

Supplementary Material for: Geometry Aware Operator Transformer as an Efficient and Accurate Neural Surrogate for PDEs on Arbitrary Domains

A Details of Problem Formulation.

Here, we introduce the core concepts of *operator learning* for both time-dependent and time-independent partial differential equations (PDEs). We begin by defining the general forms of PDEs under consideration and then discuss the associated operator-learning tasks.

A.1 General Forms of PDEs

We focus on two broad classes of PDEs: time-dependent and time-independent.

Time-Dependent PDE. Let $D \subset \mathbb{R}^d$ be a d -dimensional spatial domain, and let $(0, T)$ denote the time interval. A general time-dependent PDE (see Main Text Eqn. (2)) can be written as

$$\begin{aligned} \frac{\partial}{\partial t} u(t, x) + \mathcal{D}(c, t, u, \nabla_x u, \nabla_x^2 u, \dots) &= 0, & \forall (t, x) \in (0, T) \times D, \\ \mathcal{B}(u, \nabla_x u, \nabla_x^2 u, \dots) &= u_b, & \forall (t, x) \in (0, T) \times \partial D, \\ u(0, x) &= u_0(x), & \forall x \in D, \end{aligned} \quad (\text{A.1})$$

where

- $u(t, x)$ is the PDE solution in $(0, T) \times D$;
- $c(t, x)$ is a known, possibly spatio-temporal parameter (e.g., a material coefficient or source term);
- \mathcal{D} is a (spatial) differential operator;
- \mathcal{B} is a boundary operator acting on ∂D ;
- $u_b(x)$ are boundary values;
- $u_0(x)$ is the initial condition at $t = 0$.

We assume $u(t, \cdot) \in \mathcal{X} \subset L^p(D; \mathbb{R}^m)$ for some $1 \leq p < \infty$ and integer $m \geq 1$. Likewise, $u_0(x) \in \mathcal{X}^0 \subset \mathcal{X}$ is an element of the initial-condition space, and $c \in \mathcal{Q} \subset L^p(\bar{D}; \mathbb{R}^m)$ is taken from a parameter space.

Time-Independent PDE. A time-independent (steady-state) PDE of the general form (Main Text Eqn. (1)) can be written as

$$\begin{aligned} \mathcal{D}(c, \bar{u}, \nabla_x \bar{u}, \nabla_x^2 \bar{u}, \dots) &= f, & \forall x \in D, \\ \mathcal{B}(\bar{u}, \nabla_x \bar{u}, \nabla_x^2 \bar{u}, \dots) &= u_b, & \forall x \in \partial D, \end{aligned} \quad (\text{A.2})$$

where $\bar{u}(x) \in \mathcal{X}$ and $c(x) \in \mathcal{Q}$ are now independent of t and f is a source term. In certain scenarios, one may view (A.2) as the long-time limit of (A.1), i.e.,

$$\bar{u}(x) = \lim_{t \rightarrow \infty} u(t, x). \quad (\text{A.3})$$

Hence, much of the theory for time-dependent PDEs can be adapted to time-independent problems by recognizing steady-state solutions as limiting cases.

A.2 Solution Operators for PDEs

Let us denote the solution to the time-dependent PDE (A.1) by

$$u(t, \cdot) = \mathcal{S}(a, c, t), \quad (\text{A.4})$$

where $\mathcal{S} : \mathcal{X}^0 \times \mathcal{Q} \times (0, T) \rightarrow \mathcal{X}$ is the *solution operator*, mapping any initial datum $u_0 \in \mathcal{X}^0$ (and parameter functions $c \in \mathcal{Q}$) to the solution $u(t)$ at time t .

655 **Time-Shifted Operator.** In many operator-learning strategies, it is useful to consider a *time-shifted*
 656 *operator* that predicts solutions at a future time from a current snapshot. Specifically, define

$$\mathcal{S}^\dagger : \mathcal{X} \times \mathcal{Q} \times (0, T) \times \mathbb{R}^+ \longrightarrow \mathcal{X},$$

657 such that

$$\mathcal{S}^\dagger(u^t, c^t, t, \tau) = \mathcal{S}^t(u^t, c^t, \tau) = u^{t+\tau}. \quad (\text{A.5})$$

658 Here, $u^t = u(t, \cdot)$ is the solution snapshot at time t , which now serves as an initial condition on the
 659 restricted time interval (t, T) . Likewise, c^t is the corresponding parameter snapshot at time t .

660 **Steady-State Operator.** For the time-independent PDE (A.2), we define

$$\bar{\mathcal{S}} : \mathcal{Q} \longrightarrow \mathcal{X} \quad (\text{A.6})$$

661 to be the analogous solution operator, such that $\bar{u} = \bar{\mathcal{S}}(c)$ solves the boundary-value problem for
 662 any parameter/boundary data c . Although many operator-learning methods primarily focus on the
 663 time-dependent form \mathcal{S} , the same ideas apply to steady-state problems by treating \bar{u} as a limiting
 664 case.

665 **Operator Learning Task (OLT).** A central goal is to approximate these solution operators without
 666 repeatedly resorting to expensive, high-fidelity numerical solvers. Formally, the OLT can be stated as:

667 *Given a data distribution $\mu \in \text{Prob}(\mathcal{X}^0) \times \mathcal{Q}$ for initial/boundary conditions*
 668 *and parameters $c \in \mathcal{Q}$, learn an approximation $\mathcal{S}^* \approx \mathcal{S}$ to the true solution*
 669 *operator \mathcal{S} . That is, for any $a \sim \mu$, we want $\mathcal{S}^*(t, a)$ to closely approximate $u(t)$*
 670 *for all $t \in [0, T]$. For time-independent problems, this goal changes accordingly to*
 671 *learning $\bar{\mathcal{S}}^* \approx \bar{\mathcal{S}}$.*

672 A.3 Discretizations.

673 In practice, we only have access to a *discretized form* of the data as the labelled data is generated
 674 either through experiments/observations or numerical simulations. In both cases, we can only evaluate
 675 the inputs and outputs to the underlying solution operator at discrete points.

676 We start by describing these discretizations for the time-independent PDE (A.2). To this end, fix
 677 the i -th sample and let $D_{\Delta(i)} = \{x_j^{(i)} \in D^{(i)}\}$, for $1 \leq j \leq J^{(i)}$ denote a set of *sampling points*
 678 on the underlying domain $D^{(i)}$. Observe that the underlying domain itself can be an input to the
 679 solution operator $\bar{\mathcal{S}}$ of (A.2). We assume access to the functions $(c^{(i)}(x_j), f^{(i)}(x_j), u^{(i)}(x_j))$ and
 680 the corresponding discretized boundary values. Denoting these discretized inputs and outputs as
 681 $a_{\Delta(i)}^{(i)}$ (where $a = (c, f, u_b)$) and $u_{\Delta(i)}^{(i)}$, respectively, the underlying learning task boils down to
 682 approximating $\bar{\mathcal{S}}_{\# \mu}$ from the discretized data-pairs $(a_{\Delta(i)}^{(i)}, u_{\Delta(i)}^{(i)})$. Note that although the data is
 683 given in a discretized form, we still require that our operator learning algorithm can provide values of
 684 the output function u at any *query point* $x \in D$.

685 For the time-dependent PDE (A.1), in addition to the spatial discretization $D_{\Delta(i)} = \{x_j^{(i)} \in D^{(i)}\}$,
 686 for $1 \leq j \leq J^{(i)}$, we only have access to data at time snapshots $t_n^{(i)} \in [0, T^{(i)}]$. Thus, the data to the
 687 time-dependent operator learning task consists of inputs $(c^{(i)}(x_j), u_0^{(i)}(x_j))$ and outputs $u(x_j^{(i)}, t_n^{(i)})$,
 688 from which the space- and time-continuous solution operator \mathcal{S}_t has to be learned at every query
 689 point $x \in D$ and time point $t \in [0, T]$.

690 Summarizing, for both time-independent and time-dependent PDEs, the operator learning task
 691 amounts to approximating the underlying (space-time) continuous solution operators, given dis-
 692 cretized data-pairs.

693 B Details of GAOT Architecture

694 This appendix provides a detailed explanation of the core components of the GAOT model architecture,
 695 including the choice of latent token grid, the Multiscale Attentional Graph Neural Operator (MAGNO)
 696 used in the encoder and decoder, the geometry embedding mechanisms, and the transformer-based
 697 processor.

698 B.1 Choice of Latent Grid

699 Given the input point cloud, denoted by D_Δ above, the first step in our design is to select a *latent*
700 point cloud, consisting of points at which our spatial tokens are going to be specified. Here, we
701 explore three distinct ways of choosing spatial tokens, each offering different trade-offs in terms
702 of computational cost, geometric coverage, and ease of patching for efficient attention. Figure B.1
703 schematically illustrates these strategies.

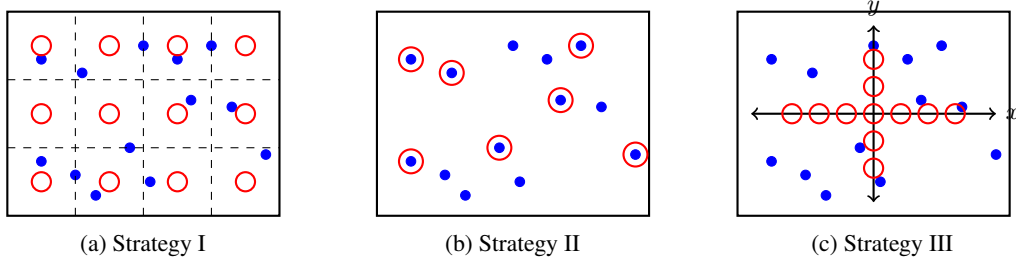


Figure B.1: Schematic illustration of the three tokenization strategies in a 2D domain. Blue points and red circles corresponding to physical grid and latent token grid, respectively. (a) A structured stencil grid overlaying the domain. (b) Downsampled unstructured points used directly as tokens. (c) Projecting the 2D domain onto low-dimensional planes.

704 **Structured Stencil Grid (Strategy I).** In this approach, we overlay a structured grid of tokens
705 across D_Δ . For 2D domains, this may be a uniform mesh of cells; for 3D domains, an analogous
706 dense grid can be used. GINO [29] is a good example for the use of such latent grids.

- 707 • *Advantages.* This grid can be quite fine if needed, ensuring adequate coverage. Moreover,
708 we can group tokens into patches before feeding them into the transformer processor (see
709 below), effectively reducing the token count (number of latent points). Our following
710 experiments in SM Sec. F.2 suggest that patch size has a negligible effect on performance;
711 hence, one can choose large patches to speed up training.
- 712 • *Limitations.* The main drawback is that token count grows exponentially with the dimension;
713 for 3D, the number of grid cells can be prohibitively large. Also, if the input data lie on
714 a low-dimensional manifold embedded in D_Δ , some tokens may remain underutilized.
715 Nonetheless, we find in practice that even empty tokens (those with no neighboring input
716 points) can still contribute to better global encoding and improved convergence.

717 **Downsampled Unstructured Points (Strategy II).** This method directly downsamples the input
718 unstructured point cloud and treat each sampled point as a token, RIGNO [39] and UPT [1] are
719 typical methods based on this strategy. If the data are denser in some regions, naturally more tokens
720 appear there.

- 721 • *Advantages.* This method avoids the pitfalls of having many tokens in empty regions, as
722 might happen if the data indeed lie on a lower-dimensional manifold. By adaptive sampling,
723 it can allocate tokens more efficiently.
- 724 • *Limitations.* Unstructured tokens are harder to patch effectively for attention mechanisms in
725 the processor. In a following experiment, we observed that this strategy can be less effective
726 than Strategy I even when the domain is indeed partially low-dimensional.

727 **Projected Low-Dimensional Grid (Strategy III).** Inspired by certain 3D computer vision mod-
728 els [7, 21], one can project the 3D domain (or higher-dimensional space) into a lower-dimensional
729 representation and then place a structured stencil grid in the reduced coordinates—for example, using
730 triplane embeddings in 3D [8].

- 731 • *Advantages.* Such a projection drastically reduces the token count in 3D, and avoids the
732 purely *low-dimensional manifold* disadvantage of Strategy I. Moreover, one can still apply
733 patching on the structured plane.
- 734 • *Limitations.* Decomposing d -dimensional coordinates into disjoint projections (e.g., splitting
735 x, y, z axes) can introduce additional approximation errors. Some local neighborhood

information is inevitably lost during the projection. This trade-off can degrade the final accuracy compared to direct methods (Strategy I or II).

In this paper, we mainly adopt Strategy I, i.e. a *structured stencil grid*, for all experiments due to its robustness and simplicity. While using a stencil grid indeed creates some empty tokens in low-density regions, we consistently observe fast convergence and strong generalization across various PDE datasets, see the ablation studies in SM Sec. F.2.

B.2 Multiscale Attentional Graph Neural Operator

Both the encoder and decoder in GAOT (see Figure 2 in the main text) employ the proposed Multiscale Attentional Graph Neural Operator (MAGNO). MAGNO is designed to augment classical Graph Neural Operators (GNOs) by incorporating multiscale information processing and attention-based weighting. A traditional GNO constructs a local graph for each query point (or token) by collecting all neighboring nodes within a specified radius, approximating a (kernel) integral operator over this local neighborhood. Below, we first recap the standard single-scale GNO scheme, then extend it to a multiscale version, and finally incorporate attention mechanisms for adaptive weighting.

Recap of Single-Scale Local Integration (GNO Basis) For any point y in the latent space \mathcal{D} (in the encoder) or a query point x in the original domain D_Δ (in the decoder), a GNO layer aims to aggregate information from its neighborhood via a kernel integral. For the encoder, given input data $a(x_j)$ on the original point cloud $D_\Delta = \{x_j\}$, the GNO transforms it into latent features $w_e(y)$. The fundamental GNO computation is given by Eq. (3) from the main text:

$$\tilde{w}_e(y) = \sum_{k=1}^{n_y} \alpha_k K(y, x_k, a(x_k)) \varphi(a(x_k)) \quad (\text{B.1})$$

where the sum is over n_y points x_k in the original point cloud D_Δ such that $|y - x_k| \leq r$. K and φ are MLPs, $a(x_k)$ is the input feature at point x_k , and α_k are given quadrature weights. This form can be seen as a discrete approximation of an integral operator:

$$\int_{B_r(y) \cap D_\Delta} K(y, x', a(x')) \varphi(a(x')) dx' \quad (\text{B.2})$$

where $B_r(y)$ is a ball of radius r centered at y .

Multiscale Neighborhood Construction The single-scale approach, while effective for capturing local interactions within a fixed radius r , may not efficiently perceive multiscale information crucial for many PDE problems. To address this, we introduce multiple radii. As described in the main text, we choose $r_m = s_m r_0$, where r_0 is a base radius and s_m are scale factors ($m = 1, \dots, \bar{m}$). For each scale m , we gather points x_k from the original point cloud D_Δ within the ball $B_{r_m}(y)$ centered at $y \in \mathcal{D}$ (for the encoder) with radius r_m . A GNO-like local integration is then performed for each scale m , as shown in Eq. (4) from the main text:

$$\tilde{w}_e^m(y) = \sum_{k=1}^{n_y^m} \alpha_k^m K^m(y, x_k, a(x_k)) \varphi(a(x_k)) \quad (\text{B.3})$$

Here, n_y^m is the number of neighbors x_k within radius r_m . The MLPs K^m and φ can be scale-specific or share parameters across scales. This paper chooses the shared parameters across all scales.

B.2.1 Attentional Weighting in Local Integration (AGNO)

In the main text, we further propose an attention-based choice for the quadrature weights α_k^m , as given by Eq. (5):

$$\alpha_k^m = \frac{\exp(e_k^m)}{\sum_{k'=1}^{n_y^m} \exp(e_{k'}^m)}, \quad e_k^m = \frac{\langle \mathbf{W}_q^m y, \mathbf{W}_\kappa^m x_k \rangle}{\sqrt{\bar{d}}} \quad (\text{B.4})$$

where $\mathbf{W}_q^m, \mathbf{W}_\kappa^m \in \mathbb{R}^{\bar{d} \times d}$ (assuming original and latent coordinate dimension d , and attention dimension \bar{d}) are learnable query and key matrices. This mechanism allows the model to dynamically assign contribution weights to each neighbor x_k based on the relationship between y and x_k . This forms the final form of our Attentional Graph Neural Operator or AGNO at each scale m .

775 B.2.2 Attentional Fusion of Multiscale Features

776 After computing the AGNO features $\hat{w}_e^m(y)$ for each scale (which are then fused with geometry
 777 embeddings, detailed in Sec. B.3, to form $\hat{w}^m(y)$), we need to integrate this information from
 778 different scales. As described in the main text (Fig. 2 and Eq. (6)), instead of simple summation or
 779 concatenation, we introduce a small MLP ψ_m to learn the relative contribution of each scale to the
 780 final encoded feature $w_e(y)$:

$$w_e(y) = \sum_{m=1}^{\bar{m}} \beta_m(y) \hat{w}^m(y), \quad \beta_m(y) = \frac{\exp(\psi_m(y))}{\sum_{m'=1}^{\bar{m}} \exp(\psi_{m'}(y))} \quad (\text{B.5})$$

781 Here, $\psi_m(y)$ is typically computed based on coordinates of y . $\beta_m(y)$ is the attention weight for the
 782 m -th scale at point y .

783 The final output of the MAGNO encoder, $w_e(y)$, is thus a feature representation that adaptively
 784 weights and fuses multiscale local information with attention mechanisms. The MAGNO in the
 785 decoder follows the exact same structure, with different inputs, outputs, and operating objects, as
 786 described in the Main Text.

787 B.3 Geometry Embeddings

788 While the Multiscale Attentional GNO already leverages geometric structure via local neighborhoods,
 789 one often needs to incorporate more explicit shape or domain information in practical PDE scenarios.
 790 For instance, when the geometry of the domain itself (e.g., the shape of an airfoil) plays a critical
 791 role in the solution operator, coordinates alone may be insufficient to encode all the necessary
 792 geometric priors. Therefore, we introduce geometry embeddings to enhance the model’s geometric
 793 awareness. These embeddings work in tandem with the MAGNO encoder (and decoder), providing a
 794 rich geometric description for each token (latent point y) and its neighborhood at various scales m .

