Imperial College London

Department of Earth Science and Engineering

MSc in Applied Computational Science and Engineering

Independent Research Project
Final Report

Domain Decomposition and Generative Adversarial Networks for modelling fluid flow

by

Jón Atli Tómasson

jon.tomasson20@imperial.ac.uk
GitHub login: acse-jat20

Supervisors:

Dr. Claire Heaney
Prof. Christopher Pain

August 2021

GitHub repository: https://github.com/acse-jat20/DD-GAN

ABSTRACT

Supplementing conventional numerical methods for modelling fluid dynamics with modern data-intensive procedures is a fast-growing field as conventional modelling is expensive and often unpractical. Modern AI methods such as generative adversarial networks that have been utilized with great success in mimicking physical phenomena in other fields show great promise for this purpose. This work presents a novel workflow for utilizing generative adversarial networks aided by reduced-order modelling for predicting fluid dynamics in time. The proposed routine includes domain decomposition to improve computational efficiency and scaling potential, which allows for variable fidelity modelling. Comprehensive validation is performed using flow past a cylinder in high aspect ratio domains, and a comparison is made to an adversarial neural network. The novel Domain Decomposition Generative Adversarial Network (DD-GAN) is the first successful implementation of a GAN that uses Domain Decomposition for the prediction of fluid flow. It does so accurately for short, medium and long term predictions using only data from two consecutive time steps as an input.

INDEX TERMS Fluid dynamics, machine learning, principal component analysis, time-series prediction, generative adversarial networks

1 Introduction

Highly computationally intensive procedures have been an integral part of fluid dynamics for decades. While advances in computational architecture and hardware have allowed for higher-fidelity simulations and speedups of orders of magnitude for fluid flow simulation, utilizing conventional numerical and heuristic methods largely remains a staple [1, 2, 3]. Thus, despite advances, the computational cost can make simulations impractical for use, especially as demand for increased resolution persists. Various modern methods have been proposed to supplement or replace computational fluid dynamics to lower computational costs [4]. Examples include modern AI methods such as neural networks [5, 6, 7], convolutional neural networks [8, 9], reinforcement learning [10], recurrent neural networks [11, 12] and generative adversarial networks or GANs [13].

A GAN is composed of a generator and a discriminator, also known as a critic, competing in a zero-sum game [13]. In this game, the generator's role is to digest the training set and to produce forgeries. The critic's role is to verify whether a particular input is a part of the original data set or a forgery. As the generator becomes better at forging images, the classifier gets better at spotting fakes. After the training process, one can use the generator to produce an image or the trained critic in classification.

The original use of GANs was image generation. Since then, newfound applications for GANs include image upscaling [14], semantic identification [15], virus modelling [16, 17], Spatio-temporal model prediction [18], text to video generation [19] and more. These are all use cases where GANs have been used successfully for extremely high dimensional systems. High-dimensional systems are made more manageable, and computational cost is kept down by using reduced-order modelling (ROM).

An example of ROM is Non-Intrusive Reduced Order Model (NIROM) constructed using proper orthogonal decomposition (POD) [20]. POD is an effective dimensionality reduction tool used in various computational physics sub-fields when it can be of benefit to reduce the dimensionality of the problem space, including fluid dynamics [21, 22, 23, 24]. The system dimensionality is a significant impact factor in the efficiency of any computational fluid dynamics solver as each measured, simulated or equivalently predicted point within an n-dimensional grid adds a problem dimension that the GAN needs to map to its latent space. By reducing the dimensionality, the problem space is kept manageable, and the training of the GANs becomes computationally cheaper and more stable.

To further improve efficiency, this project includes domain decomposition [25]. Domain decomposition is immensely valuable, for instance, in cases where aspect ratios are tricky, such as where the domain is long and thin, e.g. flow in a long pipe, or when dealing with localized complex problems, e.g. around a turbine blade. In these cases, one can split a large simulation into multiple subdomains depending on problem complexity by utilizing domain decomposition. Then each section can have the exact POD basis functions required to model the existing physics accurately. The effects of this are both that the computing time can be drastically improved, and that the fidelity of simulations improves by allowing hard-to-model sections access to more coefficients.

To demonstrate domain decomposition and generative adversarial network's (DD-GAN) performance by comparing the achieved results to a predictive adversarial network [26]. The test case here is a high-fidelity flow past a cylinder simulation at transitional Reynolds number flows in high aspect ratio domains. Quantitative results suggest the method to be stable and qualitative results indicate that the DD-GAN is suitable as a fluid flow forecasting tool with almost no quality degradation.

2 Methodology

This section formulates a fluid flow problem as a time series prediction problem. Subsequently, a framework for using generative adversarial networks to solve a time series prediction problem is discussed. Finally, the problem is reformulated with domain decomposition in mind, and an outline of the algorithm will be provided.

2.1 Fluid flow as time series prediction problem

Consider a sample matrix $\boldsymbol{X} = [\boldsymbol{x}^1 \dots x^{n_\mathrm{S}}]$ where $\boldsymbol{x}^i$ constitutes the i-th velocity snapshot array of n nodes, each containing the x, y, z directional velocity components u, v, w, where

$$\boldsymbol{x}^i = [u_1^i \dots u_n^i, v_1^i \dots v_n^i, w_1^i \dots w_n^i]^T \tag{1}$$

and n_S is the number of snapshots. The goal is to predict a velocity snapshot given several sequential snapshots. These snapshots are created using conventional simulation methods. An example of a plotted velocity snapshot can be seen in Figure 1a.

By considering the modulus of the velocity at each time step as a point in observable $3n$ dimensional space, the velocities represent a set of directions with correlations identified via principal component analysis. To do this, consider $\boldsymbol{S}$ as the covariance matrix of the mean subtracted $\boldsymbol{X}$ and apply diagonalization as follows:

$$\boldsymbol{S} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{V}^{-1} \tag{2}$$

where $\boldsymbol{V} = [\boldsymbol{v}_1 \dots \boldsymbol{v}_{3n}]$ is a matrix of normalized eigenvectors corresponding to the eigenvalues α_j arranged in decreasing magnitude in the diagonal matrix $\boldsymbol{\Sigma}$. The eigenvalues represent the variances of all points in the direction of the corresponding eigenvectors. The normalized eigenvalues represent the relative variance along the jth direction.

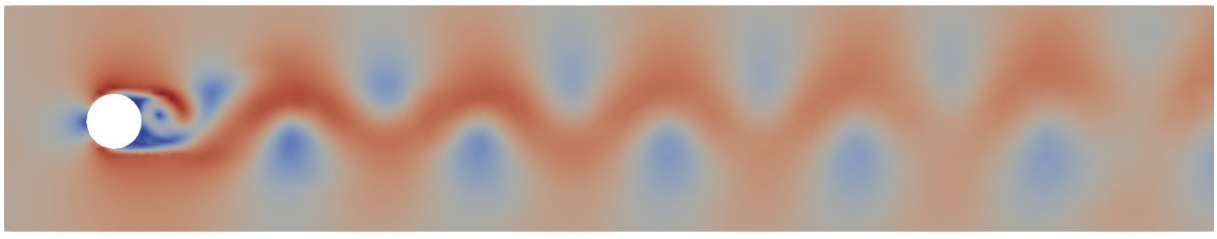
$$\varphi_j = \frac{\alpha_j}{\sum_{k=1}^{n} \alpha_k} \tag{3}$$

The rationale for decomposing the system is to limit the solution subspace to directions corresponding to a subset of the principal directions selected by their relative importance governed by the values of α. Therefore, reducing the problem complexity transforms the problem into a m dimensional time series problem, where m is a design variable chosen in a fidelity/cost trade-off. After this decomposition, a velocity snapshot $\boldsymbol{x}$ can be expressed by

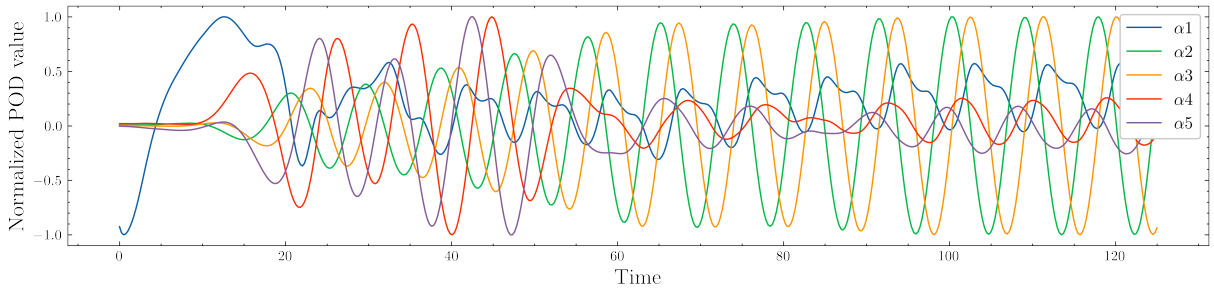
$$\boldsymbol{x} = \bar{\boldsymbol{x}} + \sum_{j=1}^{m} \alpha_j \boldsymbol{v}_j \tag{4}$$

where $\bar{\boldsymbol{x}}$ is the mean of $\boldsymbol{x}$ [27]. By the effect of this reduction, the output of the GAN will be a combination of normalized POD coefficients with values ranging from -1 to 1 over several time levels that represent the system's dynamics, as seen in Figure 1b.

