ABSTRACT

Several applications in the scientific simulation of physical systems can be formulated as control/optimization problems. The computational models for such systems generally contain hyperparameters, which control solution fidelity and computational expense. The tuning of these parameters is non-trivial and the general approach is to manually ‘spot-check’ for good combinations. This is because optimal hyperparameter configuration search becomes impractical when the parameter space is large and when they may vary dynamically. To address this issue, we present a framework based on deep reinforcement learning (RL) to train a deep neural network agent that controls a model solve by varying parameters dynamically. First, we validate our RL framework for the problem of controlling chaos in chaotic systems by dynamically changing the parameters of the system. Subsequently, we illustrate the capabilities of our framework for accelerating the convergence of a steady-state CFD solver by automatically adjusting the relaxation factors of discretized Navier-Stokes equations during run-time. The results indicate that the run-time control of the relaxation factors by the learned policy leads to a significant reduction in the number of iterations for convergence compared to the random selection of the relaxation factors. Our results point to potential benefits for learning adaptive hyperparameter learning strategies across different geometries and boundary conditions with implications for reduced computational campaign expenses.

Keywords  Reinforcement learning, Neural network, Systems control, Fluid dynamics

1 Introduction

In recent years, there has been a proliferation in the use of machine learning for fluid mechanics problems that use the vast amount of data generated from experiments, field measurements, and large-scale high fidelity simulations [1–3]. A key ML algorithm is given by the artificial neural network and it has been widely used to tackle challenges in fluid mechanics. The artificial neural network is popular due to its ability to approximate complex nonlinear functions from large amounts of data, although this comes at an expense of reduced interpretability. Different types of neural networks have been adopted for a variety of tasks such as turbulence closure modeling [4–8], reduced order modeling [9–13], super-resolution and forecasting [14–16], solving partial differential equations [17–18], flow control [19,20], etc. Apart from fluid mechanics, there is an ever-increasing trend in the use of machine learning in material science [21,22], computational chemistry [23–25], and medical imaging [26,27]. A common thread that links a large majority of these investigations is that machine learning has been deployed in a supervised learning fashion. Supervised learning relies on labeled data to learn relationships from observables to targets and is often limited by the lack of pre-existing data.

In contrast, reinforcement learning (RL) is another type of machine learning philosophy, where an agent interacts with an environment and learns a policy that will maximize a predefined long-term reward [28]. Multiple methods can be employed as an agent in RL, but the most successful is a neural network in conjunction with RL known as the deep RL. Deep RL is gaining widespread popularity due to its success in robotics, playing Atari games, the game of GO, and process control [29–33]. Recently, the field of quantum reinforcement learning is emerging which aims at

---

*This work was performed during a Research Aide appointment at the Argonne Leadership Computing Facility, Argonne National Laboratory

2Data and codes available at https://github.com/Romit-Maulik/RLLib_Theta/
developing quantum agent that can interact with the outer world and adapt to it with the strategy of achieving certain final foal [34,36]. Deep RL is also making inroads in fluid mechanics, as a means to solve fundamental problems in flow control. RL has been utilized to learn collective hydrodynamics of self-propelled swimmers to optimally achieve a specified goal [37,38], to train a glider to autonomously navigate in turbulent atmosphere [39,40], for controlled gliding and perching of ellipsoid shaped bodies [41], and navigation of smart microswimmers in complex flows [42]. RL has also been used for closed-loop control of drag of a cylinder flow [43], control of two-dimensional convection-diffusion partial differential equation [44], designing control strategies for active flow control [45,46], discovering computational models from the data [47], and shape optimization [48]. RL’s attractiveness stems from the ability to perform both exploitation and exploration to learn about a particular environment.

Many real-world system optimization problems can be formulated as RL problems and offer a potential for the application of the recent advances in deep RL [49]. Several problems in scientific simulations share a similarity with system optimization problems. For example, the iterative solver in numerical simulations of fluid flows may be considered as a plant to be controlled. The iterative solver consists of different parameters such as relaxation factors, smoothing operations, type of solver, residual tolerance, etc., and all these parameters could be controlled during run-time with RL to achieve faster convergence. Another example is the turbulence closure model employed in computational fluid dynamics (CFD) simulations to represent unresolved physics. The closure coefficients of the turbulence model can be calibrated similarly to the optimization of hyperparameters in neural networks with RL [50]. Indeed, recently, the multi-agent RL was introduced as an automated turbulence model discovery tool in the context of sub-grid scale closure modeling in large eddy simulation of homogeneous and isotropic turbulence [51].

To illustrate the application of RL for the control of the parameters in computational models employed in scientific simulation, we consider the problem of accelerating the convergence of steady-state iterative solvers in CFD simulations by dynamically controlling under-relaxation factors of the discretized Navier-Stokes equations. More concretely, we train the RL agent to control under-relaxation factors to optimize the computational time or the number of iterations it takes for the solution to converge. This control problem differs from standard control problems, as the target state (i.e., the converged solution) is not known a priori. There have been some studies done for dynamically updating under-relaxation factors with the goal to maximize the speed of convergence. One of the methods is to use a fuzzy control algorithm based on the oscillations of the solution norm observed over a large interval prior to the current iteration [52]. Other methods based on neural network and fuzzy logic have also been proposed to automate the convergence of CFD simulations [53,54]. Some of the limitations of the fuzzy logic methods are that they are based on certain assumptions about the system that might not always be accurate, and they do not scale well to larger or complex systems. On the other hand, the neural network agent trained using RL can discover a more accurate decision-making policy to achieve a certain objective and can be applied for complex systems. The problem of controlling underrelaxation factors in CFD simulations is analogous to controlling the learning rate of stochastic gradient descent (SGD) algorithm and RL has been demonstrated to achieve better convergence of SGD than human-designed learning rate [55]. In the present work, we introduce a novel method based on RL to accelerate the convergence of CFD simulations applied to turbulent flows and demonstrate the robustness of the method for the backward-facing step test case.