795 Prior work on including geometric information in neural PDE solvers typically resorts to two major
 796 approaches: (i) appending geometry features directly into node/edge attributes [39], or (ii) using a
 797 signed distance function (SDF) [29]. However, we argue that:

- 798 • Simply merging geometry and physical features at the node level may entangle them
 799 prematurely, potentially hurting performance when the geometry is complex or when
 800 additional modalities (e.g. material properties) must be fused.
- 801 • Computing SDF to represent geometry is often cumbersome, especially for unstructured
 802 datasets or when the boundary is only partially known. Each new shape would require
 803 re-computation, and the SDF values may be inaccurate if the surface is not well-defined.

804 Instead, we advocate two more direct and flexible mechanisms for extracting geometric descriptors:
 805 local *Statistical embedding* and *PointNet-based embedding*, shown in Figure B.2.

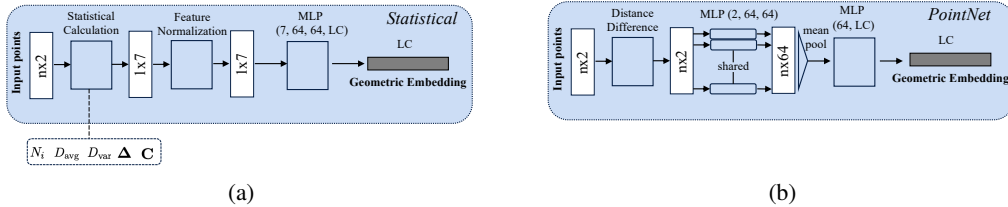


Figure B.2: Schematic of the Geometric Embedding for Statistical Embedding (a) and PointNet-based Embedding (b). LC denotes the lifting channels for MAGNO output.

806 **Local Statistical Embedding** The core idea is to extract statistical descriptors from the neighbor-
 807 hood $B_{r_m}(y)$ (or $B_{\hat{r}_m}(x)$ for the decoder) of original point cloud points x_k (or latent points y_ℓ for
 808 the decoder) for each latent point y (for the encoder) or query point x (for the decoder) at each scale
 809 m . Taking the encoder as an example, for a latent point $y \in \mathcal{D}$ and scale m , its neighborhood is
 810 $N_m(y) = \{x_k \in \mathcal{D}_\Delta : |y - x_k| \leq r_m\}$, containing n_y^m points. We compute the following statistics:

- 811 • Number of Neighbors n_y^m : Measures local density around point y .

- 812 • Average Distance $D_{\text{avg}}^m(y)$:

$$D_{\text{avg}}^m(y) = \frac{1}{n_y^m} \sum_{k=1}^{n_y^m} |y - x_k| \quad (\text{B.6})$$

813 Describes the average spatial extent of the neighborhood.

- 814 • Distance Variance $D_{\text{var}}^m(y)$:

$$D_{\text{var}}^m(y) = \frac{1}{n_y^m} \sum_{k=1}^{n_y^m} (|y - x_k| - D_{\text{avg}}^m(y))^2 \quad (\text{B.7})$$

815 Reflects the dispersion of points within the neighborhood.

- 816 • Neighbor Centroid Offset Vector Δ_y^m :

$$\Delta_y^m = \left(\frac{1}{n_y^m} \sum_{k=1}^{n_y^m} x_k \right) - y \quad (\text{B.8})$$

817 The vector from y to the centroid of its neighbors x_k .

- 818 • PCA Features: These features aim to capture the local shape anisotropy of the distribution
819 of the n_y^m neighbor points $\{x_k\}$ within the m -th scale ball $B_{r_m}(y)$. This is achieved by
820 performing PCA on the set of these neighbor coordinates $\{x_k\}$. Using the centroid of
821 the neighbors $\bar{x}_{\text{nbrs},y}^m = \left(\frac{1}{n_y^m} \sum_{k=1}^{n_y^m} x_k \right)$, the $d \times d$ covariance matrix of the neighbor
822 coordinates is calculated as:

$$\mathbf{C}_y^m = \frac{1}{n_y^m} \sum_{k=1}^{n_y^m} (x_k - \bar{x}_{\text{nbrs},y}^m)(x_k - \bar{x}_{\text{nbrs},y}^m)^\top \quad (\text{B.9})$$

823 If $n_y^m = 0$ (or too few points for a meaningful covariance, e.g. $n_y^m < d$), the covariance
824 matrix \mathbf{C}_y^m is treated as a zero matrix, leading to zero eigenvalues. Otherwise, the d real
825 eigenvalues of this symmetric, positive semi-definite covariance matrix, sorted in descending
826 order ($\lambda_1^m \geq \lambda_2^m \geq \dots \geq \lambda_d^m \geq 0$), are used as the PCA features. These eigenvalues
827 represent the variance of the neighbor data along the principal component directions, thus
828 describing the extent and orientation of the local point cloud cluster.

829 These statistical descriptors, computed for each scale m and each point $y \in \mathcal{D}$, are concatenated into
830 a vector z_y^m , normalized (e.g., to have zero mean and unit variance for each component), and then fed
831 into an MLP to yield the geometry embedding $g^m(y)$ for that scale:

$$g^m(y) = \text{MLP}_{\text{geo}}(\text{Normalize}(z_y^m)) \quad (\text{B.10})$$

832 This MLP_{geo} is typically shared across all points and scales.

833 **Point-Based Embedding** As an alternative, we can train a PointNet-style network [43] to derive
834 a compact geometric descriptor from each token's neighborhood. Classical PointNet architectures
835 typically include:

- 836 • Input Transformer: aligns input points to a canonical space (optional),
837 • Shared MLP: processes each point individually,
838 • Symmetric Pooling: aggregates per-point features into a global descriptor, ensuring permu-
839 tation invariance.

840 In our PDE setting, we do not necessarily need an input transformer; the local coordinates can directly
841 serve as input features. We replace the typical max-pooling with mean-pooling to produce smoother
842 local embeddings (though other pooling strategies are also possible). For a point y and scale m , we
843 collect the relative coordinates of its neighbors $\{\delta_k^m = x_k - y\}_{k=1}^{n_y^m}$. These relative coordinates are
844 fed into a shared MLP (point-wise MLP):

$$h_k^m = \text{MLP}_{\text{pt}}(\delta_k^m) \quad (\text{B.11})$$

Then, a symmetric pooling operation (e.g., mean pooling or max pooling) aggregates these per-point features into a global geometric feature:

$$\bar{h}^m(y) = \text{MeanPool}(\{h_k^m\}_{k=1}^{n_y^m}) = \frac{1}{n_y^m} \sum_{k=1}^{n_y^m} h_k^m \quad (\text{B.12})$$

This aggregated feature $\bar{h}^m(y)$ can optionally be passed through another small MLP to produce the final geometry embedding $g^m(y)$.

B.3.1 Integration of Geometry Embeddings in MAGNO

As depicted in Fig. 2 of the main text and described in the MAGNO paragraph, in the MAGNO component of the encoder (or decoder), the geometry embedding is fused with the AGNO output at each scale. The specific workflow (for the encoder) is as follows:

1. **Scale-Specific AGNO Features:** For latent point y and scale m , compute the AGNO output $\tilde{w}_e^m(y)$ (as described in Sec. B.2.1).
2. **Scale-Specific Geometry Embedding:** In parallel, using methods from Sec. B.3, compute the geometry embedding $g^m(y)$ from the same neighborhood $N_m(y)$.
3. **Feature Fusion:** Concatenate the AGNO features $\tilde{w}_e^m(y)$ and the geometry embedding $g^m(y)$, and pass them through an MLP for fusion, yielding the scale-specific latent feature function $\hat{w}^m(y)$:

$$\hat{w}^m(y) = \text{MLP}_{\text{fuse}}^m([\tilde{w}_e^m(y) \parallel g^m(y)]) \quad (\text{B.13})$$

where \parallel denotes concatenation. And $\text{MLP}_{\text{fuse}}^m$ is shared across scales.

4. **Multiscale Aggregation:** Finally, as described in Sec. B.2.2, an attention mechanism is used to perform a weighted sum of the fused features $\{\hat{w}^m(y)\}_{m=1}^{\bar{m}}$ from all scales, yielding the final encoder output $w_e(y)$ (Eq. (B.5)).

Compared to merging geometry and PDE features at the node level before any operator updates, this *per-scale* integration offers several benefits:

- **Scale-Adapted Geometry.** Each scale has a correspondingly sized neighborhood, allowing the geometric embedding to reflect local shape details at the appropriate radius. Small radii capture fine-grained features (e.g. sharp corners), while large radii convey coarse global context.
- **Modular Flexibility.** Both MAGNO and Geometric Embeddings act as distinct modules. One can upgrade either component (e.g. adopting a more customized local aggregator or geometry encoder) without changing the overall pipeline.
- **Unified Per-Token Fusion.** The final aggregated feature $w_e(y)$ collects information from all relevant scales and from geometric descriptors, leading to a richer token representation. This is particularly advantageous in settings with complex boundaries (e.g. airfoils, porous media) where multiple length scales and shape cues matter.

This design preserves the encode-process-decode philosophy: each token gains geometry-aware, multiscale PDE features during the encoder stage, facilitating global attention and final decoding later in the pipeline.

B.4 Processor

After constructing geometry-aware tokens, we employ a Transformer-based processor to enable global message passing among all tokens. Depending on the chosen tokenization strategy (SM B.1), we can choose the following strategies, respectively:

- **Regular Grid (Strategy I or III):** If the latent points $\{y_\ell \in \mathcal{D}\}$ lie on a regular grid (e.g., via a structured stencil or a projected low-dimensional regular grid), we adopt a strategy similar to vision transformers (ViTs) [10]. The latent points are grouped into non-overlapping "patches." All token features $w_e(y)$ within each patch are flattened and linearly projected into a single patch token embedding. These patch tokens then serve as the input sequence to the Transformer.

- Randomly Downsampled Points (Strategy II): If the latent points $\{y_\ell\}$ are randomly downsampled from the original point cloud D_Δ , they lack a regular grid structure. In this case, there is no obvious "patching" method, and each latent token $w_e(y_\ell)$ directly serves as an element in the Transformer's input sequence.

Positional Encoding Transformers themselves are permutation-invariant and do not inherently process sequential order or spatial position. Thus, positional information must be injected. In GAOT, we use the Relative Positional Embeddings (RoPE) [48], which is a method that integrates relative positional information directly into the self-attention mechanism. It achieves this by applying rotations, dependent on their relative positions, to the Query and Key vectors. This has shown strong performance in many Transformer models.

Transformer Block Structure For the transformer Blocks, we adopt an RMS norm $\text{RMSNorm}(\cdot)$ at the beginning of attention and feedforward layers:

$$\mathbf{z} = \text{RMSNorm}(\mathbf{x}), \quad \text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\text{mean}(\mathbf{x}^2)}} \odot \boldsymbol{\alpha} \quad (\text{B.14})$$

where $\boldsymbol{\alpha}$ is a learned scaling parameter. This approach is akin to LayerNorm but uses the root mean square of feature magnitudes rather than computing mean-and-variance separately. This prenorm design helps stabilize the gradient flow compared to the conventions in [49]. Each block has the structure,

$$\mathbf{Z}_{\text{attn}} = \mathbf{X} + \text{MultiHeadAttn}(\text{RMSNorm}(\mathbf{X})), \quad \mathbf{Z}_{\text{ffn}} = \mathbf{Z}_{\text{attn}} + \text{FFN}(\text{RMSNorm}(\mathbf{Z}_{\text{attn}})). \quad (\text{B.15})$$

Furthermore, we use *Group Query* and *Flash Attention* in the code for efficient multi-head self-attention.

Long-Range Skip Connections In addition to the intra-block residual connections, we also introduce long-range skip connections across multiple Transformer blocks, as suggested in works like [3]. For instance, the Transformer blocks can be divided into an earlier part and a later part, and layers can be symmetrically connected (e.g., the first with the last, the second with the second-to-last, etc.), allowing later blocks to directly receive information from earlier blocks, further improving information flow.

By stacking these blocks, the Transformer processor learns complex global dependencies among tokens, transforming the locally geometry-aware tokens $w_e(y_\ell)$ from the encoder into processed tokens $w_p(y_\ell)$ that incorporate richer contextual information. These processed tokens are then converted by the MAGNO decoder to the desired approximation of the output of the underlying solution operator (see Main Text and Fig. ??)

B.5 Training Details.

This section discusses the details of how the GAOT models were trained, including the loss functions, data normalization procedures, general training hyperparameters, and default model configurations. In our experiments, we address both time-independent and time-dependent PDEs. The application of GAOT to time-independent PDEs is straightforward. For time-dependent PDEs, we employ three different time-stepping methods and all2all training [20], as discussed in the main text. More details on these methods can be found in [39].

B.5.1 Loss Function

The loss function used for training GAOT is the Mean Squared Error (MSE), computed between the model's final predictions and the true physical quantities. For a set of N_s samples and N_p spatial points, the loss is:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N_s N_p} \sum_{i=1}^{N_s} \sum_{j=1}^{N_p} \|\mathcal{S}_\theta(\cdot)_i(x_j) - \mathbf{u}_{\text{true},i}(x_j)\|_2^2 \quad (\text{B.16})$$

where $\mathcal{S}_\theta(\cdot)_i(x_j)$ is the model's prediction for sample i at point x_j , and $\mathbf{u}_{\text{true},i}(x_j)$ is the corresponding ground truth. The exact form of $\mathcal{S}_\theta(\cdot)$ depends on whether the problem is time-independent or time-dependent.

932 **Time-Independent PDEs.** For time-independent PDEs, given an input $a(x_j)$ (e.g., boundary
 933 conditions, coefficients c), GAOT directly predicts the solution $u(x_j)$. Thus, $\mathcal{S}_\theta(a)(x_j)$ is the direct
 934 output of the GAOT architecture, and the MSE loss is computed between $\mathcal{S}_\theta(a)(x_j)$ and the true
 935 steady-state solution $u_{\text{true}}(x_j)$ of PDE (A.2).

936 **Time-Dependent PDEs.** To learn the solution operator for time-dependent PDEs, GAOT is used to
 937 update the solution forward in time. Given the solution $u(t)$ at time t and coefficients c (forming the
 938 augmented input $a(t) = (c, u(t))$), the model predicts the solution at $t + \tau$. The GAOT architecture
 939 produces an output $\hat{\mathcal{S}}_\theta(x, t, \tau, a(t))$. The final prediction for $u(t + \tau)$, denoted $\mathcal{S}_\theta(t, \tau, a(t))$, is
 940 constructed using a general time-stepping strategy as per Eq. (7) from the main text, see also [39]:

$$\mathcal{S}_\theta(t, \tau, a(t)) = \gamma u(t) + \delta \hat{\mathcal{S}}_\theta(x, t, \tau, a(t)) \quad (\text{B.17})$$

941 The MSE loss is then computed between this $\mathcal{S}_\theta(t, \tau, a(t))$ and the true solution $u_{\text{true}}(t + \tau)$. The
 942 choice of parameters (γ, δ) determines the time-stepping strategy and what the network output $\hat{\mathcal{S}}_\theta$
 943 effectively learns:

- 944 • **Output Stepping** ($\gamma = 0, \delta = 1$): The final prediction is $\mathcal{S}_\theta(t, \tau, a(t)) = \hat{\mathcal{S}}_\theta(x, t, \tau, a(t))$.
 945 The network output $\hat{\mathcal{S}}_\theta$ directly learns to approximate $u(t + \tau)$.
- 946 • **Residual Stepping** ($\gamma = 1, \delta = 1$): The final prediction is $\mathcal{S}_\theta(t, \tau, a(t)) = u(t) +$
 947 $\hat{\mathcal{S}}_\theta(x, t, \tau, a(t))$. The network output $\hat{\mathcal{S}}_\theta$ learns to approximate the residual, $u(t + \tau) - u(t)$.
- 948 • **Time-Derivative Stepping** ($\gamma = 1, \delta = \tau$): The final prediction is $\mathcal{S}_\theta(t, \tau, a(t)) = u(t) +$
 949 $\tau \cdot \hat{\mathcal{S}}_\theta(x, t, \tau, a(t))$. The network output $\hat{\mathcal{S}}_\theta$ learns to approximate the time-derivative,
 950 $(u(t + \tau) - u(t))/\tau$.

951 GAOT offers the flexibility to use any of these strategies. A detailed ablation of their comparative
 952 performance is described in SM Sec. F.

953 B.5.2 Data Normalization

954 Data normalization is applied to stabilize training. We typically use Z-score normalization, where for
 955 a quantity X , its normalized version \hat{X} is $(X - \mu_X)/\sigma_X$. The mean μ_X and standard deviation σ_X
 956 are computed over the training dataset.

957 **Time-Independent PDEs.** For input features $a(x_j)$ and output solution fields $u(x_j)$, normalization
 958 parameters are computed across all samples and spatial points in the training set for each channel
 959 independently. The model is trained on normalized inputs to predict normalized outputs.