The latent space of a well trained GAN with this data will then contain every possible combination of POD coefficients as they appear in a training set. By pattern matching sets of POD coefficients over several time levels within the latent space, one can then predict the following step by finding the logical continuation of the previous time levels. By appending the forecast to the pattern matching set, one can continue making predictions one time level at a time (cf. Section 2.2).



(a)



(b)

Figure 1: Two different representations of flow past cylinder: (a) Velocity snapshot example. Red denoted areas of high velocity and blue of low velocity, (b) Normalized first 5 POD coefficients for the first 125 seconds of the flow past a cylinder simulation. The original problem is reformulated to be the prediction of these coefficients.

The basis behind domain decomposition is implementing this routine several times within a single simulation. It revolves around splitting the simulated domain into multiple sub-domains to find the dominating POD coefficients and basis functions for each sub-domain. By doing so, it is possible to describe the typical changes of each domain more accurately with tailored basis functions and coefficients to each sub-domain. Domain decomposition can also positively affect the computational cost as efficiency improves by reducing POD coefficients in simpler domains, where the velocity variance is easily decomposable (cf. Section 2.3).

2.2 GANs for time series prediction

Here an outline of a simplified PredGAN is presented [17] to explain how a GAN can be used for a simulation prediction in time. This section also covers an explanation of the Wasserstein GAN with a gradient penalty (WGAN-GP) for time series prediction [28]. Finally, the two introduced concepts will be tied together to form a basic layout of the DD-GAN framework.

As mentioned above the fluid flow prediction can be put forward as a time series prediction problem where the goal is to predict a time step given consecutive POD coefficients (cf. Section 2.1). In this case a set of real samples, $\boldsymbol{Z} = [\boldsymbol{z}^1 \dots \boldsymbol{z}^{n_\mathrm{S}-n_\mathrm{P}-1}]^T$, containing n_P equally spaced sets of POD coefficients from $\boldsymbol{A} = [\boldsymbol{\alpha}^1 \dots \boldsymbol{\alpha}^{n_\mathrm{P}}]$, where $\boldsymbol{\alpha^i} = [\alpha_1^i \dots \alpha_m^i]$ are provided. Assuming that the spacing, Δi equals 1, the data can be represented as

$$\boldsymbol{Z} = \begin{bmatrix} \boldsymbol{\alpha}^1 & \boldsymbol{\alpha}^2 & \cdots & \boldsymbol{\alpha}^{n_\mathrm{P}} \\ \boldsymbol{\alpha}^2 & \boldsymbol{\alpha}^3 & \cdots & \boldsymbol{\alpha}^{n_\mathrm{P}+1} \\ \vdots & \vdots & & \vdots \\ \boldsymbol{\alpha}^{n_\mathrm{S}-n_\mathrm{P}-1} & \boldsymbol{\alpha}^{n_\mathrm{T}+1} & \cdots & \boldsymbol{\alpha}^{n_\mathrm{S}} \end{bmatrix}. \tag{5}$$

In practice, a suitable size for Δi should be large enough to make significant differences between time steps, but not too large so that all simulation details are lost (cf. Section 2.4). The generator G

role is to forge these values $\tilde{\boldsymbol{z}} = G(\boldsymbol{t})$, where the input $\boldsymbol{t}$ is the latent vector sampled from a normal distribution $\boldsymbol{t} \sim p(\boldsymbol{t})$ which the generator deterministically transforms to an ever more accurate forgery of actual data points. Here the role of the critic is to identify the validity of a sample to see if it is a generated sample or originates from the original set. The zero-sum game between the critic D and generator G is thus constructed using the Wasserstein-1 distance $W(q, p)$ and is

$$\min_{G} \max_{D \in \mathcal{D}} \mathop{\mathbb{E}}_{\boldsymbol{z} \sim \mathbb{P}_r} [D(\boldsymbol{z})] - \mathop{\mathbb{E}}_{\tilde{\boldsymbol{z}} \sim \mathbb{P}_g} [D(\tilde{\boldsymbol{z}})] \tag{6}$$

where $\mathcal{D}$ represents a set of 1-Lipschitz functions. Here the two distribution $\mathbb{P}_r$ and $\mathbb{P}_g$ are the training data distribution and model distribution, respectively.

This means the generator tries to map the sampled arrays from the normal distribution to the pattern of actual samples. This is an incremental procedure and can often be tricky, especially if the critic becomes better at recognizing the actual samples faster than the generator improves. In that case, the development of the generator plateaus due to a possible variety of reasons, including vanishing gradients and mode collapse.

Some standard methods to combat the overbearing behaviour of the critic include making the critic explicitly worse by performing gradient ascent regularly, making its job more challenging by adding noise to the fake and actual samples [29], and clipping the critics weights [30]. Despite these methods, GANs are extremely hard to train in practice, and remedies remain an active field of research.

One way to ease the training process is to add a gradient penalty term. The penalty term has the consequence that it encourages the 1-Lipschitz property mentioned above, which should improve the trainability of the GAN [28]. The critic loss function L_c therefore becomes

$$L_c = \mathop{\mathbb{E}}_{\boldsymbol{z} \sim \mathbb{P}_r} [D(\boldsymbol{z})] - \mathop{\mathbb{E}}_{\tilde{\boldsymbol{z}} \sim \mathbb{P}_g} [D(\tilde{\boldsymbol{z}})] + \lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{z}} \sim \mathbb{P}_{\hat{\boldsymbol{z}}}} [(\|\nabla_{\hat{\boldsymbol{z}}} D(\hat{\boldsymbol{z}})\|_2 - 1)^2] \tag{7}$$

where $\hat{\boldsymbol{z}} \leftarrow \epsilon \tilde{\boldsymbol{z}} + (1 - \epsilon)\boldsymbol{z}$ and ϵ is a random number $\epsilon \sim U[0, 1]$. This means that $\mathbb{P}_{\hat{\boldsymbol{z}}}$ is implicitly defined as a uniform sampled points somewhere between points from the two distributions $\mathbb{P}_r$ and $\mathbb{P}_g$. The purpose of the gradient penalty coefficient λ is to decide the strength of the penalty term. During a cycle of training the GAN, the critic might be updated multiple times using equation (7). Following this, the generator is trained by minimizing the following loss function

$$L_g = -D(G(\boldsymbol{t})). \tag{8}$$

Once the generator and critic are sufficiently trained, the generators mapping from the latent space to the real space can be explored. By inputting a latent vector $\boldsymbol{t}$, the output of the generator will be a plausible sample at some unknown time. By setting a role model sample at a decided time $\boldsymbol{z}^i$ we can search the latent space for a best fitting latent vector $\boldsymbol{t}^{i+1}$ that minimizes the scalar merit function $U(G(\boldsymbol{t}), \boldsymbol{z}^i)$. The task is thus the nonlinear minimization

$$\mathbf{t}^{i+1} = \arg\min_{\mathbf{t}} U(G(\boldsymbol{t}), \boldsymbol{z}^i), \tag{9}$$

where U is a root-mean-square deviation of the first $n_{\mathrm{P}} - 1$ sets of POD coefficients in $\boldsymbol{z}^{\boldsymbol{i}}$ and $\tilde{\boldsymbol{z}}^i$, multiplied by the variance explained by each coefficient. The sensitivities of equation (7) can be automatically generated and backpropagated through the generator. This results in pattern matching where a single set of POD coefficients are forecasted. By appending the forecasted set of coefficients and removing the first set, one can indefinitely predict the future POD coefficients simply by repeating the process.