One of the major challenges in using RL for scientific simulations is the computational cost of simulating an environment [38,50]. For example, the cost of CFD solver depends on several factors such as mesh refinement, complexity of the problem, level of required convergence, and can range between the order of minutes to order of several hours. RL is known to be less sample efficient and most popular RL strategies require a large number of interactions with the environment. Therefore, the RL framework should take the computational overhead of simulating an environment into account for scientific computing problems. One of the approaches to accelerate RL training is to run multiple environments concurrently on different processors and then collect their experience to train the agent. We refer to this multi-environment approach as distributed RL in this study. There are many open-source packages such as RLLib [57], Tensorforce [58], SEED RL [59]. Acme [60] that can be exploited to achieve faster training with distributed RL. In addition to the distributed execution of the environment, the CFD simulations can also be parallelized to run on multiple processing elements [61,62]. The list of contributions of this work can be summarized as follows:

- A distributed deep RL framework is presented for the optimization and control of systems parameters encountered in scientific simulations.
- We apply the deep RL framework to the task of restoring chaos in transiently chaotic systems, and for accelerating the convergence of a steady-state CFD solver.
- We highlight the importance of distributed RL and how it aids in accelerated training for scientific applications.
- We also provide an open-source code and set of tutorials for deploying the introduced framework on distributed multiprocessing systems.
This paper is organized as follows. Section 2 provides background information on RL problem formulation and present proximal policy optimization (PPO) algorithm. In Section 3, we validate our framework using the test case of restoring chaos in chaotic systems. Then, we describe how controlling relaxation factors in CFD simulations converted to sequential decision-making problem and discuss the performance of an RL agent to generalize for different boundary conditions and different geometry setup. Finally, we conclude with the summary of our study and outline the future research directions in Section 4.

2 Reinforcement Learning

In this section, we discuss the formulation of the RL problem and briefly describe the proximal policy optimization (PPO) algorithm [63] that belongs to a class of policy-gradient methods. In RL, the agent observes the state of the environment and then based on these observations takes an action. The objective of the agent is the maximization of the expected value of the cumulative sum of the reward. This problem statement can be framed as a Markov decision process (MDP). At each time step $t$, the agent observes some representation of the state of the system, $s_t \in \mathcal{S}$, and based on this observation selects an action, $a_t \in \mathcal{A}$. As a consequence of the action, the agent receives the reward, $r_t \in \mathcal{R}$, and the environment enters in a new state $s_{t+1}$. Therefore, the interaction of an agent with the environment gives rise to a trajectory as shown below

$$\tau = \{s_0, a_0, r_0, s_1, a_1, r_1, \ldots \}.$$  \hfill (1)

The advantage of the MDP framework is that it is flexible and can be applied to different problems in different ways. For example, the time steps in the MDP formulation need not refer to the fixed time step as used in CFD simulations; they can refer to arbitrary successive stages of taking the action. The goal of the RL agent is to choose an action that will maximize the expected discounted return over the trajectory $\tau$ and mathematically, it can be written as

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t,$$  \hfill (2)

where $\gamma$ is a parameter called a discount rate and it lies between $[0, 1]$, and $T$ is the horizon of the trajectory. The discount rate determines how much importance to be given to the long term reward compared to immediate reward. If $\gamma \to 0$, then the agent is concerned with maximizing immediate rewards and as $\gamma \to 1$, the agent takes future reward into account more strongly. The RL tasks are also classified based on whether they terminate or not. Some tasks involve agent-environment interactions that break into a sequence of episodes, where each episode ends in a state called the terminal state. On the other hand, there are tasks, such as process control, that that goes on continually without limit. The horizon $T$ of the trajectory for these continuing tasks goes to infinity in Equation (2).

In RL, the agent’s decision making procedure is characterized by a policy $\pi(s, a) \in \Pi$. The RL agent is trained to find a policy so as to optimize the expected return when starting in the state $s$ at time step $t$ and is called as V-value function. We can write the the V-value function as

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, \pi \right],$$  \hfill (3)

Similarly, the expected return starting in a state $s$, taking an action $a$, and thereafter following a policy $\pi$ is called as the Q-value function and can be written as

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a, \pi \right].$$  \hfill (4)

We also define an advantage function that can be considered as another version of Q-value function with lower variance by taking the V-value function as the baseline. The advantage function can be written as

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$  \hfill (5)

In deep RL, the neural network is used as an RL agent, and therefore, the weights and biases of the neural network are the parameters of the policy [64]. We use $\pi_w(\cdot)$ to denote that the policy is parameterized by $w \in \mathbb{R}^d$. The agent is trained with an objective function defined as [28]

$$J(w) = V^{\pi_w}(s_0),$$  \hfill (6)

where an episode starts in some particular state $s_0$, and $V^{\pi_w}$ is the value function for the policy $\pi_w$. The policy parameters $w$ are learned by estimating the gradient of an objective function and plugging it into a gradient ascent algorithm as follow

$$w \leftarrow w + \alpha \nabla_w J(w),$$  \hfill (7)
where $\alpha$ is the learning rate of the optimization algorithm. The gradient of an objective function can be computed using the policy gradient theorem [64] as follow

$$\nabla_w V^\pi_w(s_0) = \mathbb{E}_{\pi_w} \left[ \nabla_w \left( \log \pi_w(s, a) \right) Q^\pi_w(s, a) \right].$$

(8)

