to [-1:1] with respect to the minima/maxima of the entire dataset.

A neural operator learning [13,83–86] framework is constructed to learn a mapping between function spaces – as needed when approximating solutions of partial differential equations (PDEs). Similar to [86], it assumes $\mathcal{U}, \mathcal{V}$ to be Banach spaces of functions on compact domains $\mathcal{X} \subset \mathbb{R}^{d_x}$ or $\mathcal{Y} \subset \mathbb{R}^{d_y}$, mapping into $\mathbb{R}^{d_u}$ or $\mathbb{R}^{d_v}$, respectively. The goal of operator learning is to learn a ground truth operator $\mathcal{G} : \mathcal{U} \to \mathcal{V}$ via an approximation $\hat{\mathcal{G}} : \mathcal{U} \to \mathcal{V}$. This is usually done in the vein of supervised learning by independent and identically distributed (i.i.d.) sampling input-output pairs, with the notable difference that in operator learning the spaces sampled from are not finite dimensional. More precisely, with a given data set consisting of N function pairs $(\mathbf{u}_i, \mathbf{v}_i) = (\mathbf{u}_i, \mathcal{G}(\mathbf{u}_i)) \subset \mathcal{U} \times \mathcal{V}$, $i = 1, \ldots N$, a neural operator aims to learn $\hat{\mathcal{G}} : \mathcal{U} \to \mathcal{V}$, so that $\mathcal{G}$ can be approximated within the bounds of the input dataset $(\mathbf{u}_i, \mathbf{v}_i)$.

The PDEarena platform [74,75] was used to train the surrogate models. Although PDEarena includes several options of neural operators, in this work only the Fourier-Neural Operator method (FNO) [13] was used. PDEarena was used as a code base and extended to support the JOREK simulation data. A modified version of the FNO configuration 'FNO-128-32m' in PDEarena is used with the number of Fourier blocks increased to 3, and where the grid discretisation is concatenated in the same dimension as the physics variable fields as it was found to improve performance, as demonstrated in [77].

The FNO method [13] trains a neural network in both the real space and a Fourier space representation of the data points (for a given number of Fourier modes). This effectively means the neural network learns a functional mapping (in this case, the Fourier transform) of the training data rather than just discrete data values. The Fourier transforms are applied to individual 2D maps at each point in time, although versions of the FNO also exist with Fourier transforms applied to both spatial and temporal dimensions. The advantage of this method is that any interpolation between data points (both spatially and in terms of the input domain) will be more reliable than for data-only neural networks. The FNO has been shown to work well on a wide number of applications, including fluid models with convection particularly relevant to the use case presented here. However, for future studies, the FNO may struggle with 3D unstructured meshes and non-standard geometries, where Fourier decomposition is inappropriate, such that other approaches like Transformers may be more suitable.

3.2. Rollout and network inputs

For a given temporal resolution, the neural operator is trained to predict k time-steps ahead, given l time-steps as input. These parameters k and l affect the performance of the predictions, but they also affect how the solver can be integrated into the Parareal framework, which will be addressed in more details in Section 4. In all cases included in this study, l is always chosen to be 1, while k is varied between 5 and 20.

In order to predict far ahead of a given set of input time-steps, the prediction is *rolled out* in an autoregressive manner. Given $[1 \to k]$ input steps, once the step $k+1$ has been predicted, a new set of inputs $[2 \to k+1]$ is fed to the network to predict the step $k+2$. Following which the inputs $[3 \to k+2]$ are fed back into the network to predict step $k+3$, and so on until the desired prediction length is achieved. This is illustrated in Figure-1 of [76]. Note that the number of autoregressive rollout steps is entirely independent from l, and may vary in the following sections in order to accommodate various sizes of Parareal windows. A short investigation into the effect of increasing l found that long rollouts were slightly improved, but short-term predictions notably affected, and thereafter it was decided to fix $l = 1$. Using a larger l for the very first coarse-solver run of Parareal (which needs to predict the entire time-domain), while using $l = 1$ for individual time-window predictions in subsequent Parareal iterations, was also considered. However the marginal improvement of larger l values did not warrant the increased complexity needed inside the framework to handle two different neural coarse solvers in a single run. In any case, for chaotic systems, loss of precision in the short term will accumulate over long rollouts, such that $l = 1$ will presumably be more appropriate to ensure higher accuracy for long simulations in general.

If $\mathcal{G}$ is the mapping from an initial condition $\mathbf{u}(0, \mathbf{x}) = \mathbf{u}^0(\mathbf{x})$ to the solutions $\mathbf{u}(t, \mathbf{x}) = \mathbf{u}^t(\mathbf{x})$ at later times, then in order to obtain accurate predictions over long time horizons, a temporal operator could either be directly trained for large Δt or recursively applied for smaller time intervals. However, in practical applications, the predictions of neural operators degrade for large Δt, while autoregressive approaches are found to perform substantially better [75,87–89].

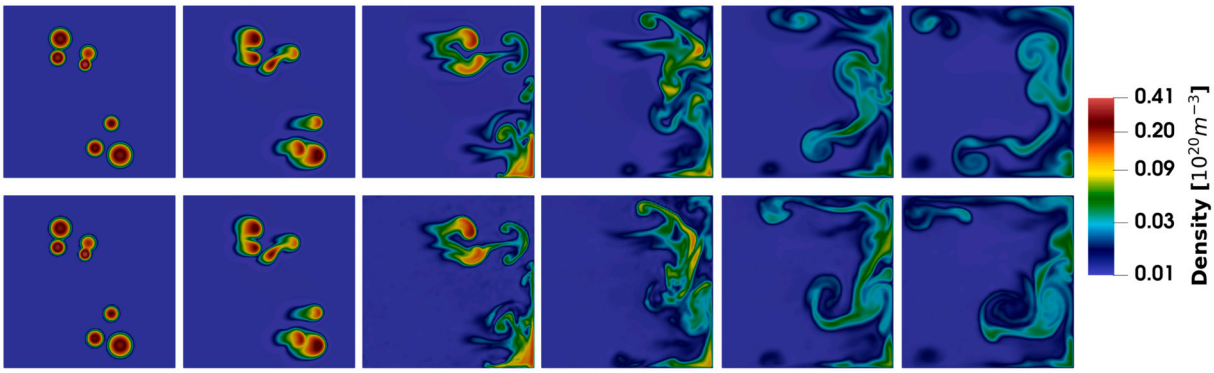


Fig. 4. Comparison of the evolution of the real simulation (top) compared to the PDEarena surrogate (bottom). As in Fig. 2, each frame corresponds to 200 time-steps, so 30 μs.

The models are trained for about 72 hours (depending on performance) using the Adam optimizer with cosine annealing learning rate scheduler with the starting learning rate of 0.0002 and minimum learning rate value of 1.e-7 for both. The number of epochs and learning rate was varied for optimum performance in some of the training runs, but systematic hyperparameter tuning of the entire model was not performed here and is being considered for future work, as described below.

Again for the same simulation example as in Fig. 2 (for the electrostatic model), the prediction of the PDEarena surrogate is rolled out and compared to the ground truth in Fig. 4. Each frame of Fig. 4 corresponds to 200 time-steps (30 μs) from the original simulation, thus 20 steps of the PDEarena temporal resolution. For this example, the neural solver was trained using 20 steps as input, meaning that the second frame is, in fact, still the actual simulation. The next 4 frames, 80 steps in total, are thus entirely predicted by the neural solver. As can be seen, the precision of the prediction is reasonably reliable for about 40 steps, beyond which it diverges from the ground truth. Note this is one of the early FNO models trained in [76], but part of this study was to explore improved models too, used in sections below.

The run-time of neural operators like the FNO is one of the main points of this study: it is extremely cheap. For a full rollout of the same length as a simulation of 2000 time-steps (i.e. 200 rollout steps), the execution time is approximately 2 minutes on a single core. Most of the execution time is in fact dominated by the NN-model load. At present the model is loaded each time an evaluation is needed, but future improvement may include having background jobs with the model loaded and awaiting external signals to process model evaluations whenever needed. This may sound excessive, considering that 2 minutes is negligible in comparison to the real simulations that require 14 hours on 48 cores, but in a Parareal framework, the speed of the coarse-solver is essential. For example, if a Parareal run is executed with 100 time-windows, and each coarse-solver evaluation takes 2 minutes, it quickly adds up.

3.3. Limitations of current versions

More work is currently under way to improve the PDEarena solvers, both for the purpose of exploration of surrogates in fusion applications in general, but also for this specific application with Parareal. Although these improvement areas are beyond the scope of this study, they are worth mentioning here for the sake of clarity.

Firstly, using higher spatial and temporal resolution may have significant effects on the precision of the neural solver. Although 200×200 bi-cubic finite-elements may be conservative, the granularity of the 100×100 grid used for the neural solver can be seen by eye when zooming on the details of Fig. 4. With a set of non-linear PDEs, any loss of precision will undoubtedly accumulate to significant deviation for long rollout predictions. Likewise, down-sampling the temporal frequency of the data may play a role. Of course, increasing resolution means the training time and cost of the models would increase significantly, which is part of the reason why this is being kept for future plans. At present, parallelising the training on multiple GPU nodes is being explored to alleviate this limitation.

Secondly, there are more neural operator options in PDEarena, besides the FNO, which ought to be explored. The efficiency of each of these methods may be strongly dependent on the use case. For example, one particular model may perform very well on 2D regular meshes, but could become unreliable when addressing unstructured meshes in 3D with more complex geometries. The most relevant aspect of this issue is to scale up towards realistic fusion applications, such as turbulence in 3D toroidal geometries, which will require larger models and larger datasets, for which Transformer architectures may be most appropriate.

Finally, hyperparameter tuning has not been done systematically in this study, as it represent another dimension to the total cost of the framework as a whole. Note that it is not just the internal parameters of each model that should be subject to optimisation, but the models themselves, as well as the data-resolution described above. It is entirely possible that, depending on the use case, as more data is made available to the neural solver training, different models and different resolutions may be more appropriate, not just model parameters.

