A composable machine-learning approach for steady-state simulations on high-resolution grids

Rishikesh Ranade
Office of CTO
Ansys Inc.
Canonsburg, PA 15317
rishikesh.ranade@ansys.com

Chris Hill
Fluids Business Unit
Ansys Inc.
Lebanon, NH, 03766
chris.hill@ansys.com

Lalit Ghule
Office of CTO
Ansys Inc.
Canonsburg, PA, 15317
lalit.ghule@ansys.com

Jay Pathak
Office of CTO
Ansys Inc.
San Jose, CA, 95134
jay.pathak@ansys.com

Abstract

In this paper we show that our Machine Learning (ML) approach, CoMLSim (**Co**mposable **M**achine **L**earning **Sim**ulator), can simulate PDEs on highly-resolved grids with higher accuracy and generalization to out-of-distribution source terms and geometries than traditional ML baselines. Our unique approach combines key principles of traditional PDE solvers with local-learning and low-dimensional manifold techniques to iteratively simulate PDEs on large computational domains. The proposed approach is validated on more than 5 steady-state PDEs across different PDE conditions on highly-resolved grids and comparisons are made with the commercial solver, Ansys Fluent as well as 4 other state-of-the-art ML methods. The numerical experiments show that our approach outperforms ML baselines in terms of 1) accuracy across quantitative metrics and 2) generalization to out-of-distribution conditions as well as domain sizes. Additionally, we provide results for a large number of ablations experiments conducted to highlight components of our approach that strongly influence the results. We conclude that our local-learning and iterative-inferencing approach reduces the challenge of generalization that most ML models face.

1 Introduction

Engineering simulations utilize solutions of partial differential equations (PDEs) to model various complex physical processes like weather prediction, combustion in a car engine, thermal cooling on electronic chips etc., where the complexity in physics is driven by several factors such as Reynolds number, heat source, geometry, etc. Traditional PDE solvers use discretization techniques to approximate PDEs on discrete computational domains and combine them with linear and non-linear equation solvers to compute PDE solutions. The PDE solutions are dependent on several conditions imposed on the PDE, such as geometry and boundary conditions of the computational domain as well as source terms such as heat generation, buoyancy etc. Moreover, many applications in engineering require highly-resolved computational domains to accurately approximate the PDEs solutions as well as to capture the intricacies of conditions such as geometry and source terms imposed on the PDEs. As a result, traditional PDE solvers, although accurate and generalizable across various PDE conditions, can be computationally slow, especially in the presence of complicated physics and

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

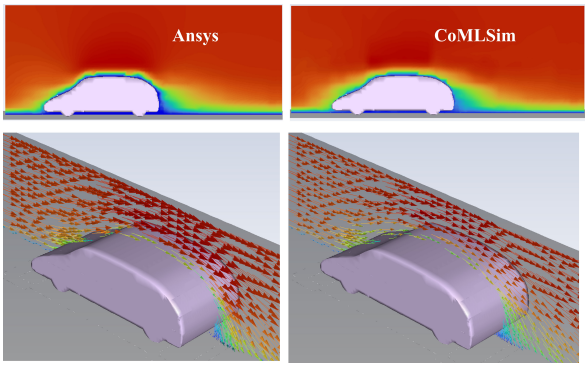

Figure 1: CoMLSim performance vs Ansys Fluent for a high Reynolds number, steady-state turbulent flow over a car (never seen during training) at 90 mph. The vectors and contours predictions match reasonably with MAE of 0.0145. High flow gradients that affect the wake pattern are resolved accurately and comparable to Fluent.

large computational domains. In our work, we specifically aim to increase the speed of engineering simulations using ML techniques on complex PDE conditions used for enterprise simulations.

The idea of using Machine Learning (ML) with PDEs is not unique and has been explored for several decades [1, 2]. ML approaches are computationally fast, but they fall short in terms of accuracy, generalization to a wide range and out-of-distribution PDE conditions and in their ability to scale to highly-resolved computational grids, when compared to traditional PDE solvers. A few shortcomings of the current methods are outlined below and subsequently are motivations for this work.

1. ML approaches use static inferencing to predict PDE solutions as a function of the PDE conditions. In many cases, solutions and conditions have high-dimensional and sparse representations, which are challenging to generalize with such black-boxed static inferencing.
2. Most engineering applications require highly-resolved computational grids to capture detailed solution features, such as hot spots on an electronic chip surface. This adversely impacts training in terms of GPU memory, computational time and data requirements.
3. Most ML approaches fail to use powerful information from traditional PDE solvers related to solver methods, numerical discretization etc.

In this work, we introduce a novel ML approach, **Co**mposable **M**achine **L**earning **Sim**ulator (CoMLSim) to work with high-resolution grids without compromising accuracy. To achieve that, our method decomposes highly-resolved computational grids into smaller local subdomains with Cartesian grids. PDE solutions and conditions on each subdomain are represented by lower-dimensional latent vectors determined using autoencoders. The latent vectors of solutions corresponding to conditions are predicted in each subdomain and stitched together to maintain local consistency, analogous to flux conservation in traditional solvers. In this paper, we describe our method (shown in fig. 2) for building CoMLSim for different PDEs, showcase different experiments to compare accuracy and generalizability with different ML methods and provide an extensive ablation study to understand the impact of CoMLSim's different components on the results.

Significant contributions of this work:

1. The CoMLSim approach combines traditional PDE solver strategies such as domain discretization, flux conservation, solution methodologies etc. with ML techniques to accurately model numerical simulations on high-resolution grids.
2. Our approach operates on local subdomains and solves PDEs in a low-dimensional space. This enables generalizing to out-of-distribution PDE conditions and scaling to bigger domains with large mesh sizes.
3. The iterative inferencing algorithm is self-supervised and allows for coupling with traditional PDE solvers.

Related Works: The use of ML for solving PDEs has gained tremendous traction in the past few years. Much of the research has focused on improving neural network architectures and optimization

techniques to enable generalizable and accurate learning of PDE solutions. More recently, there has been a lot of focus on learning PDE operators with neural networks (NNs) [3, 4, 5, 6, 7, 8, 9]. The neural operators are trained on high-fidelity solutions generated by traditional PDE solvers on a computational grid of specific resolution and do not require any knowledge of the PDE. However, accuracy and out-of-distribution generalizability deteriorates as the resolution of computational grid increases (for example, 2048^2 in 2-D or 256^3 in 3-D). Furthermore, these methods are limited by an upper cap on GPU memory.

A different research direction focuses on training neural networks with physics constrained optimization [10, 11]. These method use Automatic Differentiation (AD) [12] to compute PDE derivatives. The physics-based approaches have been extended to solve complicated PDEs representing complex physics [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. More recently, alternate approaches that use discretization techniques using higher order derivatives and specialize numerical schemes to compute derivatives have shown to provide better regularization for faster convergence [31, 32, 33, 34]. However, the use of optimization techniques to solve PDEs, although accurate, has proved to be extremely slow as compared to traditional solvers and hence, non-scalable to high resolution meshes.

Domain decomposition techniques have been successfully used in the context of ML applied to PDEs [35]. These techniques enable local learning from smaller restricted domains and have proved to accelerate learning of neural networks. In [36, 35], the authors proposed a hybrid method combining ML with domain decomposition to reduce the computational cost of finite element solvers. Similarly, domain decomposition has been extensively used in the context of physics informed neural networks to reduce training costs by enabling distributed training on multiple GPUs [37, 38]. [39] and [40] learn on localized domains but infer on larger computational domains using a stitching process. [41] and [42] learn coefficients of numerical discretization schemes from high fidelity data, which is sub-sampled on coarse grids. [43] learn surrogate models for smaller components to allow for cheaper simulations. [44] learn local prolongation operators from discretization matrices, to improve the rate of convergence of the multigrid linear solvers. Alternatively, other methods such as [45, 46, 47] make us of graph-based models to improve generalization by learning on local elements. In this work, we use a Cartesian grid to discretize each subdomain. However, GNNs [48, 49, 50, 51, 52] as well as FEM use other discretizations such as triangle or polygonal meshes. In future, these ideas can be used to handle unstructured meshes in subdomains.Additionally, other ML methods improve the learning of ML models by compressing PDE solutions on to lower-dimensional manifolds. This has shown to improve accuracy and generalization capability of neural networks [53, 54, 55, 56, 57, 58]. Although, here we propose an iterative inferencing approach in the context of solving PDEs, it is inspired from the implicit deep learning approach used in a variety of machine learning tasks [59, 60, 61, 62]. In this work, domain decomposition is combined with latent space learning is employed to accurately represent solutions on local subdomains and allow scalability to bigger computational domains.

2 CoMLSim approach

2.1 Similarities with traditional PDE solvers

Consider a set of coupled steady-state PDEs with n solution variables ($n = 2$ for example). The coupled PDEs are defined as follows:

$$L_1(u, v) - F_1 = 0; L_2(u, v) - F_2 = 0 \tag{1}$$

$u(x, y, z)$ and $v(x, y, z)$ are defined on a computational domain Ω with boundary conditions specified on the boundary of the computational domain, Ω_b. Here, L_1, L_2 denote PDE operators, F_1, F_2

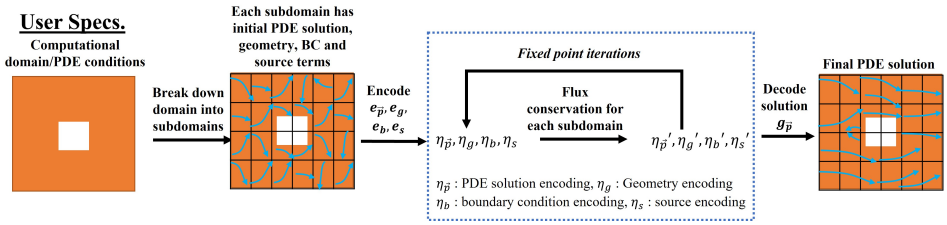

Figure 2: CoMLSim solution algorithm

represent PDE source terms and $u = u_b, v = v_b$ for $\Omega = \Omega_b$. The PDE operators can vary for different PDEs. For example, in a non-linear PDE such as the unsteady, incompressible Navier-Stokes equation the operator, $L = \vec{a}.\vec{\nabla} - \vec{\nabla}.\vec{\nabla}$ Traditional FVM/FDM based PDE solvers solve PDEs in Eq. 1 by computing solutions variables, u, v, and their linear/non-linear derivatives on a discretized computational domain. Iterative solution algorithms are used to conserve fluxes between neighboring computational elements and determine consistent PDE solutions over the entire domain at convergence. The CoMLSim approach is designed to perform similar operations but at the level of subdomains (n^3 elements) using ML techniques. Additional details are provided in supplementary materials sections A.4.

2.2 Solution algorithm for steady-state PDEs

The solution algorithm of CoMLSim approach is shown in Fig. 2 and described in detail in Alg. 1. Similar to traditional solvers, the CoMLSim approach discretizes the computational domain into smaller subdomains contain n^p computational elements, where p refers to spatial dimensionality and n is predetermined. Each subdomain has a constant physical size and represents both PDE solutions and conditions. For example, a subdomain cutting across the cylinder in Fig. 2 represents the geometry and boundary conditions of the part-cylinder as well as the corresponding solution. In a steady-state problem, the solutions on the domain are initialized either with uniform solutions or they are generated from coarse-grid PDE solvers. The initial solution $\vec{p} = (u(\vec{x}), v(\vec{x}))$ is shown with randomly oriented flow vectors in Fig. 2. Pretrained encoders $e_{\vec{p}}, e_g, e_b, e_s$ are used to encode the initial solutions as well as user-specified PDE conditions into lower-dimensional latent vectors, $\eta_{\vec{p}}$ corresponding to PDE solutions and η_g, η_b, η_s corresponding to geometry, boundary conditions and source terms, respectively. Solution and condition latent vectors on groups of neighboring subdomains are concatenated together evaluated using a pre-trained flux conservation autoencoder (Θ) to get a new set of solution latent vectors ($\eta'_{\vec{p}}$) on each subdomain, which are more locally consistent than the original vectors. The new solution latent vectors are iteratively passed through Θ to improve their local consistency and the iteration is stopped when the L_2 norm of change in solution latent vectors meets a specified tolerance, otherwise the iteration continues with the updated latent vectors. The latent vectors of the PDE conditions are not updated and help in steering the solution latent vectors to an equilibrium state that is decoded to PDE solutions using pretrained decoders ($g_{\vec{p}}$) on the computational domain. The converged solution in Fig. 2 is represented with flow vectors that are locally consistent with neighboring subdomains.The iterative procedure used in the CoMLSim approach can be implemented using several linear or non-linear equation solvers, such as Fixed point iterations [59], Gauss Seidel, Newton's method etc., that are used in commercial PDE solvers. Out of those we have explored Point Jacobi and Gauss Seidel, which are described below. The use of autoencoders for compressed solution and condition representation in tandem with iterative inferencing solution algorithm is inspired from [58, 31] but the main difference in this work is that the solution procedure is carried out on local subdomains as opposed to entire computational domains to align with principles used in traditional PDE solvers. Supplementary material section G explains our algorithm in more details using an example.

Algorithm 1: Solution methodology of CoMLSim approach

Domain Decomposition: Computational domain $\Omega \rightarrow$ Subdomains Ω_c
Initialize solution on all Ω_c: $\vec{p}(\vec{x}) = 0.0$ for all $\vec{x} \in \Omega_c$
Encode solutions on Ω_c: $\eta_{\vec{p}} = e_u(\vec{p}(\Omega_c))$
Encode conditions on Ω_c:
$\eta_g = e_g(g(\Omega_c))$, $\eta_b = e_b(b(\Omega_c))$, $\eta_s = e_s(s(\Omega_c))$
Set convergence tolerance: $\epsilon_t = 1e^{-8}$
while $\epsilon > \epsilon_t$ **do**
  for $\Omega_c \in \Omega$ **do**
    Gather neighbors of Ω_c: $\Omega_{nb} = [\Omega_c, \Omega_{left}, \Omega_{right}, ...]$
    $\eta'_{\vec{p}} = \Theta(\eta^{nb}_{\vec{p}}, \eta^{nb}_b, \eta^{nb}_g, \eta^{nb}_s)$
  Compute L_2 norm: $\epsilon = ||\eta_{\vec{p}} - \eta'_{\vec{p}}||^2_2$
  Update: $\eta_{\vec{p}} \leftarrow \eta'_{\vec{p}}$ for all $\Omega_c \in \Omega$
Decode PDE solution on all Ω_c: $\vec{p} = g_{\vec{p}}(\eta_{\vec{p}\Omega_c})$

Why flux conservation networks work? The flux conservation network (Θ) is simply an autoencoder which takes encoding of solutions and conditions on a group of neighboring subdomains as both inputs and outputs. Similar to traditional solvers that use approximations to represent relationships between neighboring solutions on elements, the flux conservation autoencoder does the same for encoded subdomain solutions and corresponding conditions. The training of this network is carried out on converged, locally consistent solutions generated for a specific PDE for arbitrary conditions. Since this network has only learnt locally consistent solutions, if one were to initialize a group of neighboring subdomains with random noise and iteratively pass it through this network the output of such a procedure would be an equilibrium solution corresponding to some locally consistent PDE solution. However, the correct PDE solution it converges to depends on the fixed condition specified in this procedure. Additional results and details are provided in the supplementary materials sections A.1, A.4 and C.1.5.