There are two main challenges in using the above empirical expectation. The first one is the large number of samples required and the second is the difficulty of obtaining stable and steady improvement. There are different families of policy-gradient algorithms that are proposed to tackle these challenges [65–67]. The performance of policy gradient methods is highly sensitive to the learning rate $\alpha$ in Equation (7). If the learning rate is large it can cause the training to be unstable. The proximal policy optimization (PPO) algorithm uses a clipped surrogate objective function to avoid excessive update in policy parameters in a simplified way [63]. The clipped objective function of the PPO algorithm is

$$J_{\text{clip}}(w) = \mathbb{E} \left[ \min(r_t(w) A^\pi_w(s, a), \text{clip}(r_t(w), 1 - \epsilon, 1 + \epsilon) A^\pi_w(s, a)) \right],$$

(9)

where $r_t(w)$ denotes the probability ratio between new and old policies as follow

$$r_t(w) = \frac{\pi_{w+\Delta w}(s, a)}{\pi_w(s, a)}.$$

(10)

The $\epsilon$ in Equation (9) is a hyperparameter that controls how much new policy is allowed to be deviated from the old. This is done using the function $\text{clip}(r_t(w), 1 - \epsilon, 1 + \epsilon)$ that enforces the ratio between new and old policy ($r_t(w)$) to stay between the limit $[1 - \epsilon, 1 + \epsilon]$.

3 Numerical Experiments

In this section, we demonstrate the application of deep RL for two problems where the parameters of the systems need to be controlled to achieve a certain objective. The first test case is restoring chaos in chaotic dynamical systems [68]. We utilize this test case as a validation example for our distributed deep RL framework. The second test problem is accelerating the convergence of steady-state CFD solver through deep RL and we illustrate this for turbulent flow over a backward-facing step example.

3.1 Chaotic process control

The chaotic state is essential in many dynamical systems and the phenomenon called crisis can cause sudden destruction of chaotic attractors [69]. This can be negative in many applications where the chaos is essential such as to enhance mixing by chaotic advection [70] or to prevent the collapse of electric power systems [71]. Some of the methods to sustain chaos is to use small perturbations based on knowledge of dynamical systems and a priori information about the escape regions from chaos [72]. Vashishtha et. al. [68] proposed an approach using deep RL to determine small perturbations to control parameters of the Lorenz system [73] in such a way that the chaotic dynamics is sustained despite the uncontrolled dynamics being transiently chaotic. This method does not require finding escape regions, target points, and therefore attractive for systems where the complete information about the dynamical system is unknown.

The Lorenz system is described by following equations

$$\frac{dx}{dt} = \sigma(y - x),$$

(11)

$$\frac{dy}{dt} = x(\rho - z) - y,$$

(12)

$$\frac{dz}{dt} = xy - \beta z,$$

(13)

$$\frac{dx}{dt} = \sigma(y - x),$$

(14)

where $x, y, z$ are the state of the Lorenz system, and $\sigma, \rho, \beta$ are the system’s parameter. The Lorenz system give rise to chaotic trajectory with $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$ as shown in Figure 1. However, the Lorenz system exhibits transient chaotic behavior for $\rho \in [13.93, 24.06]$ [74]. For example, if we use $\rho = 20$ for the Lorenz system, the solution converges to a specific fixed point $P^* = (-7.12, -7.12, 19)$ as illustrated in Figure 1.

In order to sustain the chaos in transiently chaotic Lorenz system, we utilize the similar RL problem formulation as Vashishtha et. al. [68]. The RL agent observes the solution vector and its time-derivative as the state of the system. Therefore, we have

$$s_t = x(t), y(t), z(t), \dot{x}(t), \dot{y}(t), \dot{z}(t),$$

(15)
where the dot represent the time-derivative, i.e., velocity. Based on this observations, the agent choose an action which is the perturbation to the control parameters of the Lorenz system as given below

$$a_t = \{\Delta \sigma, \Delta \rho, \Delta \beta\}. \quad (16)$$

The perturbation to the control parameters is restricted to lie within a certain interval which is $\pm 10\%$ of its values. Therefore, we have $\Delta \sigma \in [-\sigma/10, \sigma/10]$, $\Delta \rho \in [-\rho/10, \rho/10]$, and $\Delta \beta \in [-\beta/10, \beta/10]$. We note here that these limits remain fixed throughout the episode based on the initial values of $(\sigma, \rho, \beta)$. For a transiently chaotic system, the trajectory converges to a fixed point with time and therefore the magnitude of velocity will decrease consistently with time. This fact is utilized to design the reward function. When the magnitude of velocity is greater than certain threshold velocity $V_{Th}$, the agent is rewarded and the agent is penalized when the magnitude of velocity is less than $V_{Th}$. In addition to assigning reward for each time step, a terminal reward is given at the end of each episode. The reward function can be written as

$$r_t = \begin{cases} 10, & V(t) > V_{Th} \\ -10, & V(t) \leq V_{Th} \end{cases}, \quad (17)$$

$$r_{\text{terminal}} = \begin{cases} -100, & \tau_t \leq -2 \\ 0, & \tau_t > -2 \end{cases}, \quad (18)$$

where $\tau_t$ is the average of stepwise reward over the last 2000 time steps of an episode. The value of the threshold velocity $V_{Th}$ is set at 40.

For training the agent, we divide each episode into 4000 time steps with time step size $dt = 2 \times 10^{-2}$. We train the agent for $2 \times 10^5$ time steps which corresponds to 50 episodes. The RL agent is a fully connected neural network with two hidden layers and 64 neurons in each layer. Additionally, the output of the second hidden layer is further processed by the LSTM layer composed of 256 cells. The hyperbolic tangent (tanh) activation function is used for all hidden layers. Figure 2 shows how the mean reward is progressing with training for a different number of workers. We note here that by a different number of workers, we mean that the agent-environments interactions are run concurrently on different processors. In distributed deep RL, the environment is simulated on different processors and their experience is collected. These sample batches encode one or more fragments of a trajectory and are concatenated into a larger batch that is then utilized to train the neural network. More concretely, in this particular example, we collect experience of an agent for 8000 time steps to train the neural network. When we employ two workers, the agent-environment interaction is simulated for 4000 time steps on each worker and then this experience is concatenated to form a training batch for the neural network. Similarly, for four workers, the agent-environment interaction is simulated for 2000 time steps on each worker to collect a training batch size of 8000 time steps. Figure 2 also reports the computational time required for training. We see that as we increase the number of workers from two to four, the computational speed is approximately reduced by 25%. However, as we increase the number of workers to 8, there is only a marginal improvement in terms
of CPU time. This is not surprising, as the communication frequency is increased with eight workers and hence there is no significant reduction in CPU time.