4. Parareal framework

Although the implementation of the Parareal algorithm is generally straight-forward, in this particular case there are two aspects that make the framework more complex than a standard situation. In particular, the fact that the JOREK code uses finite-elements, and that the neural coarse solver requires several input time-steps, as opposed to a single initial-value state in typical Parareal applications.

4.1. Parareal with a finite-element fine-solver

As a first demonstration, the Parareal framework was first implemented using a *classical* coarse-solver, namely the same simulation code JOREK, but with (optionally) coarser spatial/temporal resolution. This step was useful not just to develop the framework itself, but as will be seen in Section 5, it also gives a practical reference for testing and evaluation.

The first technical aspect in the implementation is to convert data from one spatial resolution to another. In this context, given two equidistant grids of point-wise data with different resolutions, simple algorithms like linear interpolation are easily applied. However, when finite-elements are involved, such conversions need to be *projected* onto the degrees-of-freedoms of each element. This projection, which is applied for each scalar variable independently, requires the weak-form integration of each element around a given node to solve for its degrees of freedom. This procedure is already an available feature in the JOREK

code, but it requires the data to be evaluated at the Gaussian integration points of each element. For bi-cubic elements, 4 Gaussian integration points in each direction are necessary for each element. Thus, whether data is being up-sampled or down-sampled between two grids, if the final destination is a finite-element grid, this sampling must be done on the Gaussian integration points. It is important to note that the location of Gaussian integration points are not equidistant, such that linear interpolation is not adequate.

Whether Parareal is run with a *classical* coarse-solver that uses finite-elements or not, interpolation between different resolutions is clearly required, and since these interpolations involve non-equidistant Gaussian integration meshes (and with the long-term goal of extending the framework to unstructured grids), a robust and generic approach is to use the Clough-Tocher 2D-Interpolator from the `scipy.interpolate` library. This method has the great advantage that it is mesh-agnostic, thus ideal for this application.

Consider a fine-solver F and coarse solver G that evolve on different grid resolutions. In practice, it is safe to assume that the fine-solver has the highest spatial resolution. Note that although this is not strictly necessary, one would assume that if the coarse-solver also has high spatial resolution, it is because its grid is the same as the fine-solver, in which case no interpolation is needed. At each Parareal cycle i_p and time-window i_t, the predictor-corrector algorithm must be applied to the outputs from the previous time-window and previous Parareal cycle, such that the new initial-value map is given by

$$U|_{(i_p,i_t)} = G|_{(i_p,i_t-1)} + F|_{(i_p-1,i_t-1)} - G|_{(i_p-1,i_t-1)}. \tag{1}$$

However, it is important to note that this operation involves data from two different grids as inputs, and that the output $U|_{(i_p,i_t)}$ must be evaluated on both the coarse grid and the fine grids in order to run the next Parareal cycle. Thus there are two choices:

(a) all inputs $G|_{(i_p,i_t-1)}$, $F|_{(i_p-1,i_t-1)}$ and $G|_{(i_p-1,i_t-1)}$ are first interpolated onto the higher-resolution grid, the predictor-corrector operation (1) is applied, and the resulting initial-value map $U|_{(i_p,i_t)}$ is interpolated onto the coarse grid.
(b) all inputs $G|_{(i_p,i_t-1)}$, $F|_{(i_p-1,i_t-1)}$ and $G|_{(i_p-1,i_t-1)}$ are interpolated onto both the fine-solver grid and the coarse-solver grid, and the predictor-corrector operation (1) is applied to both sets of interpolated inputs.

While the difference between these two options may seem irrelevant, there is one important point to consider: at each Parareal cycle i_p, and at each time-window i_t, the predictor-corrector step can only be applied once the coarse-solver solution has been obtained for the previous time-window i_t-1. In other words, the predictor-corrector step is sequential across time-windows, just like the coarse-solver evaluation. However, interpolations with the Clough-Tocher method can be non-negligible, particularly when running realistic use cases with high resolution grids. Note, it is most expensive to interpolate *from* a high-resolution grid, but once the Clough-Tocher spline has been calculated, evaluation is relatively cheap. This means that for option-(a), the final interpolation onto the coarse-solver grid will be expensive, and sequential. The advantage of the second option-(b) above is that the interpolation of all F and G solutions, on both coarse- and fine-solver grids, can be executed at the end of each evaluation independently. That means the most expensive interpolation (*from* F) can be done in parallel when the fine-solver evaluation is deployed. And only the (cheaper) interpolation of $G|_{(i_p,i_t-1)}$ remains sequential since the interpolation of $G|_{(i_p-1,i_t-1)}$ is already available from the previous cycle.

There is one additional aspect to this interpolation issue. The interpolation *from* a given fine-solver input $F|_{(i_p-1,i_t-1)}$ does not necessarily have to use the Clough-Tocher method (or any interpolation method), since the fine-solver grid is in fact already a bi-cubic spline itself. This means that the $F|_{(i_p-1,i_t-1)}$ inputs can be directly *evaluated* onto the required grids (either coarse-solver or Gaussian-integration points), and no spline calculation is required. Note, however, that this step is fast provided the local-coordinates (s,t), in finite-element space, are already known for each evaluation point. This means that for each evaluation point with poloidal coordinates (R,Z) in real-space, the corresponding FEM local-coordinates (s,t) have to be calculated and recorded at the beginning of each Parareal run, since these evaluations will be repeated for each time-window at each Parareal cycle (i.e. potentially thousands of times). This local-coordinates mapping is achieved using the Newton-Raphson method for each point and the resulting map saved for future evaluations. Contrary to this optimal approach, if option-(a) is employed, then the final interpolation of the high-resolution $U|_{(i_p,i_t)}$ has to be done using an interpolator like the Clough-Tocher method, since that $U|_{(i_p,i_t)}$ map is already a mixture of 3 solutions on a grid which is not a spline (typically the Gaussian-integration mesh of the high-resolution finite-element grid).

Option-(a) is manageable for small tests and code-development examples, but for realistic applications it becomes prohibitively expensive, such that the interpolation i/o can even dominate over everything else. This becomes particularly problematic for very large numbers of time-windows, where the evaluation of each fine-solver becomes faster (fewer time-steps to compute) but where the number of predictor-corrector operations increase.

Nevertheless, as will become evident in the next section, there is one disadvantage to option-(b). Namely, the more natural implementation of option-(a) is doing all the data i/o and processing 'on-the-fly', meaning that for each predictor-corrector operation, all the mappings required (on fine- and/or coarse-grids) can be generated instantly, even in-memory, and discarded as soon as the output of the predictor-corrector is obtained. With option-(b), that data must be written on the file-system and saved until at least the end of the next Parareal cycle, since the predictor-corrector operation requires the interpolated data from $F|_{(i_p-1,i_t-1)}$ and $G|_{(i_p-1,i_t-1)}$.

4.2. Parareal with a neural coarse solver

In the case where a neural coarse solver is used, like the FNO method in PDEarena, the input data needed for each coarse-solver run, at each time-window, is not just a single initial-value state. As described in Section 3, the neural solver requires several time-steps as input. In practice, this means that the predictor-corrector step described above must be applied for several time-steps.

While this may sound like a simple matter of iterating the predictor-corrector step over each input-step, in practice this characteristic of Neural-Parareal has several implications, both in terms of data-processing and data-management, as explained below.

4.2.1. Checkpoint synchronisation

Firstly, this multiple-input constraint implies that the correct frequency of checkpoints is required for both the fine-solver and coarse-solver, to coincide with the required data-input of the neural solver, compared to standard Parareal applications, where only the total duration of the time-window needs to be the same for both solvers, with only one final checkpoint for each time-window. Note that checkpoint here refers to a *state* file, or a *restart* file which holds all the information necessary from which to continue the simulation.

Additionally, since the neural coarse-solver requires multiple input checkpoints, the Neural-Parareal framework cannot be run like a standard simulation with an initial condition. It requires a first *pre-run* time-window with the fine-solver to create the initial set of multiple input checkpoints, from which the full Parareal simulation is initiated.

Finally, the size of the Parareal time-windows cannot be smaller than the number of input-steps required by the neural coarse-solver.

4.2.2. Performance and i/o parallelisation

As mentioned above, each predictor-corrector operation is not negligible, and if it needs to be run for several checkpoints, possibly 10 to

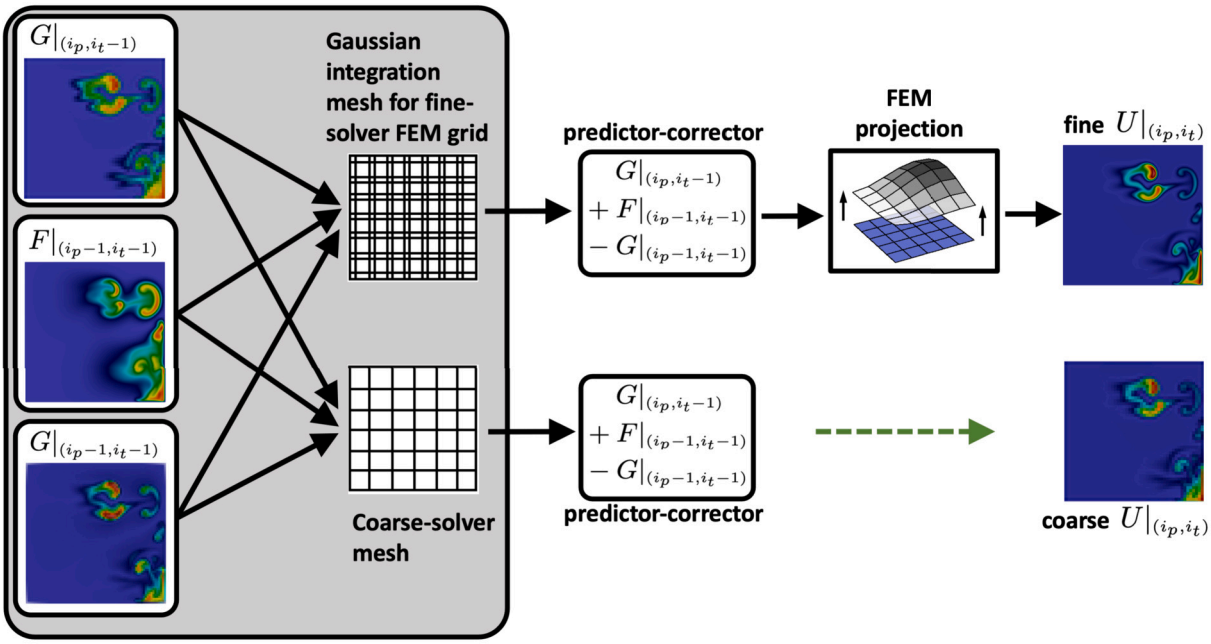


Fig. 5. Schema of the Parareal predictor-corrector step for an application with finite-element grids. Note that if the coarse-solver also uses a finite-element grid (such as the 'classical' coarse-solver), the last step (dashed-green arrow) also involves a FEM projection. In order to avoid i/o overheads at run-time, the evaluation/interpolation onto the two grid domains (grey box) is executed together with the deployment of each $F|_{(i_p-1,i_t-1)}$ and $G|_{(i_p-1,i_t-1)}$. This is particularly important for the (most expensive) interpolation of the fine-solver $F|_{(i_p-1,i_t-1)}$, which can therefore be run in parallel, rather than sequentially at the end of each time-window's coarse-solver pass $G|_{(i_p,i_t-1)}$. Note that the 2D frames of the blobs in this figure are representative, and the coarseness is exaggerated on purpose to illustrate the coarse-solver. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