960 **Time-Dependent PDEs.** The input $u(t)$ is normalized using its global mean and standard deviation
 961 computed over all time steps and samples in the training set. The normalization of the target for the
 962 network output $\hat{\mathcal{S}}_\theta(x, t, \tau, a(t))$ depends on the chosen time-stepping strategy, as $\hat{\mathcal{S}}_\theta$ learns a different
 963 physical quantity in each case:

- 964 • **Output Stepping:** The network $\hat{\mathcal{S}}_\theta$ aims to predict $u(t + \tau)$. Thus, the ground truth values
 965 $u(t + \tau)$ are normalized, and $\hat{\mathcal{S}}_\theta$ is trained to predict these normalized values. Statistics μ_u
 966 and σ_u are computed from all values $u(t')$ in the training set. The normalized target for $\hat{\mathcal{S}}_\theta$ is
 967 $\hat{u}(t + \tau) = (u(t + \tau) - \mu_u)/\sigma_u$.
- 968 • **Residual Stepping:** The network $\hat{\mathcal{S}}_\theta$ aims to predict the residual $R(t, \tau) = u(t + \tau) - u(t)$. Thus,
 969 these true residual values are computed from the training data, and their statistics (μ_R, σ_R) are
 970 used for normalization. The normalized target for $\hat{\mathcal{S}}_\theta$ is $\hat{R}(t, \tau) = (R(t, \tau) - \mu_R)/\sigma_R$.
- 971 • **Time-Derivative Stepping:** The network $\hat{\mathcal{S}}_\theta$ aims to predict the time-derivative $D(t, \tau) = (u(t +$
 972 $\tau) - u(t))/\tau$. These true derivative values are computed, and their statistics (μ_D, σ_D) are used for
 973 normalization. The normalized target for $\hat{\mathcal{S}}_\theta$ is $\hat{D}(t, \tau) = (D(t, \tau) - \mu_D)/\sigma_D$.

974 Time t and lead-time τ inputs are also typically scaled or normalized. Further details on these
 975 normalizations can be found in [39].

B.5.3 General Training Setup

This section provides an overview of our training hyperparameters. Unless otherwise noted, all experiments follow these settings. Table B.1 summarizes the primary hyperparameters and training schedules. In particular, we distinguish between time-dependent and time-independent PDE tasks in terms of epoch count, and highlight the differences in hardware usage for the DrivAerNet++ dataset. All models except DrivAerNet++ run on a single GeForce RTX 4090 GPU. For the DrivAerNet++ dataset, we use 4 GeForce A100 (40GB) GPUs in data parallel mode, and each GPU holds a batch size of 1. For the scheduler, we warm up to mitigate instability at early epochs, then adopt a cosine schedule for gradual decay, and finalize with a step-based drop for fine-tuning the last epoch range.

Table B.1: Key training hyperparameters and schedulers used for all models, unless otherwise specified.

Hardware	<ul style="list-style-type: none"> • Single-GPU: All models (except DrivAerNet++) are trained on a single GeForce RTX 4090 with batch size = 64. • Four-GPU: For the DrivAerNet++ dataset, we use four GeForce A100 (40GB) GPUs, each with batch size = 1.
Optimizer	<ul style="list-style-type: none"> • Algorithm: AdamW • Weight Decay: 1×10^{-5}
Epochs	<ul style="list-style-type: none"> • Time-Dependent PDEs: 500 epochs • Time-Independent PDEs: 1000 epochs except the DrivAerNet++ dataset, which is trained on 200 epochs.
Learning Rate Scheduler	<ul style="list-style-type: none"> • Warmup (first 10% epochs): LR increases linearly from 8×10^{-4} to 1×10^{-3}. • Cosine Decay (next 85% epochs): LR decays from 1×10^{-3} to 1×10^{-4}. • StepLR (final 5% epochs): LR drops from 1×10^{-4} to 5×10^{-5}.

B.5.4 GAOT Model Configuration

Table B.2 outlines the default configuration of our GAOT framework. This includes the MAGNO used in both the encoder and decoder stages, as well as the Transformer-based global processor. MAGNO converts node features into geometry-aware tokens (encoder) and reconstructs continuous fields (decoder). By default, the coordinates will be rescaled in the domain $[-1, 1]^d$, and we use a single aggregation radius 0.033 for adequate coverage. If multiscale is enabled, we adopt radii $\{0.022, 0.033, 0.044\}$. The default geometric embedding method is local *Statistical Embedding* (e.g. as SM Sec. B.3), and will typically be implemented for unstructured datasets. The Transformer processes geometry-aware tokens globally via multi-head self-attention. We set the hidden dimension to 256 with a 1024-dim feed-forward layer, residual connections, RMSNorm, and RoPE for positional embeddings. By adjusting the patch size and the number of tokens, we can trade off computational cost and model resolution.

B.6 Inference

When predicting solutions for *time-independent* PDEs, we simply feed the input parameters a (e.g. boundary conditions, coefficients, or geometric shape) into our learned operator $\mathcal{S}_\theta(a)$ and obtain the steady-state output $u(x)$ directly. However, for *time-dependent* PDEs, there are two different strategies for forecasting the solution at a future time, using the learned one-step advancement operator $\mathcal{S}_\theta(t, \tau, a(t))$ which, as defined in Eq. (B.17), takes the current time t , a lead-time τ , and the augmented input $a(t) = (c, u(t))$ to predict the solution at $t + \tau$. The two main inference strategies are *direct inference* and *autoregressive inference*.

Direct Inference (DR) Recall from the main text that our learned operator \mathcal{S}_θ takes the lead-time τ as an explicit input, allowing for predictions over variable time steps. Given a snapshot of the solution $u(t_n)$ (which is part of $a(t_n)$), the network can directly predict the solution at any later time $t_n + \tau_{target}$, up to a maximum trained horizon t_{max} , by evaluating $\mathcal{S}_\theta(t_n, \tau_{target}, a(t_n))$. Hence, for each possible time increment $\tau_{target} = k \cdot \Delta t$ (where Δt is a base time step in the dataset, and

Table B.2: Default architectural hyperparameters for GAOT.

Abbreviation	Default Value	Description
MAGNO (Encoder / Decoder)		
PJC	256	Dimensionality for MAGNO’s internal hidden layers.
ENC-MLP	[64, 64, 64]	Hidden layers of the encoder MLP in MAGNO.
DEC-MLP	[64, 64]	Hidden layers of the decoder MLP in MAGNO.
LC	32	Output/Lifting channels after MAGNO (both encoder and decoder).
TS	I	Tokenization Strategy I.
NT	[64, 64]	Number of tokens for Strategy I (e.g., a 64×64 stencil grid).
GR	0.033	Aggregation radius (single-scale) for every token. If multiscale: $\{0.022, 0.033, 0.044\}$.
GeoEmb	statistical	Geometric Embedding for the encoder and decoder.
EM	0.3	Edge masking ratio for the MAGNO, used for 3D drivearnet++ dataset.
Transformer		
PS	2	Default patch size for token grouping if Strategy I or III is used.
Norm	RMSNorm	Normalization used in attention and MLP layers. Pre-norm configuration.
PE	RoPE	Positional embedding used in the Transformer. Rotary positional embeddings.
RES-CON	True	Residual connections between transformer blocks.
TL	5	Number of Transformer blocks.
THS	256	Hidden dimension per self-attention block.
HEAD	8	Number of attention heads.
Dropout	0.2	Dropout ratio in the attention module.
FFN	1024	$4 \times$ hidden size (THS) for the feedforward layer.

1010 $1 \leq k \leq k_{\max}$), we can produce the model’s estimate $u(t_n + \tau_{target})$ from $u(t_n)$ in a single step,
1011 without iterating through intermediate time steps. Concretely, if our dataset is discretized at times
1012 $\Omega_t^\Delta = \{t_0, t_1, \dots, t_N\}$, we can directly evaluate $\mathcal{S}_\theta(t_n, \tau_{target}, a(t_n))$ for various τ_{target} values
1013 originating from any $t_n \in \Omega_t^\Delta$. This provides a sequence of direct predictions at each possible time
1014 offset τ_{target} from any initial time t_n .

1015 **Autoregressive Inference (AR)** While direct inference estimates the solution at a single future
1016 time, an alternative is to iterate the operator in multiple, typically smaller, sub-steps to reach the
1017 final time. This approach is called *autoregressive* (AR) inference. Formally, given an initial snapshot
1018 $u(t_0)$, we repeatedly apply the learned operator \mathcal{S}_θ with a chosen fixed time increment for each step,
1019 Δt_{AR} , to advance the solution:

$$u(t_{k+1}) = \mathcal{S}_\theta(t_k, \Delta t_{AR}, a(t_k)) \quad (\text{B.18})$$

1020 where $t_{k+1} = t_k + \Delta t_{AR}$, and $a(t_k) = (c, u(t_k))$ uses the solution $u(t_k)$ from the previous step (or
1021 the initial condition if $k = 0$). This process is repeated until the desired final time t_{final} is reached.

1022 We examine two types of autoregressive step sizes in our experiments:

- 1023 • **AR-2:** Use an autoregressive time increment of $\Delta t_{AR} = 2$ (assuming time units are
1024 consistent with the dataset). Starting from $u(t_0)$, we compute $u(t_0 + 2) = \mathcal{S}_\theta(t_0, 2, a(t_0))$.
1025 Then, using $u(t_0 + 2)$, we compute $u(t_0 + 4) = \mathcal{S}_\theta(t_0 + 2, 2, a(t_0 + 2))$, and so on, up to
1026 t_{14} . In total, we perform 7 consecutive evaluations of \mathcal{S}_θ .
- 1027 • **AR-4:** Use an autoregressive time increment of $\Delta t_{AR} = 4$. In this scenario, we predict
1028 $u(t_0 + 4) = \mathcal{S}_\theta(t_0, 4, a(t_0))$, then from $u(t_0 + 4)$, $u(t_0 + 8) = \mathcal{S}_\theta(t_0 + 4, 4, a(t_0 + 4))$,
1029 and so on, eventually reaching t_{14} in just 4 iterations (assuming $t_0 = 0$ and $t_{\text{final}} = 16$ for
1030 this example, or if t_{14} is the target after some steps).

1031 Generally, the choice of the AR step size Δt_{AR} is flexible. One could select any valid $\Delta t_{AR} \leq \tau_{\max}$
 1032 (where τ_{\max} is the maximum lead-time the model was reliably trained for in a single step) at each
 1033 sub-step. Note that using fewer, larger time steps (e.g., $\Delta t_{AR} = 4$) can reduce computational cost
 1034 but potentially compounds prediction errors more quickly if the operator \mathcal{S}_θ is less accurate for
 1035 larger single-step lead-times. Conversely, smaller increments (e.g. $\Delta t_{AR} = 2$) tend to accumulate
 1036 errors more gradually but require more iterations (and thus more computation) to reach the final time.
 1037 Details can be found in [20, 39].

1038 C Baselines

1039 For the time-dependent benchmarks (including those on unstructured grids detailed in Table 1 and
 1040 regular grids in Table E.2, the corresponding baseline results are primarily obtained from the work
 1041 by [39]. These baseline models include RIGNO-12, RIGNO-18, CNO, scOT, FNO, GeoFNO,
 1042 FNO DSE, and GINO. For further details on these methods, please refer to the paper [39]. In this
 1043 work, we have additionally included three more recent models for a comprehensive comparison:
 1044 Transolver [50], GNOT [18], and UPT [1]. Brief descriptions of these newly added models and the
 1045 specific hyperparameters adopted in our experiments are provided below.

1046 C.1 UPT

1047 *Universal Physics Transformers* (UPT) [1] form a neural-operator framework that fits into the
 1048 canonical encode–process–decode pipeline:

$$\mathcal{U} = \mathcal{D} \circ \mathcal{A} \circ \mathcal{E}, \quad (\text{C.1})$$

1049 where \mathcal{E} (Encoder) compresses k input points—coming from an arbitrary Eulerian mesh or Lagrangian
 1050 particle cloud—into a fixed set of n_{latent} tokens. It first embeds the features and coordinates through
 1051 a radius-graph message-passing layer that aggregates information into n_s supernodes, and finally
 1052 employs transformer and perceiver pooling blocks to obtain the latent representation $z_t \in \mathbb{R}^{n_{\text{latent}} \times h}$.

1053 \mathcal{A} (Approximator) is a stack of transformer blocks that advances the latent state in time, $\mathcal{A} : z_t \mapsto$
 1054 $z_{t+\Delta t}$, enabling fast *latent roll-outs* without repeatedly decoding to the spatial domain.

1055 \mathcal{D} (Decoder) is a Perceiver-style cross-attention module that evaluates the latent field at any set of
 1056 query positions $\{y_i\}_{i=1}^{k'}$, yielding $u_{t+\Delta t}(y_i) = \mathcal{D}(z_{t+\Delta t}, y_i)$ with $\mathcal{O}(n_{\text{latent}})$ complexity independent
 1057 of k' .

1058 In the setting of time-independent problems, we bypass the latent roll-out stage, and adopt a
 1059 lightweight configuration in our experiments. Specifically, we use latent tokens = 64 and em-
 1060 bedding dimensions = 64, which results in a model size of $0.74M$. A large variant with 256 latent
 1061 tokens and 192 embedding dimension as setup in [1] was found to suffer from optimization difficulties
 1062 and was not adopted in our baseline results. The same optimizer setup as GAOT is used here.

1063 Considered Hyperparameters

<i>Architecture</i>	
Trainable parameters	0.74M
Number of supernodes n_s	2048
Radius for message passing	0.033
Embedding (feature) channels	64
Encoder transformer blocks	4
Encoder attention heads	4
Latent tokens n_{latent}	64
Latent dimension h	64
Approximator transformer blocks	4
Approximator attention heads	4
Decoder attention heads	4
<i>Training</i>	
Optimizer	AdamW
Scheduler	same as in B.2
Initial learning rate	$1 \cdot 10^{-3}$
Weight decay	10^{-5}
Number of epochs	500
Batch size	64

1065 C.2 Transolver

1066 *Transolver* [50] is a transformer-based operator learning model designed for PDEs on unstructured
 1067 grids. It follows an encode-process-decode paradigm by stacking multiple Transolver blocks. The
 1068 core of each block is the Physics-Attention mechanism.

1069 Given input features $X_{\text{phys}} \in \mathbb{R}^{N \times C}$ for N mesh points:

- 1070 1. Encoding to Tokens: First, for each mesh point feature $x_i \in X_{\text{phys}}$, M slice weights
 1071 $w_i \in \mathbb{R}^{1 \times M}$ are learned, typically via a projection followed by a Softmax function: $w_i =$
 1072 $\text{Softmax}(\text{Project}(x_i))$. These weights determine the assignment of mesh points to M
 1073 learnable "slices". The j -th physics-aware token $z_j \in \mathbb{R}^{1 \times C}$ is then encoded by a weighted
 1074 aggregation of all mesh point features, using the slice weights:

$$z_j = \frac{\sum_{i=1}^N w_{i,j} x_i}{\sum_{i=1}^N w_{i,j}} \quad (\text{C.2})$$

1075 This results in M tokens $Z = \{z_j\}_{j=1}^M \in \mathbb{R}^{M \times C}$.

- 1076 2. Token Processing: These M tokens Z are processed by a standard attention mechanism
 1077 (e.g., multi-head self-attention) to capture correlations between different physical states
 1078 represented by the tokens:

$$Z'_{\text{proc}} = \text{Attention}(Z) \quad (\text{C.3})$$

1079 The processed tokens are $Z'_{\text{proc}} = \{z'_j\}_{j=1}^M \in \mathbb{R}^{M \times C}$.

- 1080 3. Decoding to Physical Grid (Deslicing): The updated token features Z'_{proc} are then broadcast
 1081 back and recomposed onto the N physical mesh points using the original slice weights w :

$$x'_i = \sum_{j=1}^M w_{i,j} z'_j \quad (\text{C.4})$$

1082 This yields the output features for the Physics-Attention block, $X'_{\text{phys}} = \{x'_i\}_{i=1}^N \in \mathbb{R}^{N \times C}$.

1083 A full Transolver layer typically incorporates this Physics-Attention mechanism within a standard
 1084 Transformer layer structure, including Layer Normalization and Feed-Forward Networks.

While the Physics-Attention mechanism itself is designed to have a computational complexity linear with respect to the number of mesh points, it is important to note that the slicing (Eq. C.2) and deslicing (Eq. C.4) operations, which involve all N points, are performed within each of the L Transolver layers. This repeated mapping can lead to significant computational costs and memory overhead, especially for large N . This contrasts with architectures like GAOT and UPT, which perform the encoding to a latent space and decoding from it only once, with intermediate processing happening entirely in the latent token domain. In our experiments with time-independent partial differential equations, we followed the settings from the original Transolver paper [50];

Considered Hyperparameters

<i>Architecture</i>	
Trainable parameters	3.85 M
Hidden channels	256
Attention heads	8
Number of Layers	8
MLP ratio	2
number of slice	32
<i>Training</i>	
Optimizer	AdamW
Scheduler	same as in B.2
Initial learning rate	$1 \cdot 10^{-3}$
Weight decay	10^{-5}
Number of epochs	500
Batch size	20

C.3 GNOT

General Neural Operator Transformer (GNOT) [18] is a Transformer-based framework designed for operator learning, particularly addressing challenges such as irregular meshes, multiple heterogeneous input functions, and multiscale problems. Its overall architecture can be represented as:

$$\mathcal{G} = \underbrace{\mathcal{F} \circ (\mathcal{B})^L}_{\text{processor}} \circ \mathcal{E}, \quad (\text{C.5})$$

where \mathcal{E} is the encoder, \mathcal{B} represents a GNOT Transformer block (repeated L times), and \mathcal{F} is the output decoder.

1. Encoder (\mathcal{E}): The encoder maps diverse input sources (geometry, fields, parameters, edges) to embeddings using dedicated MLPs. This yields query embeddings $Q \in \mathbb{R}^{N_q \times d}$ for target points and a set of m conditional embeddings $\{Y^{(\ell)} \in \mathbb{R}^{N_\ell \times d}\}_{\ell=1}^m$ from other input functions.
2. GNOT Transformer Block (\mathcal{B}): Each block refines query embeddings Q using conditional embeddings $Y^{(\ell)}$ via:
 - Heterogeneous Normalized linear Cross-Attention (HNA): Fuses Q with each $Y^{(\ell)}$ using separate MLPs for keys/values from different $Y^{(\ell)}$, followed by normalization and averaging.

$$Q'_{\text{cross}} = Q + \frac{1}{L_c} \sum_{\ell=1}^{L_c} \text{NormLinearCrossAttn}(Q, Y^{(\ell)}) \quad (\text{C.6})$$

- Normalized Self-Attention: Applies normalized linear self-attention to Q'_{cross} for further refinement.

$$Q'_{\text{self}} = \text{NormLinearSelfAttn}(Q'_{\text{cross}}) \quad (\text{C.7})$$

- Geometric Gating FFN: A Mixture-of-Experts (MoE) FFN where expert FFNs (E_k) are weighted by $p_k(x_{\text{coord}})$. These weights are predicted by a gating network $G(\cdot)$ using query point coordinates x_{coord} , enabling soft domain decomposition for multiscale problems.

$$\text{FFN}_{\text{Gated}}(X) = \sum_{k=1}^K p_k(x_{\text{coord}}) \cdot E_k(X), \quad p_k(x_{\text{coord}}) = \text{Softmax}(G_k(x_{\text{coord}})) \quad (\text{C.8})$$

- These components, with Layer Normalization and residual connections, form the block.
3. Decoder (\mathcal{F}): After L blocks, a final decoder (typically an MLP) maps processed query features to the output solution.