Figure 2: Evolution of the moving average mean reward for different number of workers trained with the PPO algorithm. The moving average window size is set at 5 episodes.

Figure 3 depicts the performance of the trained RL agent in sustaining chaos in a transiently chaotic Lorenz system. In Figure 3, the black trajectory is for uncontrolled solution of the Lorenz system for parameters \((\sigma, \rho, \beta) = (10, 20, 8/3)\). When the policy learned by an RL agent is used for the control, the Lorenz system does not converge to a fixed point of \(P^-\). We can also see that the RL agent trained with a different number of workers can learn the policy effectively and is able to sustain chaos over a period of temporal integration. This observation is very important for employing RL in physical systems where the simulation of a computationally intensive environment is the bottleneck and distributed deep RL can be employed to address this issue.

3.2 Solver convergence acceleration

In CFD, the Navier-Stokes equations are discretized which leads to a system of linear equations. This system of linear equations is usually solved using iterative techniques like Gauss-Seidel methods, Krylov subspace methods, multigrid methods, etc. [75]. The main logic behind iterative methods is to start with some approximate solution and then reduce the residual vector at each iteration, called relaxation steps. The relaxation steps are performed until the convergence criteria are reached. In the relaxation step, the value of the variable at the next time step is obtained by adding some fraction of the difference between the current value and predicted value to the value of the variable at the current time step. The SIMPLE algorithm [76] is widely used in CFD and it utilizes underrelaxation method to update the solution from one iteration to another. The underrelaxation methods improve the convergence by slowing down the update to the solution field. The value of the variable at the \(k^{th}\) iteration is obtained as the linear interpolation between the value from the previous iteration and the value obtained in the current iteration as follow

\[
x_P^{(k)} = x_P^{(k-1)} + \alpha \left( \sum_{a} a_{ab} x_{ab} + \frac{b}{a_P} - x_P^{(k-1)} \right),
\]

where \(0 < \alpha < 1\) is the relaxation factor, \(x_P^{(k)}\) is the value of the variable at node \(P\) to be used for the next iteration, \(x_P^{(k-1)}\) is the value of the variable at node \(P\) from the previous iteration, \(x_{ab}\) is the values of the variables at the neighboring nodes, and \(a_P\), \(a_{ab}\), \(b\) are the constants obtained by discretizing Navier-Stokes equations.

The underrelaxation factor \(\alpha\) in Equation 19 should be chosen in such a way that it is small enough to ensure stable computation and large enough to move the iterative process forward quickly. In practice, a constant value of the relaxation factor is employed throughout the computation and this value is usually decided based on some previous computations. Also, the suitable value of the relaxation factor is problem-dependent and a small change in it can result in a large difference in the number of iterations needed to obtain the converged solution. In Figure 4, the number of iterations required for convergence with different values of relaxation factors for momentum and pressure equation is shown for the problem investigated in this study (i.e., the backward-facing step example). It can be noticed that
Figure 3: Comparison of the evolution of the Lorenz system without applying the control and with an application of the control policy learned by an RL agent. The parameters of the Lorenz system are \((\sigma, \beta, \rho) = (10, 8/3, 20)\) and the initial condition is \((x, y, z) = (1, 12, 9)\).

The number of iterations is less for \(\alpha_u = 0.9\) (momentum equation relaxation factor), for all values of \(\alpha_p\) (pressure equation relaxation factor). However, for \(\alpha_u = 0.7\), the solution converges only for \(\alpha_p = 0.9\), and for other values of \(\alpha_p\), the solution does not converge within 3000 iterations. Even though for most of the CFD simulations, a good value of relaxation factors can be found through trial and error, the process can become intractable for large parameter space. The complexity increases even further when the relaxation factor needs to be updated on the fly during run-time for multiphysics problems. Therefore, we attempt to automate the procedure of dynamic update of the relaxation factors using deep RL to accelerate the convergence of CFD simulations.

We illustrate our RL framework to accelerate the convergence of CFD simulations by considering a two-dimensional backward-facing step example. First, we validate our CFD model using the experimental data provided in [77]. The height of the step in the experimental setup was \(H = 0.0127\) m and the free-stream reference velocity was \(U_{\text{ref}} = 44.2\) m/s. The corresponding Reynolds number based on step height and reference free-stream velocity is approximately \(Re_H = 36,000\). The turbulent intensity at the inlet is \(I_t = 6.1 \times 10^{-4}\) and the eddy-viscosity-to-molecular-viscosity ratio at the inlet is \(\mu_t/\mu = 0.009\). We use the computational mesh distributed by NASA [78] for our study. We utilize the steady-state simpleFoam solver in OpenFoam [79] and apply the \(k-\omega\) SST turbulent model [80] with the standard values of closure coefficients to simulate the turbulent flow. Figure 5 displays the contour plot of the magnitude of the velocity along with the normalized stream-wise velocity at four different axial locations obtained experimentally and numerically. We can observe that there is a very good agreement between the experimental results and numerical results for normalized stream-wise velocity.

To learn the policy that is generalizable for different inlet boundary condition, we train the RL agent for multiple values of inlet velocities. For each episode, the inlet velocity is selected randomly from the interval between [35,65] m/s.
Figure 4: Number of iterations required for the convergence with different combinations of relaxation factor for momentum and pressure equation (left). The convergence of the state of an environment with iterations for two values of momentum equation relaxation factor (right). The pressure relaxation factor is set at $\alpha_p = 0.5$ for both cases.