If the GAN is unstable and the latent space is ill-conditioned, searching the latent space becomes immediately problematic as the optimization will stagnate and not predict the system's dynamics. Fixing the stagnated optimization is not an easily resolvable problem, but the architecture and hyperparameters play a considerable role (cf. Section 2.4).

2.3 Domain decomposition

Here the previously introduced forecasting problem is reformulated to include domain decomposition. Subsequently, the dedicated updating regiment between subdomains and forecasting procedure is proposed.

Assuming that one has a collection of velocity snapshots where each snapshot is decomposable into n_{D} subdomains where $1 \leq d \leq n_{\mathrm{D}}$ denotes the domain number. By performing POD on the entire domain and finding the best fitting coefficients for each subdomain, one can describe the dynamics of each individual domain. A plot of this can be seen in Figure 2 where the initial flow propagates through the domains, one by one. The rationale for the decomposition of a larger domain into smaller subdomains includes this propagation effect as it allows for prediction in time and space by extending the collection of domains.

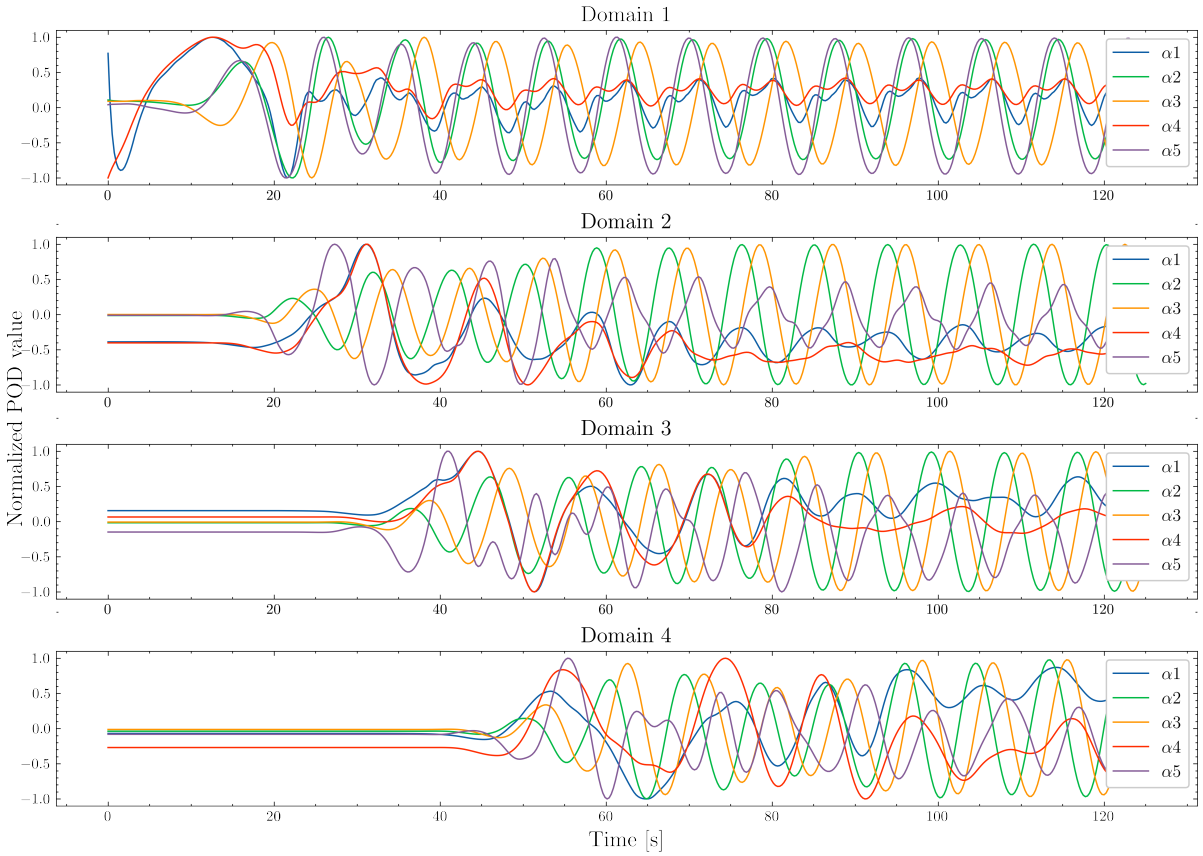


Figure 2: Normalized first 5 POD coefficients for the first 500 time steps (125 seconds) of the flow past a cylinder test case split into 4 subdomains. As can be seen the initial disturbance takes just over 40 seconds to reach the final domain.

In order to train the GAN to understand this propagation, it requires the testing data to be not only a collection of a single domains POD coefficients. It also requires the input data to include coefficients from immediate neighbours as well. By constructing the problem in this way, there must be a boundary at some point. One can either create an artificial one or use known values in the edge domains.

The test case outlined here assumes that the inlet and outlet (left- and rightmost subdomain) are boundary conditions, and the two central subdomains are the subject of prediction. In this case, the problem then becomes predicting the POD coefficients in the inner subdomains. In a two dimensional example such as a flow past a cylinder, this means the number of predicted domains is equal to $n_{\mathrm{D}} - 2 = 2$, and that the number of neighbours is 2. The number of time steps from each neighbour in a single training data sample can be denoted as the integer array $\mathbf{n} = [n_{\mathrm{b}}, n_{\mathrm{a}}, n_{\mathrm{self}}]$. Here n_{b} is the number of time steps in the domain before the one examined (neighbour on the left), n_{a} is the number of time steps in the domain after the one examined (neighbour on the right) and n_{self} is the number of time steps in the inspected domain. A value of 1 means that only the future time level is used.

The central idea of using domain decomposition is to utilize the information from neighbouring domains in the prediction stage. This can allow the GAN to learn pattern matching that can lead to forecasting past the training data in space and time and introduce variable fidelity models. In a variable fidelity GAN, the number of predicted coefficients depend on the complexity of the flow located within them. This transfers into extremely high fidelity within more complex domains and instantly realizable computational savings by decreasing the number of coefficients used in simpler domains. The simplicity in which these gains are realized is huge as the only thing changing in the training process is simply a reformulation of the input data.

A visualization of the DD-GAN training process can be seen in Figure 3. The only difference in training the generator and critic between a DD and non-DD GAN is the structure of the coefficients. However, there is quite a difference in the way the two methods predict.