Stability of flux conservation autoencoder: The iterative inferencing approach proposed in this work is similar to traditional approaches used in solving system of linear equations. Our algorithm uses fixed point iterations to solve Eq. 2

$$\eta'_{\vec{p}} = \Theta(\eta^{nb}_{\vec{p}}, \eta^{nb}_{b}, \eta^{nb}_{g}, \eta^{nb}_{s}) \tag{2}$$

where, Θ corresponds to the weights of flux conservation network, $\eta_{()}$ refers to the latent vectors and nb refers to neighboring subdomains defined below in Eq. 3.

$$\eta^{nb}_{\vec{p}} = \left[\eta^{s}_{\vec{p}}, \eta^{left}_{\vec{p}}, \eta^{right}_{\vec{p}}, \eta^{top}_{\vec{p}}, \eta^{bottom}_{\vec{p}}\right] \tag{3}$$

Similar to linear systems, the stability of our iterative inferencing approach is governed by the condition number [63] defined in Eq. 4

$$||\frac{\partial\Theta(\eta_{\vec{p}}, \eta_b, \eta_g, \eta_s)}{\partial\eta_{\vec{p}}}|| < 1 \tag{4}$$

Solution algorithms: Point Jacobi vs Gauss Seidel: Since the algorithm loops over all the subdomains in a specified order, while updating the solution encoding of a subdomain c, there are subdomains in the neighborhood that already have an updated solution encoding. The Gauss Seidel method uses these updated solution encodings ($\eta'_{\vec{p}}$) on neighboring subdomains to update the solution encoding on subdomain, c. On the other hand, Point Jacobi method does not make use of this and hence can be easily vectorized for significantly faster computation. The implementation has a remarkable similarity with how traditional solvers use these methods [64] and leaves the door open to use other non-linear optimizers with physics-based constraints.

2.3 Neural network components in CoMLSim

Our algorithm employs two types of autoencoders, CNN autoencoders to establish a lower-dimensional representation of PDE solutions and conditions from local subdomains grids and FCNN Autoencoders for flux conservation, where the goal is to learn a reduced representation of solution and condition latent vectors in a local neighborhood. Let us consider an example of solving the Laplace equation, $\nabla^2 u = 0$ in 2D for arbitrarily shaped computational domains, Ω_g. In this example, the CoMLSim algorithm will require 3 autoencoders, described in Eq. 5, to learn lower-dimensional representations on local subdomains.

$$\left.\begin{aligned} u' &= S(u, \eta_u) \\ g' &= G(g, \eta_g) \\ \eta'_{nb} &= \Theta(\eta_{nb}, \zeta) \end{aligned}\right\} \tag{5}$$

where, u, η_u, S refers to the solution, its latent vector and the weights of the autoencoder, respectively, g, η_g, G refers to a representation of geometry, such as Signed Distance Fields (SDF) [65], its latent vector and the weights of the autoencoder, Θ refers to the weights of flux conservation autoencoder and η_{nb} represents a set of concatenated latent vectors, η_u and η_g on a group of neighboring subdomains. All the autoencoders are trained with samples of PDE solutions generated for the same use case. In the case of coupled PDEs, a single autoencoder is trained for all solution variables. Additional details related to the autoencoder networks are provided in the supplementary materials section A.

Why Autoencoders?: Solutions to classical PDEs such as the Laplace equation can be represented by homogeneous solutions as follows:

$$\phi(x, y) = a_0 + a_1 x + a_2 y + a_3(x^2 - y^2) + a_4(2xy) + ... \tag{6}$$

where, $\vec{A} = a_0, a_1, a_2, ..., a_n$ are constant coefficients that can be used to reconstruct the PDE solution on any local subdomain. $\vec{A}$ can be considered as a compressed encoding of the Laplace solutions. Since, it is not possible to explicitly derive such compressed encodings for other high dimensional and non-linear PDEs, the CoMLSim approach relies on autoencoders to compute them. It is known that non-linear autoencoders with good compression ratios can learn powerful non-linear generalizations [66, 67, 68]. Autoencoders also have great denoising abilities, which improve robustness and stability, when used in iterative settings [58]. In their paper, Park and Lee [69] demonstrate that the latent manifold established by a trained autoencoder is stable for varying intensities of Gaussian noise. This quality of autoencoders is useful for the convergence of the iterative inferencing approach.

3 Experiments

In this section as well as in the supplementary materials section C, we consider a number of use cases with varying degrees of difficulty resulting from the PDE formulation as well as source terms, geometry and boundary conditions. The PDEs have applications in fluid mechanics, structural mechanics and semiconductor simulations.

Details of experiments: Here we provide some details about the experiments. Additional details and experiments on Laplace and Darcy equation may be found in the supplementary materials section D.

1. **2D Poisson equation:** The Poisson's equation, shown in Eq. 7, is very popular in engineering simulations, for example, chip temperature prediction, pressure equation in fluids etc.

$$\nabla^2 u = f \tag{7}$$

where, u is the solution variable and f is the source term. This PDE is solved on a 1024x1024 grid to resolve the high-frequency features of the source term. The source, f, is sampled from a Gaussian mixture model shown in Eq. 8.

$$\sum_{j=0}^{1024}\sum_{j=0}^{1024} f_{i,j} = \sum_{j=0}^{1024}\sum_{j=0}^{1024}\sum_{k=0}^{30} A_k \exp\left(-\left(\frac{x-\mu_{x,k}}{\sigma_{x,k}}\right)^2 - \left(\frac{y-\mu_{y,k}}{\sigma_{y,k}}\right)^2\right) \tag{8}$$

where, x, y correspond to the grid coordinates. A_k randomly assumes either 0 or 1 to vary the number of active Gaussians in the model. μ_x, μ_y and σ_x, σ_y are the mean and standard deviations of Gaussians in x and y directions, respectively. The means and standard deviations vary randomly between 0 to 1 and 0.001 to 0.01, respectively. The smaller magnitude of standard deviation results in hot spots that require highly-resolved grids. In this case, 256 training and 100 testing solutions are generated using Ansys Fluent.

2. **2D non-linear coupled Poisson equation:** The coupled non-linear Poisson's equation is shown below in Eq. 9. u, v are the solution variables and f is the source term, similar to the description in Eq. 8. This PDE has applications in reactive flow simulations.

$$\nabla^2 u = f - u^2; \qquad \nabla^2 v = \frac{1}{u^2 + \epsilon} - v^2 \tag{9}$$

The data generation is similar to Experiment 2 with a slight difference that the source term is less stiff with a standard deviation that varies between 0.005 to 0.05.

3. **3D Reynolds-Averaged Navier-Stokes external flow:** This use case consists of a 3-D channel flow with resolution 304x64x64 at high Reynolds number over arbitrarily shaped objects. The corresponding PDEs are presented in supplementary materials section C.3 or may be referred from [70]. The characteristics of flow generated on the downstream has important applications in the design of automobiles and airplanes. In this case, we use 5 primitive 3-D geometries namely, cylinder, cuboid, trapezoid, airfoil wing and wedge and their random combinations and rotational augmentations to create 150 training geometries and 50 testing geometries, which are solved with Ansys Fluent to generate the data. Out-of-distribution testing is carried out on simplistic automobile geometries as shown in Figure 1.

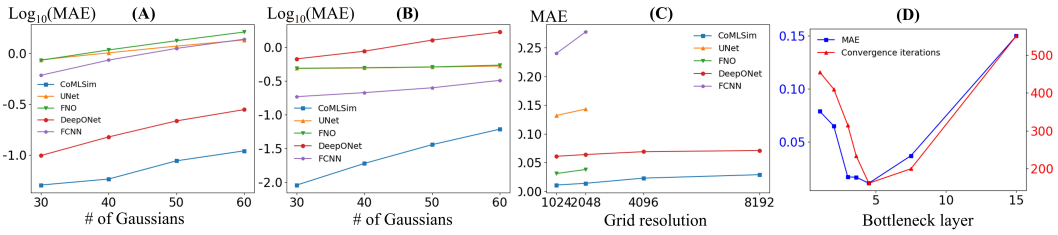

Figure 3: (A): Error vs No. of Gaussians for linear Poisson's; (B): Error vs No. of Gaussians for non-linear Poisson's; (C): Error vs grid resolution for linear Poisson's; (D): MAE and convergence iterations vs flux conservation bottleneck size

4. **3D chip cooling with Natural convection:** In this case, we extend the complexity of Experiments $1, 2$ and 3 to a industrial 3-D case of chip cooling with natural convection. In this problem, the computational grid (128^3 resolution) consists of a chip that is subjected to the powermap specified by Eq. 8. The heating of the chip results in generation of velocity in the fluid and at steady-state, a balance is achieved. We use Ansys Fluent to generate 300 training and 100 testing solutions for velocity, pressure, temperature for arbitrary sources.

4 Results and Ablation studies

In this section, we provide a variety of results for the experiments outlined in Section 3. Additionally, we conduct thorough ablation studies to understand the different components of the CoMLSim approach in more detail. We use the 2-D Linear Poisson's equation experiment for these studies unless otherwise specified. Details regarding the CoMLSim set up for each experiment, baseline network architectures as well as additional results corresponding to contour and line plots and comparisons across other metrics are provided in supplementary materials section C.

4.1 Comparison with Ansys Fluent and other ML baselines

The CoMLSim is compared with other ML baselines namely, UNet [71], FNO [8] and DeepONet [7] for the experiments outlined in Section 3. We compare the mean absolute error with respect to Ansys Fluent and averaged over the unseen testing cases and all solution variables. In the case of 3-D chip cooling we report the L_∞ norm of temperature as this metric is more suited to this industrial application. It may be observed from Table 1 that the CoMLSim approach performs better than other ML baselines. All the ML methods perform better for the non-linear Poisson's case as compared to the linear Poisson's because the source term is less stiff but the CoMLSim approach does a better job in modeling the non-linearity arising due solution coupling. It must be noted that CoMLSim as well as the baselines are trained with reasonably training samples. The baseline performance can have different outcomes with increasing the size of the training data.

Table 1: Comparison with baselines

Experiment	Metric	CoMLSim	UNet	FNO	DeepONet	FCNN
2-D Linear Poisson's	L_1	0.011	0.132	0.031	0.061	0.267
2-D non linear Poisson's	L_1	0.0053	0.0877	0.0278	0.527	0.172
3-D NS external flow	L_1	0.012	0.0625	0.038	0.81	0.125
3-D chip cooling	L_∞	15.2	95.21	60.836	45.27	192.7

4.2 Assessment of generalizability

In this section, we assess the generalizability of our method specifically with respect to out-of-distribution PDE conditions and scaling to higher resolution grids. The results are compared with Ansys Fluent as well as other ML baselines discussed in section 4.1.

Out-of-distribution source term: The source term for linear and non-linear Poisson equations are sampled from a Gaussian mixture model described in Eq. 8, where the number of Gaussian is randomly chosen between 0 and 30. In this experiment, we evaluate

the generalizability of our approach for source terms with exactly 30, 40, 50 and 60 number of Gaussians, corresponding to out-of-distribution for the training data distributions. For each case, the results averaged over 10 testing samples are compared with Ansys Fluent. It may be observed from 3A and B, that the accuracy of all ML approaches decreases as you move further away from the source term distribution. However, the CoMLSim approach performs significantly better than other ML baselines and the accuracy is reasonable even for the case with 60 Gaussian mixture model, which is substantially different from the training distribution.

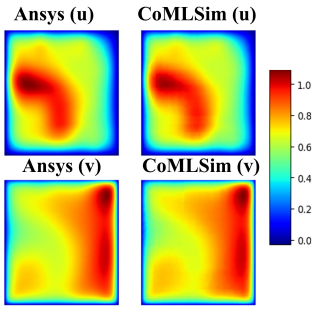

Figure 4: Example of out-of-distribution source (60 Gaussians) comparison for non-linear Poisson's

Increasing mesh size by increasing domain size: Next, we evaluate the performance of CoMLSim for bigger domains with larger grid size and compare it with other ML baselines. In this experiment, we compute solutions for the 100 source term conditions in the test set but on 4 different domains with mesh sizes, 1024^2, 2048^2, 4096^2, 8192^2, respectively. The mean and standard deviation of Gaussians in the testing set are proportionally increased with the mesh size to ensure that the problem definition does not change. It is important to note that none of the networks are retrained for larger sized grids. It maybe observed from table 3 that the CoMLSim continues to scale to larger mesh sizes with similar accuracy as the original mesh size. The slight drop in accuracy due to increase in mesh size is attributed to accumulation of error caused by the flux conservation networks resulting from the increase in number of subdomains. Other ML baselines such as UNet and FNO scale up to a grid size of 2048^2 but cannot evaluate beyond that due to GPU memory constraints. DeepONet has a point-wise inference and can also scale to larger-sized grids but its accuracy is lower than CoMLSim.

Out-of-distribution geometries: Next, we evaluate the performance of CoMLSim on 2 unseen car models presented at different 3-D angles of rotation to the external flow. It should be noted that these geometries are more complicated than the primitive objects considered for training our approach 3. The mean absolute errors with respect to Ansys Fluent across all solution variables are 0.0145, 0.06924, 0.04175, 0.91 and 0.13725 for CoMLSim, UNet, FNO, DeepONet and FCNN, respectively. Additional results and analysis are provided in supplementary materials C.3.3 and C.3.4.

4.3 Analysis of Subdomain size

In this experiment, we analyze the effect of subdomain resolution on accuracy and computational cost of the CoMLSim approach. We train 3 instances at resolutions of 32^2, 64^2 and 128^2. The compression ratio in autoencoders is kept the same for the different subdomain resolutions. The mean absolute errors on testing set are 0.015, 0.011 and 0.029, respectively. Additionally, the iterations required to convergence are 420, 162 and 89, respectively. The convergence history is shown in Fig. 5A. The 128^2 subdomain resolution has a lower accuracy because it is challenging to train accurate autoencoders as bigger subdomains capture a large amount of information. On the other hand, the computational cost is the lowest for the 128^2 resolution because the number of subdomains in the entire computational domain are significantly less and hence, the solution algorithm converges faster.

4.4 Flux conservation bottleneck layer size

The flux conservation autoencoder as it is the primary workhorse of the CoMLSim solution algorithm. As a result, we evaluate the effect of the bottleneck layer size on the accuracy and computational speed. Each instance of CoMLSim with different bottleneck size is tested on 100 unseen test cases. We verify that all models satisfy the stability criterion specified in Eq. 3. It may be observed from Figure 3D that as the compression ratio of the flux conservation autoencoder decreases, it begins to overfit and the testing error as well as the number of convergence iterations and computational time significantly increase. On the other hand, if the compression ratio is too large the testing error increases because the autoencoders underfit. In alignment with the collective intuition about autoencoders, there exists an optimum bottleneck size compression ratio where the best testing error is obtained for small computational times.

4.5 Robustness and stability

A long standing challenge in the field of numerical simulation is to guarantee the stability and convergence of non-linear PDE solvers. However, we believe that the denoising capability of autoencoders [72, 66, 73, 74, 75] used in our iterative solution algorithm presents a unique benefit, irrespective of the choice of initial conditions. Here, we empirically demonstrate the robustness and stability of our approach. In scenario A, we randomly sample 25 initial solutions from a uniform distribution and in scenario B we sample from 6 different distributions, namely Uniform, Gumbel, Laplace, Gamma, Normal, Logistic etc. The mean convergence trajectory is plotted in Fig. 5C and 5D show that the log of L_2 norm of the convergence error falls below an acceptable tolerance for all cases and demonstrates the stability of our approach.

4.6 CoMLSim solution methods: Point Jacobi vs Gauss Seidel

Next, we compare the performance of the Point Jacobi and Gauss Seidel implementations in the CoMLSim approach across 3 metrics, mean absolute error, computational cost and iterations to converge. Both methods have a similar prediction accuracy with a mean absolute error of 0.013 and 0.017, respectively but the Point Jacobi method is computationally cheaper requiring 4.3 seconds to converge as compared to 72 seconds of Gauss Seidel. It may be observed from Fig. 5E that the Gauss Seidel method takes significantly fewer convergence iterations, but the Point Jacobi method is faster because it can be efficiently vectorized on a GPU.