20 of them, then this operation must be parallelised. Note that in the predictor-corrector schema described in Fig. 5, there are two locations where parallelisation is required.

The interpolation/evaluation of each input, as described in the grey box of Fig. 5, can be run in parallel at the end of each coarse-solver and fine-solver run, simply deploying across however many time-steps are required by the neural solver.

The predictor-corrector step, however, occurs sequentially after each coarse-solver pass through the time-windows of each new Parareal cycle. For the fine-solver (top row of Fig. 5, only a single input is required (the junction between time-windows), but for the coarse-solver (bottom row), the predictor-corrector must be applied to all time-steps required as input for the neural solver. This operation must also be parallelised.

4.2.3. Data generation (and annihilation)

Since all operations to generate the mappings of the data onto the Gaussian-integration points of the fine-solver grid and the coarse-solver grid (grey box in Fig. 5) are not happening 'on-the-fly' like the predictor-corrector, this data and must be saved onto the file-system, and can be deleted later (or not).

Apart from the data generation/annihilation necessary for the predictor-corrector, the particularity of the Parareal algorithm is that it creates a lot more data than a normal simulation. Assuming that the framework is run with a large number of time-windows, then each Parareal cycle includes the equivalent amount of fine-solver time-steps as the entire run would (except that they are disjointed since they start with separate initial-conditions). Therefore, even if the Parareal run achieves the desired level of precision/convergence after 4 cycles, it means that approximately 4 times more data has been produced than for a single simulation. This large amount of extra data per run is ideal for training the neural operator. Note that the precision/convergence of the Parareal cycles acts as an uncertainty quantification of the underlying neural coarse solver, which could be used to assess whether new data points add actual value to the training of future surrogates. However this is not addressed in this study: all data is saved and used for the training.

A normal JOREK simulation (without Parareal) with 200 checkpoint files out of 2000 timesteps is about 5.8 GB. This contains the full spatial resolution with all variables on the high-order finite-element grid. A full simulation (without Parareal) in the down-sampled format for the neural operator training (100×100 grid, 200 time frames) is 93 MB. A Parareal simulation with 20 time-windows, run for the full 20 Parareal iterations, and saving all of the intermediary data (grey box in Fig. 5), is about 265 GB. After clean-up, the same Parareal simulation, without saving all these intermediary files, and keeping only strategic data, is about 61 GB.

Note that it would be possible to run Parareal without saving the data to disk, by keeping only the essential data in-memory on the parent node. For the settings described here, this essential data accumulates to approximately 16 GB for each Parareal cycle, which is needed until all predictor-corrector steps of the following cycle have been obtained. Thus, on a cluster like Cineca-Marconi, which has 120 GB of RAM per node, this would be feasible (although it may become challenging for larger simulations). However this was not implemented in the current framework, since the nature of the Neural-Parareal framework relies on saving this extra data to disk, so that the Neural Operator can be re-trained later. But such a setting could be useful in practice, for example if a given Neural-Parareal simulation set-up has achieved such accurate neural coarse-solvers that no more training is necessary.

4.2.4. Implementation with Slurm scheduler

All runs were executed on HPC clusters that operate with a Slurm Workload Manager. The current implementation requires a Slurm job 'parent' that orchestrates the whole Parareal run, its i/o and the deployment of children Slurm jobs. This parent job submits new children Slurm jobs for each fine-solver time-window, but the coarse-solver time-windows can either be submitted as separate Slurm job or be executed inside the parent job, assuming it does not require too many resources. This can be advantageous in case queuing times are elevated, since the

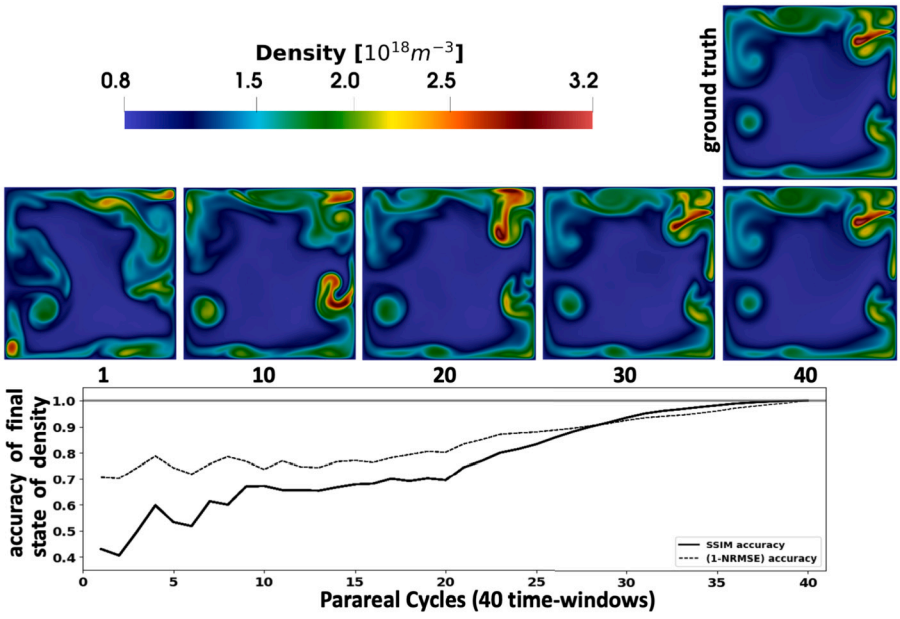


Fig. 6. Evolution of the last time-step of the last (40^{th}) time-window of a Parareal simulation (showing the density map), compared to the ground-truth from the exact simulation. As the algorithm evolves through each cycle, the final state progressively converges to the ground truth. As expected, at the last cycle, the result exactly coincides with the ground-truth simulation. The bottom plot shows the evolution of the SSIM difference between the final state and the ground truth, as a function of Parareal cycles. For comparison, the (1-NRMSE) accuracy measure is also potted.

parent job will remain idle while waiting for its children jobs to return.

5. Neural-Parareal results

5.1. Initial demonstration and tests

In order to validate the implementation of the Parareal framework, a series of tests were run. The first result is a test that the framework works and runs to completion. By definition, the Parareal algorithm can be run for as many cycles as there are time-windows. At each new cycle, the first time-window becomes the actual fine-solver target simulation. At cycle 1, time-window 1 will be run with the fine-solver starting from the initial conditions. At cycle 2, time-window 1 does not need to be run at all anymore, and time-window 2 will be run with the fine-solver, starting from where time-window 1 finished, i.e. the exact simulation is continuing. At cycle 3, time-window 3 becomes the continuation of the real simulation, and so on. Thus, at cycle n, where n is the number of time-windows, the entire simulation has been run from start to end. This is illustrated in further details in Appendix A and in [30]. In practice, one would never run all cycles of a Parareal simulation, since that becomes at least as slow as the simulation itself (i/o and coarse-solver timing added), but it is computationally much more expensive, namely $\sum_{i=1}^{n} \frac{i}{n} = \frac{1}{2}(n+1)$ times more expensive, since at each cycle i, there are $n-i$ time-windows to be run, each costing $\frac{1}{n}$ times the cost of the full simulation. Nevertheless, for testing and demonstration, it is useful to be able to compare the Parareal to the real simulation. For this purpose, the last time-step of the last time-window of the Parareal simulation should converge towards the last time-step of the full simulation. If that convergence occurs quickly, it means the Parareal simulation is efficient.

Fig. 6 displays the evolution of the last time-step of the last time-window for a Parareal simulation with 40 time-windows, showing how the Parareal result progressively converges to the ground truth. This simulation is run with the electrostatic model, using a neural operator that takes 5 input steps, similar to the surrogates shown in [76]. At the last cycle, as expected, the Parareal result is identical to the ground truth simulation.

In order to evaluate the efficiency of subsequent Parareal tests, the Structural Similarity Index Measure (SSIM) [90] algorithms is used. As can be observed in Fig. 6, even though the general structure of the density map at cycles 20 and 30 are similar to the ground truth, the fine details are so different that an MSE comparison between the two would be dominated by point-wise differences rather than inform on the similarity of the blob structures. The SSIM algorithm is designed to provide a better focus on general structures of image maps rather than their exact details. The SSIM is relatively simple to implement in Python and available from the `skimage.metrics` library. An SSIM value of zero means the two images are completely different, while an SSIM of 1.0 means the two images are identical. The bottom part of Fig. 6 shows the SSIM measure from that Parareal simulation, aligned with the corresponding 2D frames for reference.