Figure 5: Validation of the CFD model with experimental measurements for normalized velocity in the $x$-direction. The experimental data for velocity measurements are published in [77].

We highlight here that, in our study, the frequency at which the relaxation factors are updated is different from the frequency of CFD iterations. In particular, we update the relaxation factor for the momentum and pressure equation after every 100 iterations of the CFD simulation. We use the average value of the square of the velocity magnitude as the characteristic quantity that represents the CFD solution. Thus, the state of the environment can be written as

$$ s_t = \frac{1}{N_b} \sum_{i=1}^{N_b} [U_b(i)]^2 + \frac{1}{N_m} \sum_{i=1}^{N_m} [U_m(i)]^2, $$

(20)

where $U_b$ is the velocity at the inlet boundary, $U_m$ is the nodal value of the velocity at internal mesh, $N_b$ is the total number of nodes at inlet boundary, and $N_m$ is the total number of nodes in the internal mesh. At the start of an episode, the velocity in the interior of the domain is initialized with zero velocity, and hence the initial state for each episode will depend only on the value of the velocity at the inlet boundary. Figure 4 displays how the state of the environment progress towards the converged state for two different values of relaxation factor for momentum equations. As the solution starts converging, the change in the state of the system from one iteration to other will also be reduced. Based on the observation of the state of the system, the agent decides the relaxation factor for the momentum and pressure equations. The action of the agent can be written as

$$ a_t = \{\alpha_u, \alpha_p\}, $$

(21)
where $\alpha_u$ and $\alpha_p$ are the relaxation factors for momentum and pressure equation, respectively. The action space for both relaxation factors lies within the interval $[0.2, 0.95]$. Even though the action space is composed of only two relaxation factors, this problem is complicated due to its dynamic nature and simple exploratory computation will not be feasible. For more complex multiphysics problems, the action space can be easily augmented in this framework to include relaxation factors for other governing equations. The goal of the RL agent is to minimize the number of iterations required for obtaining the converged solution. We give the total number of iterations of the CFD simulation between two time steps as the reward for that time step. Accordingly, the reward at each time step can be written as

$$r_t = -(K_t - K_{t-1}),$$

(22)

where $K$ is the iteration count of CFD simulation. Since each episode comes to end with the terminal state, the task of accelerating CFD simulations can be considered as an episodic task. Therefore, we set the discount factor $\gamma = 1.0$ for this test case. The stepwise reward function computed using Equation 22 and with $\gamma = 1.0$ will give cumulative reward equal to the total number of iterations it took for CFD simulation to converge. Specifically, if the CFD simulation required 450 iterations to converge and the relaxation factor is updated every 100 iterations, then the trajectory for the reward will look like $\{-100, -100, -100, -100, -50\}$.

For this example, the RL agent is a fully connected neural network with 64 neurons in each hidden layer. The RL agent is trained for 2000 episodes and the learning rate is set to $2.5 \times 10^{-4}$. We assume that the convergence for CFD simulation is reached once the residual for momentum equation falls below $1 \times 10^{-3}$ and the residual for pressure equation is reduced below $5 \times 10^{-2}$. Figure 6 shows how the moving average mean reward progress with training for a different number of workers. The moving average window size is set at 100 episodes, i.e., the average of the reward is taken over the last 100 episodes. At the beginning of the training, the agent is dominantly exploring and the mean reward at the start of training is in the range of 650-700 iterations for CFD simulation to converge. The mean reward improves over the training and the final mean reward is around 450 iterations for the convergence of CFD simulations. This corresponds to approximately 30% improvement in the number of iterations required for convergence. We highlight here that, for many CFD problems, suitable values of relaxation factors can be chosen to achieve faster convergence based on prior experience. However, for complex systems, the process of learning a rule to dynamically update relaxation factors through exploratory computations can become unmanageable. For such systems, the proposed RL framework can be attractive to achieve acceleration in the convergence. In Figure 6, the CPU time for training is reported for a different number of workers. As we increase the number of workers/environments from 4 to 8, we get around 40% improvement in the CPU time for the training. The improvement in CPU speed is only marginal as the number of processor is increased to 16, which might be due to increased communication between the processors. We note here that our environment, i.e. the CFD simulation utilizes a single core as the backward-facing step is a relatively small problem and can be simulated within an order of seconds or minutes depending upon the computer architecture. For high-dimensional problems, the parallelization through domain-decomposition can also be exploited to reduce the CPU time of each CFD simulation run.

Figure 6: Evolution of the moving average mean reward for the PPO algorithm trained using different number of workers (left). The moving average window size is set at 100 episodes. The CPU wall time for agent’s training for different number of workers (right)

Once the agent is trained, we test the agent’s performance for three different values of inlet velocity, $V = 25.0, 50.0, 75.0$. Among these three velocities, the velocity $V = 50.0$ lies within the training velocity range,
i.e., $[35.0, 65.0]$. We run ten different samples of CFD simulation for these three inlet velocities, and utilize the RL agent to select the relaxation factor after every 100 iterations. Figure 7 shows the box plot to depict the data obtained from ten samples through their quartiles. It should be noted that the policy learned by an agent is not deterministic, and hence the number iterations required for convergence for a single inlet velocity is not constant for all samples. The mean and standard deviation of these samples are also shown in Figure 7. From both these plots, we can see that the mean of the samples for all three different velocities is around 450 iterations. Therefore, we can conclude that the policy learned by the RL is generalizable to inlet boundary condition different than the training boundary condition. We plot the variation of relaxation factor for momentum equation and pressure equation in Figure 8 to understand the policy learned by an RL agent. The RL agent has learned to keep the relaxation factor for momentum equation at its highest value (i.e., 0.95) throughout the CFD simulation. We do not see any specific pattern for the variation of pressure equation relaxation factor. Since, the relaxation factor for momentum equations is constant, the difference in number of iterations to convergence for different samples is mainly due to how the pressure relaxation factor is decided by an RL agent at various stages of the CFD simulation.