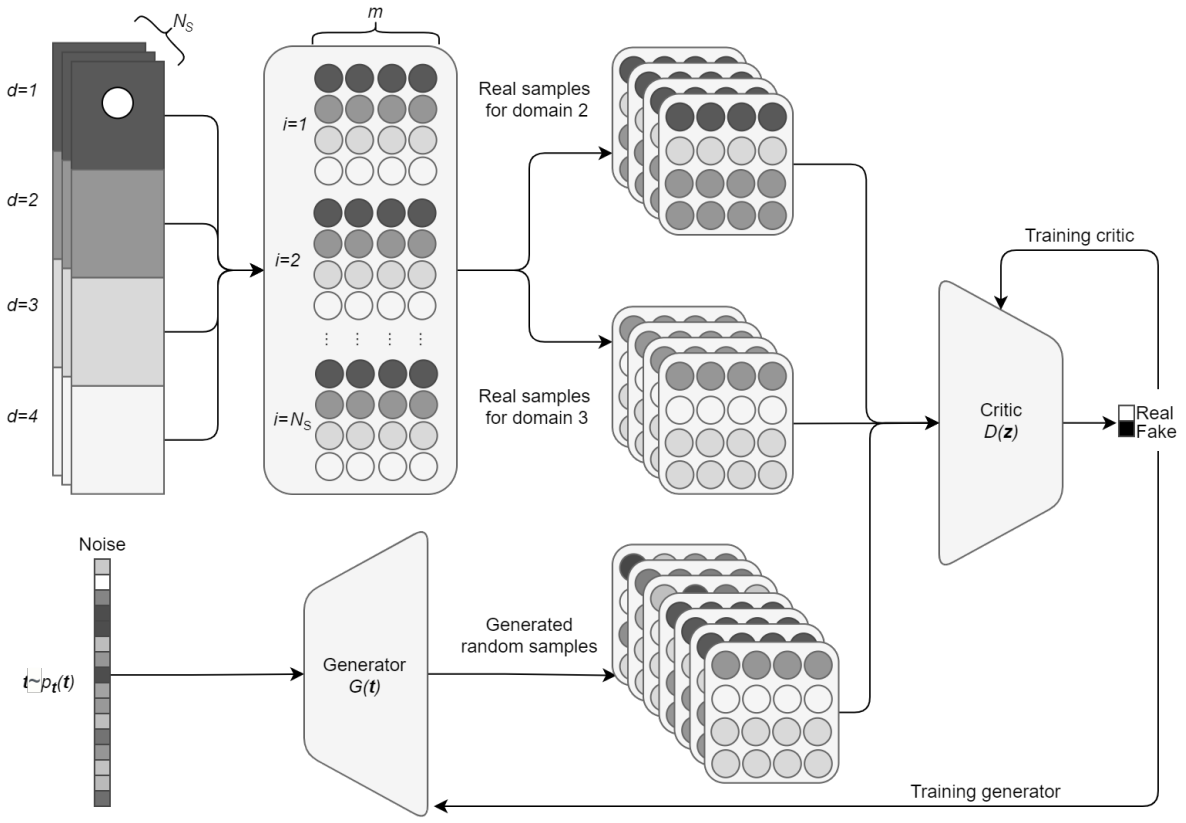


Figure 3: Flowchart illustrating training of the generator and critic assuming four subdomains. In the scenario here domain 1 and 4 act as boundary conditions and domains 2 and 3 are predicted. Here $\boldsymbol{n} = [1, 1, 2]$ and individual POD coefficients are described as circles. In actuality, critic inputs are one-dimensional versions of what is depicted here.

As outlined in Section 2.2, the GAN can generate POD coefficients by pattern matching within the latent space and solving the nonlinear minimization problem outlined in (9). When performing this routine within a domain decomposition framework, the same principles apply. Coefficients from all-time levels are used, except for predicted future time levels within an expected domain. However, not all values are known as they were in Section 2.2. When starting a prediction, there might be multiple unknowns. When dealing with four subdomains in total, the values of the two central subdomains are unknown at any predicted time. Furthermore, the unknown values for the two subdomains are semi-reliant on one another, and only one of these values can be predicted at a time.

Thus for every time step predicted, there must be an initial assumption of an almost realistic value for the non-boundary laying neighbours before then iterating through the domains for each prediction. Once an initial guess has been made and the GAN has made a prediction, the predictions can be broadcasted to the neighbours. This process is performed up and down the subdomains, first following the direction of the flow and then reversing, each time broadcasting the updated values. This iteration through the domains is repeated until convergence.

While this process is close to being stable, using pure noise as initial values for the unknown coefficients is not recommended as the GAN has not been explicitly conditioned for poor inputs. Thus using random guesses might result in slow or no convergence. Alternatively, these initial guesses can be a previous value or a value found using first or second-order estimation based on previous time levels. This helps ensure stability in the model and improves convergence by a significant margin. A rough illustration of this process can be seen in Figure 4.

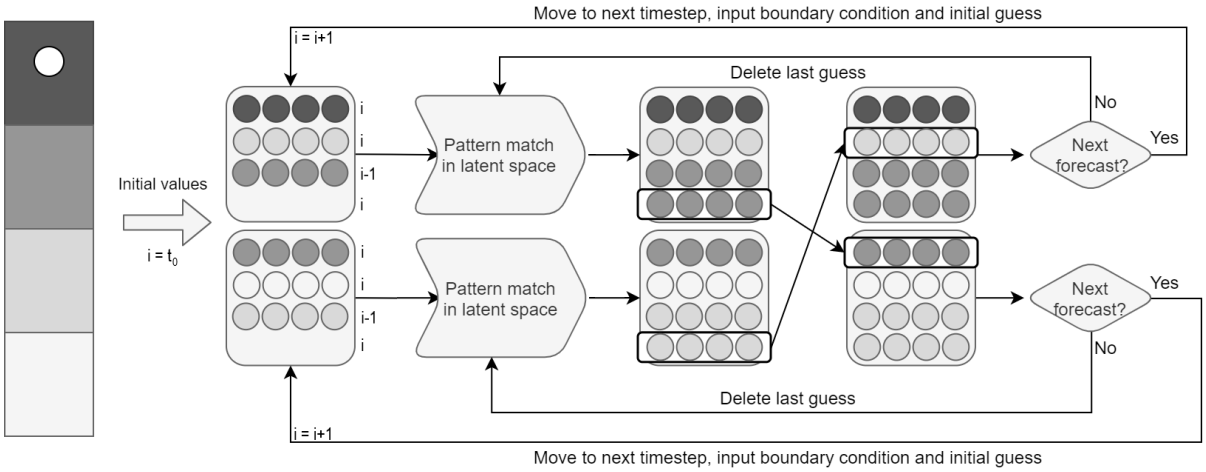


Figure 4: Flowchart illustrating an infinite optimization/prediction stage for domains 2 and 3 with $\boldsymbol{n} = [1, 1, 2]$ assuming $m = 4$. The pattern matched time values match the time value being predicted, except for the domain that is being predicted, as can be seen from the missing values in the figure. In actuality, the data is a ravelled one-dimensional version of what is depicted here in the squares.

2.4 Algorithm workflow

This section contains a review of the algorithm flow in its entirety, along with a review of the methods utilized to improve the training of the GAN and its predictions. Firstly the training of the GANs will be covered.

Remedies to assist with the training of GANs remain an active field of research. Some standard methods aim to improve the training of the GAN by using data augmentation strategies and regular gradient ascent for the critic. In training the GAN as presented here, these methods were of little use. The existence of stability when training the DD-GAN was mostly determined by the architecture of

the generator/critic pair and the setup of the input data. Therefore, the primary update schemes used to produce the results in Section 4 are exactly as described in Algorithm 1, without the addition of data augmentation or other methods.

Algorithm 1 Training stage of the modified WGAN with gradient penalty. Adapted from the original paper [28].

Require: Adam hyperparameters ε_{C}, β_{C1}, β_{C2}
Require: Nadam hyperparameters ε_{G}, β_{G1}, β_{G2}
Require: Gradient penalty coefficient λ, critic iterations per generation n_{critic}, batch size n_{batch}
Initialize generator weights θ_0 and initial critic parameters w_0 randomly
while θ has not converged **do**
  for c $= 1, \ldots, n_{\mathrm{critic}}$ **do**
    for i $= 1, \ldots, n_{\mathrm{batch}}$ **do**
      Sample actual data $\boldsymbol{z}^{(i)} \sim \mathbb{P}_r$
      Sample latent variables $\boldsymbol{t}^{(i)} \sim p(\boldsymbol{t})$
      Sample a random number $\epsilon \sim U[0,1]$
      $\tilde{\boldsymbol{z}}^{(i)} \leftarrow G_\theta(\boldsymbol{z}^{(i)})$
      $\hat{\boldsymbol{z}}^{(i)} \leftarrow \epsilon\tilde{\boldsymbol{z}}^{(i)} + (1-\epsilon)\boldsymbol{z}^{(i)}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{z}}^{(i)}) - D_w(\boldsymbol{z}^{(i)}) + \lambda(||\nabla_{\hat{\boldsymbol{z}}} D_w(\hat{\boldsymbol{z}}^{(i)})||_2 - 1)^2$
    end for
    $w \leftarrow \mathrm{Adam}\left(\nabla_w \frac{1}{n_{\mathrm{batch}}} \sum_{i=1}^{n_{\mathrm{batch}}} L^{(i)}, w, \varepsilon_{\mathrm{C}}, \beta_{\mathrm{C1}}, \beta_{\mathrm{C2}}\right)$
  end for
  $\theta \leftarrow \mathrm{Nadam}\left(\nabla_\theta \frac{1}{n_{\mathrm{batch}}} \sum_{i=1}^{n_{\mathrm{batch}}} -D_w\left(G_\theta(\boldsymbol{t})\right), \theta, \varepsilon_{\mathrm{G}}, \beta_{\mathrm{G1}}, \beta_{\mathrm{G2}}\right)$
  Sample latent variables $\left\{\boldsymbol{t}^{(i)}\right\}_{i=1}^{n_{\mathrm{batch}}} \sim p(\boldsymbol{t})$
end while

While other parameters such as the learning rates of the GAN were impactful, their tuning could not make up for a poorly adjusted data set. An adjusted data set in this case would be one with a good value of Δi, a good sample size, and appropriately sized values in $\boldsymbol{n}$. From testing, the results suggest that running two separate predictions with a value of $\Delta i = 4$, starting at two time steps apart, and combining them would make for a better result than a single one with a $\Delta i = 2$ or 1. This appears to have to do with the GAN having a hard time describing small changes while still fooling the critic and not tending to plateau when forecasting. Therefore using a $\Delta i = 4$ and weaving predictions together is done when generating results in Section 4.