In order to compare the SSIM measure with a more conventional measure, Fig. 6 also includes the inverse normalised root-mean-square error (1-NRMSE). It illustrates why the SSIM is more useful for global accuracy: between frames 1 and 20 of Fig. 6, although there is a clear improvement in the overall structure of the solution compared to the ground truth, the (1-NRMSE) barely evolves from 0.7 to 0.8, while the SSIM evolves from 0.4 to 0.7. In other words, SSIM is better at measuring the accuracy of 'imprecise' frames. However, for more advanced studies, where high precision becomes essential, it is expected that the (1-NRMSE) measure will be more appropriate. In addition, for realistic tokamak geometries, with unstructured meshes, the SSIM would not be applicable anyway, since it is designed for 2D rectangular frames of regular resolution. Nevertheless, for this initial demonstration of the Neural-Parareal concept, the SSIM is more appropriate, and such high-precision studies are left for future works.

Note that since the last time-step coincides with the ground truth simulation at the last Parareal cycle, the SSIM will converge to 1.0 at the final cycle. In other words, no matter how bad the coarse-solver is, and how slowly Parareal converges, the SSIM and the (1-NRMSE) will always go to 1.0 at the final Parareal cycle.

In order to obtain some measure of the efficacy of the Neural-Parareal framework, and the corresponding neural coarse-solver, it is useful to run the same cases with *classical* coarse-solvers. This is achieved by using the actual JOREK code for the coarse solver, with

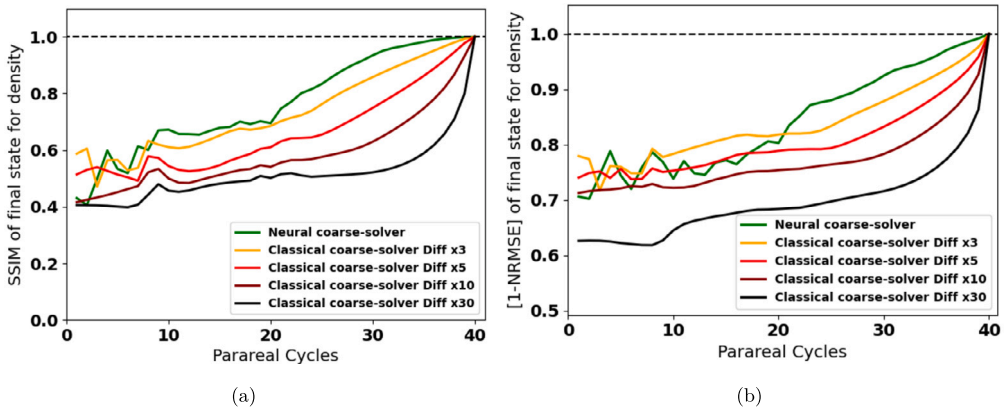


Fig. 7. Accuracy of the Parareal implementation using a Neural Operator as the coarse solver, compared to a 'classical' coarse solver. (a) Evolution of the SSIM computed for the density map (with the electrostatic model) at the last time-step of the last (10^{th}) time-window of a Parareal simulation, compared to the ground-truth from the exact simulation. The evolution is plotted for 5 cases: the Parareal framework with a PDEarena neural operator as the coarse-solver, and 4 'classical' coarse-solvers with increasing levels of additional diffusion. Even for the best 'classical' coarse-solver with only 3 times the level of extra diffusion, the SSIM evolution is still outperformed by the Neural-Parareal case. (b) Same but using (1-NRMSE) accuracy instead of SSIM, which shows the neural coarse solver performing closer to $\gamma = 5$ in the early phase of the Parareal cycles.

exactly the same physics model, but with a reduced resolution grid of 90×90 instead of 200×200, and increased diffusion coefficients. For this exercise, 4 cases are run with 4 levels of increased diffusion, with all parameters D, κ, μ increased together by a given factor coefficient γ. The 4 cases are run with a γ of 3, 5, 10 and 30. Effectively, this means that the classical coarse-solver with $\gamma = 30$ is a very *bad* or *inaccurate* coarse-solver, while the one with $\gamma = 3$ is much more reliable. All cases are run with 40 time-windows, using the electrostatic model, which was the maximum number of time-windows possible for the neural operator which was trained to use 5 time-samples as input. The comparison between these 4 cases and the real neural coarse solver from PDEarena is done by observing the evolution of the SSIM between the last time-step of the last time-window and the ground truth, as a function of Parareal cycles, which is shown in Fig. 7. The (1-NRMSE) equivalent is also included for comparison, showing slightly lower performance in the early phase of the run, with the Neural coarse-solver being closer to the classical coarse solver with $\gamma = 5$.

As can be seen in this result, the Neural-Parareal framework with a PDEarena surrogate as coarse-solver performs similarly to the best *classical* coarse-solvers with $\gamma = 3$ and $\gamma = 5$ (depending on the SSIM or NRMSE measure). Presumably, a classical coarse solver with $\gamma < 3$ would outperform the neural coarse solver. There was no particular reason for the choice of $\gamma = [3, 5, 10, 30]$, except that some (about 3%) of the 2000 simulations in the electrostatic dataset used in [76,77] suffered from numerical instabilities, and therefore it was expected that the coarse solver's lower resolution of 90×90 would require higher diffusion parameters anyway, to ensure numerical stability. Also, it should be noted that, as far as the Parareal method is concerned, such a *classical* coarse-solver with diffusion coefficients increased by only a factor 3, with exactly the same physics model, should be considered an extremely reliable coarse-solver. Effectively, anything better than that should be considered almost identical to the real-solver. Now, evidently, in a non-linear system, any deviation, even at numerical accuracy, can lead to drastic differences for long simulation times, but at least this comparison provides an informative initial measure of how well the neural coarse-solver performs.

Nevertheless, as can be seen from Fig. 7, after 10/40 cycles (i.e. up to 4× speedup), even the best Parareal run has barely reached 65% SSIM accuracy, and by the 20/40 cycle (i.e. up to 2× speedup), barely 70%. Although the point of this study is not to investigate the performance of Parareal as a speed-up option for the JOREK code, a relevant point is to investigate how far the neural operators can be improved to increase the precision (and/or speed-up) of the Parareal framework.

5.2. Integrated framework demonstration

As mentioned earlier, for a specific number n of time-windows, assuming the i/o and orchestration operations are negligible compared to the fine-solver, the Parareal algorithm will provide a real-time speed-up of $\frac{n}{i}$, where i is the number of Parareal cycles needed to reach the desired level of convergence (the exact speed-up of $\frac{n}{i}$ might be reduced depending on the cost of the coarse-solver, the i/o operations and the orchestration of all jobs). However, assuming n is large, then the total amount of fine-solver time-steps simulated will be i times the amount that a single (non-Parareal) simulation would provide. Given this large amount of data produced by the Parareal framework, a sensible approach is to consider a situation where a scientist would need to produce new simulations progressively within an input-parameter domain to explore new features of the problem at hand. Initially, since no data is available, it is impossible to train any neural operator, but after a reasonable number of simulations are achieved, a first (very) coarse neural solver can be trained, thus enabling simulations to be run with the Parareal framework. From that point onward, all simulations run with Parareal produce more data, which can then be bootstrapped into the training of progressively more accurate neural coarse-solvers. Ideally, the more simulations are run, the more training data, the better the neural coarse-solver, and thus the higher Parareal speed-up. In the ideal scenario that the neural coarse-solver can become as accurate as the fine-solver itself, this means simulations with Parareal could then reach a speed-up efficiency equal to the number of time-windows, i.e. with $n = 100$, simulations would be achieved 100 times faster.

This optimistic vision will be highly dependent on the use case. For simple problems where neural coarse-solvers can indeed reach high accuracy levels, such speed-up results might be achievable, but one would argue that simple sets of PDEs do not need to be parallelised in the first place, as they would be easily and quickly evaluated with ordinary numerical solver methods. A demonstration of this bootstrap method is provided here with the electromagnetic model of Reduced-MHD. Since a lot of data was already available for the electrostatic model, it was intriguing to explore an entirely new case where no data was available, in order to provide a genuine test of the idea of Neural-Parareal.

First, a set of 20 simulations is run, without Parareal. Each one with a random sampling of the initial conditions of multiple blobs. Once these 20 simulations are obtained, the time-trajectories of these are each split into segments of the size of the time-windows. In order to enable enough data within a time-window to be used for the training of the neural solver, the time-domain is decomposed into 20 time-windows. This results in segments of data containing 100 time-steps of simula-

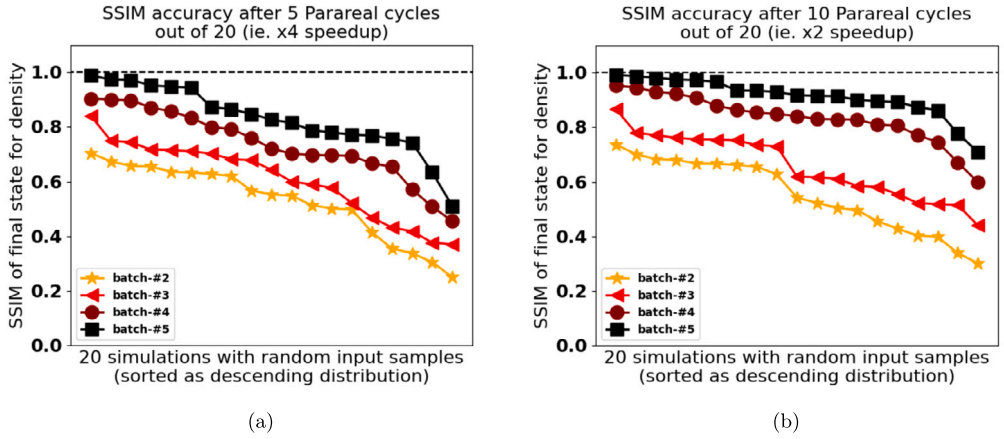


Fig. 8. Performance of the Neural-Parareal framework evaluated at (a) cycle #5 out of 20 (i.e. speed-up of 4), and at (b) cycle #10 out of 20 (i.e. speed-up of 2). For each batch (different colours/signs), all simulations are ordered in descending order to provide a distribution-like view of the performance.