Considered Hyperparameters

<i>Architecture</i>	
Trainable parameters	4.87 M
Hidden channels	128
Attention heads	8
Number of Layers	8
MLP ratio	2
<i>Training</i>	
Optimizer	AdamW
Scheduler	same as in B.2
Initial learning rate	$1 \cdot 10^{-3}$
Weight decay	10^{-5}
Number of epochs	500
Batch size	20

D Datasets

In this work, we test GAOT on 24 benchmarks for both time-independent and time-dependent PDEs of various types, ranging from regular grids to random point clouds to highly unstructured adapted grids. The time-dependent and Poisson-Gauss dataset are sourced from [20] and [39], respectively. A static elasticity dataset is from [26]. We have generated five additional challenging datasets: a Poisson-C-Sines dataset exhibiting multiscale properties, and four datasets for compressible fluid dynamics with highly unstructured adapted grids. Detailed information regarding these datasets can be found in the Tab [D.1](#).

We focus on the following PDE Types under various initial/boundary conditions and domain geometries:

Hyper-Elastic Equation (HEE) :

$$\rho^s \frac{\partial^2 \mathbf{u}}{\partial t^2} + \nabla \cdot \boldsymbol{\sigma} = 0, \quad (\text{D.1})$$

where ρ^s is the mass density, \mathbf{u} is the displacement vector, and $\boldsymbol{\sigma}$ is the stress tensor. A constitutive model links the strain tensor $\boldsymbol{\epsilon}$ to the stress tensor. The material is the incompressible Rivlin-Saunders type, characterized by $\boldsymbol{\sigma} = \frac{\partial w(\boldsymbol{\epsilon})}{\partial \boldsymbol{\epsilon}}$ with $w(\boldsymbol{\epsilon}) = C_1(I_1 - 3) + C_2(I_2 - 3)$.

Poisson Equation (PE) :

$$-\Delta u = f, \quad \text{in } (0, 1)^2, \quad (\text{D.2})$$

with homogeneous Dirichlet boundary conditions. The dataset related to poisson equation use either sinusoidal or Gaussian-like source terms on square or circular domains (see Table [D.1](#)).

Table D.1: Overview of the datasets used in this work. Datasets listed above line are time-independent, while those below are time-dependent. Geometry variation (GeoVar) describes whether all data samples share the same geometry. Characteristic briefly describes each dataset’s geometry or PDE setup. The PDE Type column indicates the corresponding class. Visualization (Vis.) provides references to visual examples; for time-dependent datasets, this may include visualizations for both unstructured partially ones and original regular grid ones. Datasets marked with * are newly proposed in this work.

Abbreviation	GeoVar	Characteristic	PDE Type	Vis.
Poisson-C-Sines*	F	Circular domain with sines f	PE	G.1
Poisson-Gauss	F	Gaussian source	PE	G.2
Elasticity	T	Hole boundary distance	HEE	G.3
NACA0012*	T	Flow past NACA0012 airfoil	CE	G.4
NACA2412*	T	Flow past NACA2412 airfoil	CE	G.5
RAE2822*	T	Flow past RAE2822 airfoil	CE	G.6
Bluff-Body*	T	Flow past bluff-bodies	CE	G.7
DrivAerNet++(p)	T	Surface pressure	INS	G.8
DrivAerNet++(wss)	T	Surface wall shear stress	INS	G.9
NS-Gauss	F	Gaussian vorticity IC	INS	G.10 , G.18
NS-PwC	F	Piecewise const. IC	INS	G.11 , G.19
NS-SL	F	Shear layer IC	INS	G.12 , G.20
NS-SVS	F	Sinusoidal vortex sheet IC	INS	G.13 , G.21
CE-Gauss	F	Gaussian vorticity IC	CE	G.14 , G.22
CE-RP	F	4-quadrant RP	CE	G.15 , G.23
Wave-Layer	F	Layered wave medium	WE	G.16 , G.24
Wave-C-Sines	F	Circular domain with sines IC	WE	G.17

1138 **Incompressible Navier–Stokes (INS)** :

$$\begin{aligned}\nabla \cdot \mathbf{v} &= 0, \\ \partial_t \mathbf{v} + (\mathbf{v} \cdot \nabla) \mathbf{v} &= -\nabla p + \nu \nabla^2 \mathbf{v},\end{aligned}$$

1139 where \mathbf{v} is the velocity field, p is the pressure, and viscosity is ν . It assumes periodic boundary
1140 conditions and sample various initial conditions (e.g., Gaussian, piecewise-constant, sinusoidal vortex
1141 sheets).

1142 **Compressible Euler (CE)** :

$$\partial_t \mathbf{u} + \nabla \cdot \mathbf{F} = 0, \quad \mathbf{u} = (\rho, \rho \mathbf{v}, E)^\top, \quad E = \frac{1}{2} \rho \|\mathbf{v}\|^2 + \frac{p}{\gamma-1}, \quad (\text{D.3})$$

1143 with $\gamma = 1.4$. Showing in [20], it imposes periodic boundary conditions and ignore gravity effects.
1144 Data are generated using random initial/boundary conditions such as Gaussian or Riemann problem
1145 (RP) setups.

1146 **Wave Equation (WE)** :

$$\partial_{tt} u - c^2(x, y) \nabla^2 u = 0, \quad (\text{D.4})$$

1147 with a spatially varying propagation speed $c(x, y)$ in an inhomogeneous medium. The dataset employs
1148 absorbing or homogeneous Dirichlet boundaries. Initial conditions (e.g., sinusoidal or layered) are
1149 drawn from parameterized distributions.

1150 All time-dependent problems are numerically integrated up to $T = 1$ (except for the Wave-
1151 C-Sines where $T=0.005$), collected (up to) $N_t = 21$ uniform snapshots per sample at $t \in$
1152 $\{0, 2, 4, 6, 8, 10, 12, 14\}$. For time-dependent PDE, as mentioned before, we use the same all2all
1153 training strategy proposed in Poseidon [20]. This means that each trajectory can generate 28 pairs for
1154 training.

1155 D.1 Poisson-C-Sines

1156 This dataset contains solutions to the two-dimensional Poisson equation with a circular domain.
 1157 The Poisson equation is a fundamental linear elliptic partial differential equation (PDE) given by
 1158 Eq. D.2. The dataset represents the mapping from the source term f to the solution u using the
 1159 solution operator $\mathcal{G}^\dagger : f \mapsto u$. The source term is defined as:

$$f(x, y) = \frac{\pi}{K^2} \sum_{i,j=1}^K a_{ij} \cdot (i^2 + j^2)^{-r} \sin(\pi i x) \sin(\pi j y), \quad \forall (x, y) \in D, \quad (\text{D.5})$$

1160 where $r = -0.5$ and $K = 16$. The coefficients a_{ij} are sampled i.i.d. uniformly from $[-1, 1]$ to
 1161 generate the dataset. The solution u is computed on a circular domain with zero Dirichlet boundary
 1162 conditions. The dataset is generated using a finite element method (FEM) on a triangular mesh in a
 1163 circular domain. The mesh is generated using the Delaunay algorithm with 16431 points and 32441
 1164 elements.

1165 D.2 Compressible Flow Past Airfoils & Bluff Bodies

1166 A classic benchmark for compressible flow physics used for testing the accuracy of neural operators
 1167 and PDE foundation models is the case of flow past airfoils [20, 26]. The datasets used in these
 1168 papers are limited to transonic flow past perturbations of a single airfoil. To capture a broader range
 1169 of rich flow phenomena, it is essential to explore the parameter space spanned by the Mach number
 1170 Ma , angle of attack α and the shape function. To address this issue, this new dataset introduces
 1171 samples comprising a range of flow phenomena from subsonic to supersonic flow for varying angles
 1172 of attack across classical airfoils and various bluff body geometries. The steady-state compressible
 1173 Euler equations govern the flow phenomena in this dataset. The equations have been solved using the
 1174 finite-volume EULER solver of the open-source software SU2 [11] on an unstructured grid generated
 1175 by Delaundo [40]. Convective flux discretization is done using the Jameson-Schmidt-Turkel (JST)
 1176 scheme that is designed especially for achieving quick convergence to steady-state solutions of the
 1177 compressible Euler equations. Figure D.1 represents an O-type unstructured mesh generated using
 1178 Delaundo for the RAE2822 airfoil. Similar O-type unstructured meshes have been generated for all
 1179 airfoils and bluff-bodies considered. The free-stream pressure and temperature conditions for all
 1180 simulations in this dataset are $p_\infty = 1$ atm and $T_\infty = 288.15$ K.

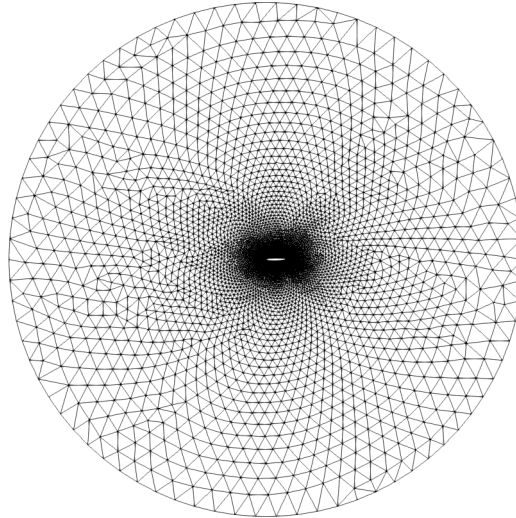


Figure D.1: O-type unstructured mesh - RAE2822 airfoil

1181 D.2.1 Airfoils

1182 Flow past airfoils is considered for $0.5 \leq Ma \leq 1.4, 0.5^\circ \leq \alpha \leq 5.0^\circ$ and for 500 unique
 1183 perturbations applied to shape functions of the NACA2412, NACA0012, and RAE2822 airfoils.

1184 Anisotropic adaptive mesh refinement for highly accurate shock resolution (oblique and bow shocks)
 1185 is performed using INRIA's pyAMG library coupled with SU2 [32]. The anisotropic mesh refinement
 1186 is done using a Mach sensor that generates refined meshes based on the simulation on a coarse grid
 1187 such as in Figure D.1. The final simulations are then performed by the SU2 EULER solver on the
 1188 new refined mesh. Figure D.2 represents the highly unstructured adapted grids for transonic and
 1189 supersonic flow past the RAE2822 airfoil.

1190 We consider the reference airfoil shapes with the upper and lower surface coordinates located at
 1191 $(x, y_{\text{ref}}^U(\xi))$ and $(x, y_{\text{ref}}^L(\xi))$ where $\xi = \frac{x}{c}$, c is the chord length. We use the Class Function/Shape
 1192 Function Transformation (CST) Method [24] for parameterizing the airfoil surfaces in terms of a
 1193 class function $C(\xi)$ and shape functions $S^U(\xi)$, $S^L(\xi)$ using an in-house MATLAB code. The airfoil
 1194 upper surface function $\eta^U(\xi)$ and lower surface function $\eta^L(\xi)$ are parametrized as follows:

$$\eta^U(\xi) = C(\xi)S^U(\xi), \quad \eta^L(\xi) = C(\xi)S^L(\xi) \quad (\text{D.6})$$

1195 where the class function for airfoils is given as:

$$C(\xi) = \sqrt{\xi(1-\xi)} \quad (\text{D.7})$$

1196 and the upper and lower surface shape functions are respectively

$$S^U(\xi) = \sum_{i=0}^n A_i \frac{n!}{i!(n-i)!} \xi^i (1-\xi)^{n-i}, \quad S^L(\xi) = \sum_{i=0}^n B_i \frac{n!}{i!(n-i)!} \xi^i (1-\xi)^{n-i} \quad (\text{D.8})$$

1197 The polynomials $S_{i,n} = \frac{n!}{i!(n-i)!} \xi^i (1-\xi)^{n-i}$ associated with the coefficients (A_i, B_i) are Bernstein
 1198 polynomials and $n = 7$ is chosen. The parameters (A_i, B_i) directly influence key airfoil design
 1199 variables such as the leading edge radius, trailing edge boattail angle, maximum airfoil thickness and
 1200 maximum thickness location. The parameters (A_0, B_0) are linked to the leading edge radius R_{LE} as
 1201 follows,

$$A_0 = -B_0 = \sqrt{2r}, \quad r = \frac{R_{\text{LE}}}{c} \quad (\text{D.9})$$

1202 and the parameters (A_n, B_n) are linked to the upper boattail angle β_U and lower boattail angle β_L :

$$A_n = \tan(\beta_U), \quad B_n = \tan(\beta_L) \quad (\text{D.10})$$

1203 To generate perturbed variations of the airfoils, minor random perturbations are made to the CST
 1204 parameters (A_i, B_i) keeping in mind the constraints $A_0 = -B_0$, $A_i > B_i$ and $\beta_U > \beta_L$. We
 1205 have randomly sampled 5384 solutions from our dataset for each classical airfoil shape with a
 1206 train/validation/test split of 5000/128/256. For each data, we sub-samples 8000 points for training,
 1207 validation and testing.

1208 D.2.2 Bluff-Body

1209 Flow past bluff-bodies is considered at $0.3 \leq Ma \leq 1.3$, $0.5^\circ \leq \alpha \leq 15.0^\circ$ for a wide variety of
 1210 simple bluff-body geometries. Steady-state solutions for the compressible Euler equation for flow
 1211 past bluff-bodies may not exist or are often unstable making it difficult to attain convergence. This
 1212 bluff-body aerodynamics dataset comprises of samples that are at pseudo-steady state in a large
 1213 finite-time limit. Figure E.6a describes all the bluff body geometries taken into consideration in this
 1214 dataset.

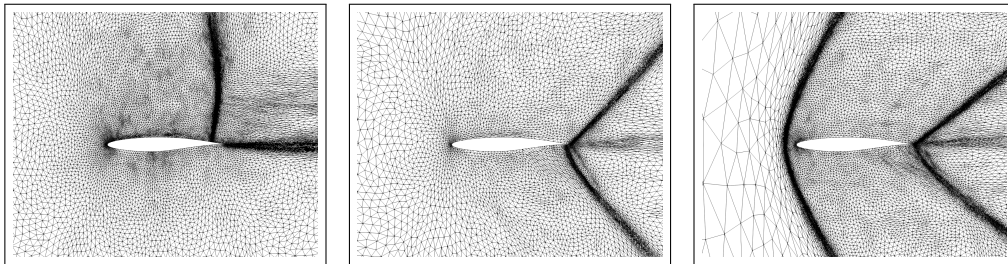


Figure D.2: Adaptively refined meshes for flow past the RAE2822 airfoil at $\alpha = 2.0$ and at different $Ma = 0.8$ (left), 1.0 (center), 1.4 (right).

1215 We sample 4384 solutions from our dataset with a train/validation/test split of 4000/128/256 for all
 1216 bluff-body geometries used for "Training and Testing". For each data, we sub-samples 14000 points
 1217 for training, validation and testing.

1218 E Additional Results

1219 E.1 Accuracy, Robustness and Computational Efficiency Metrics

1220 **Accuracy** For benchmarks in Table 1 and Table E.3, we adopt the relative L^1 error metric, following
 1221 the manner of CNO [45], to measure the discrepancy between the ground-truth operator output $\mathcal{S}(a)$
 1222 and the model's prediction $\mathcal{S}_\theta(a)$ over a discrete set of points. Suppose a given sample is discretized
 1223 into N points (either on a regular grid or an unstructured mesh). For a single-component solution
 1224 field, the discretized relative L^1 error ε is defined as

$$\varepsilon = \frac{1}{N} \sum_{i=1}^N \frac{|(\mathcal{S}(a))_i - (\mathcal{S}_\theta(a))_i|}{|(\mathcal{S}(a))_i|}. \quad (\text{E.1})$$

1225 Because the test set contains multiple input–output pairs $\{(a, \mathcal{S}(a))\}$, we obtain a distribution of
 1226 errors. We report the median of these errors—rather than the mean—to mitigate the influence of
 1227 strong outliers. For multi-component PDE solutions (e.g., velocity and pressure fields), we compute
 1228 the median error per component, then average these medians to obtain a single scalar metric. In
 1229 time-dependent tasks, we specifically report the relative L^1 error at the final time snapshot, as errors
 1230 usually accumulate over time and thus the last snapshot often poses the greatest challenge.

1231 For the pressure and wall shear stress (WSS) in the DrivAerNet++ dataset, we evaluated the model
 1232 on 1154 samples according to the official leaderboard. The errors are calculated based on normalized
 1233 pressure and WSS. For pressure, the mean and standard deviation (std) for normalization were
 1234 obtained from the open-sourced code of [12]. However, for WSS, as the normalization statistics
 1235 were not open-sourced, we calculated the mean and variance for the x, y, and z components over
 1236 8000 samples to be used for normalization. The Mean Squared Error (MSE) and Mean Absolute
 1237 Error (MAE) are first computed for each individual sample. Then, the average of these errors across
 1238 the 1154 test samples is reported as the final result. This entire procedure strictly follows their
 1239 open-sourced code methodology of [12].