4.7 Coupling with traditional PDE solvers and super-resolution

In this experiment, we analyze the potential of coupling the CoMLSim approach with a traditional PDE solver such as Ansys Fluent and subsequently demonstrate how our approach inherently possesses the capability to perform super-resolution. We mainly carry out 5 experiments, where we start from a coarse grid solution for the 100 test samples of linear Poisson's equation solved by Ansys Fluent on a 64x64 grid (256x smaller than the fine resolution), add random noise to the coarse initial solution with varying amplitudes (10%, 25% and 50%), and use that as an initial solution in the CoMLSim approach. The convergence trajectory in Fig. 5B shows that the case with coarse grid initialization converges the fastest, followed by initialization with noise and zero initialization. All solutions converge to the same accuracy.

4.8 Computational speed

We observe that the CoMLSim approach is about 40-50x faster as compared to commercial steady-state PDE solvers such as Ansys Fluent for the same mesh resolution and physical domain size in all the experiments presented in this work. In comparison to the ML baselines, our approach is expected to be slightly slower because it adopts an iterative inferencing approach. But, this is compensated by solution accuracy, generalizability and robustness on high-resolution grids. A detailed analysis is provided in supplementary materials section E.

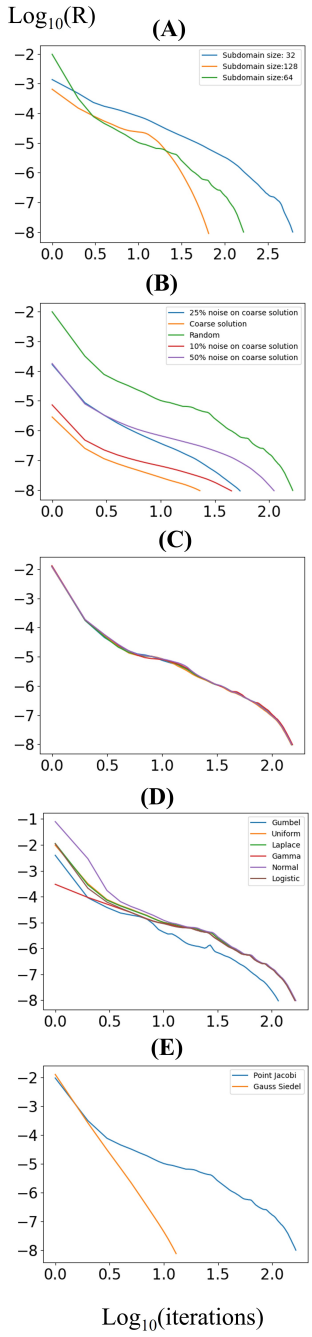

Figure 5: Convergence history for (A): varying subdomain sizes; (B): coupling with traditional PDE solvers; (C, D): Robustness and stability; (E): Point Jacobi vs Gauss Seidel

4.9 Importance of local learning and latent-space representation

To understand the effect of local learning, we set up the CoMLSim approach for a single subdomain (size equal to the computational domain). From this experiment, we conclude that the solution errors with a single subdomain is significantly higher than having multiple subdomains (See supplementary material section C.1.5.1). Additionally, to study the impact of latent space representations we experiment with the latent vector sizes as well as a case where CoMLSim solves in the solution space. We conclude that latent space representations have a larger impact on the computational cost than accuracy (See supplementary material section C.1.5.2).

4.10 Solution evolution during inference

In Figure. 6, we present the evolution of the PDE solution for the Poisson's equation experiment at different iterations of the iterative inferencing. The results are plotted for 3 unseen test cases with different source term distribution. In each case, the solution on the domain is initialized with random noise sampled from a uniform distribution between -1 and 1. The contour plot at iteration 0 shows an attenuated noisy solution because the initial solution is passed through the solution encoder and decoder. It may be observed that at iteration 1, the PDE solution starts evolving from the regions of high source term. As iterations progress, the solution begins to diffuse through the solution domain due to the repeated operation of the flux conservation network, until it converges. The diffusion process is dominant in the case of Poisson's equations and is effectively captured by the flux conservation autoencoder in CoMLSim. The error plot shows the solution accuracy at different iterations during the inference procedure. The log error is the largest initially and drops linearly as the iterations progress.

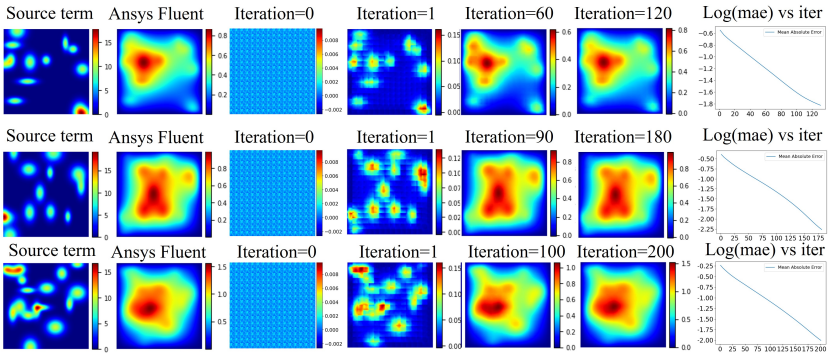

Figure 6: Evolution of PDE solution during inference

5 Conclusion

In this work, we introduce the CoMLSim approach, which is a self-supervised, low-dimensional and local machine learning approach for solving PDE on highly-resolved grids and generalizing across a wide range of PDE conditions. Our approach is inspired from strategies employed in traditional PDE solvers and adopts iterative inferencing. The proposed approach is demonstrated to predict accurate solutions for a range of PDEs, generalize reasonably across geometries, source terms and BCs. Moreover, it scales to bigger domains with larger mesh size.

Broader impact, future work & limitations Although the proposed ML-model can generalize to out-of-distribution geometries, source terms and BCs, but like other ML approaches, extrapolation to any and all PDE conditions that are significantly different from the original distribution still remains a challenge. However, this work takes a big step towards laying down the framework on how truly generalizable ML-based solvers can be developed. In future, we would like to address these challenges of generalizability and scalability by training autoencoders on random, application agnostic PDE solutions and enforcing PDE-based constraints in the iterative inferencing procedure. Future work will also investigate the potential for hybrid solvers and extensions to transient PDEs and inverse problems. Finally, we will also address extension of the current approach to unstructured meshes, which is a current limitation.

References

[1] James P Crutchfield and BS McNamara. Equations of motion from a data series. *Complex systems*, 1(417-452):121, 1987.

[2] Ioannis G Kevrekidis, C William Gear, James M Hyman, Panagiotis G Kevrekidid, Olof Runborg, Constantinos Theodoropoulos, et al. Equation-free, coarse-grained multiscale computation: Enabling mocroscopic simulators to perform system-level analysis. *Communications in Mathematical Sciences*, 1(4):715–762, 2003.

[3] Kaushik Bhattacharya, Bamdad Hosseini, Nikola B Kovachki, and Andrew M Stuart. Model reduction and neural networks for parametric pdes. *arXiv preprint arXiv:2005.03180*, 2020.

[4] Anima Anandkumar, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Nikola Kovachki, Zongyi Li, Burigede Liu, and Andrew Stuart. Neural operator: Graph kernel network for partial differential equations. In *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020.

[5] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Multipole graph neural operator for parametric partial differential equations. *arXiv preprint arXiv:2006.09535*, 2020.

[6] Ravi G Patel, Nathaniel A Trask, Mitchell A Wood, and Eric C Cyr. A physics-informed operator regression framework for extracting data-driven continuum models. *Computer Methods in Applied Mechanics and Engineering*, 373:113500, 2021.

[7] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

[8] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[9] Zongyi Li, Hongkai Zheng, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, and Anima Anandkumar. Physics-informed neural operator for learning partial differential equations. *arXiv preprint arXiv:2111.03794*, 2021.

[10] Maziar Raissi, 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.

[11] Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

[12] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[13] Xiaowei Jin, Shengze Cai, Hui Li, and George Em Karniadakis. Nsfnets (navier-stokes flow nets): Physics-informed neural networks for the incompressible navier-stokes equations. *Journal of Computational Physics*, 426:109951, 2021.

[14] Zhiping Mao, Ameya D Jagtap, and George Em Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, 360:112789, 2020.

[15] Chengping Rao, Hao Sun, and Yang Liu. Physics-informed deep learning for incompressible laminar flows. *Theoretical and Applied Mechanics Letters*, 10(3):207–212, 2020.

[16] Jin-Long Wu, Heng Xiao, and Eric Paterson. Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework. *Physical Review Fluids*, 3(7): 074602, 2018.

[17] Elizabeth Qian, Boris Kramer, Benjamin Peherstorfer, and Karen Willcox. Lift & learn: Physics-informed machine learning for large-scale nonlinear dynamical systems. *Physica D: Nonlinear Phenomena*, 406:132401, 2020.

[18] Vikas Dwivedi, Nishant Parashar, and Balaji Srinivasan. Distributed learning machines for solving forward and inverse problems in partial differential equations. *Neurocomputing*, 420: 299–316, 2021.

[19] Ehsan Haghighat, Maziar Raissi, Adrian Moure, Hector Gomez, and Ruben Juanes. A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 379:113741, 2021.

[20] Ehsan Haghighat and Ruben Juanes. Sciann: A keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *Computer Methods in Applied Mechanics and Engineering*, 373:113552, 2021.

[21] Mohammad Amin Nabian, Rini Jasmine Gladstone, and Hadi Meidani. Efficient training of physics-informed neural networks via importance sampling. *Computer-Aided Civil and Infrastructure Engineering*, 2021.

[22] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. hp-vpinns: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, 374:113547, 2021.

[23] Shengze Cai, Zhicheng Wang, Frederik Fuest, Young Jin Jeon, Callum Gray, and George Em Karniadakis. Flow over an espresso cup: inferring 3-d velocity and pressure fields from tomographic background oriented schlieren via physics-informed neural networks. *Journal of Fluid Mechanics*, 915, 2021.

[24] Shengze Cai, Zhicheng Wang, Sifan Wang, Paris Perdikaris, and George Karniadakis. Physics-informed neural networks (pinns) for heat transfer problems. *Journal of Heat Transfer*, 2021.

[25] Mathis Bode, Michael Gauding, Zeyu Lian, Dominik Denker, Marco Davidovic, Konstantin Kleinheinz, Jenia Jitsev, and Heinz Pitsch. Using physics-informed enhanced super-resolution generative adversarial networks for subfilter modeling in turbulent reactive flows. *Proceedings of the Combustion Institute*, 38(2):2617–2625, 2021.

[26] Ehsan Taghizadeh, Helen M Byrne, and Brian D Wood. Explicit physics-informed neural networks for non-linear upscaling closure: the case of transport in tissues. *arXiv preprint arXiv:2104.01476*, 2021.

[27] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[28] Khemraj Shukla, Ameya D Jagtap, and George Em Karniadakis. Parallel physics-informed neural networks via domain decomposition. *arXiv preprint arXiv:2104.10013*, 2021.

[29] Oliver Hennigh, Susheela Narasimhan, Mohammad Amin Nabian, Akshay Subramaniam, Kaustubh Tangsali, Max Rietmann, Jose del Aguila Ferrandis, Wonmin Byeon, Zhiwei Fang, and Sanjay Choudhry. Nvidia simnet^{TM}: an ai-accelerated multi-physics simulation framework. *arXiv preprint arXiv:2012.07938*, 2020.

[30] Li Li, Stephan Hoyer, Ryan Pederson, Ruoxi Sun, Ekin D Cubuk, Patrick Riley, Kieron Burke, et al. Kohn-sham equations as regularizer: Building prior knowledge into machine-learned physics. *Physical review letters*, 126(3):036401, 2021.

[31] Rishikesh Ranade, Chris Hill, and Jay Pathak. Discretizationnet: A machine-learning based solver for navier–stokes equations using finite volume discretization. *Computer Methods in Applied Mechanics and Engineering*, 378:113722, 2021.

[32] Han Gao, Luning Sun, and Jian-Xun Wang. Phygeonet: physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state pdes on irregular domain. *Journal of Computational Physics*, 428:110079, 2021.

[33] Nils Wandel, Michael Weinmann, and Reinhard Klein. Learning incompressible fluid dynamics from scratch towards fast, differentiable fluid models that generalize. *arXiv preprint arXiv:2006.08762*, 2020.

[34] Haiyang He and Jay Pathak. An unsupervised learning approach to solving heat equations on chip based on auto encoder and image gradient. *arXiv preprint arXiv:2007.09684*, 2020.

[35] Alexander Heinlein, Axel Klawonn, Martin Lanser, and Janine Weber. Combining machine learning and domain decomposition methods for the solution of partial differential equations—a review. *GAMM-Mitteilungen*, 44(1):e202100001, 2021.

[36] Alexander Heinlein, Axel Klawonn, Martin Lanser, and Janine Weber. Machine learning in adaptive domain decomposition methods—predicting the geometric location of constraints. *SIAM Journal on Scientific Computing*, 41(6):A3887–A3912, 2019.

[37] Ke Li, Kejun Tang, Tianfan Wu, and Qifeng Liao. D3m: A deep domain decomposition method for partial differential equations. *IEEE Access*, 8:5283–5294, 2019.

[38] Ameya D Jagtap, Ehsan Kharazmi, and George Em Karniadakis. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering*, 365:113028, 2020.

[39] Lu Lu, Haiyang He, Priya Kasimbeg, Rishikesh Ranade, and Jay Pathak. One-shot learning for solution operators of partial differential equations. *arXiv preprint arXiv:2104.05512*, 2021.

[40] Hengjie Wang, Robert Planas, Aparna Chandramowlishwaran, and Ramin Bostanabad. Train once and use forever: Solving boundary value problems in unseen domains with pre-trained deep learning models. *arXiv preprint arXiv:2104.10873*, 2021.

[41] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

[42] Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics. *arXiv preprint arXiv:2102.01010*, 2021.

[43] Alex Beatson, Jordan Ash, Geoffrey Roeder, Tianju Xue, and Ryan P Adams. Learning composable energy surrogates for pde order reduction. *Advances in Neural Information Processing Systems*, 33, 2020.

[44] Daniel Greenfeld, Meirav Galun, Ronen Basri, Irad Yavneh, and Ron Kimmel. Learning to optimize multigrid pde solvers. In *International Conference on Machine Learning*, pages 2415–2423. PMLR, 2019.

[45] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

[46] Jiayang Xu, Aniruddhe Pradhan, and Karthikeyan Duraisamy. Conditionally parameterized, discretization-aware neural networks for mesh-based modeling of physical systems. *Advances in Neural Information Processing Systems*, 34:1634–1645, 2021.

[47] Lukas Harsch and Stefan Riedelbauch. Direct prediction of steady-state flow fields in meshed domain with graph networks. *arXiv preprint arXiv:2105.02575*, 2021.

[48] Ferran Alet, Adarsh Keshav Jeewajee, Maria Bauza Villalonga, Alberto Rodriguez, Tomas Lozano-Perez, and Leslie Kaelbling. Graph element networks: adaptive, structured computation and memory. In *International Conference on Machine Learning*, pages 212–222. PMLR, 2019.

[49] Valerii Iakovlev, Markus Heinonen, and Harri Lähdesmäki. Learning continuous-time pdes from sparse data with graph neural networks. *arXiv preprint arXiv:2006.08956*, 2020.