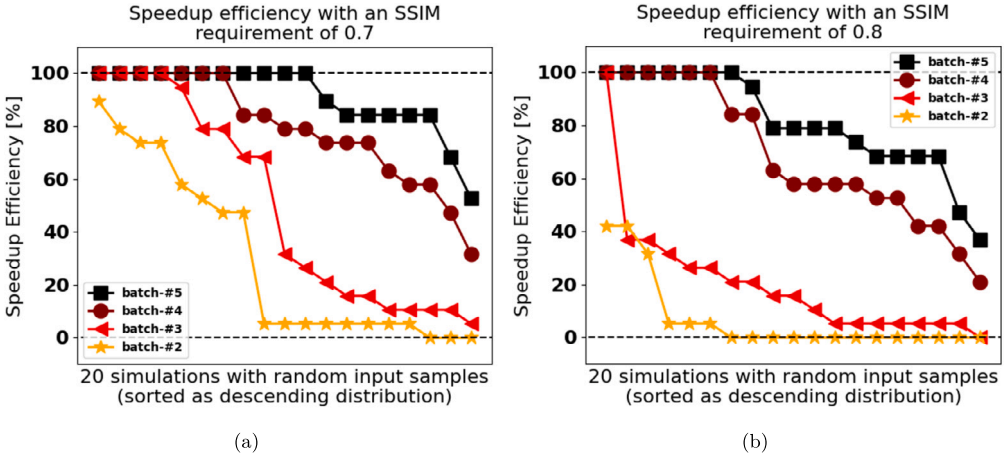


Fig. 9. Speed-up efficiency of the Neural-Parareal framework for a required SSIM-accuracy of (a) 70%, and (b) 80%. For each batch (different colours/signs), all simulations are ordered in descending order to provide a distribution-like view of the speed-up. In this particular example, since the Parareal simulations have 20 time-windows, a 100% speed-up efficiency means a time-to-solution accelerated by a factor 20 (for the required SSIM-accuracy), whereas a speed-up efficiency of 50% means an acceleration of a factor 10.

tions, meaning 10 data frames (due to down-sampling of the frequency agreed for the neural solver inputs). The neural solver itself is trained to use 5 time-samples as inputs (and outputs a single one, which can be rolled-out).

After the first neural coarse-solver is trained, another set of 20 simulations is run with Parareal using the neural coarse-solver, starting from a new set of random inputs. Each simulation is run for the full 20 Parareal cycles. Thus the total number of time-windows run for each Parareal simulation is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, which gives 210 with $n = 20$. This new batch of 20 simulations produces a total of 4200 fine-solver segments that can be added to the initial data set to re-train the neural coarse-solver.

With this new coarse-solver, another set of 20 simulations is run with Parareal, and the resulting data aggregated to the existing dataset for a new training, and so on. This Neural-Parareal loop, described in Fig. 1 is run for 4 iterations, and the result is shown in Fig. 8, demonstrating clear performance improvement at each iteration of the framework. The framework implementation, although specific to the JOREK code for now, is available on [91].

Although in a normal situation one may wish to stop after a fixed number of parareal cycles if convergence is not achieved, for the purpose of this exercise, running all the Parareal cycles ensures more data is created for future training, and it also enables a direct comparison against the ground truth, which would not be available otherwise. All final simulation trajectories (without the additional Parareal intermediate windows) can be downloaded from Zenodo [78].

The result in Fig. 8 demonstrates that in a Neural-Parareal framework, the more simulations are run, the more precise the neural coarse-solver becomes. As the neural coarse-solver improves, the speed-up obtained by the Parareal framework will increase. This is best illustrated in Fig. 9, by plotting the speed-up efficiency of each Parareal simulation for a given SSIM accuracy requirement. As explained in detail in Appendix A, the maximal speedup of a Parareal simulation is the number of time-windows (assuming the number of computing resources are scaled linearly with the number of time-windows). If it takes a single evaluation of the coarse-solver (and fine-solver) on each time-window to obtain the required accuracy level, then the speed-up (time-to-solution acceleration) is equivalent to the number of time-windows. In other words, the speed-up efficiency is 100%. With every Parareal iteration, the speed-up efficiency diminishes. As shown in Fig. 9, the speed-up efficiency increases significantly for each batch of simulation, when the neural operator coarse-solver is updated.

Finally, although this particular aspect of the framework is not addressed in detail here, the accuracy level in Fig. 8 and/or the speed-up efficiency in Fig. 9 can be considered to be a reflection of the Neural Operator's accuracy. As such, the Neural-Parareal effectively provides an uncertainty quantification of the Neural Operator surrogate, which could be used for further improvements, such as Active Learning to

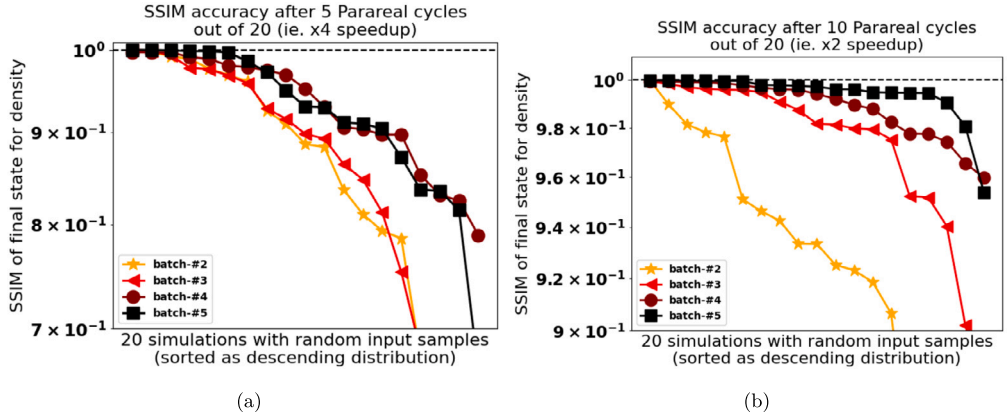


Fig. 10. Performance of the Neural-Parareal framework evaluated at (a) cycle #5 out of 20 (i.e. speed-up of 4), and at (b) cycle #10 out of 20 (i.e. speed-up of 2). Here the SSIM is evaluated against the final time-step of the previous Parareal iteration, as opposed to the ground truth (assuming the ground truth is not available).

determine the most informative choice of input parameter for the following batch of simulations.

6. Conclusion

6.1. Summary

This paper presents the development of an integrated Neural-Parareal framework, which bootstraps the training of neural coarse-solvers as more simulations are being produced by users, progressively leading to more and more accurate neural operators, and thus larger speed-ups of the Parareal simulations. By design, the convergence characteristics of the Parareal algorithm provides a natural quantification of the uncertainty of the underlying Neural Operator surrogate, a valuable measure in the domain of surrogate modelling.

The framework exploits the large amount of data that Parareal produces by design. The Parareal algorithm can provide real-time speedup (i.e. time-to-solution) of simulations for a given precision requirement. Provided a very fast and precise coarse-solver is available, and given large amounts of High-Performance-Computing resources if the simulation can be split into n time-windows on n parallel computing resources, the speedup can potentially become $n/2$ or $n/3$. If n is large, of the order of hundreds, this speedup can be significant. This trade-off between computing cost versus time-to-solution also comes with increased simulation data production, which is ideal for training deep learning models and, in this context, neural operators.

This self-improving framework is demonstrated using MHD simulations relevant to fusion research, with radially evolving blobs in a 2D poloidal slab with toroidal axisymmetry. The neural operators are trained using FNO inside the PDEarena platform, and the MHD simulations are performed with the JOREK code. Parallelisation of the framework is implemented with SLURM on HPC clusters. For a set of Reduced-MHD equations, the framework is evolved for 4 iterations using 20 new simulations at each iteration, clearly showing rapidly improving performance.

Beyond this demonstration, several improvements could be implemented. Particularly in the urgent context of Digital Twins for fusion, this self-improving platform could take full advantage of the rapidly evolving domain of Artificial Intelligence and exascale computing, revealing an interesting avenue in the convergence of AI and HPC.

6.2. Potential extensions and improvements

In order to build upon the work presented here, the following ideas could be addressed in the future:

1. **Using a real Parareal convergence measure**
   In the demonstration above, each Parareal simulation is run for the full number of iterations possible. This is partly to create more data for the training, but also to provide a clean performance measure by comparing against the real simulation. In the future, instead of using SSIM (or MSE) against the ground truth of the final simulation result, one could compare each Parareal iteration against the previous iteration to check the relative convergence. How to properly measure Parareal convergence is an active area of research in itself. A preview of this is given in Fig. 10, with the SSIM accuracy compared to the previous Parareal iteration (instead of the ground truth) for a speed-up of 2 and a speed-up of 4.
2. **Higher resolution surrogates**
   As mentioned in Section 3, the spatial/temporal resolution of the neural operator is down-sampled from the simulation. Although the full resolution equivalent to the simulation may not be required, it will undoubtedly affect the precision of the Parareal framework as a whole, and should therefore be investigated.
3. **Other PDEarena options**
   The high performance of the FNO option in PDEarena may be dependent on the use case, as addressed in Section 3, and it would be interesting to investigate other options, ideally using multiple options at once and choosing the optimal one as part of a broader hyperparameter tuning.
4. **Using existing foundation models**
   Instead of training neural operators from scratch, one could directly fine-tune existing foundation models to create the first coarse-solvers of a given simulation set-up if the amount of data is sparse. This would be particularly relevant to realistic use cases, where engineers and researchers are regularly faced with new problems, including new geometries, new meshes and new physics models. While the FNO works well in this 2D context, it may struggle for 3D unstructured meshes and non-standard geometries, where Fourier decomposition is inappropriate, and where, instead, transformer encoding may be more efficient.
5. **Filtering and discarding data**
   A particularly attractive aspect of the above point is that fine-tuning of large models would be more suited to the potential requirement of discarding data after simulations have been run, with respect to available data-storage capabilities. Regardless, even with the current framework, one could retain only the data of simulations that have struggled to converge, and discard data from simulations that have converged quickly, and thus are already covered by the surrogate.
6. **Better input-domain sampling with Active Learning**
   For a self-improving framework like the one presented here, it may be more efficient to sample the input-domain space with advanced methods like Active Learning, rather than letting users choose new simulations at random. In other words, a modern framework would ingest new input requests from users, and provide a first estimation

of whether the new simulation is actually needed, or whether several simulations have already been performed in the vicinity, thus implying the coarse neural solver is already reliable, and requesting confirmation before this specific simulation is run.