1240 In Table E.1, we further provide an aggregate performance comparison of GAOT and three represen-
 1241 tative baseline models (Transolver, GINO, RIGNO-18) on both time-dependent and time-independent
 1242 dataset categories, described in Table 1. Specifically, for each individual dataset, we calculate the
 1243 normalized scores for every model. The best-performing model is assigned a score of 1. The scores
 1244 for other models are calculated as the ratio of the best model's error to their respective errors:

$$S_{\text{norm}} = \frac{\text{error}_{\text{best}}}{\text{error}_{\text{model}}} \quad (\text{E.2})$$

1245 These dataset scores are then summed for each model to derive total scores for the time-dependent and
 1246 time-independent dataset categories, respectively, offering a complete view of model performance.

1247 **Robustness** To evaluate the consistency of model performance across different datasets, we intro-
 1248 duce a Robustness Score. This score is calculated for both the time-dependent and time-independent
 1249 categories of datasets. Leveraging the normalized scores obtained by each model on the individual
 1250 datasets within these categories, the Robustness Score for a model is defined as:

$$\text{Robustness Score} = \bar{S}_{\text{norm}} \times (1 - \text{CV}), \quad (\text{E.3})$$

1251 where \bar{S}_{norm} is the mean of the model's normalized scores across all datasets in a specific category
 1252 (either time-dependent or time-independent). The term CV represents the Coefficient of Variation of
 1253 these normalized scores, calculated as:

$$\text{CV} = \frac{\sigma_{S_{\text{norm}}}}{\bar{S}_{\text{norm}}}, \quad (\text{E.4})$$

1254 where $\sigma_{S_{\text{norm}}}$ is the standard deviation of the model's normalized scores within that same category. A
 1255 higher Robustness Score suggests that a model not only achieves high average performance (high
 1256 mean normalized score) but also exhibits less variability in its performance across the different
 1257 datasets within the category (low CV), indicating greater reliability. The robustness scores for GAOT,
 1258 Transolver, GINO and RIGNO-18 are shown in Table E.1.

Computational Efficiency To provide a comprehensive characterization of model performance and analyze the inherent accuracy-efficiency trade-off, we further evaluate the computational efficiency of the models during both training and inference phases.

Training efficiency is quantified by the *training throughput*, defined as the number of samples the model can process per second during training, encompassing the forward pass, backward pass, and gradient update. A high training throughput is indicative of a model’s ability to learn quickly from data. This is essential for handling large-scale datasets or developing large foundation models where training time can be a significant bottleneck. For measuring throughput, the batch size for each model was determined first by identifying the maximum value that could be run without encountering Out-of-Memory (OOM) errors on the target hardware. The actual batch size used for the throughput measurement was then set to approximately half of this maximum. This heuristic is based on the observation that peak throughput is often not achieved at the absolute maximum batch size, but rather at a point (frequently around half the maximum) where GPU resources, such as shared memory bandwidth, are optimally utilized, leading to the highest processing rates.

Inference efficiency is measured by the *inference latency*, which is the time taken for the model to perform a single forward pass on an individual sample (i.e., batch size of 1). Low inference latency is a critical attribute for the practical deployment of models, particularly in applications requiring real-time or near real-time predictions, such as in engineering simulations, interactive design tools, or control systems.

All computational efficiency metrics were benchmarked on the Bluff-Body dataset with one NVIDIA-4090 hardware. To ensure reliable and stable measurements, the GPU was warmed up prior to data collection, and each reported metric is the average of 100 repeated measurements.

E.2 Results for Radar Chart in Main Text.

Table E.1 presents the raw data used to generate the radar chart in Figure 1 of the main text. The metrics depicted in the radar chart include:

- **Accuracy (Acc. and Acc.(t)):** Overall accuracy on time-independent (Acc.) and time-dependent (Acc.(t)) datasets.
- **Robustness (Robust. and Robust.(t)):** Robustness on time-independent (Robust.) and time-dependent (Robust.(t)) datasets.
- **Training Throughput (Tput.(train)):** The number of samples processed per second during training.
- **Inference Latency (Infer. Latency):** The time (ms) taken for a single forward pass on one sample during inference.

The precise definitions and calculation methodologies for these metrics are detailed in Sec. E.1

Model	acc.(t)	acc.	Tput(train)	Infer Latency	Peak memory	InputScal.	ModelScal	robust(t)	robust
GAOT	7.45	6.30	97.5	6.966	101.7	68.12	48.7	0.80	0.77
Transolver	0	4.12	39.5	15.295	144.0	8.96	6.69	0	0.22
GINO	2.77	2.94	60.4	8.455	556.8	30.53	40.00	0.15	0.19
RIGNO-18	7.37	4.38	50.3	12.749	188.8	12.52	7.51	0.85	0.29

Table E.1: Data for Radar Chart

Table E.1 also includes **Peak Memory (MB)**, which records the peak GPU memory consumption of each model during inference with a batch size of 1. Although not visualized in the radar chart (Figure 1), the data indicates that GAOT exhibits the lowest peak memory usage among the compared models. Furthermore, Figure E.1 (a, c) illustrates the scaling of peak memory with increasing input grid size and model size, respectively. These plots demonstrate GAOT’s superior memory utilization capabilities.

The **Input Scalability** and **Model Scalability** scores presented in the radar chart are derived from the training throughput measured under specific conditions:

- Input Scalability is based on throughput at an input grid size of 50,000 points.
- Model Scalability is based on throughput for a model size of approximately 70 million parameters.

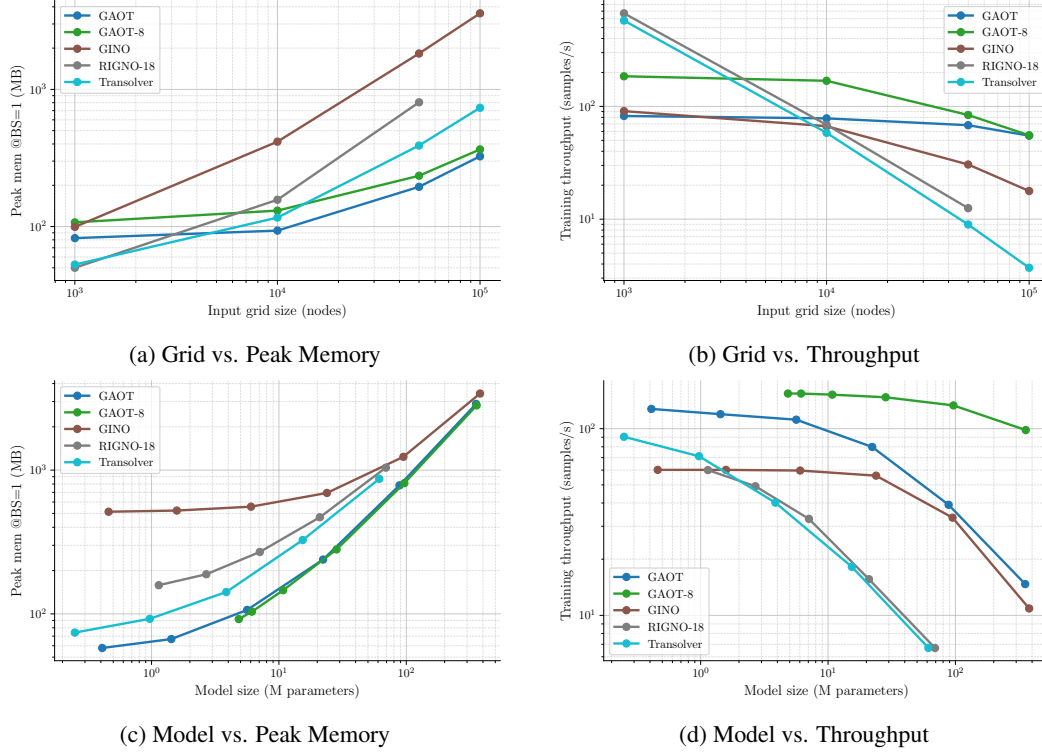


Figure E.1: Performance scaling comparisons across different metrics.

These particular evaluation points were chosen due to the performance limitations encountered with models like RIGNO-18 and Transolver on a single NVIDIA 4090 GPU, which prevented us from benchmarking them at larger scales. It is important to note that GAOT's architecture allows it to scale significantly beyond these tested limits.

SM Figure E.1 provides detailed scaling curves for both peak memory and training throughput as functions of input grid size and model size. To vary the model size for these comparisons, we systematically adjusted key architectural width parameters for each model:

- For Transolver, we scaled its hidden channel dimension through [64, 128, 256, 512, 1024].
- For RIGNO, the hidden channel dimensions of its node and edge functions were varied across [64, 128, 256, 512, 1024].
- For GINO, the hidden channel dimension of its FNO processor layers was selected from [16, 32, 64, 128, 256, 512].
- For our GAOT model, we scaled the hidden channel dimension of its attention layers using values from [64, 128, 256, 512, 1024, 2048], while the hidden dimension of its FFN layers was maintained at four times the attention layer's hidden dimension.

This figure also introduces results for GAOT-8, a variant of GAOT where the patch size in the transformer processor is set to 8 (the default GAOT employs a patch size of 2). As shown, GAOT-8 can achieve enhanced computational performance. Furthermore, as detailed in our ablation studies (Section F.2), this improvement in efficiency with GAOT-8 does not give rise to the substantial accuracy degradation.

E.3 Regular Grid Dataset

In addition to datasets with arbitrary point cloud geometries, we also evaluated the performance of our GAOT model on time-dependent PDE datasets where the inputs are provided on regular (structured)

1327 grids. The results for GAOT are compared against several baselines, including RIGNO (RIGNO-18
1328 and RIGNO-12), CNO, scOT, and FNO. The performance data for these baseline models are sourced
1329 from the original RIGNO paper [39].

1330 As demonstrated in Table E.2, GAOT also performs well on these structured grid datasets. Our
1331 model consistently ranks within the top two across six of the seven benchmark datasets, achieving
1332 the leading (first place) performance on five of them. This highlights GAOT’s robustness and strong
1333 generalization capabilities across different input discretizations.

Table E.2: Benchmarks with time-dependent datasets with regular grid inputs. Best and 2nd best models are shown in blue and orange fonts for each dataset.

Dataset	Median relative L^1 error [%]					
Structured	GAOT	RIGNO-18	RIGNO-12	CNO	scOT	FNO
NS-Gauss	2.29	2.74	3.78	10.9	2.92	14.41
NS-PwC	1.23	1.12	1.82	5.03	7.12	12.55
NS-SL	0.98	1.13	1.82	2.12	2.49	2.08
NS-SVS	0.46	0.56	0.75	0.70	1.01	7.52
CE-Gauss	5.28	5.47	7.56	22.0	9.44	28.69
CE-RP	4.98	3.49	4.43	18.4	9.74	38.48
Wave-Layer	5.40	6.75	8.97	8.28	13.44	28.13

1334 E.4 Model and Dataset Scaling

1335 **Model Size** To further investigate the scalability of our approach, we conduct an ablation study
1336 on how different model sizes affect performance. We focus on the two compressible Euler datasets,
1337 CE-Gauss and CE-RP, each with 1,024 training trajectories. We measure the final-time relative L^1
1338 error ($t = t_{14}$) and record the total number of parameters and per-epoch training time under various
1339 hyperparameter configurations.

1340 We vary the following components of our GAOT architecture as explained in the Tab B.2:

- 1341 • LC (Lifting Channels): The number of channels used during the encoder stage to project
1342 from the unstructured node features to latent tokens. Intuitively, a larger LC can preserve
1343 more local features when mapping from the input domain to the latent space.
- 1344 • TL (Transformer Layers): The depth of the transformer-based processor. Increasing TL
1345 typically increases modeling capacity for global interactions.
- 1346 • THS (Transformer Hidden Size): The hidden dimension of each self-attention block. A
1347 larger THS can capture richer representations.
- 1348 • FFN (Feed-Forward Network Size): The hidden dimension inside the FFN sub-layer, which
1349 we set to $4 \times$ THS following standard vision transformer practice.

1350 Table E.3 summarizes the performance across a range of these hyperparameters. We also record the
1351 total number of trainable parameters (in millions) and the approximate epoch time (in seconds) on
1352 one NVIDIA 4090 GPU with a batch size of 64. Here, all experiments are done with patch size equal
1353 to 2.

1354 From the top block of Table E.3 (rows 1–4), we observe that as we increase THS from 32 to 256
1355 (keeping TL=5 and LC=32), the final-time errors on both CE-Gauss and CE-RP decrease significantly.
1356 For example, on CE-Gauss, the error drops from 48.4% down to 6.88%. This trend reflects the
1357 transformer’s ability to scale with hidden dimension. The training time per epoch grows from roughly
1358 84 seconds to 143 seconds. While performance improves, larger THS demands more computational
1359 resources.

1360 Next, we fix (TL,THS)=(5,256) and vary LC in the middle block (rows 5–7). Setting LC=32
1361 consistently achieves strong results. Lowering LC to 16 slightly degrades performance, while
1362 pushing LC to 64 or 128 yields only marginal gains. Hence, LC=32 appears sufficient to capture the
1363 encoder-level geometry information.

1364 Finally, the bottom block (rows 8–10) examines the effect of transformer layers TL from 1 up to 10.
1365 With TL=1, errors remain quite high (25.0% on CE-Gauss); adding layers substantially reduces error

Table E.3: Relative L^1 test errors at $t = t_{14}$ with different architectural hyperparameters. Time refers to training time with batch size equals to 64 on 1 NVIDIA-4090 GPU, and the patch size is set to 2. The size of training trajectories is 1024.

Model size		Hyperparameters				Median relative L^1 error [%]	
Parameters [M]	Time [s]	LC	TL	THS	FFN	CE-Gauss	CE-RP
0.14	84	32	5	32	128	48.4	26.5
0.41	89	32	5	64	256	13.2	12.0
1.42	100	32	5	128	512	9.17	7.90
5.6	143	32	5	256	1024	6.88	5.28
5.5	142	16	5	256	1024	7.97	5.94
5.6	154	64	5	256	1024	6.94	5.18
6.1	181	128	5	256	1024	7.33	5.20
1.16	50	32	1	256	1024	25.0	14.5
3.39	98	32	3	256	1024	9.00	6.80
11.2	260	32	10	256	1024	5.28	5.35

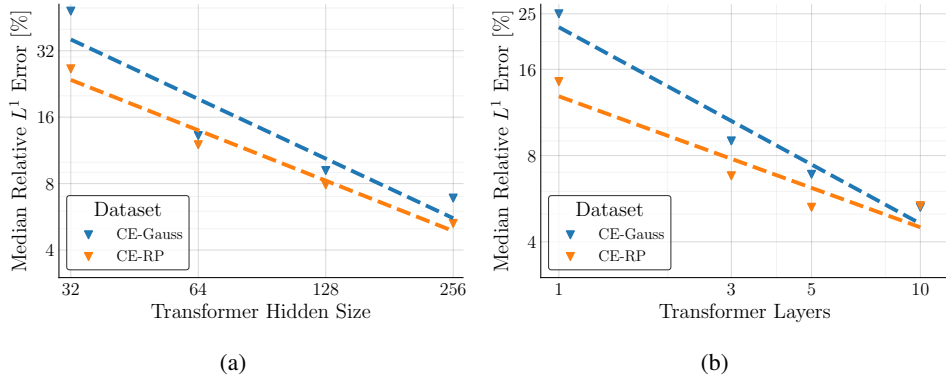


Figure E.2: Relative median L^1 test errors at $t = t_{14}$ with different strategies for scaling model sizes. The x-axis in the left plot corresponds to all the following hyperparameters: THS and TL.

1366 to 9.0% at $TL = 3$, and ultimately down to 5.28% at $TL=10$ on CE-Gauss. Increasing TL to 10 also
1367 expands the parameter count to 11.2M, nearly doubling the training time per epoch (260s).

1368 Figure E.2 illustrates how errors decrease as we scale THS or TL, while Figure E.3 shows the effect
1369 of changing LC or the total parameter count. The largest model tested reaches 11.2M parameters and
1370 attains around 5% error on CE-Gauss, demonstrating the potential to improve accuracy by investing
1371 in more computational resources.

1372 **Data Size** So far, we have discussed how increasing model size affects accuracy. In this subsection,
1373 we turn our attention to data scaling: we examine how the learned operator’s performance changes as
1374 the number of training samples (trajectories or static solutions) grows. Figure E.4 illustrates two sets
1375 of experiments:

1376 **(a) Time-Dependent (Fluid) Datasets.** We plot the final-time ($t = t_{14}$) error on multiple fluid PDE
1377 benchmarks as a function of the training set size $\{128, 256, 512, 1024\}$. All of these datasets use
1378 partial-grid subsampling. The results confirm that as we increase the number of training trajectories,
1379 errors consistently drop across all fluid datasets, often in a near-linear fashion with respect to the
1380 number of training trajectories. This trend highlights the model’s capacity to benefit from additional
1381 time-series diversity.

1382 **(b) Time-Independent (Static) Datasets.** We similarly measure how the final solution error decreases
1383 when expanding the dataset size to $\{128, 256, 512, 1024, 2048\}$ for three static PDE tasks: Poisson-
1384 Gauss, Poisson-C-Sines, and Elasticity. Note that elasticity is limited to at most 1024 samples due to

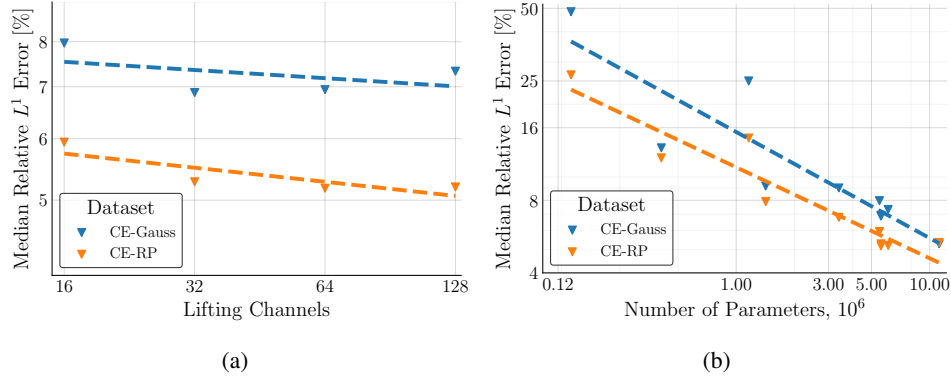


Figure E.3: Relative L^1 test errors at $t = t_{14}$ with different strategies for scaling model sizes. The x-axis in the left plot corresponds to all the following hyperparameters: LC, and the total number of trainable parameters.