The step size is fairly connected to the number of consecutive time steps in an inspected subdomain and its neighbours $\boldsymbol{n} = [n_{\mathrm{b}}, n_{\mathrm{a}}, n_{\mathrm{self}}]$. Initial results suggest that training of the GANs becomes harder as the number of POD coefficient sets increase. The value used therefore was the minimum viable amount of $\boldsymbol{n} = [1, 1, 3]$.

The subdomain update-communication regiment seen in Algorithm 2 is the backbone of the method proposed here. Supplying initial guesses for unknown values, as seen in line 6 of Algorithm 2, plays a significant role in the convergence speed. Experimentation suggests that the better the guess, the better the convergence, but a carryover of values from a previous time step is sufficient for a working implementation. Section 4 demonstrates this as previous values make up the initial guesses when generating the results.

In the shape proposed here, the forecasting scheme seems to be somewhat insensitive to the hyperparameters selected. Examples of working values for n_{cycles} range from 10 and upward. Similarly, the number of optimizer epochs do not seem to be especially relevant, assuming that the total number of epochs crosses a value of around 1000. The most probable reason for a failing forecasting round alternatively seems to be the quality of the trained generator.

Algorithm 2 Prediction stage of the modified WGAN with gradient penalty

Require: Adam hyperparameters ε_{L}, β_{L1}, β_{L2}
Require: Number of prediction iteration cycles n_{cycles}, number of predictions n_{preds}, latent vector disturbance standard deviation σ_{latent}
Require: Initial POD value arrays for each of the n_{domains} predicted domains $\boldsymbol{Z}^{(0)} = [\boldsymbol{z}_1^{(0)}, \ldots, \boldsymbol{z}_{n_{\mathrm{domains}}}^{(0)}]$
Require: Set of boundary conditions for each prediction
Initialize latent arrays for each domain $\boldsymbol{T}^{(0)} = [\boldsymbol{t}_1^{(0)}, \ldots, \boldsymbol{t}_{n_{\mathrm{domains}}}^{(0)}]$, randomly
for $i = 1, \ldots, n_{\mathrm{preds}}$ **do**
  $\boldsymbol{T}^{(i)} \leftarrow \boldsymbol{T}^{(i-1)}$
  $\boldsymbol{Z}^{(i)} \leftarrow \boldsymbol{Z}^{(i-1)}$
  Update $\boldsymbol{z}_1^{(i)}$ and $\boldsymbol{z}_{n_{\mathrm{domains}}}^{(i)}$ with boundary values
  Supply an initial guess for unknown values
  for $j = 1, \ldots, n_{\mathrm{cycles}}$ **do**
    for $k = 1, \ldots, n_{\mathrm{domains}}, \ldots, 2$ **do**
      Sample perturbation array $\boldsymbol{\epsilon} \sim N(0, \sigma_{\mathrm{latent}})$
      $\boldsymbol{t}_k^{(i)} \leftarrow \boldsymbol{t}_k^{(i)} + \boldsymbol{\epsilon}$
      $\boldsymbol{t}_k^{(i)} \leftarrow \mathrm{Adam}(U(G(\boldsymbol{t}_k^{(i)}), \boldsymbol{z}_k^{(i)}), \boldsymbol{t}_k^{(i)}, \varepsilon_{\mathrm{L}}, \beta_{\mathrm{L1}}, \beta_{\mathrm{L2}})$
      $\boldsymbol{z}_k^{(i)} \leftarrow G(\boldsymbol{t}_k^{(i)})$
      Broadcast the predicted coefficients in $\boldsymbol{z}_k^{(i)}$ to non-boundary neighbours
    end for
  end for
  Sample perturbation array $\boldsymbol{\epsilon} \sim N(0, \sigma_{\mathrm{latent}})$
  $\boldsymbol{t}_1^{(i)} \leftarrow \boldsymbol{t}_1^{(i)} + \boldsymbol{\epsilon}$
  $\boldsymbol{t}_1^{(i)} \leftarrow \mathrm{Adam}(U(G(\boldsymbol{t}_1^{(i)}), \boldsymbol{z}_1^{(i)}), \boldsymbol{t}_1^{(i)}, \varepsilon_{\mathrm{L}}, \beta_{\mathrm{L1}}, \beta_{\mathrm{L2}})$
  $\boldsymbol{z}_1^{(i)} \leftarrow G(\boldsymbol{t}_1^{(i)})$
  Broadcast the predicted POD coefficients in $\boldsymbol{z}_1^{(i)}$ to $\boldsymbol{z}_2^{(i)}$
  Append the latest predictions from $\boldsymbol{Z}^{(i)}$ to a $results$ array
end for
return $results$

3 Code metadata and software

Data collection and forecasting was performed using the following: Processor Intel(R)Core(TM) i7-8650U CPU @ 1.90GHz, 2112 Mhz, 4 Core(s), 8 Logical Processor(s) running on Ubuntu 20.04. The code was run using Python version 3.9.5. For the creation of GAN, *TensorFlow* 2.5.0 [31] was used. Other packages include *Scikit-learn* [32] for the shuffling of data and for the generation of truncated normal distribution, *Numpy* [33] for mathematical operations and data processing, *Matplotlib* [34] and *SciencePlots* [35] for visualization. *ParaView* [36] was used for visualization of fluid flow.

The prepossessing code used for the generation is a pre-compiled Fortran library. Prof. Christopher Pain and Dr. Claire Heaney supplied the Fortran code and the appropriate simulation .vtu files, with Python 2 wrapper functions written by Dr. Claire Heaney and Zef Wolff. The version used to run these files is *Python* 2.7.18 with *Numpy* version 1.16.6 installed.

The original implementation of a non-DD Wasserstein GAN was supplied with this project by Claire Heaney. The backbone of the code is two wrapper classes written by the author. The first one contains the necessary functions for the training of the GAN, and the second includes the necessary functions for the prediction. Both these classes were made to work with domain decomposition and

non-domain decomposition frameworks in mind. Usage instructions are included in readme files and in the example folder in the GitHub repository.

The repository, linked at the top of the paper, also includes a set of example notebooks used to generate the final results and a set of readme files that guide the user through installing and importing the required packages. The repository also includes the data used and everything necessary to replicate the results. All code written has been extensively tested and includes a GitHub workflow with a set of weak tests that guarantee that everything is operational. Section 4 includes both qualitative and quantitative results confirming the final functionality of the method.

4 Results

This section provides numerical and qualitative experimental verification of the proposed domain decomposition and generative adversarial network. The results include a comparison to a predictive adversarial network [26] using hyperparameters outlined in Table 1. Discussion on the selection of some of the more important parameters are outlined in Section 2.4. The repeatability of the results presented here is validated by statistical analysis performed for multiple fluid prediction runs executed from random initial times.

Table 1: DD-GAN hyperparameters and descriptions.

Feature	Parameter	Description	Value
GAN	ε_{C}	Critic learning rate	0.0001
	β_{C1}	Critic decay rate 1	0.9
	β_{C2}	Critic decay rate 2	0.999
	ε_{G}	Generator learning rate	0.0001
	β_{G1}	Generator decay rate 1	0.0
	β_{G2}	Generator decay rate 2	0.9
	λ	Gradient penalty coefficient	10
	Δi	Time step size	4
	n_{critic}	Critic iterations per generation	5
	n_{batch}	Training batch size	45
	n_{latent}	Dimensionality of latent space	100
	$epochs_{\mathrm{t}}$	Training epochs	2000
	$\boldsymbol{n}$	Number of time steps per domain	[1,1,3]
Prediction	ε_{L}	Latent space search learning rate	0.005
	β_{L1}	Latent space search decay rate	0.9
	β_{L2}	Latent space search decay rate	0.999
	σ_{latent}	Latent vector disturbance standard deviation	0.05
	$epochs_{\mathrm{p}}$	Training epochs per cycle	50
	n_{cycles}	Number of prediction iteration cycles	20
	n_{preds}	Number of time step predictions	100

4.1 Experimental setup

The verification case is an incompressible high Reynolds number ($Re = \frac{\rho V R}{\mu}$) flow past a cylinder stimulation with a $Re = 3900$ [37]. The behaviour of the simulated fluid flow is initially complex during a transient start-up phase of about 400 time levels ($100\,s$), after which the flow reaches a quasi steady-state.