Furthermore, we test the policy for different backward-facing step geometry. The test geometry has the step height twice that of the train geometry. In Figure 9, the flow feature for train and test geometry is shown, and it can be seen that there is a sufficient difference in terms of the flow in the recirculation region. We run a similar experiment with the test geometry for three different values of inlet velocity. Figure 10 displays the summary of the RL agent’s performance in accelerating CFD simulation of the test geometry. Overall, the number of iterations required for the convergence for the test geometry is higher than the train geometry. The mean of the samples for test geometry is around 550 iterations for all three velocities. Interestingly, the variance of test geometry samples employed with the policy trained using 8 workers is less compared to the policy trained using 4 and 16 workers.

Figure 7: Evaluation of the trained agent’s PPO policy for different inlet velocities - $V = 25.0$ m/s (left), $V = 50.0$ m/s (middle), $V = 75.0$ m/s (right). The top plot is the box plot in which the green marker is for the mean, red marker is for outliers, and the orange line is for the median of the data. The bottom plot shows the mean of the sample with error bars denoting the standard deviation of samples.

4 Concluding Remarks

In this study, we have investigated the application of distributed reinforcement learning (RL) to automatically update the computational hyperparameters of numerical simulations, which is a common task in many scientific applications. We illustrate our framework for two problems. The first problem is tasked with sustaining chaos in chaotic systems, and the second problem is the acceleration of convergence for steady-state CFD solver. In the first example, we demonstrate...
that the policy learned by an RL agent trained using proximal policy optimization (PPO) algorithm can automatically adjust the system’s parameters to maintain chaos in transiently chaotic systems. The problem of controlling relaxation factors in steady-state CFD solvers is different from the standard control problem. In a standard control problem, the target value is known in advance and this target value is used to change action variables in such a way that the state of the system is guided towards the target state. However, the problem of accelerating convergence of steady-state CFD solvers is different in a way that the target point, i.e., the converged solution is not known. Therefore, RL is a suitable algorithm for this problem to discover the decision-making strategy that will dynamically update the relaxation factor with an objective to minimize the number of iterations required for convergence. Our results of numerical experiments indicate that the learned policy leads to acceleration in the convergence of steady-state CFD solver and the learned policy is generalizable to unseen boundary conditions and different geometry setup.

Despite the relative simplicity in terms of the RL problem formulation and its application to backward-facing step example, this framework can be readily extended for more complex multiphysics systems. The state and the action space of an RL agent can be augmented to tackle complexities in these systems. Our future work is centered around applying this framework for more challenging problems like a differentially heated cavity, and to improve the computational efficiency of the software pipeline through direct I/O communication between OpenFoam solver and RL libraries. One of the major challenges in using RL for scientific simulations is the computational time due to a slow environment, and we believe this challenge can be addressed by utilizing distributed environment evaluation along with high-performance computing to reduce the CPU time of the solver.

Acknowledgement

We acknowledge the helpful discussions with Venkat Vishwanath and the Argonne Leadership Computing Facility Datascience team. This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research was funded in part and used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science.
Figure 9: Flow features in the vicinity of backward facing step for train and test geometry setup. The agent is trained with a step height $h = 0.0127 \text{ m}$ and the test geometry has step height $h = 0.0254 \text{ m}$.

User Facility supported under Contract DE-AC02-06CH11357. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. DOE or the United States Government.

References

[1] Steven L Brunton, Bernd R Noack, and Petros Koumoutsakos. Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics*, 52, 2019.

[2] MP Brenner, JD Eldredge, and JB Freund. Perspective on machine learning for advancing fluid mechanics. *Physical Review Fluids*, 4(10):100501, 2019.

[3] Paul Garnier, Jonathan Viquerat, Jean Rabault, Aurélien Larcher, Alexander Kuhnle, and Elie Hachem. A review on deep reinforcement learning for fluid mechanics. *arXiv preprint arXiv:1908.04127*, 2019.

[4] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[5] Karthik Duraisamy, Gianluca Iaccarino, and Heng Xiao. Turbulence modeling in the age of data. *Annual Review of Fluid Mechanics*, 51:357–377, 2019.

[6] Jian-Xun Wang, Jin-Long Wu, and Heng Xiao. Physics-informed machine learning approach for reconstructing Reynolds stress modeling discrepancies based on dns data. *Physical Review Fluids*, 2(3):034603, 2017.

[7] Romit Maulik, Omer San, Adil Rasheed, and Prakash Vedula. Subgrid modelling for two-dimensional turbulence using neural networks. *Journal of Fluid Mechanics*, 858:122–144, 2019.
Figure 10: Evaluation of the trained agent’s PPO policy for different inlet velocities in test geometry setup - $V = 25.0$ m/s (left), $V = 50.0$ m/s (middle), $V = 75.0$ m/s (right). The top plot is the box plot in which the green marker is for the mean, red marker is for outliers, and the orange line is for the median of the data. The bottom plot shows the mean of the sample with error bars denoting the standard deviation of samples.

[8] PA Srinivasan, L Guastoni, Hossein Azizpour, PHILIPP Schlatter, and Ricardo Vinuesa. Predictions of turbulent shear flows using deep neural networks. Physical Review Fluids, 4(5), 2019.

[9] Michele Milano and Petros Koumoutsakos. Neural network modeling for near wall turbulent flow. Journal of Computational Physics, 182(1):1–26, 2002.

[10] Arvind Mohan, Don Daniel, Michael Chertkov, and Daniel Livescu. Compressed convolutional lstm: An efficient deep learning framework to model high fidelity 3d turbulence. arXiv preprint arXiv:1903.00033, 2019.