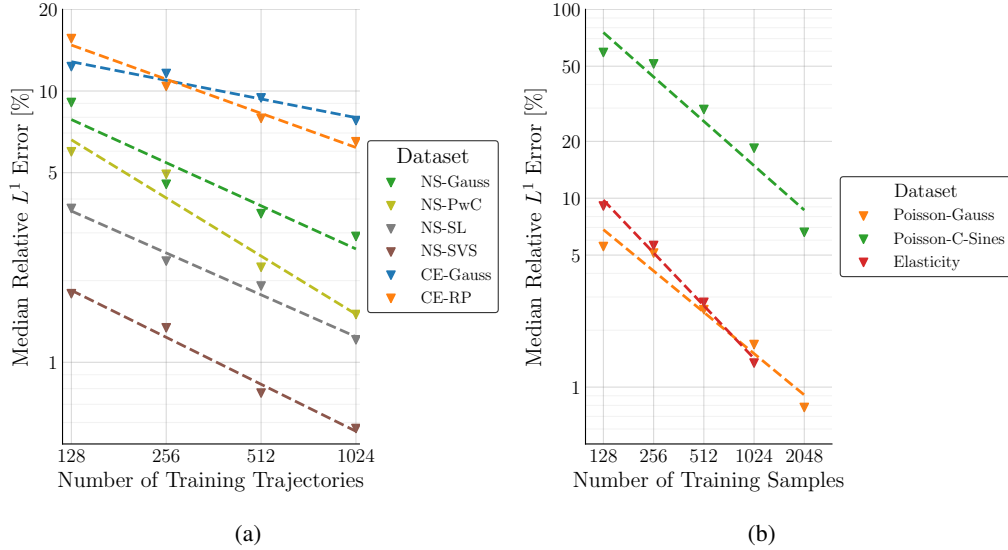


Figure E.4: Relative L^1 test errors against the size of training dataset. The left plot shows these results for the time-dependent fluid datasets, and the right plot for time-independent datasets. The lines show linear regression slopes for each dataset.

1385 data availability. Across all these static problems, we observe a consistent downward slope in error as
 1386 the number of samples increases, again underscoring the advantage of larger training sets.

1387 Overall, in both dynamic (time-dependent) and static (time-independent) scenarios, GAOT exhibits a
 1388 scalable relationship between training set size and error reduction. As the training data grows, the
 1389 learned operator converges more reliably to the underlying PDE solution. This robust data scaling
 1390 property supports our premise that GAOT can serve as a strong foundation model backbone for PDE
 1391 tasks, becoming increasingly accurate with more extensive datasets.

1392 E.5 Resolution Invariance

1393 One of the core properties for operator learning is resolution invariance—the ability to train on a specific
 1394 discretization yet accurately predict solutions at higher/lower resolutions. To validate this property in
 1395 our GAOT framework, we conduct experiments on a time-independent PDE, Poisson-Gauss.

1396 Figure E.5 illustrates the resolution invariance capabilities of GAOT. In this experiment, GAOT was
 1397 trained using data discretized by 2048 points. Its performance was then evaluated across a spectrum

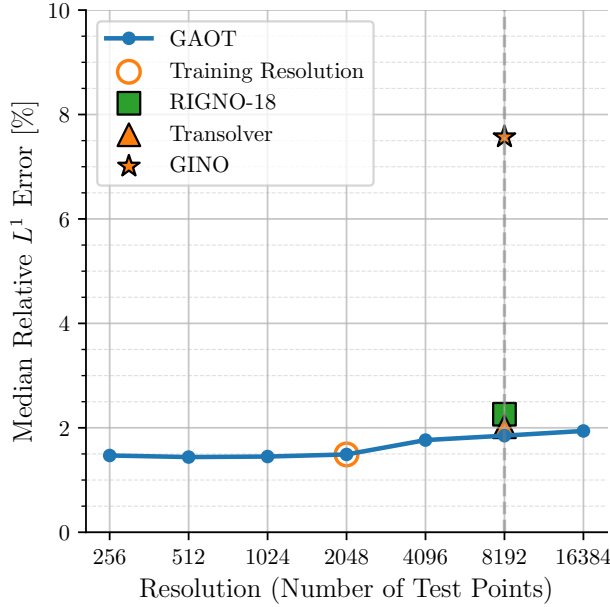


Figure E.5: The GAOT model is trained at a resolution of 2048 and evaluate at various test resolutions. The results for RIGNO-18, Transolver and GINO correspond to models trained and tested at a resolution of 8192.

of seven distinct resolutions: three sub-resolution settings (256, 512, and 1024 points), the training resolution itself (2048 points), and three super-resolution settings (4096, 8192, and 16384 points). For comparative purposes, Figure E.5 also displays the performance of the top three baseline models from Table 1 (excluding GAOT itself), namely RIGNO-18, Transolver, and GINO. These baseline results were obtained from models that were both trained and tested at a resolution of 8192 points.

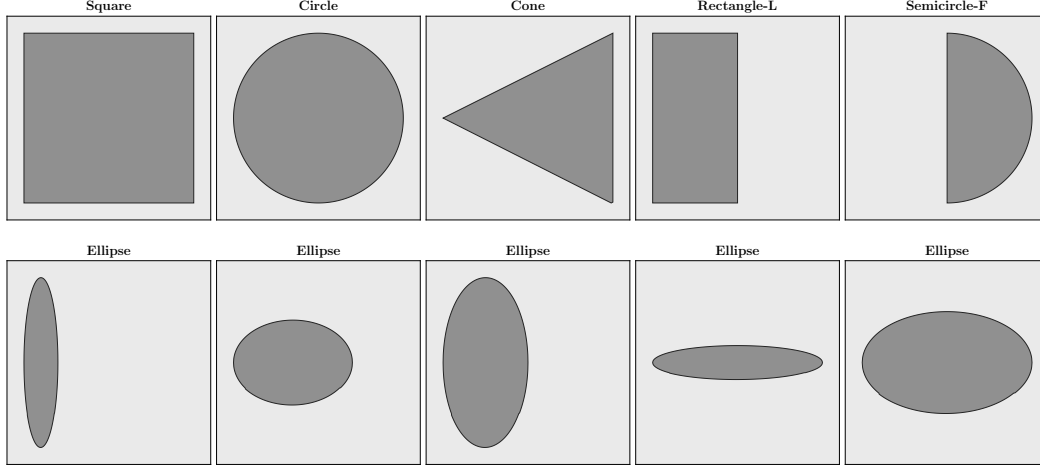
The results demonstrate that GAOT possesses excellent resolution invariance. Notably, even when trained at a resolution of 2048 points, GAOT not only generalizes well to higher resolutions but also achieves the best performance when tested at 8192 points. It outperforms the baseline models which were specifically trained for and tested at this higher resolution (8192 points), underscoring GAOT’s efficiency and robustness in learning resolution-independent solution operators.

E.6 Transfer Learning

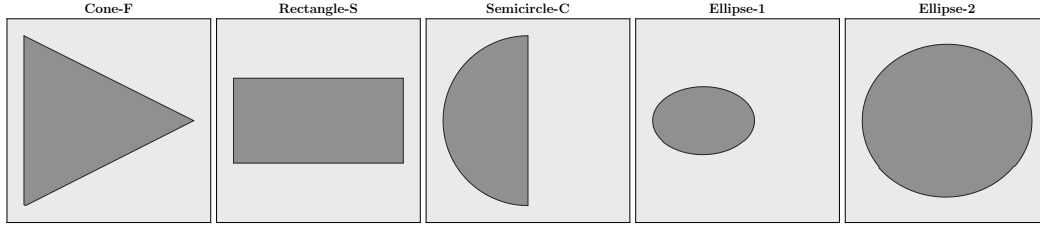
The set of geometries illustrated in Figure E.6a, represent the varying bluff-body geometries in the Bluff-Body dataset, which is one of the benchmarks presented in Tab. 1 of the main text. This dataset is constructed by simulating compressible flow across diverse bluff-body geometries at varying Ma and α , as described in Sec. D.2. The distinct shapes depicted in Figure E.6b were specifically employed for the fine-tuning stage of our transfer learning experiments. The corresponding transfer learning performance, demonstrating the model’s ability to adapt from the shapes used for pretraining to the novel ones indicated in Figure E.6b, is presented in Figure 3(c) in the main text.

E.7 Training Randomness

In order to quantify the dependence of the final model performance on the inherent randomness in the training process, such as in weight initialization, we trained the GAOT model six independent times. Each training run utilized a different seed for the pseudo-random number generator. These experiments were conducted on the Bluff-Body dataset. The statistics of the resulting relative L^1 test errors across these six runs are summarized in Table E.4. The standard deviation of these errors is 0.12. This relatively small standard deviation suggests that the GAOT model exhibits good stability with respect to the random aspects of the training procedure.



(a) Shapes utilized for pretraining in the transfer learning experiments.



(b) Bluff body shapes employed for the fine-tuning (FT) phase of the transfer learning task.

Figure E.6: The geometries in (a) are included in the Bluff-Body dataset in Tab. 1. Shape-* (*: C, F, S, L; Shape: Semicircle, Cone, Rectangle) indicates shapes and their contact surfaces (*) with respect to the flow. Here, F - flat surface, C - curved surface, L - larger side, S - smaller side.

Table E.4: Statistics of relative L^1 test errors with different random seeds. We train GAOT on Bluff-Body dataset, which is repeated 6 times.

Dataset	Error [%]
	Mean \pm Standard deviation
Bluff-Body	2.39 ± 0.12

F Ablation Studies

F.1 Encode-Process-Decode

In this subsection, we investigate the performance of four different *encode-process-decode* architectures on both time-dependent and time-independent PDE benchmarks. The four models considered are GAOT (ours), Regional Attentional Neural Operator (RANO), Regional Fourier Neural Operator (RFNO) and GINO [29], with components are *Message-Passing (MP)* graph neural network [16], *Transformer* [49], *Fourier Neural Operator (FNO)* [27], *Graph Neural Operator (GNO)* [28] and proposed *Multi-scal Attentional GNO (MAGNO)*. Table F.1 summarizes the components of each model. All variants follow an *encode-process-decode* pipeline but differ in how graph, Fourier, or transformer-based mechanisms are deployed.

Figure F.1 shows the *median relative L^1 error* for each model on six PDE datasets (4 time-dependent PDEs, 2 time-independent PDEs). All models are trained for 500 epochs under the same data splits and hyperparameter conditions. We can see that GAOT consistently achieves strong performance and robustness across all six datasets. Its errors remain low, highlighting the effectiveness of combining MAGNO for local geometric encoding with transformer-based global attention. GINO ranks second

Model	Encode	Process	Decode
GAOT	MAGNO	Transformer	MAGNO
RANO	MP	Transformer	MP
RFNO	MP	FNO	MP
GINO	GNO	FNO	GNO

Table F.1: Components for different encode–process–decode designs.

in overall accuracy, yet exhibits noticeable difficulties on NS-Gauss and Poisson-C-Sines. RANO and RFNO perform moderately well on simpler datasets (e.g. Elasticity), but show instability on more challenging tasks (e.g. NS-SVS or NS-Gauss). This indicates that reliance on message-passing or FNO-based processors alone may not be sufficient to handle diverse PDE and geometry conditions with the same level of robustness.

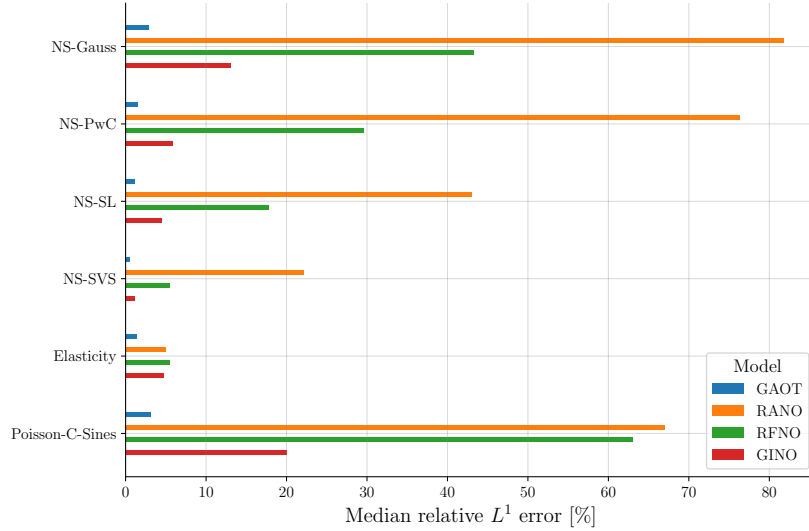


Figure F.1: Median relative L^1 errors (%) of GAOT, RANO, RFNO, and GINO on six PDE benchmarks.

Overall, these results reinforce GAOT’s stability across multiple PDE settings. Even with a fixed training protocol (500 epochs for each dataset), GAOT consistently converges faster and more reliably, underscoring the advantage of geometry-aware tokens, multiscale attention, and the flexible transformer backbone.

F.2 Tokenization Strategies

In Section B.1, we have introduced three tokenization methods. Here, we compare these strategies on two datasets, Elasticity and Poisson-C-Sines. Figure F.2 shows the final median relative L^1 errors for each approach. The Strategy I consistently achieves the best performance on both unstructured datasets. Strategy II & III perform similarly to Strategy I on simpler datasets (elasticity), but can fail to converge on the more challenging one, Poisson-C-Sines. Similar situations also happen on models like UPT and GNOT in Tab. 1 of main text. Overall, Strategy I emerges as the most robust approach in our current experiments. While Strategies II and III show promise, they require more careful optimization to match Strategy I’s reliability.

Next, we focus on Strategy I and study how varying the *number of latent tokens (LT)*, *patch size (PS)*, and *radius (GR)* affect performance. Note that here we do not use multiscale radii; each token has a single radius. Table F.2 summarizes experiments on the elasticity and Poisson-Gauss datasets. Results show that fewer tokens (e.g., [32,32]) can degrade performance in some cases (elasticity), presumably because the domain coverage becomes coarser, making it harder to capture local variations. More tokens ([64,64] or [128,128]) typically improve accuracy and stabilize convergence. Nevertheless, computational costs rise when the number of tokens grows, as transformer

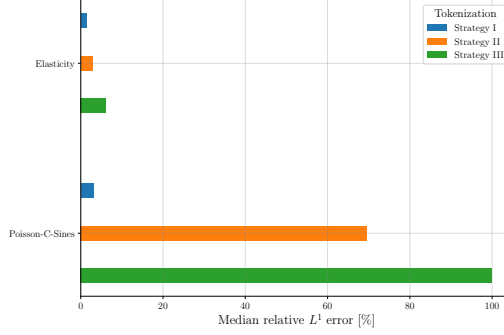


Figure F.2: Median relative L^1 errors (%) comparing three tokenization strategies on Elasticity, and Poisson-C-Sines. The Strategy I, II, III corresponds to the methods discussed in Section B.1.

Table F.2: Median relative L^1 errors (%), parameter counts, and training time with different numbers of latent tokens (LT), patch sizes (PS), and radii (GR).

Model Size		Hyperparameters			Median Relative L^1 Error [%]	
Params [M]	Time [s/it]	LT	PS	GR	Elasticity	Poisson-Gauss
5.60	10.1	[64, 64]	2	0.033	1.80	1.05
6.00	3.06	[64, 64]	4	0.033	1.71	1.57
10.7	2.20	[64, 64]	8	0.033	1.60	1.65
5.56	10.1	[32, 32]	1	0.066	3.41	1.22
5.60	3.00	[32, 32]	2	0.066	2.25	1.22
6.00	11.8	[128, 128]	4	0.033	1.67	1.72
10.7	5.02	[128, 128]	8	0.033	1.62	1.22

attention scales quadratically with token count. Increasing the patch size (PS) reduces the number of tokens entering the transformer, lowering the training time. Encouragingly, performance does not degrade sharply with larger patches. For instance, going from $PS = 2$ to 8 is fairly stable across datasets. Note that the overall parameter count can increase if each token aggregates larger local features, but in practice, training runs faster due to fewer tokens in self-attention. Radius (GR) grows if we reduce the number of latent tokens because we need to ensure coverage of the entire physical domain by enlarging the receptive field. This is critical for unstructured or irregular samples, especially if tokens must capture a bigger subregion.

F.3 Time-Stepping Method

We now investigate how different *time-stepping* formulations (see Section B.5) affect performance on time-dependent PDEs. Specifically, we compare the output, residual, and derivative stepping strategies. Table F.3 reports the median relative L^1 errors for six representative fluid dynamics benchmarks on regular grids.

Table F.3: Median relative L^1 errors (%) at final time t_{14} for GAOT with three different time-stepping methods.

Dataset	Median relative L^1 error [%]		
	Output	Residual	Derivative
NS-Gauss	3.57	3.60	2.52
NS-PwC	1.95	1.70	1.23
NS-SL	1.78	1.49	1.29
NS-SVS	0.60	0.60	0.56
CE-Gauss	8.80	8.93	7.97
CE-RP	5.17	6.12	5.94

As shown, modeling the operator as a time derivative (derivative column) often yields the lowest final-time errors on all but one dataset (CE-RP, where the *Output* strategy slightly outperforms the others). We hypothesize that treating the operator as $\partial_t u$ naturally enforces a continuous dependence on time, analogous to neural ODEs or residual networks [19, 9], which can improve stability and accuracy over multiple steps. In experiments involving time-dependent PDEs, we therefore use derivative time stepping as the default unless stated otherwise. This approach not only achieves strong final-time accuracy, but also aligns with our design goal of a differentiable, time-continuous operator.

F.4 Geometric Embedding

As discussed in Section B.3, our framework incorporates a *geometric embedding network* to encode shape and domain information separately from the physical (PDE) state. Table F.4 compares these geometric embedding approaches against a baseline "original" (i.e., no additional geometry embedding) on two original unstructured datasets (Wave-C-Sines, Poisson-C-Sines).