The selection of the test case was principally so that enough complex behaviour and details could be contained within the simulation without the behaviour being completely random as would be found

with higher Reynolds number flows. The domain inspected is $2.2\,m$ x $0.41\,m$ with a cylinder located $0.2\,m$ from the left boundary. No-slip and no normal flow boundary conditions are applied on the horizontal walls and cylinder. Zero shear and zero normal stress are applied at the outlet.

The first 1800 velocity snapshots were used as a training set for the construction over $450\,s$. Each snapshot contains x and y velocity components at 20550 mesh nodes. For dimensionality reduction purposes, basis functions are constructed using all velocity data for all snapshots. Each snapshot is then decomposed into four subdomains and the velocity in each subdomain is then described using 10 POD coefficients. In all following numerical experiments, $n_{\mathrm{P}} = 10$ coefficients have been selected as they were found to explain $> 98\%$ of the variance of the simulations. This was chosen as a good trade-off between the fidelity of the simulations and the computational expense. The qualitative effects of this decomposition can be seen in Figure 5.

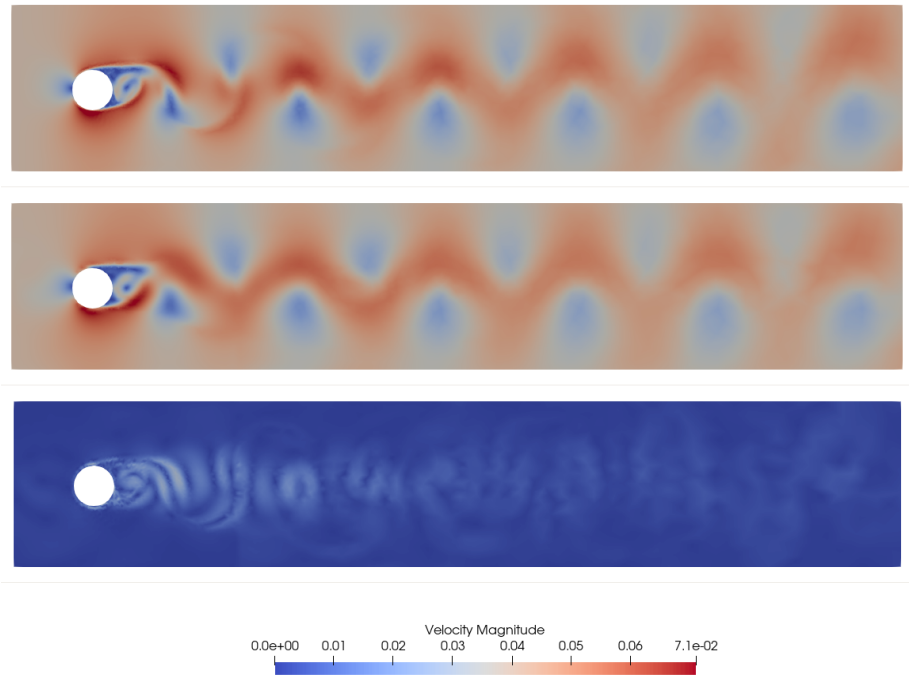


Figure 5: Two flow past a cylinder velocity snapshots at $i = 950$, and their differences. The snapshots are (in order, top to bottom) the original snapshot data, its reconstruction using 10 POD coefficients and a velocity difference representation.

As can be seen, the POD does an excellent job of describing the system with only 10 coefficients. However, there are clear signs that the reconstruction is not perfect. The signs are most prevalent in the wake of the cylinder, where an area of high velocity meets an area of low velocity.

4.2 Numerical results and discussion

In order to verify the training of the GAN, one can look at the development of the samples drawn from the GAN as it is being trained. Originally the samples for the untrained GAN are drawn from the distribution $N(0, 0.01)$. Figure 6 displays a comparison of histograms of the training data to value histograms produced by the GAN after training. As can be seen, the overall shape for each coefficient is firstly unique and secondly being imitated well by the generator.

Figure 7 demonstrates qualitatively that the combination of samples being used when traversing the latent space leads to good predictions with small quality degradation.

As can be seen from Figure 7, the value differences in the two central subdomains are minimal, seen only in a difference plot.

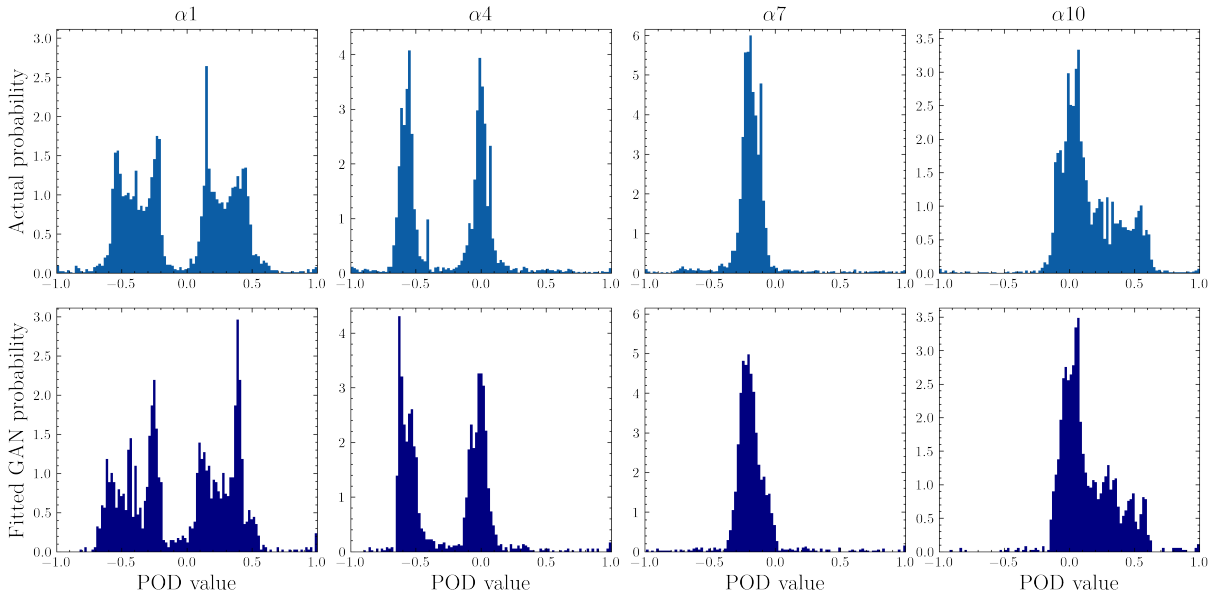


Figure 6: Histograms displaying the distributions of values for various POD coefficients. Here the top row displays the actual occurrences within the training set and the bottom row contains values from the generator