[50] Wenzhuo Liu, Mouadh Yagoubi, and Marc Schoenauer. Multi-resolution graph neural networks for pde approximation. In *International Conference on Artificial Neural Networks*, pages 151–163. Springer, 2021.

[51] Zijie Li and Amir Barati Farimani. Graph neural network-accelerated lagrangian fluid simulation. *Computers & Graphics*, 103:201–211, 2022.

[52] J Chen, E Hachem, and J Viquerat. Graph neural networks for laminar flow prediction around random two-dimensional shapes. *Physics of Fluids*, 33(12):123607, 2021.

[53] Steffen Wiewel, Byungsoo Kim, Vinicius C Azevedo, Barbara Solenthaler, and Nils Thuerey. Latent space subdivision: stable and controllable time predictions for fluid flow. In *Computer Graphics Forum*, volume 39, pages 15–25. Wiley Online Library, 2020.

[54] 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.

[55] Byungsoo Kim, Vinicius C Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep fluids: A generative network for parameterized fluid simulations. In *Computer Graphics Forum*, volume 38, pages 59–70. Wiley Online Library, 2019.

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

[57] Kai Fukami, Taichi Nakamura, and Koji Fukagata. Convolutional neural network based hierarchical autoencoder for nonlinear mode decomposition of fluid field data. *Physics of Fluids*, 32(9):095110, 2020.

[58] Rishikesh Ranade, Chris Hill, Haiyang He, Amir Maleki, and Jay Pathak. A latent space solver for pde generalization. *arXiv preprint arXiv:2104.02452*, 2021.

[59] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *arXiv preprint arXiv:1909.01377*, 2019.

[60] Yulia Rubanova, Alvaro Sanchez-Gonzalez, Tobias Pfaff, and Peter Battaglia. Constraint-based graph network simulator. *arXiv preprint arXiv:2112.09161*, 2021.

[61] Edoardo Remelli, Artem Lukoianov, Stephan Richter, Benoît Guillard, Timur Bagautdinov, Pierre Baque, and Pascal Fua. Meshsdf: Differentiable iso-surface extraction. *Advances in Neural Information Processing Systems*, 33:22468–22478, 2020.

[62] Yilun Du, Shuang Li, Joshua Tenenbaum, and Igor Mordatch. Learning iterative reasoning through energy minimization. In *International Conference on Machine Learning*, pages 5570–5582. PMLR, 2022.

[63] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

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

[65] Amir Maleki, Jan Heyse, Rishikesh Ranade, Haiyang He, Priya Kasimbeg, and Jay Pathak. Geometry encoding for numerical simulations. *arXiv preprint arXiv:2104.07792*, 2021.

[66] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[67] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[68] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *arXiv preprint arXiv:2003.05991*, 2020.

[69] Saerom Park and Jaewook Lee. Stability analysis of denoising autoencoders based on dynamical projection system. *IEEE Transactions on Knowledge and Data Engineering*, 33(8):3155–3159, 2020.

[70] Alexandre Joel Chorin. Numerical solution of the navier-stokes equations. *Mathematics of computation*, 22(104):745–762, 1968.

[71] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[72] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

[73] Bo Du, Wei Xiong, Jia Wu, Lefei Zhang, Liangpei Zhang, and Dacheng Tao. Stacked convolutional denoising auto-encoders for feature representation. *IEEE transactions on cybernetics*, 47 (4):1017–1027, 2016.

[74] Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. *arXiv preprint arXiv:1305.6663*, 2013.

[75] Marc Ranzato, Christopher Poultney, Sumit Chopra, Yann LeCun, et al. Efficient learning of sparse representations with an energy-based model. *Advances in neural information processing systems*, 19:1137, 2007.

[76] Frederic Gibou, Ronald Fedkiw, and Stanley Osher. A review of level-set methods and some recent applications. *Journal of Computational Physics*, 353:82–109, 2018.

[77] Yinhao Zhu and Nicholas Zabaras. Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification. *Journal of Computational Physics*, 366: 415–447, 2018.

[78] Giancarlo Alfonsi. Reynolds-averaged navier–stokes equations for turbulence modeling. *Applied Mechanics Reviews*, 62(4), 2009.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Yes
   (b) Did you describe the limitations of your work? [Yes] Yes. See Section 5
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] N/A. This is not relevant to our work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] Yes, our paper conforms to the ethical guidelines.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] N/A
   (b) Did you include complete proofs of all theoretical results? [N/A] N/A
3. If you ran experiments...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] No. The source code is proprietary. But, we will provide the data and code for the main experiment once the paper is accepted. Instructions for generalization are provided in the supplementary materials section F.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Yes, all the details are provided in the main paper as well as supplementary materials.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We test the robustness and stability of our solver for different initial conditions and provide the mean and standard deviation of convergence error for these experiments in Section 4.5

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Yes, all the details are provided in the main paper as well as supplementary materials.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [N/A]

(b) Did you mention the license of the assets? [N/A]

(c) Did you include any new assets either in the supplemental material or as a URL? [N/A]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Supplementary Materials for A composable machine-learning approach for steady-state simulations on high-resolution grids

In the supplementary materials, we provide additional details about our approach and to support and validate the claims established in the main body of the paper. We have divided the supplementary materials into 6 sections. Sections A and B provide details about the network architectures and training mechanics used in the CoMLSim approach as well as the ML baselines considered in this work. This is followed by additional experimental results in Sections C and D for PDEs considered in the main paper as well as additional canonical PDEs, namely Laplace and Darcy equations. Finally, we expand on the computational performance of CoMLSim in Section E and provide details of reproducibility in Section F.

A CoMLSim network architectures and training mechanics

In this section, we will provide details about the typical network architectures used in CoMLSim followed by the training mechanics. The training portion of the CoMLSim approach corresponds to training of several autoencoders to learn the representations of PDE solutions, conditions, such as geometry and PDE source terms as well as flux conservation. In this work, we mainly employ 2 autoencoder architectures, a CNN-based autoencoder to train the PDE solutions and conditions and a DNN-based autoencoder to train the flux conservation network.

A.1 Solution/Condition Autoencoder

These autoencoders learn to represent solutions and conditions on subdomains into corresponding lower dimensional vectors. CNN-based encoders and decoders are employed here to achieve this compression because subdomains consist of structured data representations. Figure 7 shows the architecture of a typical autoencoder used in this work to learn PDE solutions and conditions. We use separate autoencoders to learn solution and representations of conditions into lower-dimensional latent vectors. In the encoder network, we use a series of convolution and max-pooling layers to extract global features from the solution. Irrespective of the size of the input, the pooling is carried out until a resolution of 4x4 in 2-D and 4x4x4 in 3-D. This is followed by flattening and a series of dense fully-connected layers to compute the latent vector. The decoder network mirrors the encoder network exactly, except that the pooling layer is replaced by an up-sampling layer. A ReLU activation function is applied after every convolution layer. The number of filters in the convolutional layers as well as number of dense layers and the bottleneck size depends on the complexity of the application, non-linearity and sparsity in the input distribution and the size of the subdomains.

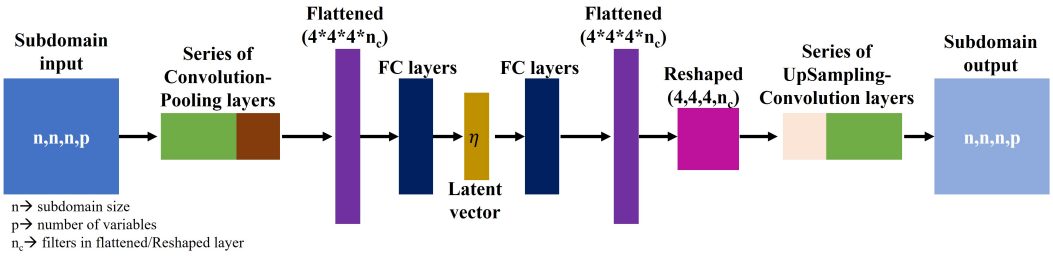

Figure 7: Schematic of Autoencoder network for solution and condition

A.1.1 Input representation of PDE conditions

The experiments considered in this paper have different types of PDE conditions associated with the PDE. For example, Poisson's and Non-Linear Poisson's solutions are influenced by the source term, Reynolds Averaged Navier-Stokes external flow by geometry, Darcy's solutions by diffusivity and Laplace solutions by boundary conditions. Each PDE condition is encoded into a lower-dimensional vector using the autoencoder shown in Figure 7. Generally, diffusivity, source terms and boundary conditions have a spatial representation on the computational domain, which can be directly used to train the autoencoder. On the other hand, an efficient representation of geometry is a topic of on-going research in the ML community. Geometry can be represented using several ways such as

point clouds, voxels, etc. In this work, we use the signed distance field (SDF) to represent geometry. Mathematically, the signed distance at any point within the geometry is defined as the normal distance between that point and closest boundary of a object. More specifically, for $x \in \mathrm{R}^n$ and object(s) $\Omega \subset \mathrm{R}^n$, the signed distance field $\phi(x)$ is defined by:

$$\phi(x) = \begin{cases} +d(x, \partial\Omega) & x \in \Omega \\ -d(x, \partial\Omega) & x \notin \Omega \end{cases}.$$

where, $\phi(x)$ is the signed distance field for $x \in \mathrm{R}^n$ and objects $\Omega \subset \mathrm{R}^n$ [76]. Maleki et al. [65] use the same representation of geometry to successfully demonstrate the encoding of geometries. However, this is a matter of choice and other valid representation can also be used in our approach.

All in all, there are two things to consider when encoding the PDE conditions, 1) the PDE condition is only encoded on subdomains that they influence and the encoding is hard-coded to a vector of zeros for all the other subdomains. For example, in an experiment of flow over a cylinder, the SDF is computed locally on each subdomain. The subdomain that cuts through the cylinder has a non-zero SDF and hence the encoding computed using the trained encoder is non-zero. Other subdomains that don't contain any parts of the cylinder can be encoded with a vector of zeros, 2) An autoencoder for PDE condition is required to be trained only if the set of conditions considered in a given problem have a spatial representation. If the PDE conditions are uniform, the magnitude can simply be considered as an encoding for a given subdomain. For example, if the source term is uniformly described on the computational domain for a given experiment, then the magnitude of the uniform source term can be used as an encoding on each subdomain.

A.2 Flux conservation Autoencoder

These autoencoders learn to represent solution and condition encodings of a collection of neighboring subdomains. Since latent vectors don't have a spatial representation, DNN-based encoder and decoders are employed to compress them. Figure 8 shows a typical DNN-autoencoder used in this work to learn relationships between neighboring subdomains. The input to this autoencoder consist of PDE solution encoding (η) and condition encodings (η_s, η_g, η_b) on neighboring subdomains. In this work, the encoder network consists of typically 3 hidden layers with $n, n/2, n/4$ hidden neurons, respectively. The decoder network is similar to the encoder network but the order of the hidden layers is exactly opposite. The choice of n, depends on the size of the input vector and the complexity of the application. In the figure, we show an example of how the input to the flux conservation network is setup for a 2-D case. Given the PDE solution and condition on a local stencil with 5 subdomains, the encoded representations are calculated using the pre-trained encoders. The encodings of solutions and conditions are concatenated together in a pre-determined order. The same approach works in 3-D, with the difference that the local stencil has 7 subdomains.

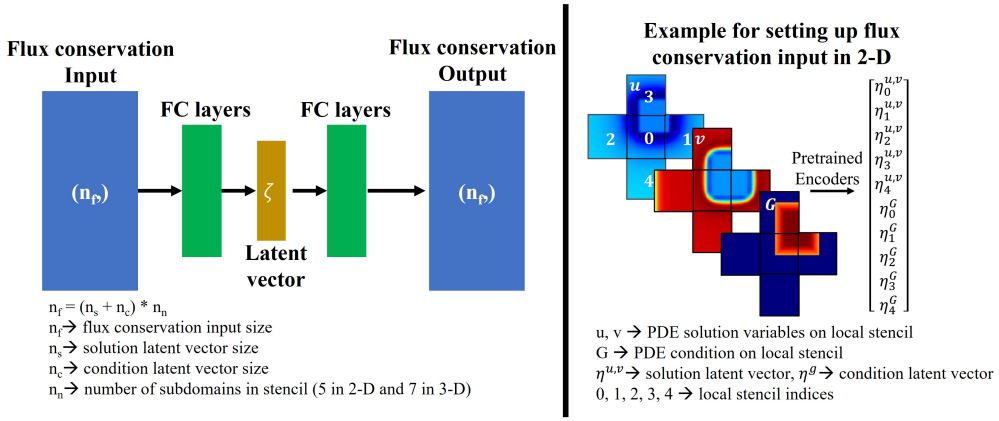

Figure 8: Schematic of Autoencoder network for flux conservation

A.3 Training mechanics

Autoencoders have overfitting tendencies and hence they are required to be trained carefully. Here, we provide general guidelines that may be used to train these autoencoders efficiently. In this work,

we train all the autoencoders until an MSE of $1e^{-6}$ or an MAE of $1e^{-3}$ is achieved on a validation set. More importantly, the compression ratio is selected such that the bottleneck layer has the smallest possible size and yet satisfies the accuracy up to these tolerances. Although, each training run is very fast but may require a decent amount of hyper-parameter tuning to obtain an optimized bottleneck size. Based on the experiments and results we have shown in the main paper, the optimum performance of our approach is observed in a range of bottleneck layer sizes. But, if the bottleneck size is too small or too large, the performance deteriorates. All the autoencoders are trained with the NVIDIA Tesla V-100 GPU using TensorFlow. The autoencoder training is a one-time cost and is reasonably fast.

A.4 Similarities between CoMLSim and Traditional PDE solvers

In this section we expand on the main similarities in between our approach on a traditional Finite Volume or Finite Difference based PDE solvers. There are 3 main similarities, 1) Domain discretization, 2) Flux conservation and 3) Iterative solution algorithm. Here we provide more details about each one.

Consider a set of coupled PDEs with n solution variables. For the sake of notation simplicity, we take $n = 2$, such that $u(x, y, z, t)$ and $v(x, y, z, t)$ are defined on a computational domain Ω with boundary conditions specified on the boundary of the computational domain, Ω_b. It should be noted that extension to more solution variables is trivial. The coupled PDEs are defined as follows:

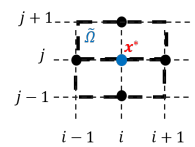

Figure 9: 2D stencil

$$L_1(u, v) - F_1 = 0; L_2(u, v) - F_2 = 0 \tag{10}$$

where, L_1, L_2 denote PDE operators and F_1, F_2 represent PDE source terms. The PDE operators can vary for different PDEs. For example, in a non-linear PDE such as the unsteady, incompressible Navier-Stokes equation the operator, $L = \frac{\partial}{\partial t} + \vec{a}.\vec{\nabla} - \vec{\nabla}.\vec{\nabla}$

1. **Domain discretization:** Traditional PDE solvers solve PDEs given in Eq. 10 by representing solutions variables, u, v, and their spatio-temporal derivatives on a discretized computational domain. The domain is discretized into a finite number of computational elements, using techniques such as Finite Difference Method (FDM), Finite Volume Method (FVM) and Finite Element Method (FEM).

   Similar to traditional PDE solvers, the first step in the CoMLSim is to decompose the computational domain into smaller subdomains. A single subdomain in the CoMLSim is analogous to a computational element in the traditional solver because the CoMLSim predicts PDE solutions directly on local subdomains.