Table F.4: Median relative L^1 errors (%) for various geometry embedding approaches. Original omits geometric embedding, while Statistical and PointNet follow Section B.3.

Dataset	Median relative L^1 error [%]		
	original	statistical	pointnet
Wave-C-Sines	6.50	5.69	6.07
Poisson-C-Sines	6.60	4.66	23.7

In the unstructured datasets, including Wave-C-Sines, and Poisson-C-Sines, explicitly encoding domain geometry yields a more pronounced benefit. In particular, the statistical strategy consistently outperforms PointNet on these irregular meshes, and in Poisson-C-Sines, training with the PointNet approach appears unstable (23.7% error). Based on these observations, we use statistical embedding by default for unstructured dataset given its stable and superior performance in most cases.

F.5 multiscale Features

As introduced in Section B.2, our encoder can capture multiscale local information by aggregating neighborhood features across multiple radii. Specifically, we compare:

- Single-scale: using a single fixed radius of 0.033 for each point.
- multiscale: using three radii [0.022, 0.033, 0.044] for each point.

Table F.5: Median relative L^1 errors (%) comparing single-scale vs. multiscale features.

Dataset	Median relative L^1 error [%]	
	Single-scale	multiscale
Wave-C-Sines	5.69	4.6
Poisson-C-Sines	4.66	3.04

Table F.5 reports the mean relative L^1 errors on unstructured datasets including Wave-C-Sines and Poisson-C-Sines. Results show that multiscale neighbors yield a clear reduction in error. For instance, in Poisson-C-Sines, the error decreases from 4.66% to 3.04%. This contrast reflects the fact that a single, fixed receptive field on a regularly spaced grid is often sufficient. However, on unstructured domains where the mesh density can vary, using multiple radii helps the network capture both fine and coarse local structures.

G Visualizations of Datasets

Estimates produced by trained models are visualized in this section for different datasets.

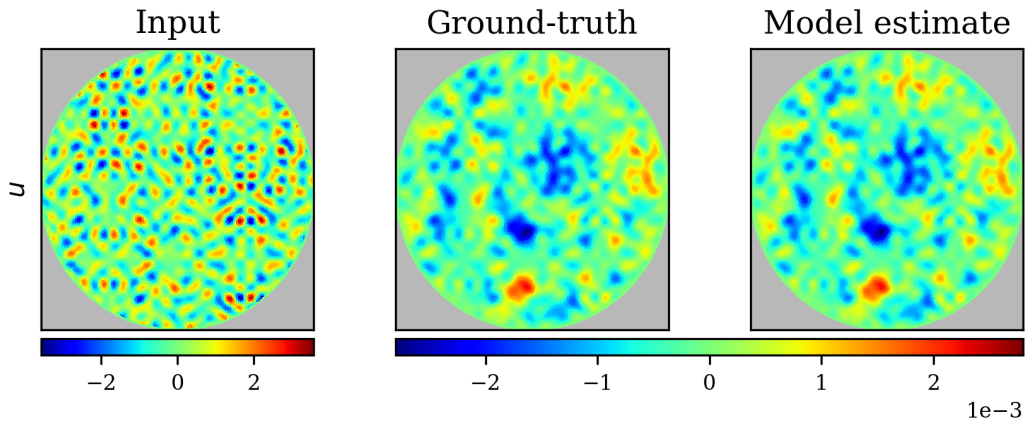


Figure G.1: Model input, ground-truth solution, and model estimate of a test sample of the Poisson-C-Sines dataset.

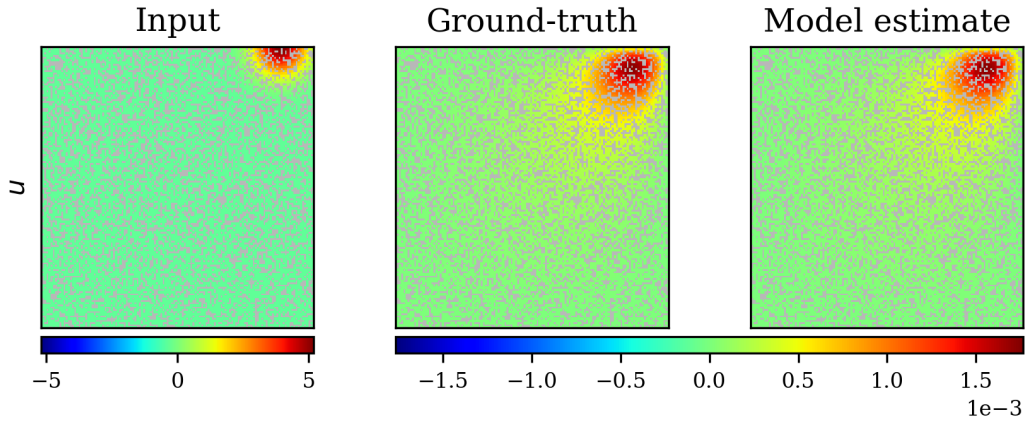


Figure G.2: Model input, ground-truth solution, and model estimate of a test sample of the Poisson-Gauss dataset.

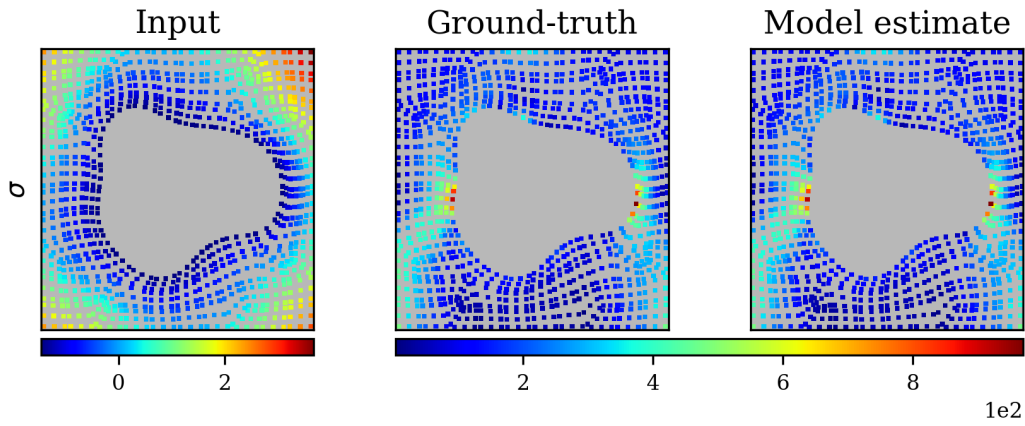


Figure G.3: Model input, ground-truth solution, and model estimate of a test sample of the Elasticity dataset.

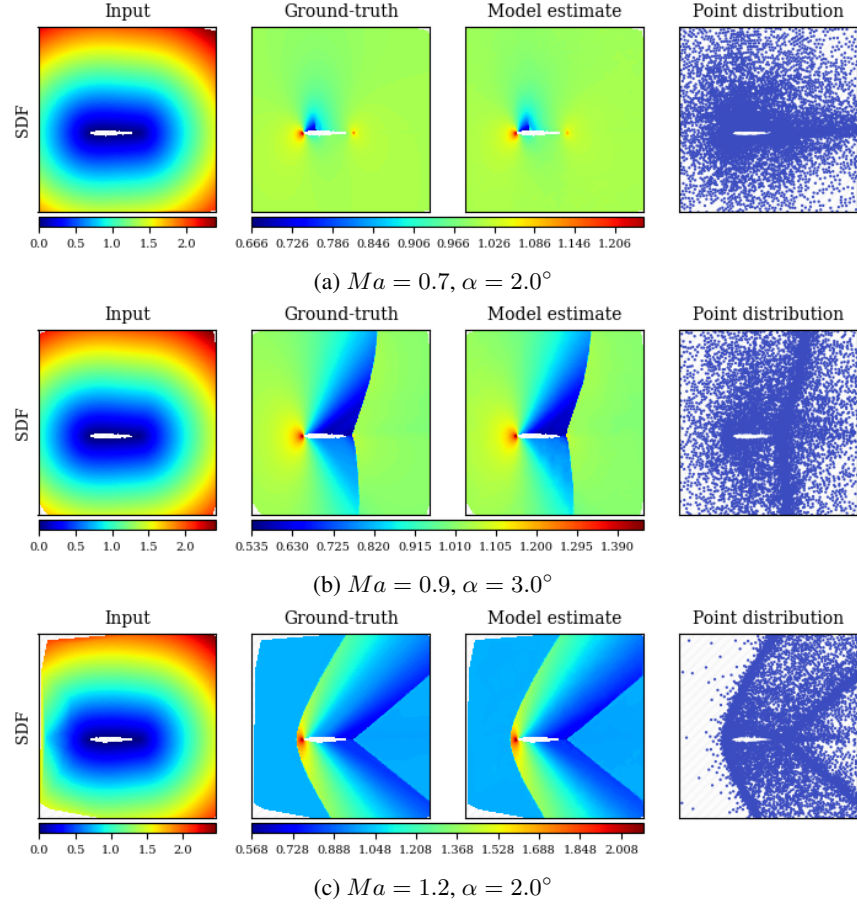


Figure G.4: Model input, ground-truth solution, model estimate and point distribution of test samples of the NACA0012 dataset.

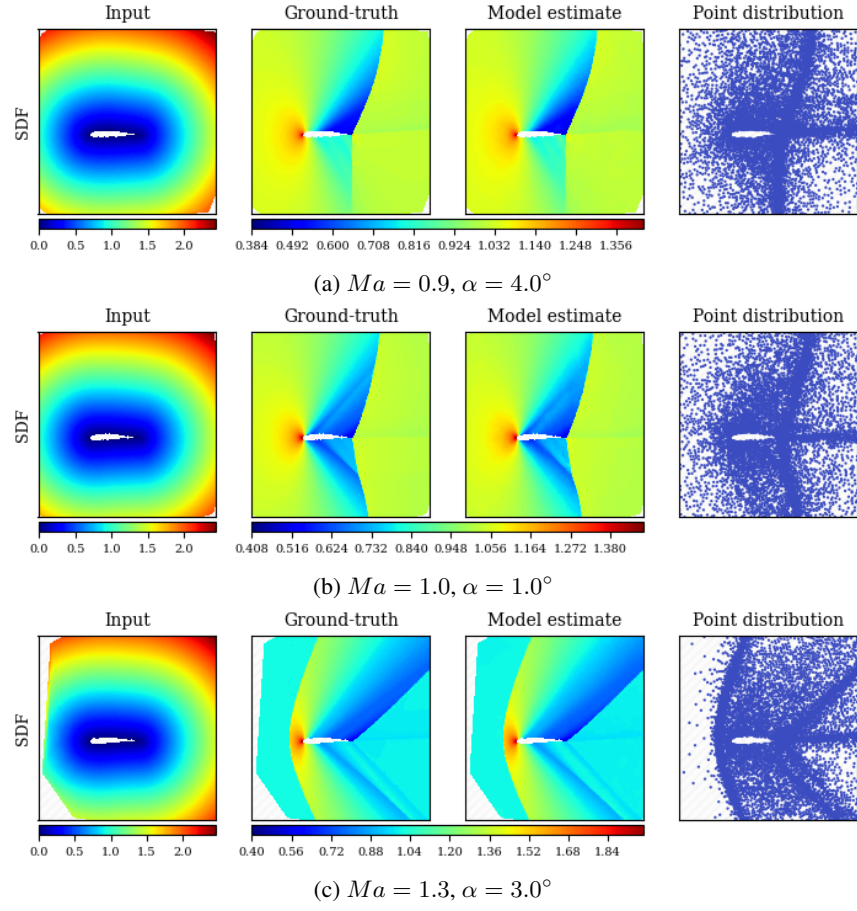


Figure G.5: Model input, ground-truth solution, model estimate and point distribution of test samples of the NACA2412 dataset.

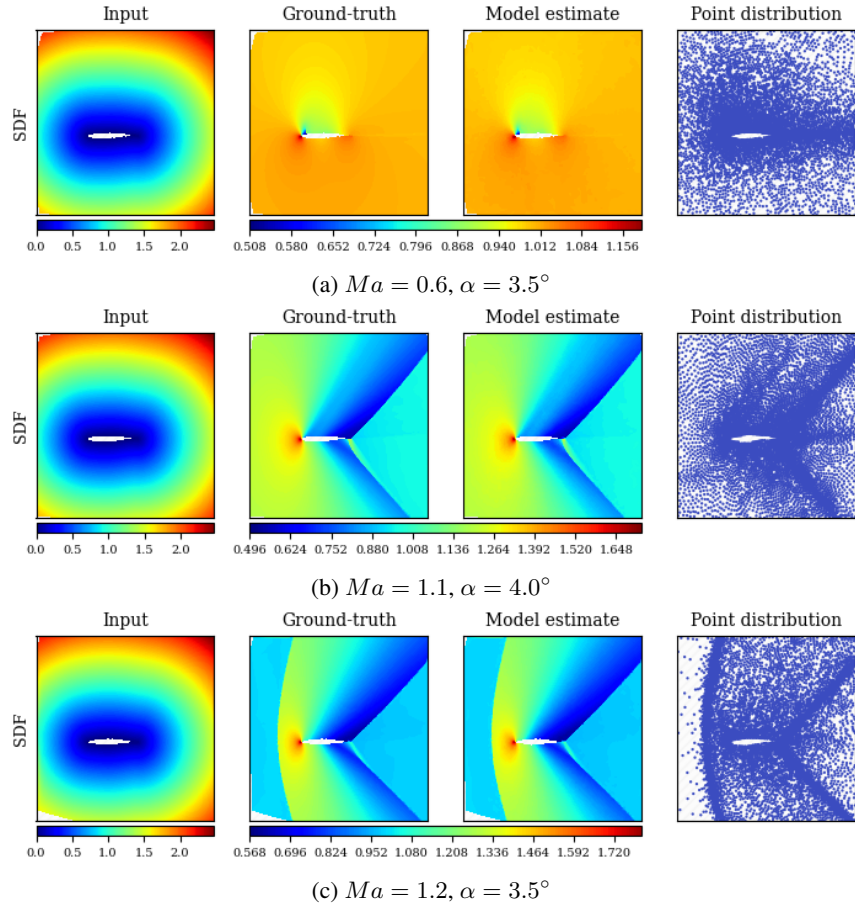


Figure G.6: Model input, ground-truth solution, model estimate and point distribution of test samples of the RAE2822 dataset.

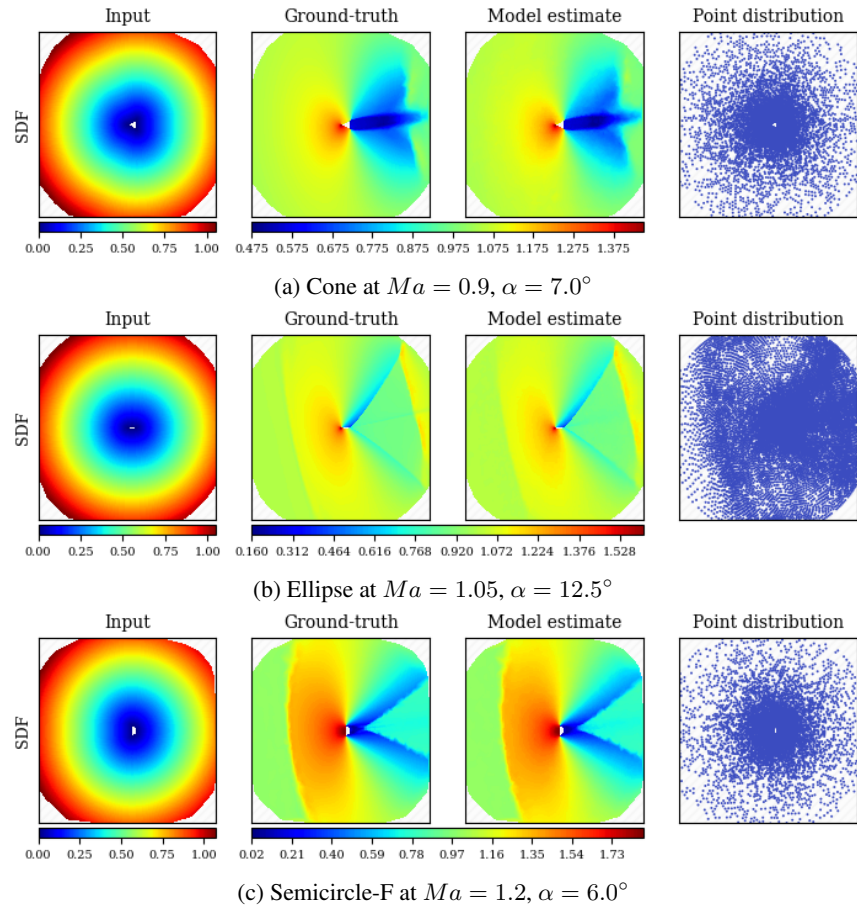


Figure G.7: Model input, ground-truth solution, model estimate and point distribution of test samples of the Bluff-Body dataset.

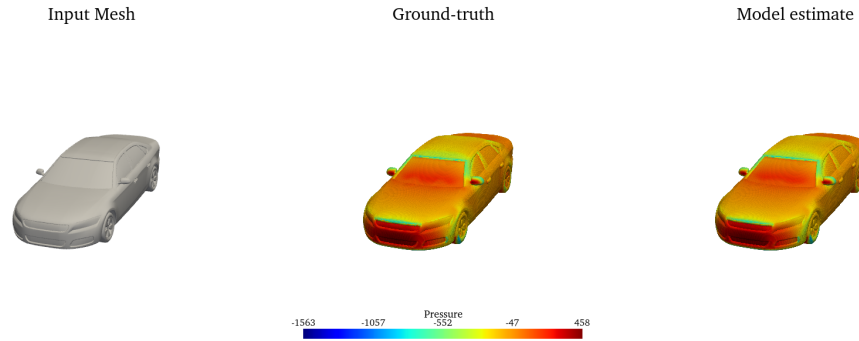


Figure G.8: Model input, ground-truth solution, model estimate of a test sample N_S_WWS_WM_172 of the surface pressure on the DrivAerNet++.