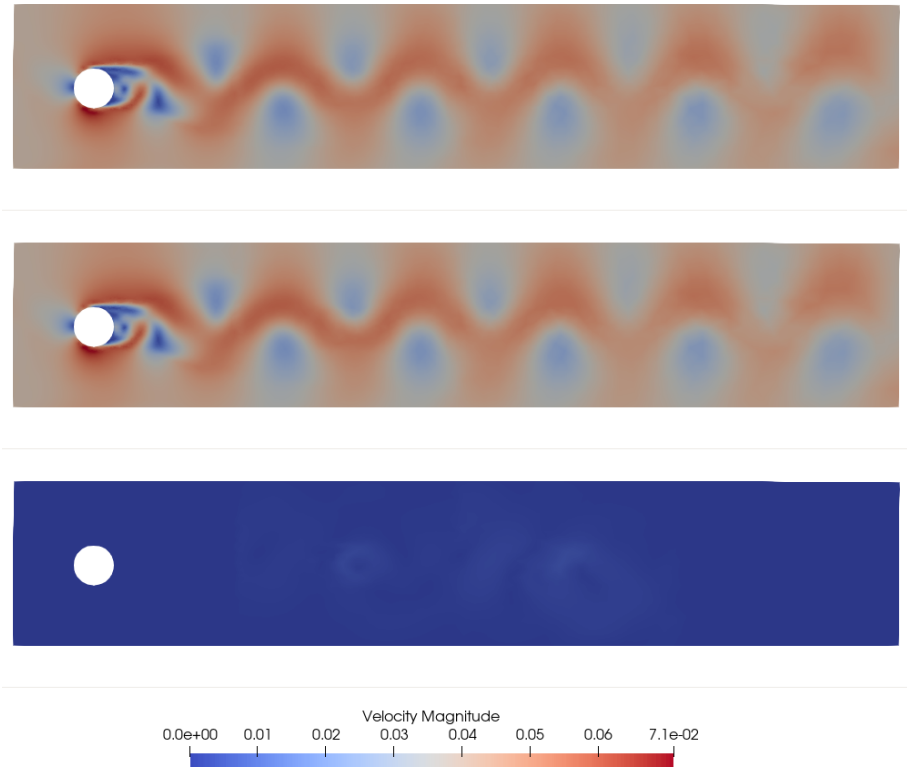
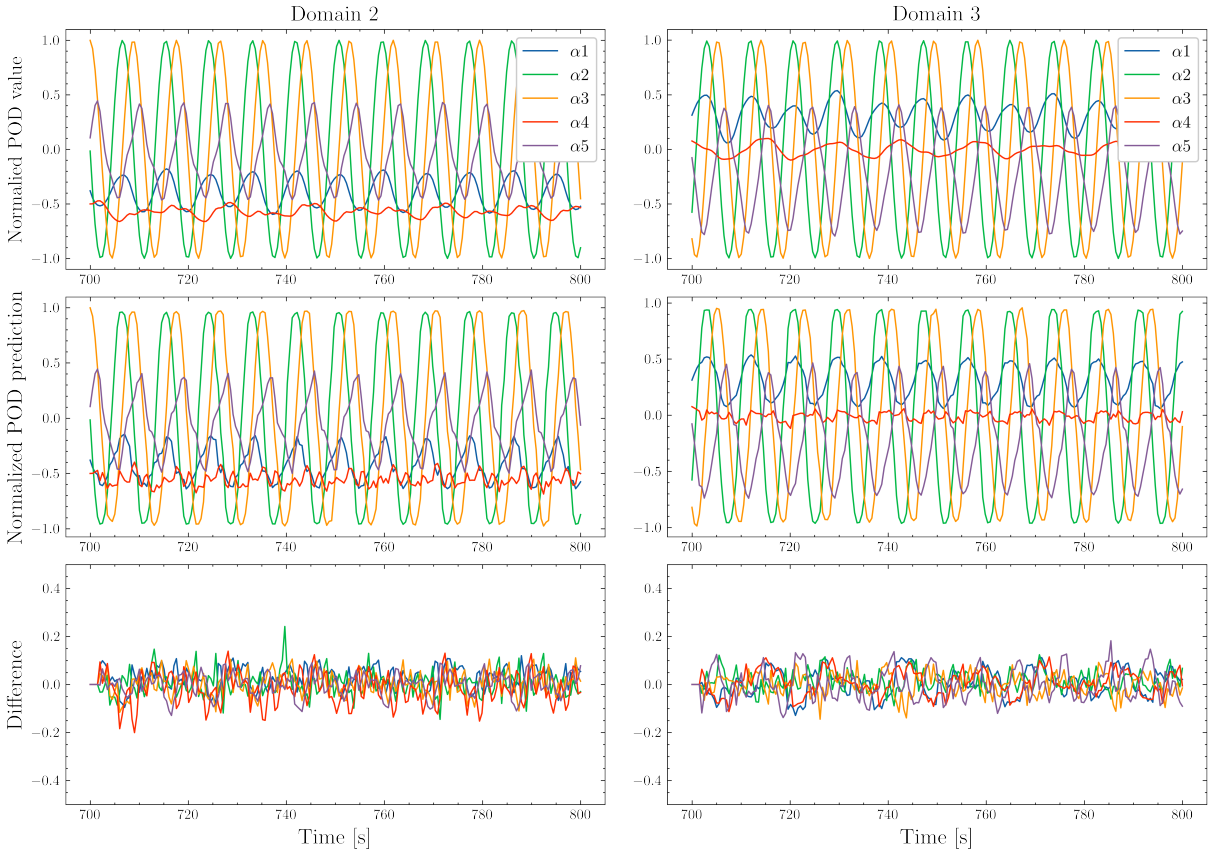
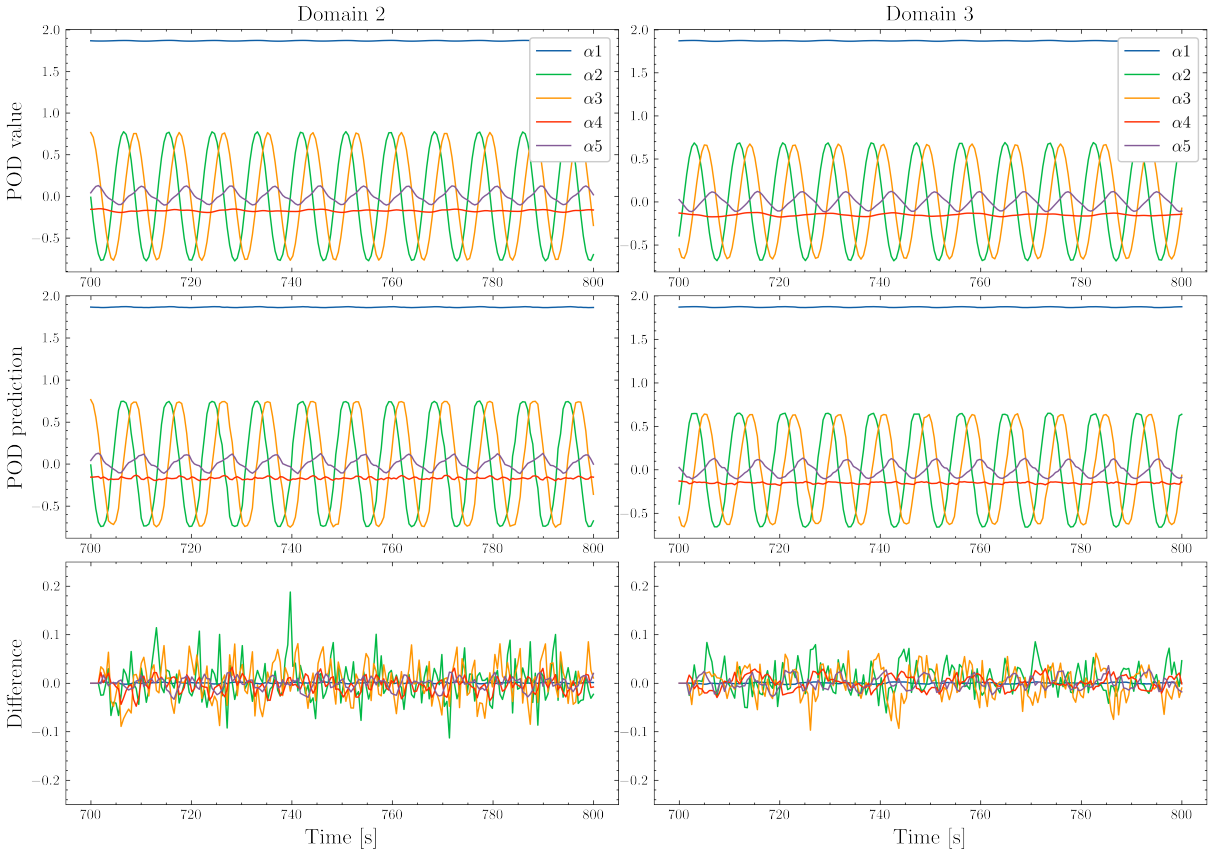


Figure 7: Two flow past a cylinder velocity snapshots at the thirtieth predicted time step at $i = 730$, and their differences. The snapshots are (in order, top to bottom) reconstructed from 10 original POD coefficients, a reconstruction from 10 predicted coefficients and the differences in velocity.

Figure 8 contains the first 5 POD coefficients for 200 consecutively forecasted time steps ($100\,s$) compared to the actual values. The forecasting was done as described in Section 2 where a time-stepping of $\Delta i = 4$ is used but 2 separate prediction runs are started and the results are braided together. Here the forecasts are initialized at $t = 700$ and $t = 702$, respectively.



(a) Normalized prediction results



(b) Rescaled prediction results

Figure 8: Line plots of the first 5 POD coefficients predicted for 200 time steps between the times 700 and 800. Each of the two charts show (from top to bottom) the prediction targets, the predicted values and the difference between the two.

It is worth restating that the POD coefficients have been normalized so the differences as displayed in the bottom third in Figure 8a are therefore independent of the original variance of the coefficients. These differences demonstrate the quality of the method, regardless of the coefficient.

The GAN is stable predicting the consecutive time steps with individual POD errors less than 12.5%, no matter the original variance of the coefficient. The plots in Figure 8b display the rescaled values used to reconstruct the simulation. An interesting pattern is the seemingly stochastic nature of the errors. Since the latent space of the generator is not ideal for searching by default, these results do make sense due to the abundance of local minima located in the high dimensional latent space that needs to be traversed for every prediction. However, due to the absolute values being relatively low, the influence these errors have on the final simulation is low, as can be seen from Figure 7.