2. **Flux conservation:** Traditional PDE solvers use numerical approximation schemes are used to compute linear and non-linear components of the PDE. For example, in Eq. 10, if $L = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$, representing the 2-D incompressible continuity equation in fluid flows, the spatial derivatives can be approximated on a uniform stencil shown in Figure 9 using a second order Euler approximation shown below in Eq. 11.

   $$L_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \tag{11}$$

   where $i, i+1, i-1, j, j+1, j-1$ are element indices and $\Delta x, \Delta y$ correspond to the size of stencil. These numerical approximations denote flux conservation between neighboring elements. Fluxes represent the flow of information between neighbors and hence, their accurate representation is crucial for information propagation within the domain.

   Similarly, flux conservation in the CoMLSim happens across neighboring subdomains to ensure local consistency and information propagation. The representation of PDE discretization on subdomains is similar to Equation Eq. 11 but the indices i, j represent subdomain indices and the numerical schemes for discretization are now represented by a neural network, Theta, as shown in Eq. 12.

   $$L_{i,j} = \Theta(\eta^u_{i,j}, \eta^u_{i+1,j}, \eta^u_{i-1,j}, \eta^u_{i,j+1}, \eta^u_{i,j-1}, \eta^v_{i,j}, \eta^v_{i+1,j}, \eta^v_{i-1,j}, \eta^v_{i,j+1}, \eta^v_{i,j-1}) \tag{12}$$

   where, η_u, η_v are encodings of solution fields on subdomains and $i, i+1, i-1, j, j+1, j-1$ are subdomain indices.

3. **Iterative solution procedure:** In traditional solvers, the discretized PDEs represent a system of linear or non-linear equations, where the number of such equations equals the number of computational elements. To solve the PDE solutions, the discretized PDE residual is minimized by enforcing flux conservation iteratively using linear and non-linear equation solvers.

   Similar to traditional solvers, the discretized PDEs represent a system of linear or non-linear equations, where the number of such equations equals the number of computational subdomains. To solve this system of equations we employ exactly the same techniques that a traditional would use. For example, in this work we have explored 2 linear iterative solution methods, such as Point Jacobi and Gauss Seidel.

A.5 Self-supervised solution algorithm

Although, this approach requires solution samples to train the autoencoders, we claim in the paper that it is self-supervised in the sense that we don't use these samples to learn an explicit relationship between the input and output distribution. Our training consists of simply training autoencoders and the inference algorithm involves solving a constrained fixed-point iteration to converge to a PDE solution. In the constrained fixed-point iteration, the solution converges to a PDE solution starting from initial random noise. Our solution algorithm is never taught this trajectory of solution convergence but discovers that by itself. Hence, we claim that the solution algorithm at inference is self-supervised.

B Description of baseline network architectures

In the main paper, we compare the performance of CoMLSim with UNet [71], FNO [8], DeepONet [7] and FCNN [77]. Here we describe the network architectures used to train the respective models for all the experiments considered in the main paper.

UNet [71]: The encoder part of the network has 10 convolutional blocks, 2 at each down-sampled size. The input is down-sampled by 4x. The decoder part of the network predicts the output by up-sampling the bottleneck and using skip connections from the encoder network by concatenating the corresponding upsampled output with the corresponding down-sampled output. The decoder part of the network also has 10 convolutional blocks, 2 after each up-sampled size and has 256 stacked channels. The total number of learnable parameters in UNet baseline is equal to 0.471 million in 2-D and 1.412 million in 3-D.

Fourier Neural Operator (FNO) [8]: The FNO model is same as the original implementation in Li et al. [8] but the number of modes are increased to 8 for 3-D experiments to achieve better training loss. The FNO model has 1.188 million parameters in 2-D and 3.689 million parameters in 3-D.

DeepONet [7]: The DeepONet architecture has two branches, a branch net and a trunk net. In all cases, the trunk net has 3 hidden layers with 512 neurons each. The branch is a convolutional neural network, which takes inputs the spatial source term. It is extremely difficult to train the DeepONet with the full resolution of the PDE conditions because of the massive data storage requirements. Hence, for all experiments the PDE conditions are uniformly subsampled to a lower grid resolution given as an input to the branch net. The branch net has 4 DownSample blocks and 10 convolutional blocks, 2 at each down-sampled size. Additionally, the DeepONet is extremely sensitive to the sampling strategy adopted in the training data. The total number of learnable parameters is equal to 1.353 million. The subsampling of PDE conditions and the random sampling used in this work may have affected the testing accuracy of DeepONet.

Fully Convolutional Neural Network (FCNN) [77]: The FCNN model is similar to the original implementation in Zhu and Zabaras [77] but the number of convolution filters and downsampling layers are tuned to accommodate the high-resolutions and non-linearity in different use cases. The FCNN model has 0.189 million parameters in 2-D and 0.578 million parameters in 3-D.

C Experiments results and details from main paper

We demonstrated the CoMLSim for 4 experiments in the main paper. Here we provide more details about the CoMLSim setup as well as additional results and discussions for each experiment. The

different experiments presented in this work are a good mix of pure research and engineering problems with varying levels of non-linear complexity, input distributions, PDEs, solution variable coupling, spatial dimension etc. It must be noted that CoMLSim as well as the baselines are trained with reasonably training samples. The baseline performance can have different outcomes with increasing the size of the training data.

C.1 2-D Linear Poisson's equation

The Poisson's equation is shown below in Eq. 13.

$$\nabla^2 u = f \tag{13}$$

where, u is the solution variable and f is the source term. In this experiment, the source term is sampled from a Gaussian mixture model, where the number of Gaussians is randomly chosen between 1 and 30 and each Gaussian has a randomly specified mean and standard deviation. The Gaussian mixture model is described below in Eq. 13. The computational domain is 2D and is discretized with a highly-resolved grid of resolution 1024x1024. The high-resolution grid is required in this case to capture the local effects of the source term distribution.

$$\sum_{j=0}^{1024}\sum_{j=0}^{1024} f_{i,j} = \sum_{j=0}^{1024}\sum_{j=0}^{1024}\sum_{k=0}^{30} A_k \exp\left(-\frac{x-\mu_{x,k}}{\sigma_{x,k}} - \frac{y-\mu_{y,k}}{\sigma_{y,k}}\right) \tag{14}$$

where, x, y correspond to the grid coordinates. A_k randomly assumes either 0 or 1 to vary the number of active Gaussians in the model. μ_x, μ_y and σ_x, σ_y are the mean and standard deviations of Gaussians in x and y directions, respectively. The means vary randomly between 0 and 1, while the standard deviations are varied between 0.001 and 0.01. The smaller magnitude of standard deviation results in hot spots that can only be captured on highly-resolved grids.

C.1.1 Training

256 solutions are generated for random Gaussian mixtures using Ansys Fluent and used to train the different components of CoMLSim. The computational domain of 1024x1024 resolution is divided into 256 subdomains each of resolution 64x64. The solution and source terms are compressed into latent vectors of size 11, respectively. The flux conservation autoencoder has a bottleneck layer of size 35.

C.1.2 Testing

100 more solutions are generated for random Gaussian mixtures using Ansys Fluent. Due to the high-dimensionality of the source term description, the testing set has no overlap with the training set. The convergence tolerance of the CoMLSim solution algorithm is set to $1e^{-8}$ and the solution method is Point Jacobi. Each solution converges in about 2 seconds and requires about 150 iterations on an average.

C.1.3 Comparisons with Ansys Fluent for in-distribution testing

The CoMLSim predictions for a selected unseen testing samples are compared with Ansys Fluent in Fig. 10. It may be observed that the contour comparisons agree well with Fluent solutions. Overall, the mean absolute error over 100 testing samples is 0.011.

C.1.4 Comparisons with Ansys Fluent for out-distribution testing

In this section, we demonstrate the generalization capability of CoMLSim as compared to Ansys Fluent and all the baselines. We have not presented results from DeepONet because it is does not perform as well as the other baselines. We consider 3 experiments outlined below.

1. **Higher number of Gaussians:** It may be observed from Eq. 14 that the maximum number of Gaussians allowed is set to 30 and this number is sampled from a uniform distribution. In this experiment, we fix the number of Gaussians to 30, 40, 50 and 60 and generate 10

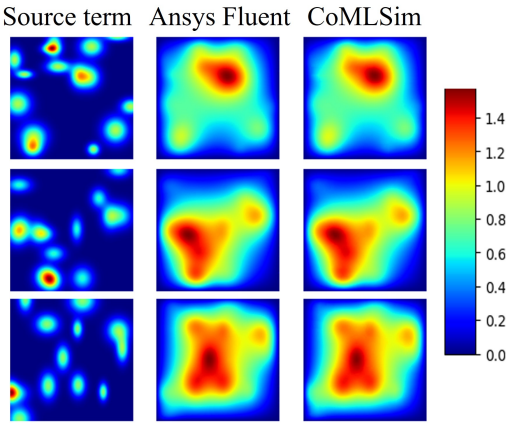

Figure 10: CoMLSim vs Ansys Fluent for linear Poisson's equation with unseen in-distribution source terms

solutions for each case using Ansys Fluent. As the number of Gaussians increase the source term distribution moves further away from the distribution in Eq. 14 used in training.

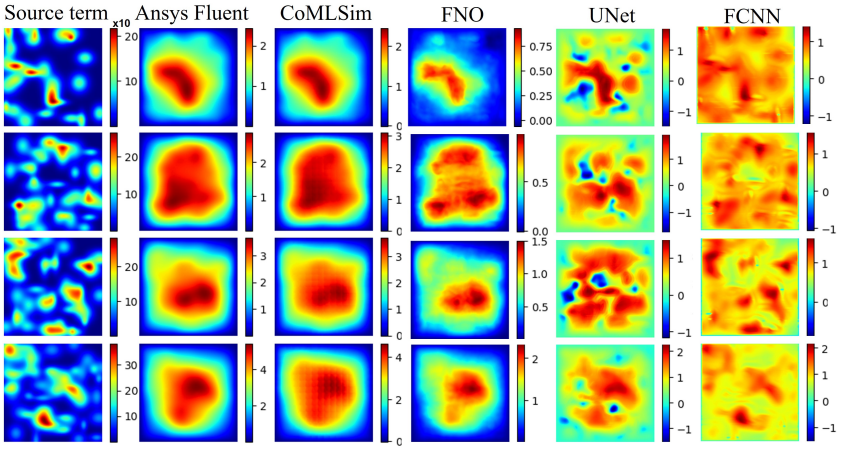

Figure 11: Comparison for linear Poisson's equation on out-of-distribution source terms. Number of Gaussians vary from 30 to 60 from top to bottom.

It may be observed from the contour plots in Figure 11, that the CoMLSim approach consistently beats all the ML baselines. As the number of Gaussians increases, the total source term applied on the computational domain increases proportionally. As a result, the magnitude of the solution is also higher and this can be observed in the color map scale in Figure 11. The CoMLSim captures the both the spatial solution pattern as well as the magnitude correctly in comparison to all ML-baselines. Amongst the ML baselines, FNO performs the best.

2. **Completely different source distribution:** In this experiment, we sample the source term from a completely different distribution. The source term distribution is discontinuous, where the computational domain is divided into either 8^2 or 16^2 tiles and a uniform source term is specified on each tile such that the total source is conserved. The resulting discontinuity in source terms between neighboring tiles makes it challenging to calculate the solution, even with traditional PDE solvers. We generate 10 solutions each with 8^2 and 16^2 tiles using Ansys Fluent. This source term distribution is significantly different from the Gaussian distribution that was initially considered for training.

   It may be observed from the contour plots in Figure 12, that the ML baselines don't predict a reasonable solution. On the other hand, the CoMLSim predicts solutions with higher accuracy. This can be attributed to the local and latent space learning strategies adopted which enables it to accurately learn and predict the local physics of the PDE. It may also be

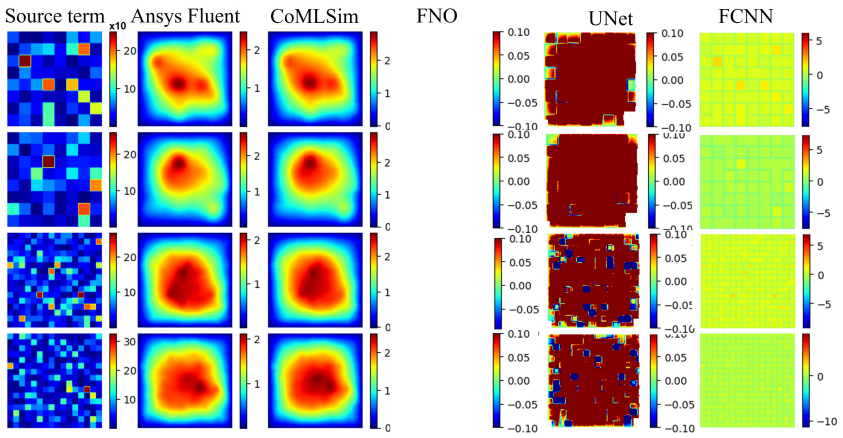

Figure 12: Comparison for linear Poisson's equation on out-of-distribution discontinuous source terms

observed that FNO predicts NaN values because the Fourier transform of the discontinuous source term in the first layer of FNO results in NaN.

3. **Bigger domain with mesh size of 2048x2048:**

   In the main paper, we showed how CoMLSim is designed to scale to bigger physical domains with larger mesh sizes. Here, we present comparisons at a mesh size of 2048^2 between CoMLSim, Ansys Fluent and the ML baselines for the first 4 samples in the testing set. It

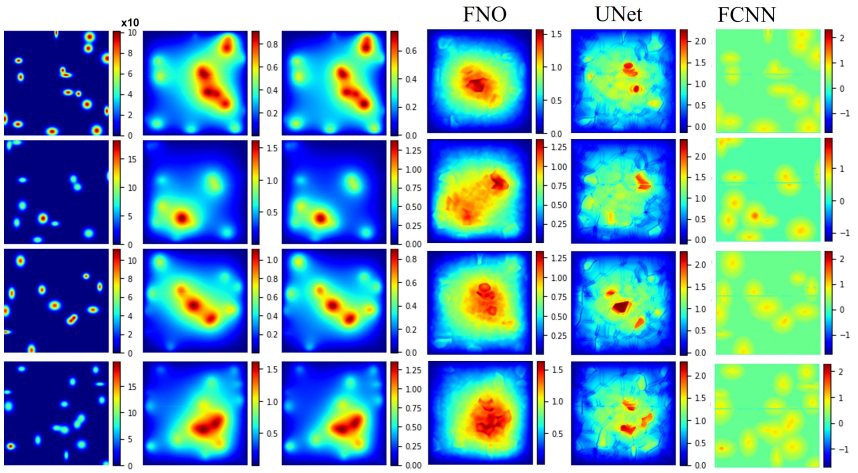

Figure 13: Comparison for linear Poisson's equation for a bigger domain with mesh size of 2048x2048

may be observed from Figure 14 that the CoMLSim outperforms all baselines and matches well with Ansys Fluent.

C.1.5 Additional Ablation Studies

In this section we carry out additional ablation studies to further understand the different aspects of the CoMLSim approach. The CoMLSim has 3 main features, 1) Local learning, 2) Latent space representations and 3) iterative inferencing. The ML components in our approach, such as the solution, condition and flux conservation autoencoders are designed to support these features. In this section, we present experiments to understand the impact of each of these features on the performance of our approach.

1) Importance of local learning:

In this experiment we setup the CoMLSim with a single subdomain. The solution and source term autoencoders, in this case, are trained on the entire computational domain (1024x1024). Since

there are no neighbors, the flux conservation autoencoder is trained on a 1-subdomain stencil. The CoMLSim approach is tested on 250 samples. The mean absolute error obtained is equal to 0.072. The cases reported in the paper in Section 4.3 with 32, 64 and 128 subdomains have mean absolute errors of 0.015, 0.011 and 0.029, respectively. In comparison to Table 1 from the paper, it may be observed that the errors from the single subdomain CoMLSim approach are similar to the ML-baselines. On the other hand, the computational time required by the single subdomain instance is about 3x more expensive than multi-subdomain instance of CoMLSim. This is due to large latent vector sizes obtained from encoding solutions on highly resolved computational domains (1024x1024) as opposed to using subdomains (64x64). Moreover, single subdomain CoMLSim instance cannot scale to bigger computational domains with larger meshes.