7. **Physics-informed accuracy measures**
   In this study, relatively conventional measures of accuracy were used, such as SSIM and NRMSE. Implementing physics-informed measures of the errors could also provide more elaborate evaluations of the Neural-Parareal progression. For example, given two consecutive time-steps, the variables could be used to reconstruct the PDEs in question, or to reconstruct other known physics quantities, such as the divergence level of the predicted magnetic field.

6.3. Final comments

While this Neural-Parareal demonstration is successful, the authors would like to point out that Parareal itself adds a significant layer of complexity on top of an already complex non-linear MHD code (although any non-linear MHD code will be, by definition, complex). A frequent argument in the Parallel-in-Time community is that Parareal is 'non-intrusive'. Indeed, this is true for the implementation: Parareal can, in principle, be applied to any code without requiring any modification to the code itself. For the end-user of the code, on the other hand, it is worth admitting that Parareal is massively intrusive. Even if a user would not need to understand the details of the Parareal implementation, they would still need to deal with the additional layer of scripts, the coordination of window sizes/numbers, and the large amount of additional data produced by the algorithm. The authors simply cannot argue that Parareal is a user-friendly approach which should be prioritised in future developments.

Instead, the authors would argue that efforts should be prioritised on methods where, even if significant intrusive code-developments are required, the end-user experience is protected. For example, a similar self-improving approach could be implemented to learn the matrix preconditioners required in PDE-solvers like JOREK. By design, this could be entirely abstracted away from the end-user, and even implemented as part of larger libraries like PETSc [92]. The implementation would also be much easier, since the entry would always be a matrix, there is no need for time-windows or tuning of the Neural Operator's input/output steps, or the Neural Operator's grid resolution. In addition, if the target of such surrogates is to provide matrix preconditioners, the main solver would still have a convergence refinement like GMRES or BiCGStab, thus removing the inherent weakness of Parareal: namely that breaking the precision of a time-dependent solution can have drastic consequences in non-linear chaotic systems.

Indeed, while this initial study demonstrates that the framework results in progressively more and more accurate neural operators (and thus larger speed-ups), this statement is entirely relative, since Parareal will always involve a trade-off between accuracy and speed-up. This may be particularly true for real-life use cases, such as realistic 3D non-linear simulations of powerplant-relevant tokamak plasmas, which would need to be assessed in future studies.

Nevertheless, it should still be reminded that, in the domain of PDE solvers, there are not many numerical algorithms that can incorporate surrogate models to accelerate the underlying simulations. Apart from matrix preconditioners, and Large-Eddy-Simulations (LES), Parareal is one of the very few available methods with this characteristic.

CRediT authorship contribution statement

S.J.P. Pamela: Writing – original draft, Visualization, Validation, Supervision, Software, Project administration, Methodology, Investigation, Conceptualization. **N. Carey:** Writing – review & editing, Visualization, Software. **J. Brandstetter:** Writing – review & editing, Software, Methodology, Investigation, Conceptualization. **R. Akers:** Writing – review & editing, Funding acquisition, Conceptualization. **L. Zanisi:** Writing – review & editing, Supervision, Methodology. **J. Buchanan:** Writing – review & editing, Supervision. **V. Gopakumar:** Writing – review & editing, Supervision, Software. **M. Hoelzl:** Writing – review & editing, Software, Funding acquisition. **G. Huijsmans:** Writing – review & editing, Software. **K. Pentland:** Writing – review & editing. **T. James:** Writing – review & editing. **G. Antonucci:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was performed with the support of the JOREK Team [see https://www.jorek.eu for the present list of team members].

This work has been carried out within the framework of the EUROfusion Consortium, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 — EUROfusion). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

This work has been carried out within the framework of the RCUK Energy Programme [grant number EP/I501045].

This work was performed using the MARCONI computer at CINECA in Italy, within the EUROfusion framework.

This work was performed using the Cambridge Service for Data Driven Discovery (CSD3), part of which is operated by the University of Cambridge Research Computing on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The DiRAC component of CSD3 was funded by BEIS capital funding via STFC capital grants ST/P002307/1 and ST/R002452/1 and STFC operations grant ST/R00689X/1. DiRAC is part of the National e-Infrastructure.

Appendix A. Parareal basics

The Parallel-in-Time method *Parareal* is relatively straightforward [12]. Although the concept of parallelising the temporal domain is abstract compared to the parallelisation of a spatial domain, it can be considered in a similar manner to an iterative method which converges to an approximation of the real solution. In order to visualise the Parareal algorithm, Fig. 11 describes the first 4 cycles of a fictional Parareal simulation which has been split into 5 time-windows (animated version here [30]). The full *true* simulation, represented by the black curve, is the target that Parareal will try to approximate. In a practical case, that target is not known, it is only plotted here for explanation. In Parareal, the fine-solver F is the same as the actual simulation code which would be used to calculate whole black curve as $F(t)$.

At the 0^{th} cycle, as shown in Fig. 11a, the coarse-solver G is run all the way through the time-windows (dashed yellow curves). This initial run illustrates that the coarse-solver run is a serial procedure which cannot be parallelised. This is why it is essential that the run-time of the coarse-solver remains negligible (hence the advantage of NN surrogates). After this coarse-solver pass through all windows, the fine-solver is run for each time-window (plain green curves), starting from the initial value left by the coarse-solver at the junction between each time-window. These fine-solver runs are independent of one another, and can be executed in parallel.

Once all fine-solver runs have returned, the *predictor-corrector* scheme (1) is applied to get the initial values of the next Parareal cycle. This is done for all time-windows, except the one starting at $t = t_1$, which is restarted straight from the exact value left by the fine-solver on the first time-window. Although this initial value at $t = t_1$ is exact, it still requires the coarse-solver to be evaluated there, because its result at the

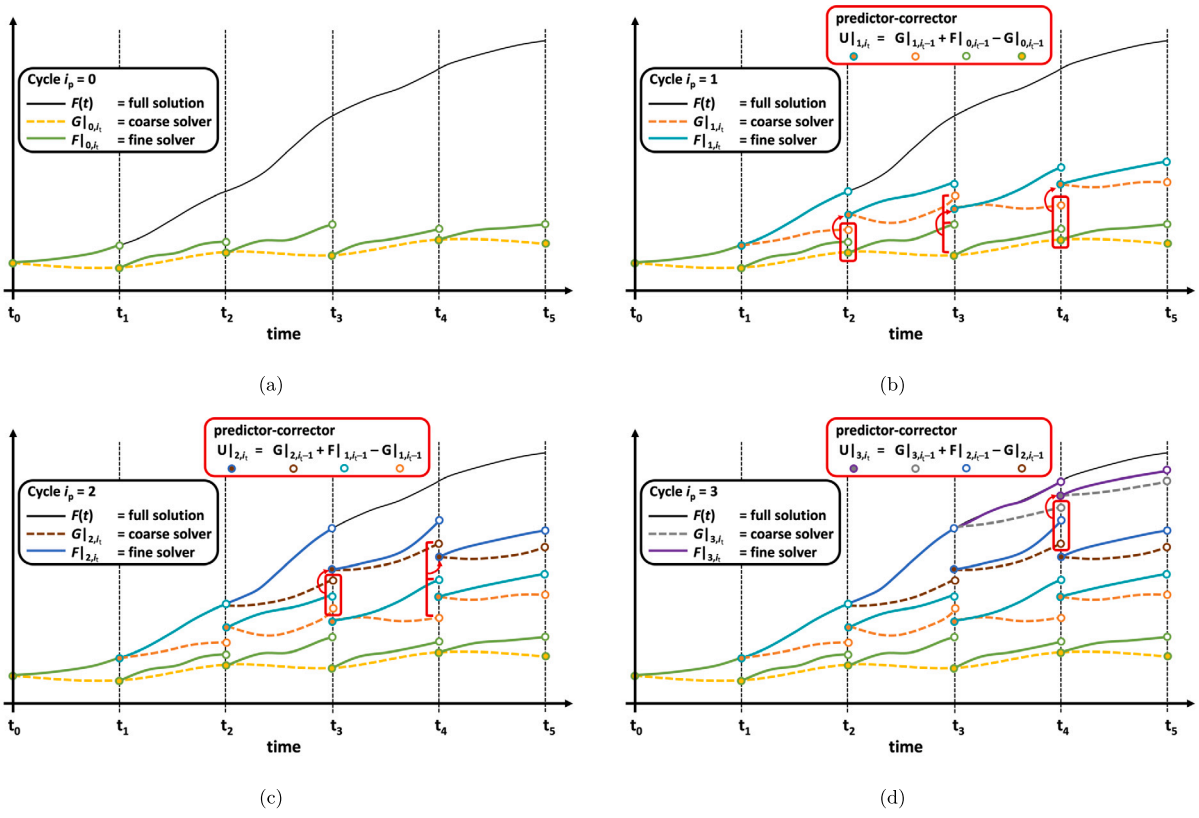


Fig. 11. Basic illustration of the Parareal algorithm, and the predictor-corrector scheme, showing 5 time-windows, with Parareal cycles $i_p = [0,1,2,3]$ in pictures (a)-(d). An animated version can be visualised here [30].

end of the window (at $t = t_2$) will be needed for the predictor-corrector calculation of the initial value at $t = t_2$ for the next window. Again, this illustrates that the coarse-solver G, together with the predictor-corrector, have to be run in a serial manner. For the next time-window, starting at $t = t_3$, the predictor-corrector requires the output of G from the previous window, before it can be run between $t = t_3$ and $t = t_4$, etc. As soon as the predictor-corrector has been calculated for a give initial value, the fine-solver F can be started at that initial value. These fine-solver runs (plain cyan curves) can be run in parallel.