[11] Takaaki Murata, Kai Fukami, and Koji Fukagata. Nonlinear mode decomposition with convolutional neural networks for fluid dynamics. Journal of Fluid Mechanics, 882, 2020.

[12] Sk Mashfiqur Rahman, Suraj Pawar, Omer San, Adil Rasheed, and Traian Iliescu. Nonintrusive reduced order modeling framework for quasigeostrophic turbulence. Physical Review E, 100:053306, 2019.

[13] Romit Maulik, Bethany Lusch, and Prasanna Balaprakash. Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders. arXiv preprint arXiv:2002.00470, 2020.

[14] Kai Fukami, Koji Fukagata, and Kunihiko Taira. Super-resolution reconstruction of turbulent flows with machine learning. Journal of Fluid Mechanics, 870:106–120, 2019.

[15] Chiyu Max Jiang, Soheil Esmaeilzadeh, Kamyar Azizzadenesheli, Karthik Kashinath, Mustafa Mustafa, Hamdi A Tchelepi, Philip Marcus, Anima Anandkumar, et al. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. arXiv preprint arXiv:2005.01463, 2020.

[16] Junhyuk Kim and Changhoon Lee. Prediction of turbulent heat transfer using convolutional neural networks. Journal of Fluid Mechanics, 882:A18, 2020.

[17] Maziar Raisi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics, 378:686–707, 2019.

[18] Zichao Long, Yiping Lu, and Bin Dong. Pde-net 2.0: Learning pdes from data with a numeric-symbolic hybrid deep network. Journal of Computational Physics, 399:108925, 2019.
[19] Changhoon Lee, John Kim, David Babcock, and Rodney Goodman. Application of neural networks to turbulence control for drag reduction. *Physics of Fluids*, 9(6):1740–1747, 1997.

[20] N Benard, Jordi Pons-Prats, Jacques Periaux, Gabriel Bugeda, P Braud, JP Bonnet, and E Moreau. Turbulent separated shear flow control by surface plasma actuator: Experimental optimization by genetic algorithm approach. *Experiments in Fluids*, 57(2):22, 2016.

[21] Brian DeCost, Jason R Hattrick-Simpers, Zachary Trautt, Aaron Gilad Kusne, Eva Campo, and Martin L Green. Scientific AI in materials science: a path to a sustainable and scalable paradigm. *Machine Learning: Science and Technology*, 2020.

[22] Keith T Butler, Daniel W Davies, Hugh Cartwright, Oxandr Isayev, and Aron Walsh. Machine learning for molecular and materials science. *Nature*, 559(7715):547–555, 2018.

[23] Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole von Lilienfeld. Big data meets quantum chemistry approximations: The δ-machine learning approach. *Journal of chemical theory and computation*, 11(5):2087–2096, 2015.

[24] Sergei Manzhos. Machine learning for the solution of the schrödinger equation. *Machine Learning: Science and Technology*, 1(1):013002, 2020.

[25] Yasemin Bozkurt Varolgüneş, Tristan Bereau, and Joseph F Rudzinski. Interpretable embeddings from molecular simulations using gaussian mixture variational autoencoders. *Machine Learning: Science and Technology*, 1(1):015012, 2020.

[26] Bradley J Erickson, Panagiotis Korfiatis, Zeynnett Akkus, and Timothy L Kline. Machine learning for medical imaging. *Radiographics*, 37(2):505–515, 2017.

[27] Chang Min Hyun, Kang Cheol Kim, Hyun Cheol Cho, Jae Kyu Choi, and Jin Keun Seo. Framelet pooling aided deep learning network: the method to process high dimensional medical data. *Machine Learning: Science and Technology*, 1(1):015009, 2020.

[28] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[30] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Reinforcement learning for humanoid robotics. In *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pages 1–20, 2003.

[31] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[32] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[33] SPK Spielberg, RB Gopaluni, and PD Loewen. Deep reinforcement learning approaches for process control. In *2017 6th international symposium on advanced control of industrial processes (AdCONIP)*, pages 201–206. IEEE, 2017.

[34] Daoyi Dong, Chunlin Chen, and Zonghai Chen. Quantum reinforcement learning. In *International Conference on Natural Computation*, pages 686–689. Springer, 2005.

[35] Lucas Lamata. Quantum machine learning and quantum biomimetics: A perspective. *Machine Learning: Science and Technology*, 2020.

[36] F Albarrán-Arriagada, JC Retamal, E Solano, and L Lamata. Reinforcement learning for semi-autonomous approximate quantum eigensolver. *Machine Learning: Science and Technology*, 1(1):015002, 2020.

[37] Mattia Gazzola, Babak Hejazialhosseini, and Petros Koumoutsakos. Reinforcement learning and wavelet adapted vortex methods for simulations of self-propelled swimmers. *SIAM Journal on Scientific Computing*, 36(3):B622–B639, 2014.

[38] Siddhartha Verma, Guido Novati, and Petros Koumoutsakos. Efficient collective swimming by harnessing vortices through deep reinforcement learning. *Proceedings of the National Academy of Sciences*, 115(23):5849–5854, 2018.

[39] Gautam Reddy, Antonio Celani, Terrence J Sejnowski, and Massimo Vergassola. Learning to soar in turbulent environments. *Proceedings of the National Academy of Sciences*, 113(33):E4877–E4884, 2016.

[40] Gautam Reddy, Jerome Wong-Ng, Antonio Celani, Terrence J Sejnowski, and Massimo Vergassola. Glider soaring via reinforcement learning in the field. *Nature*, 562(7726):236–239, 2018.
[41] Guido Novati, Lakshminarayanan Mahadevan, and Petros Koumoutsakos. Controlled gliding and perching through deep-reinforcement-learning. Physical Review Fluids, 4(9):093902, 2019.