2) Importance of latent space representation:

In this experiment, we set the encoder and decoder of solution and condition autoencoders to identity. By doing this we show that as we reduce to a standard domain decomposition, the accuracy of the approach is maintained but the computational time significantly increases. As stated earlier, we use a subdomain resolution of 64x64 to discretize the domain of 1024x1024 resolution into 256 smaller subdomains. With identity encoders and decoders, the latent size of solutions and source terms on each subdomain is equal to 4096, which is the number of computational elements in a subdomain. As a result, the flux conservation autoencoder has an input of (4096 + 4096) * 5 = 40960, where 5 is the number of surrounding neighbors. Training a fully connected neural network with 40960 input features requires networks with billions of parameters and computationally prohibitive GPU memory. Due to these reasons, we reduce the subdomain size to 16x16 to generate a comparison for accuracy. The flux conservation compression ratio (bottleneck size / input size) is consistent with the experiment in the paper. The accuracy observed on 250 testing samples is around 0.017 and is consistent with the results reported Table 1. However, the computational time required in this case is about 2-3x larger than the results reported in Section E.

3) Impact of latent vector size:

In this experiment we evaluate the impact of solution and source term latent sizes with accuracy and computational time of CoMLSim approach averaging over 250 testing samples. It may be observed from the table 2 that the accuracy decreases as we increase the latent sizes. This is due to overfitting of the autoencoder, which makes the latent vector less pronounced. The computational iterations also increase. This observation is consistent with the experiment in Section 4.4 of the main paper where we vary the size of the latent size in flux conservation autoencoder.

It is important to note that the result of this experiment cannot be compared to standard domain decomposition. The solution and condition autoencoders in the case of standard domain decomposition were identity and resemble error-free encoding. In this case, autoencoders are overfit for larger latent sizes and substantially diminish the prominence of the latent vector, thereby resulting in bad performance.

Table 2: Solution/Condition Encoding size vs accuracy and number of convergence iterations

Encoding size	Mean Absolute Error	Avg. Iterations
7	0.047	75
11	0.017	150
32	0.029	250
64	0.1	600
128	0.21	700
256	0.29	800

C.2 2-D Non-linear coupled Poisson's equation

The coupled non-linear Poisson's equation is shown below in Eq. 15.

$$\left.\begin{aligned} \nabla^2 u &= f - u^2 \\ \nabla^2 v &= \frac{1}{u^2 + \epsilon} - v^2 \end{aligned}\right\} \tag{15}$$

where, u, v are the solution variables and f is the source term. The source term is similar to the experiment in Section C.1 except that the x, y standard deviations are varied between 0.001 and 0.05.

As opposed to linear Poisson's, in this problem the complexities result from the coupling of solution variables as well as the source term distribution. A closer look at the PDE in Eq. 15 shows us that the variable v is implicitly coupled with the source term and this relationship is challenging to discover for ML methods.

C.2.1 Training

256 solutions are generated for random Gaussian mixtures using Ansys Fluent and used to train the different components of CoMLSim. The computational domain is divided into 256 subdomains each of resolution 64x64. The solution and source terms are compressed into latent vectors of size 11, respectively. The flux conservation autoencoder has a bottleneck layer of size 35.

C.2.2 Testing

100 more solutions are generated for random Gaussian mixtures using Ansys Fluent. The convergence tolerance of the CoMLSim solution algorithm is set to $1e^{-8}$ and the solution method is Point Jacobi. Each solution converges in about 2 seconds and requires about 150 iterations on an average.

C.2.3 Comparisons with Ansys Fluent for in-distribution testing

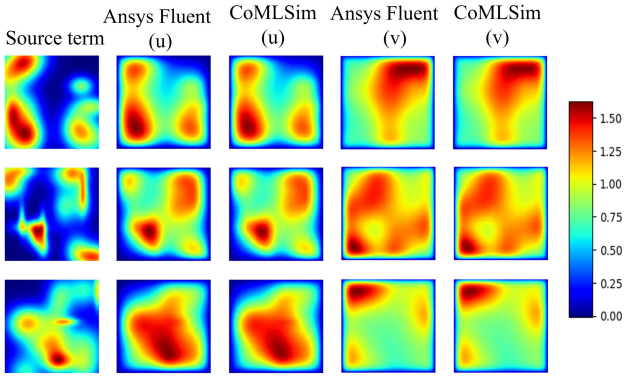

Figure 14: CoMLSim vs Ansys Fluent for non-linear Poisson's equation for in-distribution unseen source terms

The CoMLSim predictions for a selected unseen testing samples are compared with Ansys Fluent in Fig. 14. It may be observed that the contour comparisons agree well with Fluent solutions. Overall, the mean absolute error over 100 testing samples is 0.0053. The mean absolute error is smaller than the linear Poisson experiment because of the smoothness introduced in the power map by increasing the Gaussian mixture standard deviation. Comparisons with other ML baselines are provided in the main paper.

C.2.4 Comparisons with Ansys Fluent for out-distribution testing

In this section, we demonstrate the generalization capability of CoMLSim as compared to Ansys Fluent and all the baselines. Similar to the generalizability test with linear Poisson's, we fix the number of Gaussians to 30, 40, 50 and 60 and generate 10 solutions for each case using Ansys Fluent. As the number of Gaussians increase the source term distribution moves further away from the distribution in Eq. 14 used in training.

It may be observed from Figures 15 and 16 that the CoMLSim outperforms all the ML baselines and matches well with Ansys Fluent for both solution variables. The out-of-distribution source term has a much stronger impact on the coupled solution variable as compared to linear Poisson's experiment. It may be observed from the figures that the ML-baselines predict reasonably for solution variable u, but the errors are much higher for the coupled variable, v.

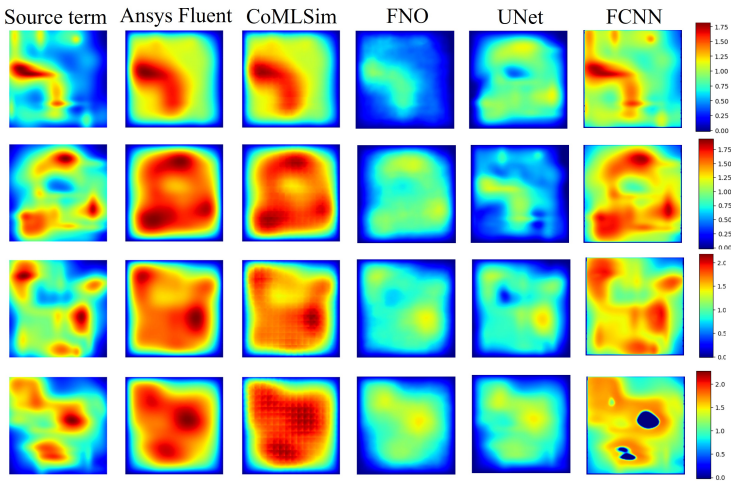

Figure 15: Comparison for variable u of non-linear Poisson's equation for out-of-distribution source terms. The number of Gaussians vary from 30 to 60 from top to bottom.

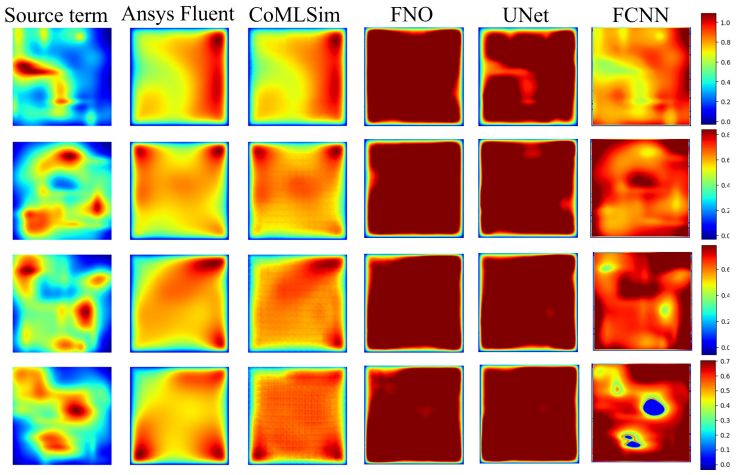

Figure 16: Comparison for variable v of non-linear Poisson's equation for out-of-distribution source terms. The number of Gaussians vary from 30 to 60 from top to bottom.

C.3 3-D Reynolds Averaged Navier-Stokes external flow

In this experiment, we apply the CoMLSim approach to model high Reynolds number turbulent flow around arbitrary geometry shapes built for primitive objects such as Cylinder, Cuboid, NURBS etc and their combinations. The governing PDEs for this case are shown in Eq.17

$$\left.\begin{aligned}
&\textbf{Continuity:} && \nabla.\mathbf{v} = 0 \\
&\textbf{Momentum:} && (\mathbf{v}.\nabla)\mathbf{v} + \nabla\mathbf{p} - \nabla.\left(\nu_t \nabla\mathbf{v}\right) = 0 \\
&\textbf{TKE:} && (\mathbf{v}.\nabla)\mathbf{k} - \nabla.\left(\frac{\nu_t}{\sigma_k}\nabla\mathbf{k}\right) - 2\nu_t E.E + \mathbf{e} = 0 \\
&\textbf{TDR:} && (\mathbf{v}.\nabla)\mathbf{e} - \nabla.\left(\frac{\nu_t}{\sigma_e}\nabla\mathbf{e}\right) - C_{1e}\frac{\mathbf{e}}{\mathbf{k}}2\nu_t E.E + C_{2e}\frac{\mathbf{e}^2}{\mathbf{k}} = 0
\end{aligned}\right\} \quad (16)$$

where, k is Turbulent Kinetic Energy (TKE), e is Turbulent Dissipation Rate (TDR), $v = (u_x, u_y, u_z)$ is the velocity vector, p is pressure, ν_t is the kinematic turbulent viscosity, $C_{1e}, C_{2e}, \sigma_k, \sigma_e$ are empirical constants. More details related to this model and approach can be found in Alfonsi [78].

Table 3: Mean Abs. Errors normalized by inlet velocity (40 m/s) in X-Velocity and Pressure

	X-Velocity	Pressure
CoMLSim	0.02025	0.088
UNet	0.0435	1.178
FNO	0.0315	0.5502
FCNN	0.1395	1.6685

Modeling of Reynolds averaged turbulence presents an additional challenge because it involves 2 additional solution variables which are strongly coupled and influence the velocity and pressure of the flow.

The use case consists of a 3-D channel flow over arbitrarily shaped objects with a computational domain that has a grid resolution of 304x64x64. The object is placed on the bottom surface of the channel to simulate conditions of automobiles moving on a road. The domain has a velocity inlet specified at 40m/s and a zero pressure outlet boundary condition, while the rest of the surfaces are walls with no-slip conditions. The high resolution grid is required because the large flow Reynolds number, of the order of 10000, results in large velocity gradients which can affect the entire flow field. In this experiment, we vary only the geometric features of the object but in future we would like to develop an external flow solver that encompasses both geometry and Reynolds number variations.

C.3.1 Training

150 solutions are generated using Ansys Fluent for geometries corresponding to 5 primitive objects such as cylinder, cuboid, trapezoid, NURB and wedge and their random rotations along all axes as well as their combinations. The computational domain of 304x64x64 is divided into smaller subdomains each with a resolution 8x8x8. The solutions and geometry are encoded to a latent vectors of size 64 and 32, respectively. The geometry is represented with a signed distance field in each subdomain. The flux conservation network has a bottleneck layer of size 64.

C.3.2 Testing

50 additional solutions are generated using Ansys Fluent for other geometries sampled from the same distribution. These geometries are not a part of of the training set. The convergence tolerance of the CoMLSim solution algorithm is set to $1e^{-8}$ and the solution method is Gauss Seidel. Each solution converges in about 35 seconds and requires about 75 iterations on an average.

C.3.3 Comparison with Ansys Fluent for geometries outside the training set

It may be observed from Figures 17 and 18 that the CoMLSim predictions match well with Ansys Fluent for both x-velocity and pressure solutions. The geometries considered in this experiment were not a part of the training set and can be considered out-of-distribution for CoMLSim and other ML baselines. The different features of the flow field are captured reasonably well. As the flow first hits the object it severely decelerates causing smaller velocities and higher pressures. This is known as the stagnation point. Beyond this point, the flow travels along the surface of the object and eventually separates from the object leaving a recirculating wake on the downstream of the object. The point of separation is dependent on the geometry of the object and the flow speed, and determines the pattern of the recirculating wake. As the flow separates, it begins to accelerate resulting in a drop in pressure. It may be observed from the figure that all of these flow features are captured accurately by CoMLSim for the objects considered in this experiment. On the other hand, the other ML baselines do not capture the flow physics accurately. The error comparisons of CoMLSim and other ML baselines in comparison to Ansys Fluent are reported in the table 3. It may be observed that CoMLSim outperforms the baselines.

Amongst the baselines, FNO performs the best and this is evident from the contours in Figures 17 and 18 as well as the errors reported in table 3. On the other hand, FCNN and UNet are severely affected by the sparsity of the PDE solutions observed in all the flow variables and hence don't perform well. We would like to clarify that all the ML baselines were trained to the best of our ability. We provide loss curves in Figure 19 for the benefit of the reader to show that each baseline is trained until the training and validation losses plateau. Additionally, it may also be observed the

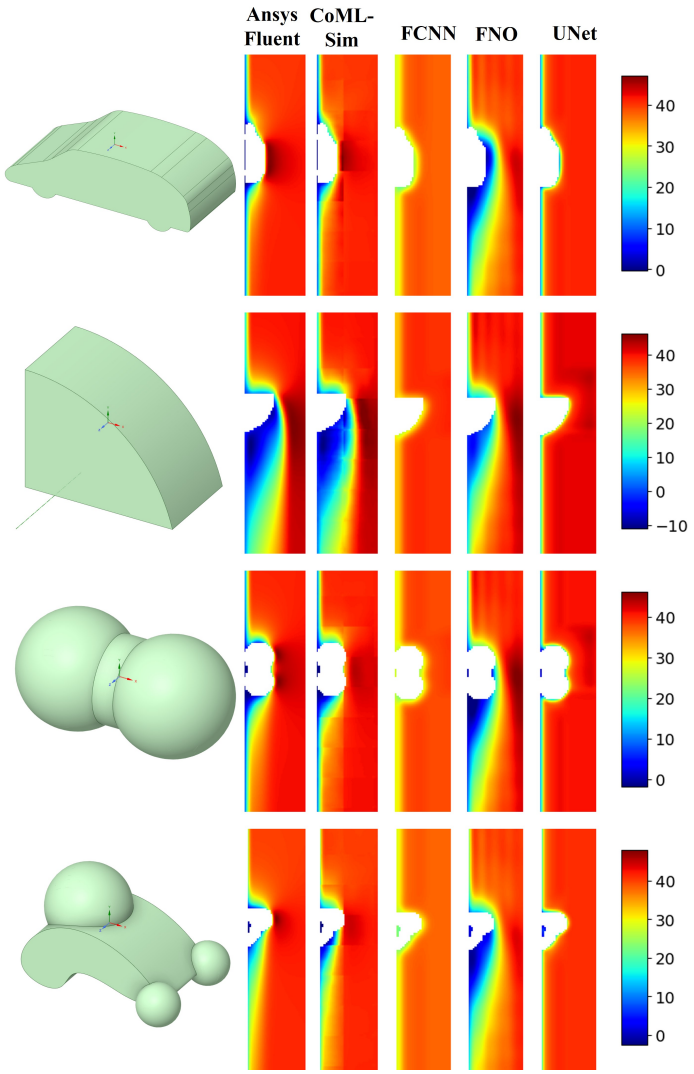

Figure 17: X-velocity contour comparisons (Figures rotated to fit on page and zoomed in on the object)

loss drops by 3 orders of magnitude for each baseline. Moreover, as described in Section B, the networks have a sufficient number of parameters to train efficiently. Finally, traditional ML methods, such as the ones considered in this paper, are data-hungry and improve results as more data becomes available. However, in this experiment, we trained all the models with only 150 solution samples on highly-resolved grids and this reflects in the poor solution accuracy of these methods. Increasing the depth of the networks and number of parameters to 5.38 million for UNet and 2.33 million for FCNN did not improve the results.