As can be observed in Fig. 11, at each Parareal cycle i_p, the exact simulation is run with F for the time-window $i_t = i_p$. Although not shown here, at the 5^{th} cycle, the exact entire simulation has been obtained. In that case the speed-up efficiency of the Parareal run is 0%, since it will take at least as much time to run as the exact simulation, but using almost 5 times more resources.

The main objective of the Parareal algorithm is to run with a very large number of time-windows, for as few Parareal cycles as possible. In the case of Fig. 11, a deliberately 'bad' coarse-solver was illustrated in order to explain the algorithm. However, consider a situation where the coarse-solver G is so precise that it gives a solution already aligned to the plain black curve $F(t)$ at the 0^{th} cycle. Then the first fine-solver runs (plain green curves) would give results that are also aligned to the coarse-solver results. In other words, if $G|_{0,i_t-1} \approx F|_{0,i_t-1}$, then the predictor-corrector would imply that

$$U|_{(1,i_t)} = G|_{(1,i_t-1)} + \cancel{F|_{(0,i_t-1)}} - \cancel{G|_{(0,i_t-1)}} \approx G|_{(0,i_t-1)}$$

If this is the case, then no more iterations are required, and the speed-up achieved by Parareal is nearly 100% (minus overhead of running the coarse solver). Therefore, if a simulation can be parallelised with 1000 time-windows, using a very fast yet very accurate coarse-solver, then Parareal will provide a speed-up of a factor 1000. Even if 3 iterations are required to converge to a satisfactory accuracy, this will still provide a speed-up of about $1000/3 \approx 333$.

Data availability

Data on Zenodo, DOI's included in paper, will be made public as soon as article published (to include Publication DOI inside datasets). Can share data with referee earlier if needed. Code on github.

References

[1] MOOSE, https://mooseframework.inl.gov/.
[2] MFEM: modular finite element methods [software], mfem.org, https://doi.org/10.11578/dc.20171025.1248.
[3] Q. Tang, L. Chacon, T.V. Kolev, J.N. Shadid, X.-Z. Tang, An adaptive scalable fully implicit algorithm based on stabilized finite element for reduced visco-resistive mhd, submitted for publication, https://arxiv.org/abs/2106.00260, 2021.
[4] Firedrake, https://www.firedrakeproject.org/.
[5] F. Laakmann, P.E. Farrell, L. Mitchell, An augmented Lagrangian preconditioner for the magnetohydrodynamics equations at high Reynolds and coupling numbers, submitted for publication, https://arxiv.org/abs/2104.14855, 2021.
[6] OpenFoam, https://www.openfoam.com/.
[7] JOREK, https://www.jorek.eu/, 2020.
[8] M. Hoelzl, G.T.A. Huijsmans, S.J.P. Pamela, M. Becoulet, E. Nardon, F.J. Artola, B. Nkonga, C.V. Atanasiu, V. Bandaru, A. Bhole, D. Bonfiglio, A. Cathey, O. Czarny, A. Dvornova, T. Feher, A. Fil, E. Franck, S. Futatani, M. Gruca, H. Guillard, J.W. Haverkort, I. Holod, D. Hu, S.K. Kim, S.Q. Korving, L. Kos, I. Krebs, L. Kripner, G. Latu, F. Liu, P. Merkel, D. Meshcheriakov, V. Mitterauer, S. Mochalskyy, J.A. Morales, R. Nies, N. Nikulsin, F. Orain, J. Pratt, R. Ramasamy, P. Ramet, C. Reux, K. Sarkimaki, N. Schwarz, P. Singh Verma, S.F. Smith, C. Sommariva, E. Strumberger, D.C. van Vugt, M. Verbeek, E. Westerhof, F. Wieschollek, J. Zielinski, The jorek non-linear extended mhd code and applications to large-scale instabilities and their control in magnetically confined fusion plasmas, Nucl. Fusion 61 (2021) 065001.
[9] ANSYS, https://www.ansys.com/.
[10] ABAQUS, https://www.3ds.com/products/simulia/abaqus/standard.
[11] SIEMENS, https://plm.sw.siemens.com/en-US/simcenter/.
[12] J.-L. Lions, Y. Maday, G. Turinici, Resolution d'edp par un schema en temps parareel, C. R. Acad. Sci., Ser. I Math. 332 (7) (2001) 661–668, https://doi.org/10.1016/S0764-4442(00)01793-6.