Table 2 includes the results of 20 prediction runs where 50 time steps were forecasted. Each starting point was random. In order to verify the reliability of the prediction process the results are compared to those obtained using the algorithm of Ref. 26. The results in Table 2 are the averages and the standard deviations of the root mean square deviation of predictions compared to actual values for all runs. The repeatability of the results is measured by the standard deviation of the root mean square error.

Table 2: Prediction results.

Algorithm	**mean RMSE**	**std RMSE**
Algorithm of Ref. 26	0.0201	0.0034
This work	0.0210	0.0035

From these results it can be observed that the proposed algorithm performs similarly to the one proposed in Ref. 26 over the testing set. The standard deviation of the algorithm proposed here is higher, speaking to the apparent stability of the compared study compared to that of the DD-GAN. This compared instability is explained by the pattern matching within the latent space, which does not occur in Ref. 26.

5 Conclusion and further work

Here a novel procedure for time series prediction using domain decomposition and generative adversarial networks was introduced. By formulating a flow past a cylinder problem to predict POD coefficients, the implementation presented here remains the first successful DD-GAN used to predict fluid flow.

The foundation of this method is the communication/prediction cycle that allows a GAN to predict with variable fidelity in time. By expanding on this method, GANs could conceivably be utilized to predict in space by creating domains between boundary conditions of the simulation being performed, essentially extending the simulation from the middle out. In a propagating flow such as the FPC displayed here, the initial conditions might be the starting conditions at the initial time. Assuming a properly trained network, one could see a wave propagating through the domains that could conceivably stretch indefinitely. Due to time constraints, these use cases are left as future work.

It is clearly apparent that the use cases for an evolved DD-GAN are plentiful. A few example include extreme high ratio flows such as slug flow in long pipes where the non-linearity of the simulations is even greater, in porous flows or in other fields where computational cost of simulation is high.

The introduced approach introduced was shown to capably predict in time indefinitely, using only two time steps as a basis, with minimal quality degradation. Numerical results demonstrate the method's stability, and the qualitative results demonstrate the quality of the GAN predictions and that predictions can be reverted back to realistic simulations.

Acknowledgements

The completion of this thesis would not have been possible without the patient guidance and plentiful ideas of my supervisors: Dr. Claire Heaney and Prof. Christopher Pain. I would like to give special thanks to Dagmar whose support and advice carried me though this project and degree, and my family and friends. To my co-students, in particular Zef Wolffs and Jakob Torben whose discussions led to good ideas and better times. Lastly to all others who so patiently provided writing advice, including Marijan Beg and James Percival.

References

[1] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. Machine Learning for Fluid Mechanics. *Annual Review of Fluid Mechanics*, 52(1):477–508, 2020.

[2] B.E. Launder and D.B. Spalding. The numerical computation of turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 3(2):269–289, 1974.

[3] Mahesh T. Dhotre, Nandkishor Krishnarao Nere, Sreepriya Vedantam, and Mandar Tabib. Advances in Computational Fluid Dynamics. *International Journal of Chemical Engineering*, 2013:917373, Mar 2013.

[4] Bo Wang and Jingtao Wang. Application of Artificial Intelligence in Computational Fluid Dynamics. *Industrial & Engineering Chemistry Research*, 60(7):2772–2790, Feb 2021.

[5] C.M. Bishop and G.D. James. Analysis of multiphase flows using dual-energy gamma densitometry and neural networks. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 327(2):580–593, 1993.

[6] Michele Milano and Petros Koumoutsakos. Neural Network Modeling for Near Wall Turbulent Flow. *Journal of Computational Physics*, 182(1):1–26, 2002.

[7] Nils Thuerey, Konstantin Weißenow, Lukas Prantl, and Xiangyu Hu. Deep learning methods for Reynolds-averaged Navier–Stokes simulations of airfoil flows. *AIAA Journal*, 58(1):25–36, 2020.

[8] Byungsoo Kim, Vinicius C. Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep Fluids: A Generative Network for Parameterized Fluid Simulations. *Computer Graphics Forum*, 38(2):59–70, 2019.

[9] Andrea Beck, David Flad, and Claus-Dieter Munz. Deep neural networks for data-driven LES closure models. *Journal of Computational Physics*, 398:108910, 2019.

[10] Guido Novati, L. Mahadevan, and Petros Koumoutsakos. Controlled gliding and perching through deep-reinforcement-learning. *Phys. Rev. Fluids*, 4:093902, Sep 2019.

[11] Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[12] Alex Graves. Generating Sequences With Recurrent Neural Networks. *CoRR*, abs/1308.0850, 2013.

[13] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, 2014.

[14] You Xie, Erik Franz, Mengyu Chu, and Nils Thuerey. tempoGAN: A Temporally Coherent, Volumetric GAN for Super-resolution Fluid Flow. *CoRR*, abs/1801.09710, 2018.

[15] Masaki Saito, Eiichi Matsumoto, and Shunta Saito. Temporal Generative Adversarial Nets with Singular Value Clipping, 2017.

[16] César Quilodrán-Casas, Vinicius Santos Silva, Rossella Arcucci, Claire E. Heaney, Yike Guo, and Christopher C. Pain. Digital twins based on bidirectional LSTM and GAN for modelling the COVID-19 pandemic, 2021.

[17] Vinicius L. S. Silva, Claire E. Heaney, Yaqi Li, and Christopher C. Pain. Data Assimilation Predictive GAN (DA-PredGAN): applied to determine the spread of COVID-19. *CoRR*, abs/2105.07729, 2021.

[18] M. Cheng, F. Fang, C.C. Pain, and I.M. Navon. Data-driven modelling of nonlinear spatio-temporal fluid flows using a deep convolutional generative adversarial network. *Computer Methods in Applied Mechanics and Engineering*, 365:113000, 2020.

[19] Kangle Deng, Tianyi Fei, Xin Huang, and Yuxin Peng. IRC-GAN: Introspective Recurrent Convolutional GAN for Text-to-video Generation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 2216–2222. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[20] Ian Jolliffe. *Principal component analysis*. Springer Verlag, New York, 2002.

[21] K. Tang, W. Graham, and J. Peraire. *Active flow control using a reduced order model and optimum control*. 1996.

[22] Rubens Sampaio and Christian Soize. Remarks on the efficiency of POD for model reduction in non-linear dynamics of continuous elastic systems. *International Journal for Numerical Methods in Engineering*, 72(1):22–45, 2007.

[23] Tomasson, Jon Atli and Koziel, Slawomir and Pietrenko-Dabrowska, Anna. Quasi-global optimization of antenna structures using principal components and affine subspace-spanned surrogates. *IEEE Access*, 8:50078–50084, 2020.

[24] G Berkooz, P Holmes, and J L Lumley. The Proper Orthogonal Decomposition in the Analysis of Turbulent Flows. *Annual Review of Fluid Mechanics*, 25(1):539–575, 1993.

[25] D. Xiao, C.E. Heaney, F. Fang, L. Mottet, R. Hu, D.A. Bistrian, E. Aristodemou, I.M. Navon, and C.C. Pain. A domain decomposition non-intrusive reduced order model for turbulent flows. *Computers & Fluids*, 182:15–27, 2019.

[26] Zef Wolffs. Domain Decomposition and Autoencoders for Modelling Fluid Flow. *MSc thesis, Applied Computational Science and Engineering, Imperial College London*, August 2021.

[27] D. Xiao, P. Yang, F. Fang, J. Xiang, C.C. Pain, I.M. Navon, and M. Chen. A non-intrusive reduced-order model for compressible fluid and fractured solid coupling and its application to blasting. *Journal of Computational Physics*, 330:221–244, 2017.

[28] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved Training of Wasserstein GANs, 2017.

[29] Martin Arjovsky and Léon Bottou. Towards Principled Methods for Training Generative Adversarial Networks, 2017.

[30] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN, 2017.

[31] TensorFlow: Large-scale machine learning on heterogeneous systems.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[34] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[35] John D. Garrett and Hsin-Hsiang Peng. garrettj403/SciencePlots. February 2021.

[36] Utkarsh Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitwatre, 2015.

[37] Claire E. Heaney, Yuling Li, Omar K. Matar, and Christopher C. Pain. Applying Convolutional Neural Networks to Data on Unstructured Meshes with Space-Filling Curves, 2021.