C.3.4 Extension to larger physical domains and complex geometries

In this section, we demonstrate the scalability of the CoMLSim approach to bigger physical domains with substantially higher mesh sizes. The results presented previously are carried out on physical domains with grid resolutions of 304x64x64. In this experiment, we increase the grid size to 562x128x128 by increasing the physical size of the domain as well as the geometry. Additionally, we use an unseen test geometry of a car model to perform this evaluation.

Figure 20 shows the streamlines and vector plots of fluid flow near the surface of the car. When compared with Ansys Fluent we observe reasonable mean absolute errors of 0.039 for velocity and 0.106 for pressure. Moreover, it is important to note that the performance and scalability of our

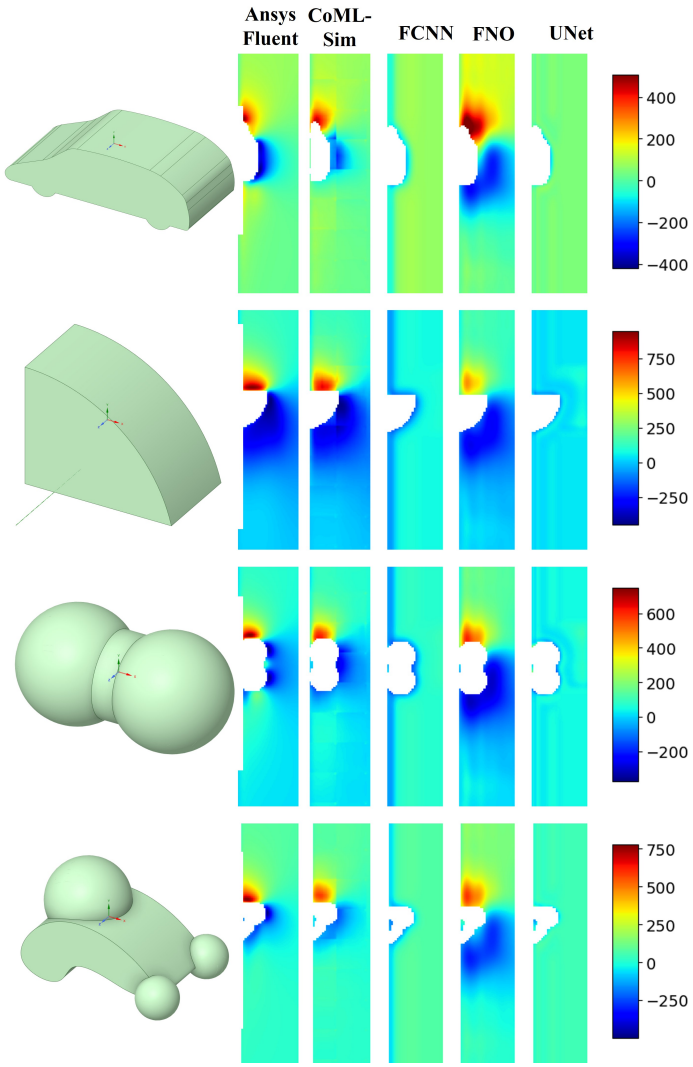

Figure 18: Pressure contour comparisons (Figures rotated to fit on page and zoomed in on the object)

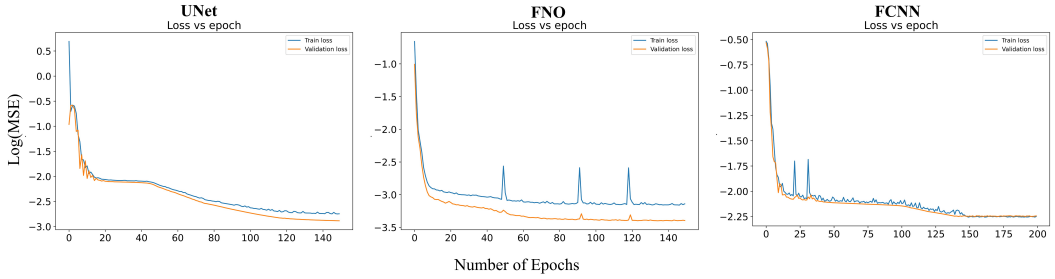

Figure 19: Training loss curves for all the ML baselines

approach to larger meshes is not affected by the 3-D spatial representation in this use case. Most ML baselines considered in this work run out of computational memory when evaluating on this mesh.

C.4 Industrial usecase: 3-D electronic chip cooling with natural convection

In this section, we demonstrate the CoMLSim approach for solving the complicated industrial use case of electronic chip cooling with a wide range of applications. The main purpose of presenting this experiment is to show the ability of our approach can be used for solving real industrial application with similar accuracy's as traditional PDE solvers.

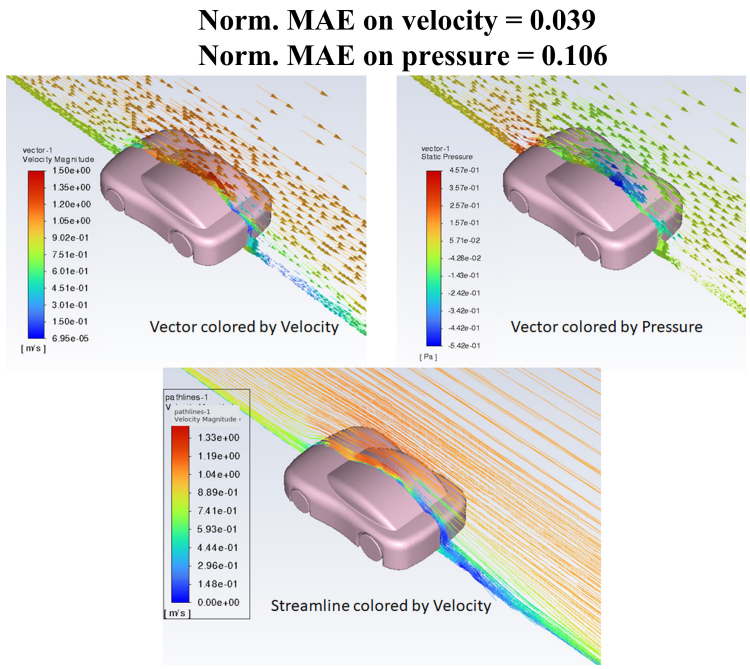

Figure 20: CoMLSim evaluation on bigger domains and unseen car model

In this case the domain consists of a chip, which is sandwiched in between an insulated mold. The chip-mold assembly is held by a PCB and the entire geometry is placed inside a fluid domain. The geometry and case setup of the electronic chip cooling case can be observed in Fig. 21. The chip is subjected to electric heating and the uncertainty in this process results in random spatial distribution of heat sources on the on the surface of the chip. The power map distribution in this case sampled from the same Gaussian mixture model described in Eq. 14, but the total number of Gaussians is limited to 15.

The physics in this problem is natural convection cooling where the power source is responsible for generating heat on the chip, resulting in an increase in chip temperature. The rising temperatures get diffused in to the fluid domain and increase the temperature of air. The air temperature induces velocity which in turn tries to cool the chip. At equilibrium, there is a balance between the chip temperature and velocity generated and both of these quantities reach a steady state. The objective of this problem is to solve for this steady state condition for an arbitrary power source sampled from a Gaussian mixture model distribution, which is extremely high dimensional. The governing PDEs that represent this problem are shown in Eqs. 17.

$$\left.\begin{aligned}
&\textbf{Continuity:} && \nabla.\mathbf{v} = 0 \\
&\textbf{Momentum:} && (\mathbf{v}.\nabla)\mathbf{v} + \nabla\mathbf{p} - \frac{1}{Re}\nabla^2\mathbf{v} + \frac{1}{\beta}\vec{g}\mathbf{T} = 0 \\
&\textbf{Heat (Solid):} && \nabla.(\alpha\nabla\mathbf{T}) - P = 0 \\
&\textbf{Energy (Fluid):} && (\mathbf{v}.\nabla)\mathbf{T} - \nabla.(\alpha\nabla\mathbf{T}) = 0
\end{aligned}\right\} \tag{17}$$

where, $v = u_x, u_y, u_z$ is the velocity field in x, y, z, p is pressure, T is temperature, Re and α are flow and thermal properties, P is the heat source term, $\frac{1}{\beta}\vec{g}T$ is the buoyancy term. P is the spatially varying power source applied on the chip center. The main challenges are in capturing the two-way coupling of velocity and temperature and generalizing over arbitrary spatial distribution of power. The coupled PDEs with 5 solution variables, $v = u_x, u_y, u_z, P, T$ are solved on a fluid and solid domain with loose coupling at the boundaries. The fluid domain is discretized with 256^3 elements in the domain and the solid domain (chip) is modeled as a 2-D domain with 64^2 elements as it is very thin in the third spatial dimension.

C.4.1 Training

The data to train the autoencoders in the CoMLSim approach is generated using Ansys Fluent and corresponds to 300 PDE solutions. The computational domain is divided into 4096 subdomains, each with 16^3 computational elements. The solution autoencoders for the 5 solution variables are trained to establish lower dimensional latent vectors with size 29 on the subdomain level.

C.4.2 Testing

100 more solutions are generated for random Gaussian mixtures using Ansys Fluent. The convergence tolerance of the CoMLSim solution algorithm is set to $1e^{-8}$ and the solution method is Gauss Seidel. Each solution converges in about 42 seconds on an average.

C.4.3 Comparison with Ansys Fluent

Contour plots across different metrics:

The CoMLSim predictions for 2 randomly selected unseen testing samples are compared with Ansys Fluent in Figs. 22 and 23. It may be observed that the contour comparisons agree well with Fluent solutions and the mean absolute errors are of the order of $1e^{-2}$ for temperature and $1e^{-3}$ for velocity. Similar results are observed for other power maps from the unseen testing set.

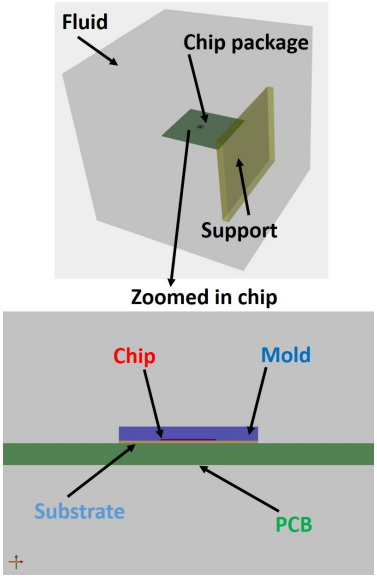

Figure 21: Electronic chip cooling geometry

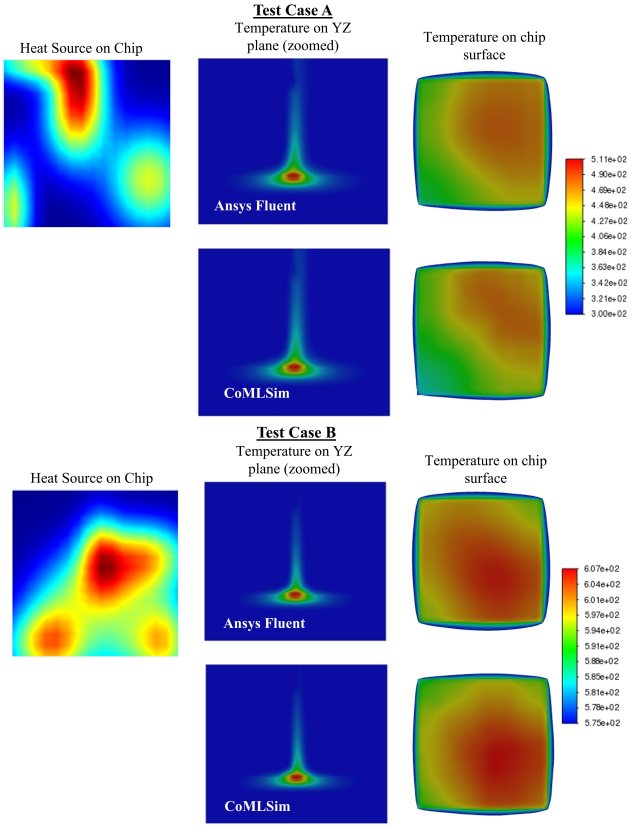

Figure 22: Temperature contour comparisons for unseen powermaps

Line plots across different metrics:

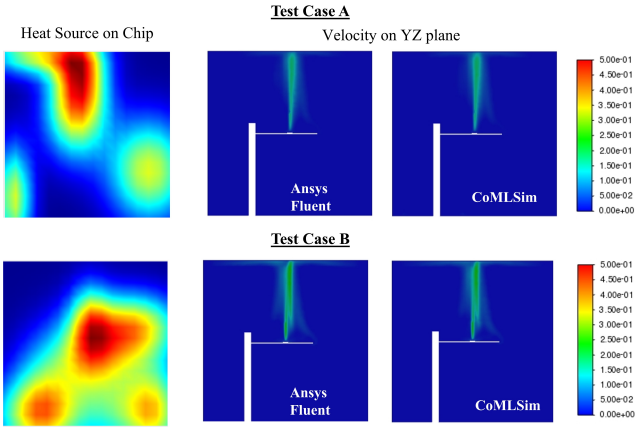

Figure 23: Velocity contour comparisons for unseen powermaps

Figures 24 and 25 shows the line comparisons of temperature and velocity plotted along the chip across all three spatial directions. These metrics provide a more quantitative representation of the accuracy of our approach. It maybe observed that the predictions of CoMLSim agree well with Ansys Fluent for the 2 test cases.