[42] Simona Colabrese, Kristian Gustavsson, Antonio Celani, and Luca Biferale. Flow navigation by smart microswimmers via reinforcement learning. Physical review letters, 118(15):158004, 2017.

[43] Florimond Guéniat, Lionel Mathelin, and M Youssuf Hussaini. A statistical learning strategy for closed-loop control of fluid flows. Theoretical and Computational Fluid Dynamics, 30(6):497–510, 2016.

[44] Amir-massoud Farahmand, Saleh Nabi, and Daniel N Nikovski. Deep reinforcement learning for partial differential equation control. In 2017 American Control Conference (ACC), pages 3120–3127. IEEE, 2017.

[45] Jean Rabault, Miroslav Kuchta, Atle Jensen, Ulysse Réglaire, and Nicolas Cerardi. Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control. Journal of Fluid Mechanics, 865:281–302, 2019.

[46] Feng Ren, Jean Rabault, and Hui Tang. Applying deep reinforcement learning to active flow control in turbulent conditions. arXiv preprint arXiv:2006.10683, 2020.

[47] Maxime Bassenne and Adrián Lozano-Durán. Computational model discovery with reinforcement learning. arXiv preprint arXiv:2001.00008, 2019.

[48] Hassan Ghrabieb, Jonathan Viquerat, Aurélien Larcher, P Meliga, and Elie Hachem. Optimization and passive flow control using single-step deep reinforcement learning. arXiv preprint arXiv:2006.02979, 2020.

[49] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. A view on deep reinforcement learning in system optimization. arXiv preprint arXiv:1908.01275, 2019.

[50] Jia Wu, SenPeng Chen, and XiYuan Liu. Efficient hyperparameter optimization through model-based reinforcement learning. Neurocomputing, 2020.

[51] Guido Novati, Hugues Lascombes de Laroussilhe, and Petros Koumoutsakos. Automating turbulence modeling by multi-agent reinforcement learning. arXiv preprint arXiv:2005.09023, 2020.

[52] Zoran Dragoljovic and Deborah A Kaminski. A fuzzy logic algorithm for acceleration of convergence in solving turbulent flow and heat transfer problems. Numerical Heat Transfer, Part B: Fundamentals, 46(4):301–327, 2004.

[53] Zoran Dragoljovic, Deborah A Kaminski, and Juntaek Ryoo. Tuning of a fuzzy rule set for controlling convergence of a cfd solver in turbulent flow. International Journal of Heat and Mass Transfer, 44(20):3811–3822, 2001.

[54] Juntaek Ryoo, Zoran Dragoljovic, and Deborah A Kaminski. Control of convergence in a computational fluid dynamics simulation using anfis. IEEE Transactions on fuzzy systems, 13(1):42–47, 2005.

[55] Chang Xu, Tao Qin, Gang Wang, and Tie-Yan Liu. Reinforcement learning for learning rate control. arXiv preprint arXiv:1705.11159, 2017.

[56] Jean Rabault and Alexander Kuhnle. Accelerating deep reinforcement learning strategies of flow control through a multi-environment approach. Physics of Fluids, 31(9):094105, 2019.

[57] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In International Conference on Machine Learning, pages 3053–3062, 2018.

[58] Michael Schaaarschmidt, Alexander Kuhnle, and Kai Fricke. Tensor force: A tensorflow library for applied reinforcement learning. Web page, 2017.

[59] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. arXiv preprint arXiv:1910.06591, 2019.

[60] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, et al. Acme: A research framework for distributed reinforcement learning. arXiv preprint arXiv:2006.00979, 2020.

[61] William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. High-performance parallel implicit cfd. Parallel Computing, 27(4):337–362, 2001.

[62] Horst D Simon. Parallel computational fluid dynamics-implementations and results. STIA, 94:11384, 1992.

[63] John Schulman, Filip Wolski, Prafulla Dharwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[64] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Advances in neural information processing systems, pages 1057–1063, 2000.
[65] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[66] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[67] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[68] Sumit Vashishtha and Siddhartha Verma. Restoring chaos using deep reinforcement learning. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(3):031102, 2020.

[69] Celso Grebogi, Edward Ott, and James A Yorke. Crises, sudden changes in chaotic attractors, and transient chaos. *Physica D: Nonlinear Phenomena*, 7(1-3):181–200, 1983.

[70] Julio M Ottino. Mixing, chaotic advection, and turbulence. *Annual Review of Fluid Mechanics*, 22(1):207–254, 1990.

[71] Ian Dobson and Hsiao-Dong Chiang. Towards a theory of voltage collapse in electric power systems. *Systems & Control Letters*, 13(3):253–262, 1989.

[72] Weiming Yang, Mingzhou Ding, Arnold J Mandell, and Edward Ott. Preserving chaos: Control strategies to preserve complex dynamics with potential relevance to biological disorders. *Physical Review E*, 51(1):102, 1995.

[73] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[74] James L Kaplan and James A Yorke. Preturbulence: a regime observed in a fluid flow model of lorenz. *Communications in Mathematical Physics*, 67(2):93–108, 1979.

[75] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[76] Suhas Patankar. *Numerical heat transfer and fluid flow*. Taylor & Francis, 2018.

[77] David M Driver and H Lee Seegmiller. Features of a reattaching turbulent shear layer in divergent channelflow. *AIAA journal*, 23(2):163–171, 1985.

[78] Christopher L Rumsey. Recent developments on the turbulence modeling resource website. In *22nd AIAA Computational Fluid Dynamics Conference*, page 2927, 2015.

[79] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. Openfoam: A c++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, volume 1000, pages 1–20. IUC Dubrovnik Croatia, 2007.

[80] Florian R Menter, Martin Kuntz, and Robin Langtry. Ten years of industrial experience with the sst turbulence model. *Turbulence, heat and mass transfer*, 4(1):625–632, 2003.