[13] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, arXiv:2010.08895, 2020.
[14] J. Brandstetter, M. Welling, D. Worrall, Message passing neural pde solvers, arXiv:2202.03376, 2022.
[15] Z. Jiang, J. Jiang, Q. Yao, G. Yang, A neural network-based pde solving algorithm with high precision, Nat. Sci. Rep. 13 (2023) 4479, https://doi.org/10.1038/s41598-023-31236-0.
[16] S. Chakraborty, S. Adhikari, R. Ganguli, The role of surrogate models in the development of digital twins of dynamic systems, Appl. Math. Model. 90 (2021) 662–681, https://doi.org/10.1016/j.apm.2020.09.037.
[17] OpenMC, https://docs.openmc.org/en/stable/.
[18] P.K. Romano, N.E. Horelik, B.R. Herman, A.G. Nelson, B. Forget, K. Smith, Openmc: a state-of-the-art Monte Carlo code for research and development, Ann. Nucl. Energy 82 (2015) 90–97, https://doi.org/10.1016/j.anucene.2014.07.048.
[19] MCNP, https://mcnp.lanl.gov/.
[20] H. Brooks, A. Davis, Scalable multi-physics for fusion reactors with aurora, Plasma Phys. Control. Fusion 65 (2022) 024002, https://doi.org/10.1088/1361-6587/aca998.
[21] C. Mistrangelo, L. Buhler, Mhd flow in curved pipes under a nonuniform magnetic field, IEEE Trans. Plasma Sci. (2024), https://doi.org/10.1109/TPS.2024.3358018.
[22] D.R. Mason, A.E. Sand, S.L. Dudarev, Atomistic-object kinetic Monte Carlo simulations of irradiation damage in tungsten, Model. Simul. Mater. Sci. Eng. 27 (2019) 055003, https://doi.org/10.1088/1361-651X/ab1a1e.
[23] F. Schwander, E. Serre, H. Bufferand, G. Ciraolo, P. Ghendrih, Global fluid simulations of edge plasma turbulence in tokamaks: a review, Comput. Fluids 270 (2024) 106141, https://doi.org/10.1016/j.compfluid.2023.106141.
[24] M. Romanelli, G. Corrigan, V. Parail, S. Wiesen, R. Ambrosino, P. Da-Silva-Aresta-Belo, L. Garzotti, D. Harting, F. Kochl, T. Koskela, L. Lauro-Taroni, C. Marchetto, M. Mattei, E. Militello-Asp, M. Filomena-Ferreira-Nave, S. Pamela, A. Salmi, P. Strand, G. Szepesi, EFDA-JET Contributors, Jintrac: a system of codes for integrated simulation of tokamak scenarios, Plasma Fusion Res. 9 (2014) 3403023, https://doi.org/10.1585/pfr.9.3403023.
[25] F. Legoll, T. Lelievre, U. Sharma, An adaptive parareal algorithm: application to the simulation of molecular dynamics trajectories, SIAM J. Sci. Comput. 44 (1) (2022) B146–B176, https://doi.org/10.1137/21M1412979.
[26] J. Guilherme-Caldas-Steinstraesser, V. Guinot, A. Rousseau, Application of a modified parareal method for speeding up the numerical resolution of the 2d shallow water equations, in: Simhydro 2021 - 6th International Conference Models for Complex and Global Water Issues, Sophia Antipolis, France, 2021, https://inria.hal.science/hal-03224056/.
[27] H. Samuel, Time domain parallelization for computational geodynamics, AGU Geochem. Soc. Tech. Brief 13 (1) (2012), https://doi.org/10.1029/2011GC003905.
[28] D. Samaddar, D.P. Coster, X. Bonnin, L.A. Berry, W.R. Elwasif, D.B. Batchelor, Application of the parareal algorithm to simulations of elms in iter plasma, Comput. Phys. Commun. 235 (2019) 246–257, https://doi.org/10.1016/j.cpc.2018.08.007.
[29] K. Pentland, M. Tamborrino, D. Samaddar, L.C. Appel, Stochastic parareal: an application of probabilistic methods to time-parallelization, SIAM J. Sci. Comput. 45 (3) (2023) S82–S102, https://doi.org/10.1137/21M1414231.
[30] Parareal cycles GIF, Zenodo 10.5281/zenodo.11217978 (available upon publication, uploaded with manuscript for review), https://doi.org/10.5281/zenodo.11217978.
[31] O. Gorynina, F. Legoll, T. Lelievre, D. Perez, Combining machine-learned and empirical force fields with the parareal algorithm: application to the diffusion of atomistic defects, arXiv:2212.10508, 2022.
[32] A. Qadir-Ibrahim, S. Gotschel, D. Ruprecht, Space-time parallel scaling of parareal with a Fourier neural operator as coarse propagator, arXiv:2404.02521, 2024.
[33] Z. Li, H. Zheng, N. Kovachki, D. Jin, H. Chen, B. Liu, K. Azizzadenesheli, A. Anandkumar, Physics-informed neural operator for learning partial differential equations, arXiv:2111.03794, 2021.
[34] K. Pentland, M. Tamborrino, T.J. Sullivan, J. Buchanan, L.C. Appel, Gparareal: a time-parallel ode solver using Gaussian process emulation, Stat. Comput. 33 (2023) 23, https://doi.org/10.1007/s11222-022-10195-y.
[35] K. Um, R. Brand, Y. Fei, P. Holl, N. Thuerey, Solver-in-the-loop: learning from differentiable physics to interact with iterative pde-solvers, in: NeurIPS, 2020, Adv. Neural Inf. Process. Syst. 33 (2020) 6111–6122, https://proceedings.neurips.cc/paper_files/paper/2020/file/43e4e6a6f341e00671e123714de019a8-Paper.pdf.
[36] A.P. Toshev, H. Ramachandran, J.A. Erbesdobler, G. Galletti, J. Brandstetter, N.A. Adams, Jax-sph: a differentiable smoothed particle hydrodynamics framework, arXiv:2403.04750, 2024.
[37] J. Castagna, F. Schiavello, L. Zanisi, J. Williams, Stylegan as an ai deconvolution operator for large eddy simulations of turbulent plasma equations in bout++, Phys. Plasmas 31 (2024) 033902, https://doi.org/10.1063/5.0189945.
[38] V. Masson-Delmotte, P. Zhai, A. Pirani, S.L. Connors, C. Pean, S. Berger, N. Caud, Y. Chen, L. Goldfarb, M.I. Gomis, M. Huang, K. Leitzell, E. Lonnoy, J.B.R. Matthews, T.K. Maycock, T. Waterfield, O. Yelekci, R. Yu, B. Zhou, Climate Change 2021: The Physical Science Basis. Contribution of Working Group I to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change, Cambridge University Press, 2023.
[39] J. Wesson, D.J. Campbell, Tokamaks, Clarendon Press/Oxford University Press, Oxford/New York, 2004.
[40] J.P. Freidberg, Ideal MHD, Cambridge University Press, ISBN 9780511795046, 2014.
[41] D.D. Schnack, D.C. Barnes, D.P. Brennan, C.C. Hegna, E. Held, C.C. Kim, S.E. Kruger, A.Y. Pankin, C.R. Sovinec, Phys. Plasmas 13 (2006) 058103.
[42] O. Sauter, et al., Phys. Plasmas 6 (1999) 7.
[43] T. Eich, A. Herrmann, J. Neuhauser, Phys. Rev. Lett. 91 (2003) 195003.
[44] B. Sieglin, M. Faitsch, T. Eich, A. Herrmann, W. Suttrop, Phys. Scr. 2017 (2017) T170.
[45] C. Ham, A. Kirk, S. Pamela, H. Wilson, Nat. Rev. Phys. 2 (2020) 159–167.
[46] A.W. Leonard, Phys. Plasmas 21 (2014) 090501.
[47] C. Cheng, L. Chen, M.S. Chance, Ann. Phys. 161 (1985) 21.
[48] A. Dvornova, G.T.A. Huijsmans, S. Sharapov, F.J. Artola Such, P. Puglia, M. Hoelzl, S. Pamela, A. Fasoli, D. Testa, Phys. Plasmas 27 (2020) 012507.
[49] M. Fitzgerald, J. Buchanan, R.J. Akers, B.N. Breizman, S.E. Sharapov, Comput. Phys. Commun. 252 (2020) 106773.
[50] S.D. Pinches, I.T. Chapman, Ph.W. Lauber, H.J.C. Oliver, S.E. Sharapov, K. Shinohara, K. Tani, Phys. Plasmas 22 (2015) 021807.
[51] A.H. Boozer, Phys. Plasmas 19 (2012) 058101.
[52] P.C. de Vries, G. Pautasso, D. Humphreys, M. Lehnen, S. Maruyama, J.A. Snipes, A. Vergara, L. Zabeo, Fusion Sci. Technol. 69 (2016) 471–484.
[53] M. Lehnen, K. Aleynikova, P.B. Aleynikov, D.J. Campbell, P. Drewelow, N.W. Eidietis, Yu. Gasparyan, R.S. Granetz, Y. Gribov, N. Hartmann, E.M. Hollmann, V.A. Izzo, et al., J. Nucl. Mater. 463 (2015) 39–48.
[54] F.J. Artola, K. Lackner, G.T.A. Huijsmans, M. Hoelzl, E. Nardon, A. Loarte, Phys. Plasmas 27 (2020) 032501.
[55] D. Hu, E. Nardon, M. Lehnen, G.T.A. Huijsmans, D.C. van Vugt, Nucl. Fusion 58 (2018) 12.
[56] V. Bandaru, M. Hoelzl, F.J. Artola, G. Papp, G.T.A. Huijsmans, Phys. Rev. E 99 (2019) 063317.
[57] G.T.A. Huysmans, O. Czarny, Nucl. Fusion 47 (2007) 659.
[58] O. Czarny, G.T.A. Huysmans, J. Comput. Phys. 227 (2008) 7423.
[59] M3D-C1, https://w3.pppl.gov/~nferraro/m3dc1.html, 2020.
[60] S.C. Jardin, J. Comput. Phys. 200 (2004) 133.
[61] NIMROD, https://nimrodteam.org/, 2020.
[62] C.R. Sovinec, A.H. Glasser, T.A. Gianakon, D.C. Barnes, R.A. Nebel, S.E. Kruger, D.D. Schnack, S.J. Plimpton, A. Tarditi, M.S. Chufth, the NIMROD Team, J. Comput. Phys. 195 (2004) 355.
[63] H. Lutjens, J.F. Luciani, J. Comput. Phys. 227 (14) (2008) 6944–6966.
[64] BOUT++, https://boutproject.github.io/, 2020.
[65] B.D. Dudson, M.V. Umansky, X.Q. Xu, P.B. Snyder, H.R. Wilson, Comput. Phys. Commun. 180 (2009) 1467–1480.
[66] Y. Todo, T. Sato, Phys. Plasmas 5 (1321) (1998).
[67] Y. Todo, H.L. Berk, B.N. Breizman, Nucl. Fusion 52 (2012) 094018.
[68] A. Konies, S. Briguglio, N. Gorelenkov, T. Feher, M. Isaev, Ph. Lauber, A. Mishchenko, D.A. Spong, Y. Todo, W.A. Cooper, R. Hatzky, R. Kleiber, M. Borchardt, G. Vlad, A. Biancalani, A. Bottino, Nucl. Fusion 58 (2018) 12.
[69] H.R. Strauss, J. Plasma Phys. 57 (part 1) (1997) 83–87.
[70] H.R. Strauss, Phys. Fluids 19 (1976) 134.
[71] S.J.P. Pamela, A. Bhole, G.T.A. Huijsmans, B. Nkonga, M. Hoelzl, I. Krebs, E. Strumberger, Phys. Plasmas 27 (2020) 102510.
[72] F. Militello, B. Dudson, L. Easy, A. Kirk, P. Naylor, On the interaction of scrape off layer filaments, Plasma Phys. Control. Fusion 59 (2017) 125013, https://doi.org/10.1088/1361-6587/aa9252.
[73] A. Ross, A. Stegmeir, P. Manz, D. Groselj, W. Zholobenko, D. Coster, F. Jenko, On the nature of blob propagation and generation in the large plasma device: global grillix studies, Phys. Plasmas 26 (2019) 102308, https://doi.org/10.1063/1.5095712.
[74] PDEarena, https://pdearena.github.io/pdearena/.
[75] J.K. Gupta, J. Brandstetter, Towards multi-spatiotemporal-scale generalized pde modeling, arXiv:2209.15616, 2020.
[76] N. Carey, L. Zanisi, S. Pamela, V. Gopakumar, J. Omotani, J. Buchanan, J. Brandstetter, Data efficiency and long term prediction capabilities for neural operator surrogate models of core and edge plasma codes, arXiv:2402.08561, 2024, https://doi.org/10.1088/1741-4326/ad313a.
[77] V. Gopakumar, S. Pamela, L. Zanisi, Z. Li, A. Gray, D. Brennand, N. Bhatia1, G. Stathopoulos, M. Kusner, M.P. Deisenroth, A. Anandkumar, JOREK Team MAST Team, Plasma surrogate modelling using Fourier neural operators, Nucl. Fusion 64 (5) (2024) 056025, https://doi.org/10.1088/1741-4326/ad313a.
[78] S. Pamela, Neural-Parareal RMHD dataset, Zenodo 10.5281/zenodo.11099527 (available upon publication), https://doi.org/10.5281/zenodo.11099527.
[79] S. Pamela, Neural-Parareal electrostatic dataset (batch 0001-0500), Zenodo 10.5281/zenodo.11099659 (available upon publication), https://doi.org/10.5281/zenodo.11099659.
[80] S. Pamela, Neural-Parareal electrostatic dataset (batch 0501-1000), Zenodo 10.5281/zenodo.11099671 (available upon publication), https://doi.org/10.5281/zenodo.11099671.
[81] S. Pamela, Neural-Parareal electrostatic dataset (batch 1001-1500), Zenodo 10.5281/zenodo.11099679 (available upon publication), https://doi.org/10.5281/zenodo.11099679.
[82] S. Pamela, Neural-Parareal electrostatic dataset (batch 1501-2000), Zenodo 10.5281/zenodo.11099685 (available upon publication), https://doi.org/10.5281/zenodo.11099685.

[83] Lu Lu, Pengzhan Jin, George Em Karniadakis, Deeponet: learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators, arXiv preprint, arXiv:1910.03193, 2019.
[84] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, George Em Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, Nat. Mach. Intell. 3 (3) (2021) 218–229.
[85] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar, Neural operator: graph kernel network for partial differential equations, arXiv preprint, arXiv:2003.03485, 2020.
[86] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar, Neural operator: learning maps between function spaces, arXiv preprint, arXiv:2108.08481, 2021.
[87] Zongyi Li, Miguel Liu-Schiaffini, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar, Learning chaotic dynamics in dissipative systems, Adv. Neural Inf. Process. Syst. 35 (2022) 16768–16781.
[88] Sifan Wang, Paris Perdikaris, Long-time integration of parametric evolution equations with physics-informed deeponets, J. Comput. Phys. 475 (2023) 111855.
[89] Phillip Lippe, Bas Veeling, Paris Perdikaris, Richard Turner, Johannes Brandstetter, Pde-refiner: achieving accurate long rollouts with neural pde solvers, Adv. Neural Inf. Process. Syst. 36 (2024).
[90] Z. Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE Trans. Image Process. 13 (4) (2004) 600–612, https://doi.org/10.1109/TIP.2003.819861.
[91] S. Pamela, Neural parareal github repo, https://github.com/spamela/neural_parareal.
[92] PETSc, https://petsc.org/.