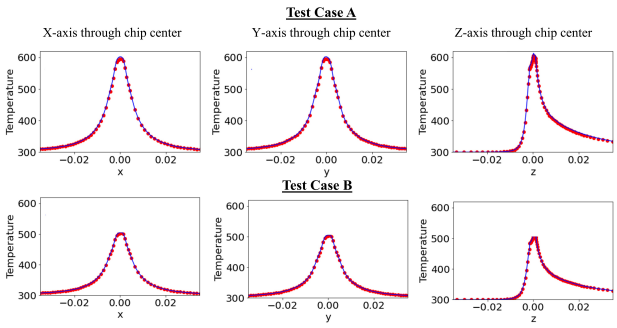

Figure 24: Line comparisons of temperature for unseen powermaps (blue:CoMLSim, red:Ansys Fluent

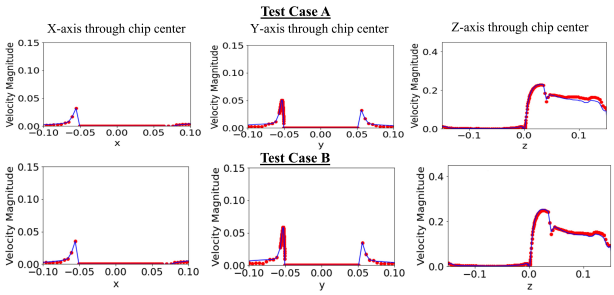

Figure 25: Line comparisons of velocity for unseen powermaps (blue:CoMLSim, red:Ansys Fluent

Comparisons with ML baselines for other quantitative metrics:

Table 4: Testing errors for different metrics

	$L_\infty(T)$	$\epsilon(T_{max})$	$\epsilon(heat\ flux)$
CoMLSim	15.2	5.09	1.26
UNet	95.21	76.2	27.56
FNO	60.83	5.97	12.12
FCNN	192.7	191.38	311.23

Finally, in Table 4 we present 3 different testing metrics for the CoMLSim predictions, 1) Error in maximum temperature in computational domain (hot spots on chip), 2) L_∞ error in temperature

in the computational domain, and 3) Error in heat flux (temperature gradient) on the chip surface. These metrics are more suited for this application and provide a much better measure for evaluating accuracy. All errors are computed relative to the Ansys Fluent solution and are averaged over 100 the unseen testing samples. It may be observed from Table 4 that CoMLSim agrees well with Ansys Fluent on all the comparison metrics and outperforms all the ML baselines.

D Experiments for additional canonical PDEs

D.1 2-D Laplace Equations:

The Laplace equation represents a canonical problem for benchmarking linear solvers. It is shown below in Eq. 18.

$$\nabla^2 \phi = 0 \tag{18}$$

where, u is the solution variable subjected to a Dirichlet boundary condition, $\phi(\vec{x_b}) = f_b$ or a Neumann boundary condition, $\frac{\partial \phi}{\partial \vec{x_b}} = f_b$. The boundary conditions (BCs) are sampled randomly and the magnitude of the BC is uniformly sampled between -1.0 and 1.0.

D.1.1 Training

300 solutions are generated for arbitrary BCs using Ansys Fluent on a computational domain with a resolution of 256x256. These solutions are used to train the different components of CoMLSim. The computational domain is divided into 64 subdomains each of resolution 32x32. The solution is compressed into latent vectors of size 7. Since the boundary condition is uniform, it is encoded discretely on each subdomain. The boundary condition encoding is a vector of length 8, such that the first 4 elements represent an index, 0 or 1 indicating Dirichlet or Neumann boundary on the 4 sides of a subdomain. The next 4 elements represent the magnitude of BCs on the respective sides. The flux conservation autoencoder has a bottleneck layer of size 35.

D.1.2 Testing

100 more solutions are generated for arbitrary BCs using Ansys Fluent. The testing set contains completely different boundary conditions than the training set with no overlaps. The convergence tolerance of the CoMLSim solution algorithm is set to $1e^{-8}$ and the solution method is Point Jacobi.

D.1.3 Comparisons with Ansys Fluent

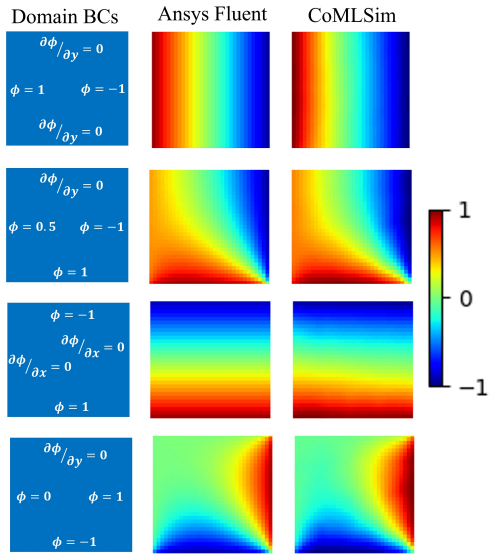

Figure 26: CoMLSim vs Ansys Fluent for Laplace equation over different unseen BCs

The CoMLSim predictions for a selected unseen testing samples are compared with Ansys Fluent in Fig. 26. It may be observed that the contour comparisons agree well with Fluent solutions. Overall, the mean absolute error over 100 testing samples is 0.015.

D.2 2-D Darcy Equations:

The Darcy equation is defined as follows:

$$-\nabla.(\alpha\nabla\phi(\vec{x}) = f \tag{19}$$

It is subjected to different diffusivity conditions, $\alpha(\vec{x}) = f(\vec{x})$ in a 2-D bounded computational domain between (0, 1). The problem setup as well as the data to train the autoencoders is taken from [8]. The main objective for including this case was to demonstrate the CoMLSim approach on a public data set and on relatively smaller mesh resolutions.

We use about 400 samples for training the flux conservation autoencoders, and 200 randomly picked samples for testing from the remaining data. The computational domain in this problem is 2D and has a resolution of 241x241. The domain is divided into 241 subdomains of resolution 21x21 each. The solution on each subdomain is encoded into a latent vector of size 11 and the diffusivity is encoded into size 11. The flux conservation network has a bottleneck size of 16. Since, this is a steady state problem, the CoMLSim iterative solution algorithm is initialized with a solution field equal to zero in all test cases. The iterative algorithm convergence tolerance is set to $1e^{-7}$.

The testing error is computed using Eq. 20 shown below:

$$\epsilon = \frac{L_2(Y_{pred} - Y_{true})}{L_2(Y_{true})} \tag{20}$$

The CoMLSim has a testing error of 0.019 averaged over 200 unseen testing samples as compared to a second-order finite difference solver. The CoMLSim error compares well with FNO [8], which has an error of 0.021. Although the testing errors are very similar in this case, the main advantage of the CoMLSim approach is to accurately model local solution features on highly resolved grids and out-of-distribution generalizability as presented in the different use cases in the main paper as well as the supplementary materials.

E Computational Speed Analysis:

We observe that the CoMLSim approach is about 40-50x faster as compared to commercial steady-state PDE solvers such as Ansys Fluent for the same mesh resolution in all the experiments presented in this work. Both CoMLSim approach and Ansys Fluent are solved on a single Xeon CPU with single precision. We expect our approach to scale to multiple CPUs as traditional PDE solvers but single CPU comparisons are provided here for benchmarking. Moreover, our algorithm is a python language interpreted code, whereas Ansys Fluent is an optimized, C language pre-compiled code. We expect the C/C++ version of our algorithm to further provide independent speedups (not included in current estimates). In Table 5, we show comparison of simulation time between Ansys Fluent and CoMLSim for all the use cases. The results are averaged over 100 testing cases. As expected, the iterative convergence achieved by the CoMLSim approach is substantially faster than Ansys Fluent.

In comparison to the ML baselines, our approach is expected to be slower because it adopts an iterative inferencing approach. But, our approach focuses on substantially improving the solution accuracy, generalizability, robustness and stability at the cost of computational speed-ups. Although our approach is still faster than traditional PDE solvers, but the ratio of is speed-up is smaller than traditional ML methods.

F Guidelines for reproducibility

In this section, we provide the necessary details for training the CoMLSim. As emphasized previously, the CoMLSim training corresponds to training several autoencoders for PDE solutions, conditions, such as geometry, boundary conditions and source terms and for flux conservation. In Figure 27, we present a flow chart of the steps that can be followed to train each of these autoencoders. The

Table 5: Simulation time (in seconds) comparison with Ansys Fluent

Experiment	Num. of Elements	CoMLSim	Ansys Fluent
Laplace	65K	0.21	10
Linear Poissons	1048K	2.75	130
Coupled non linear Poissons	1048K	2.81	540
3D Navier-Stokes flow	1245K	35	1900
3D Electronic Chip Cooling	2097K	42	1600

specific training details and network architectures are described in the previous Section. For a given set of coupled PDEs and depending of the use case, we start with 100-1000 sample solutions on a computational domain with n, m, p computational elements in spatial directions x, y, z, respectively. The solutions are divided into smaller subdomains of resolution, $n/16, m/16, p/16$. Each subdomain can has PDEs solutions and PDE conditions associated with it. The PDE solutions are used as training samples to the PDE solution autoencoder to learn a compressed encoding of all the variables on the subdomain. On the other hand, the autoencoders for PDE conditions can be trained with completely random samples, which may or may not be related to sample solutions. Once the PDE solution and condition autoencoders are trained, neighboring subdomains are grouped together and the solution and PDE condition latent vectors on groups of neighboring subdomains are stacked together. These groups of stacked latent vectors are used for training the flux conservation autoencoder. The trained PDE solution, condition and flux conservation autoencoders combine to form the primary components of the CoMLSim.

The solution algorithm of the CoMLSim has been described in detail in the main body of the paper and may be used for solving for unseen PDE conditions. We are happy to share the source code and the dataset for some experiments if deemed necessary by the reviewers.

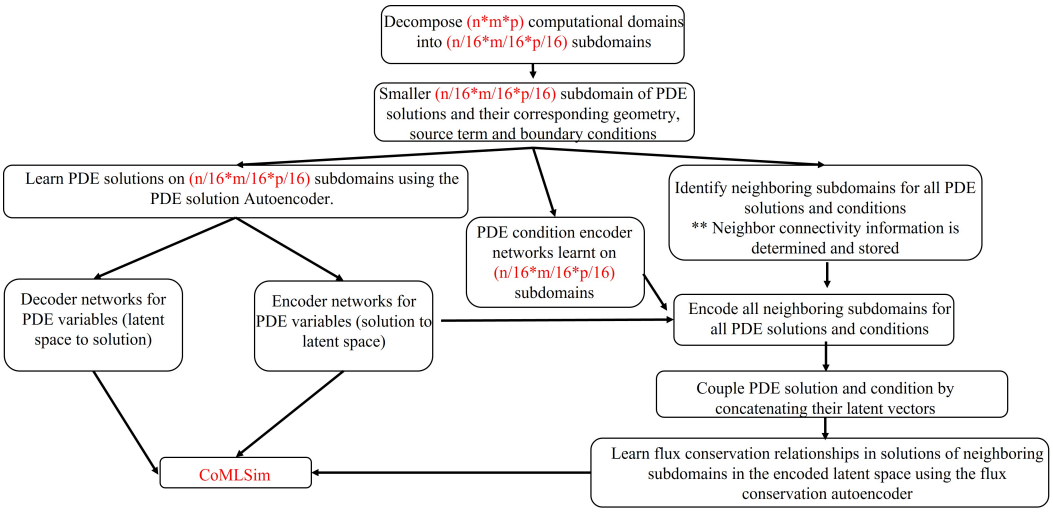

Figure 27: Flow chart for training the CoMLSim approach

F.1 Step-by-step instructions for Poisson's equation

The Poisson's equation is shown in Eq. 13. The solution to this equation is governed by a source term applied on the computational domain. Here we consider this 2D case to explain the training and inference modes of the CoMLSim approach.

F.1.1 CoMLSim training

The data in this use case corresponds to 256 training and testing PDE solutions generated for random source terms defined by Eq. 14 using Ansys Fluent on a 1024x1024 resolution grid. This data is preprocessed such that it is divided into subdomains of size 64x64 before being used for training.

The training portion of CoMLSim corresponds to training the solution autoencoder, source term autoencoder and the flux conservation autoencoder. Training solutions and the corresponding conditions on 64x64 subdomains are used for training the solution and condition autoencoders. The Network architecture and training mechanics described in Figure 7 are used to perform this training. Separate 2D CNN autoencoders are used to train solutions and source terms.

The flux conservation networks require further preprocessing of the data before it is trainable. The trained encoders are used to encode the solutions and conditions on each subdomain and its top, bottom, left and right neighbors. These neighborhood encodings are concatenated. For example, if the solution latent size is 16 and the condition latent size is 16, then the input size for the flux conservation autoencoder is (16+16)*5=160. The flux conservation autoencoder is trained to learn this group of concatenated neighborhood encodings for all the subdomains in the training dataset. For the subdomains on the edges, we simply use zero vectors to pad the missing subdomains. For 3-D cases, we include two additional subdomains, one in front and behind the center subdomain. The network architecture and training mechanics used to perform this training is described in more detail in Figure 8.

The trained autoencoders are used to evaluate the our approach on other unseen source terms.

F.1.2 CoMLSim inference

During the inference time, we evaluate the CoMLSim approach for a user-specified source term. A schematic of the inference algorithm for this case is shown in Figure. 28.

Similar to training, we decompose the domain into same sized smaller subdomains and encode the solution and source terms on all the subdomains using the pretrained encoders. Since the solution is unknown, a constant or randomly initialized solution is assumed on the domain. In each fixed point iteration, we loop over all the subdomains. On each subdomain, we gather the solution and source term latent vectors of the subdomain and its neighbors. The order in which the neighboring latent vectors are concatenated is consistent with the order used during training. The concatenated neighborhood latent vectors are then passed through the flux conservation autoencoder to obtain a new set of solution and source term encodings for each subdomain and its neighbors. Depending on the algorithm being used, Point Jacobi or Gauss Seidel, we modify the solution encoding of the subdomains while maintaining the source term encoding fixed. The modified solution encodings along with the fixed source term encodings are iteratively passed through the flux conservation autoencoder until the L-2 norm of the change in the solution latent vectors on all subdomains drops below a specified tolerance. The constant source term encodings steer the solution encoding towards an equilibrium. Finally, the converged solution encodings are decoded using the trained solution decoder. The decoded solutions are then stitched together to obtain the full domain solution.

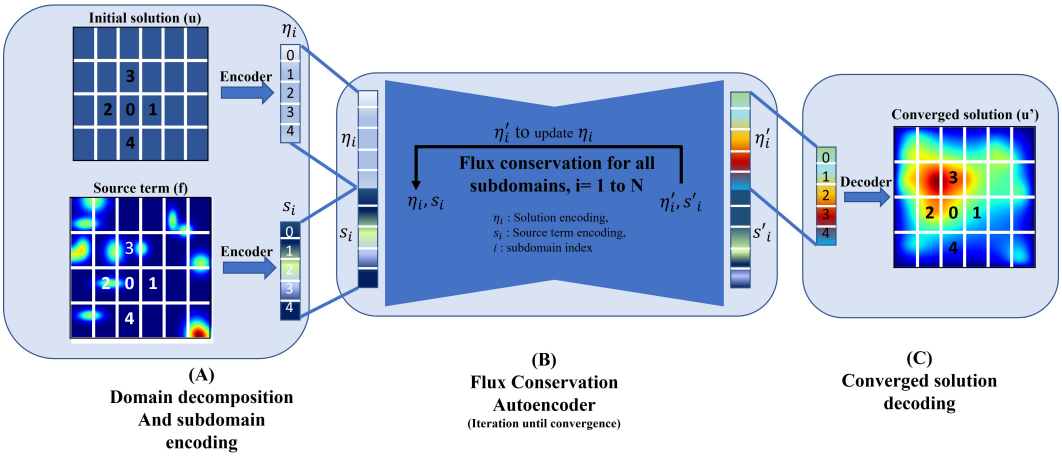

Figure 28: Schematic of CoMLSim inference for Poisson's PDE: The inference process consists of 3 steps. (A) Domain decomposition and encoding: The solution and source term are divided into subdomains. The central subdomain and the neighboring subdomains are encoded individually using the trained autoencoders. η_i represents the encoded initial solutions and s_i the encoded subdomains. The solution and source latent vectors on neighboring subdomains are concatenated together. (B) Flux conservation: The concatenated encodings are passed through the flux conservation autoencoder iteratively until convergence. The source term s_i, part of the encoding is held constant. Only the solution η_i, part is updated. In each flux conservation iteration, the algorithm loops over all the subdomains before moving to the next iteration. (C) The converged solution encoding is then extracted from the complete encoding and decoded per subdomain using the solution decoder trained during the training phase.