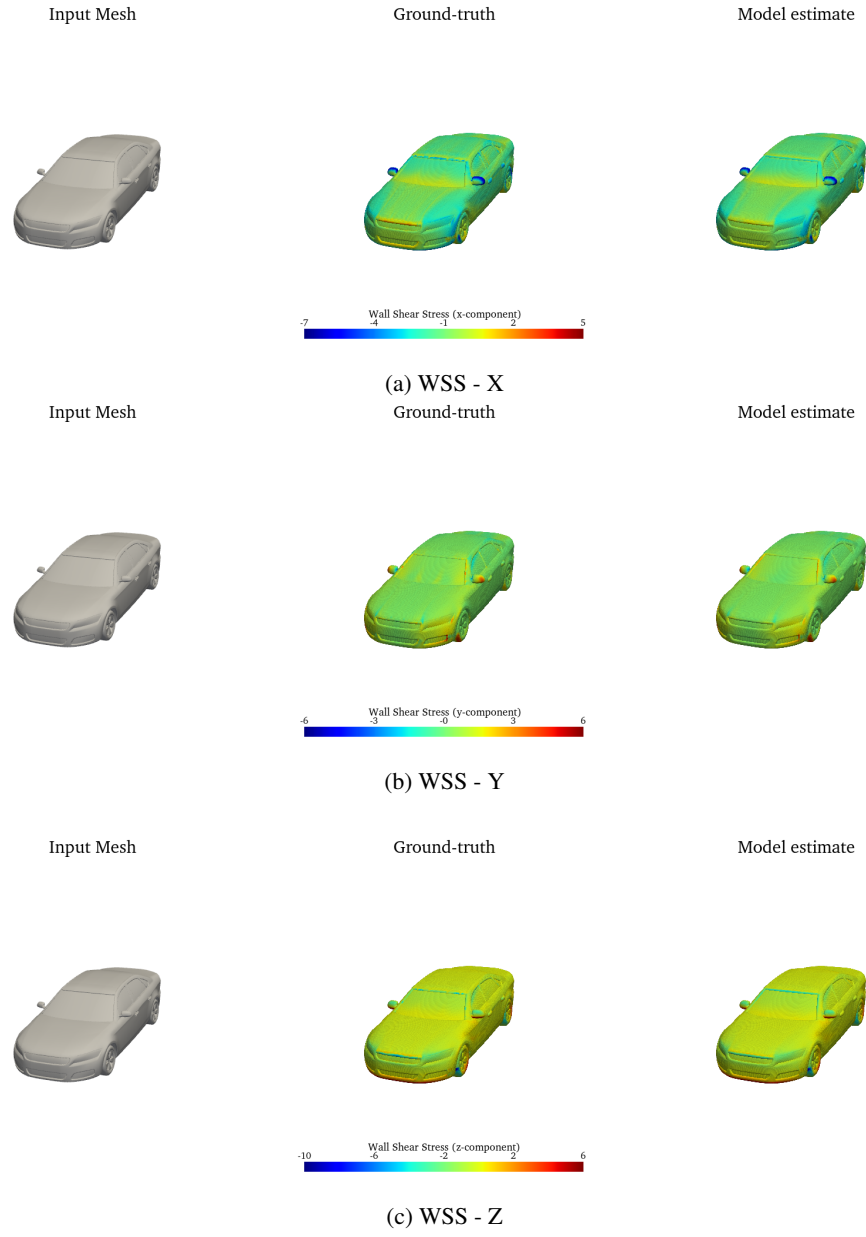


Figure G.9: Model input, ground-truth solution, model estimate of a test sample N_S_WWS_WM_172 of the surface wall shear stress on the DrivAerNet++.

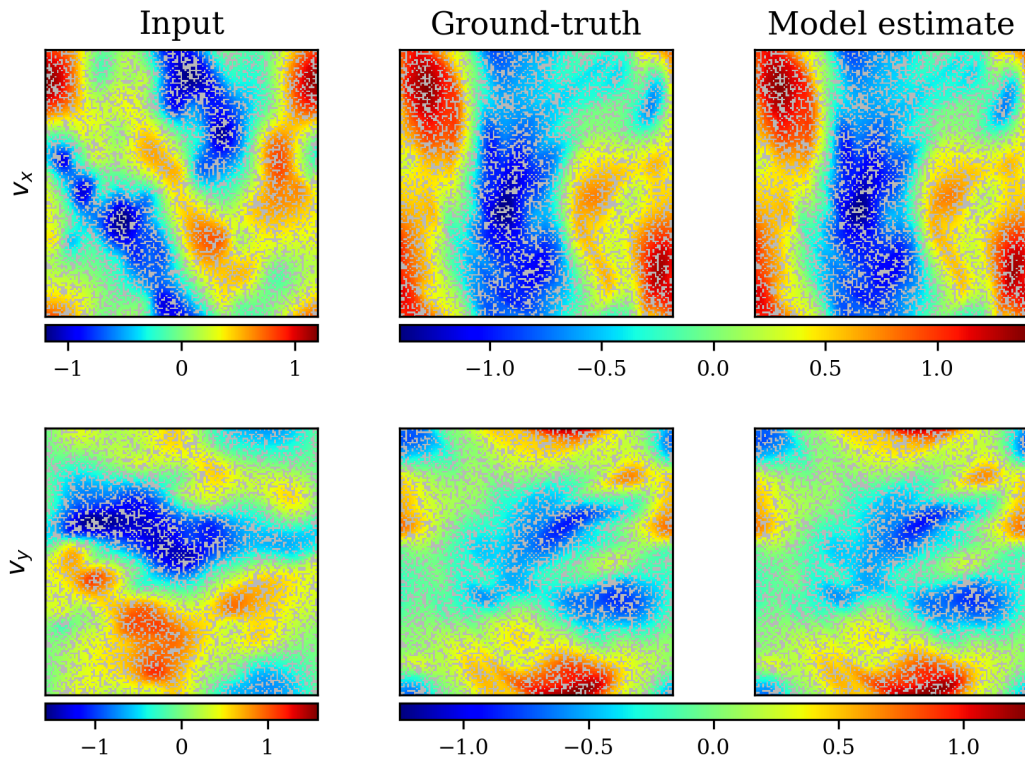


Figure G.10: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured NS-Gauss dataset.

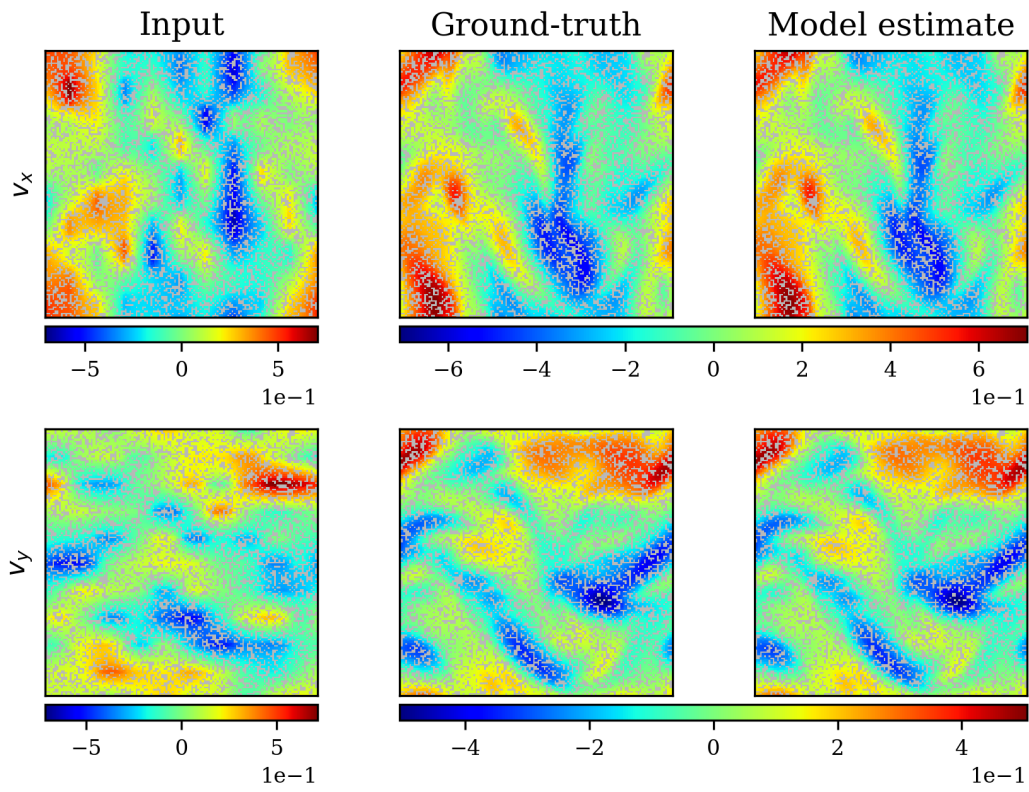


Figure G.11: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured NS-PwC dataset.

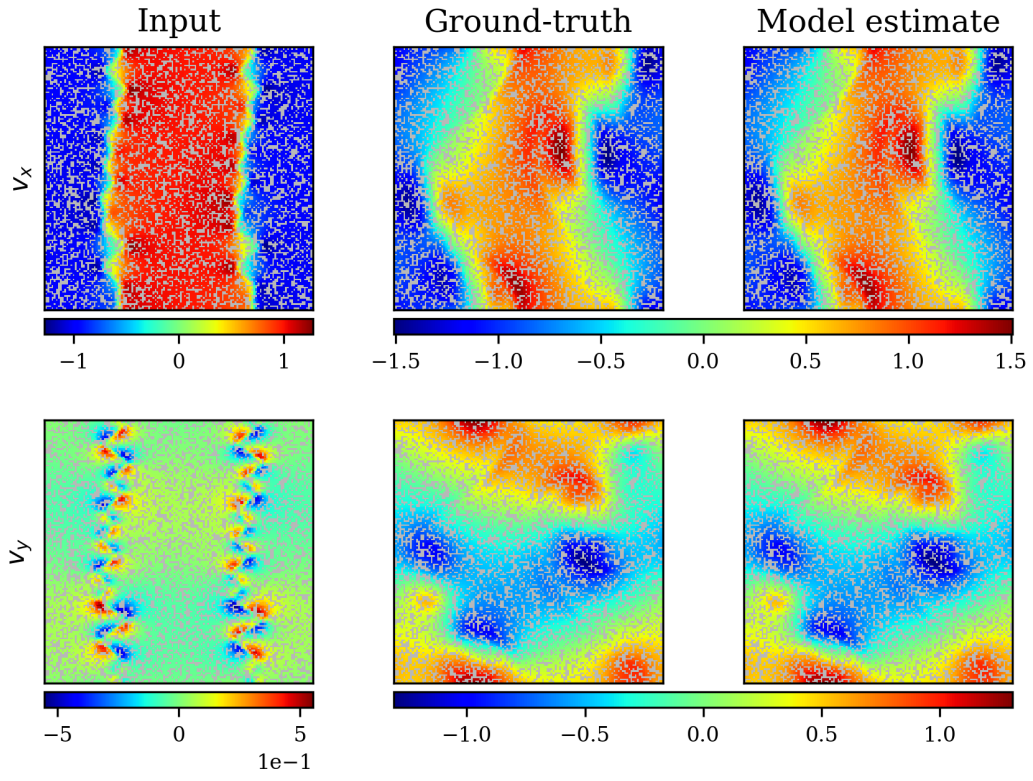


Figure G.12: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured NS-SL dataset.

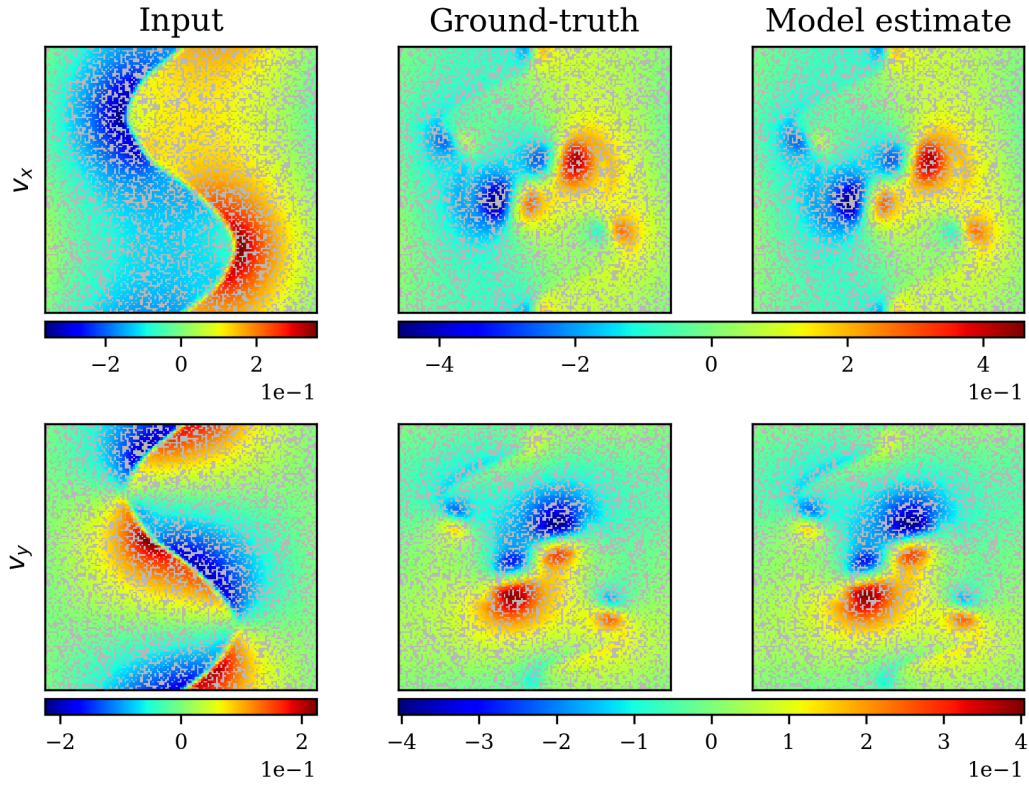


Figure G.13: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured NS-SVS dataset.

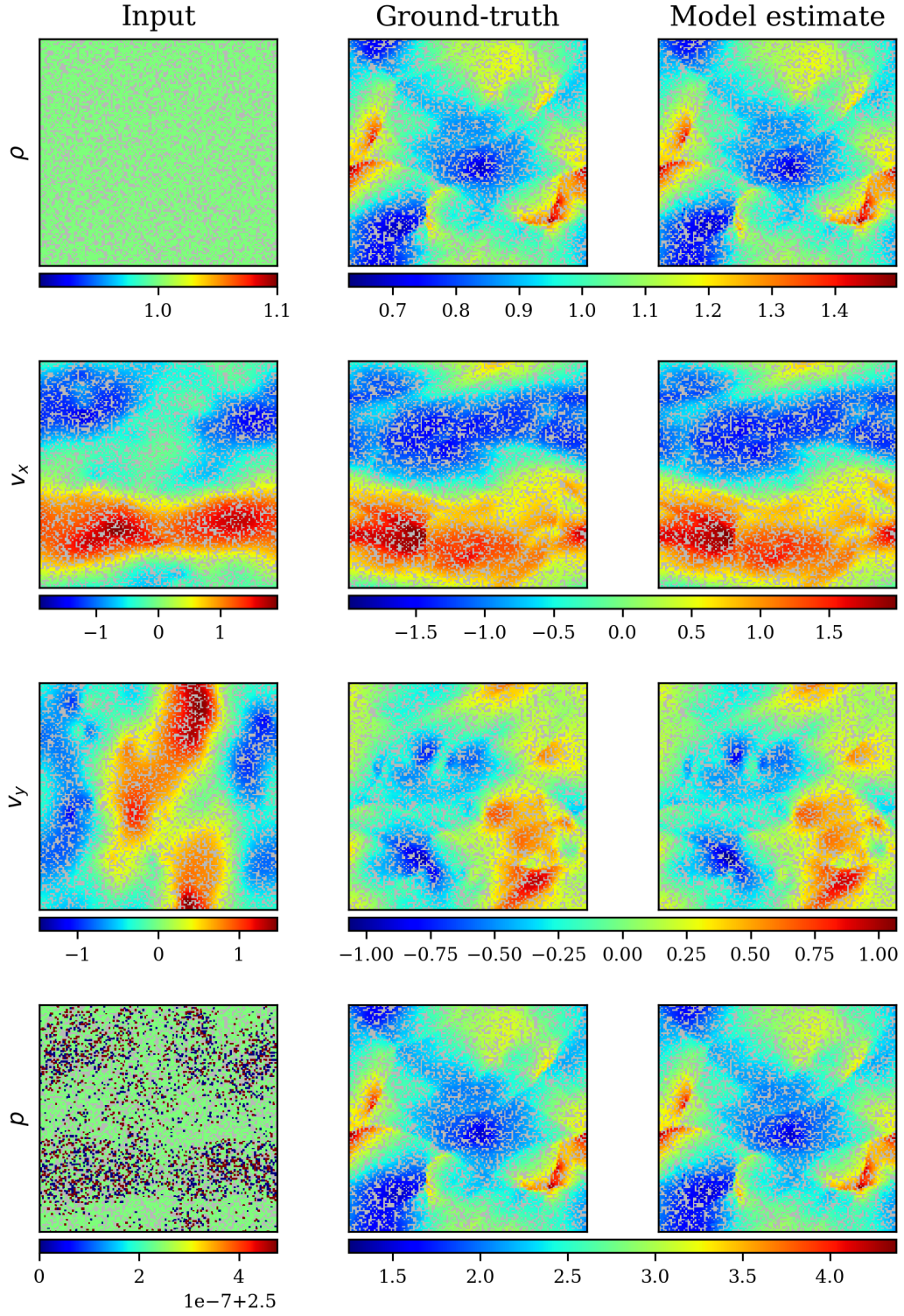


Figure G.14: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured CE-Gauss dataset.

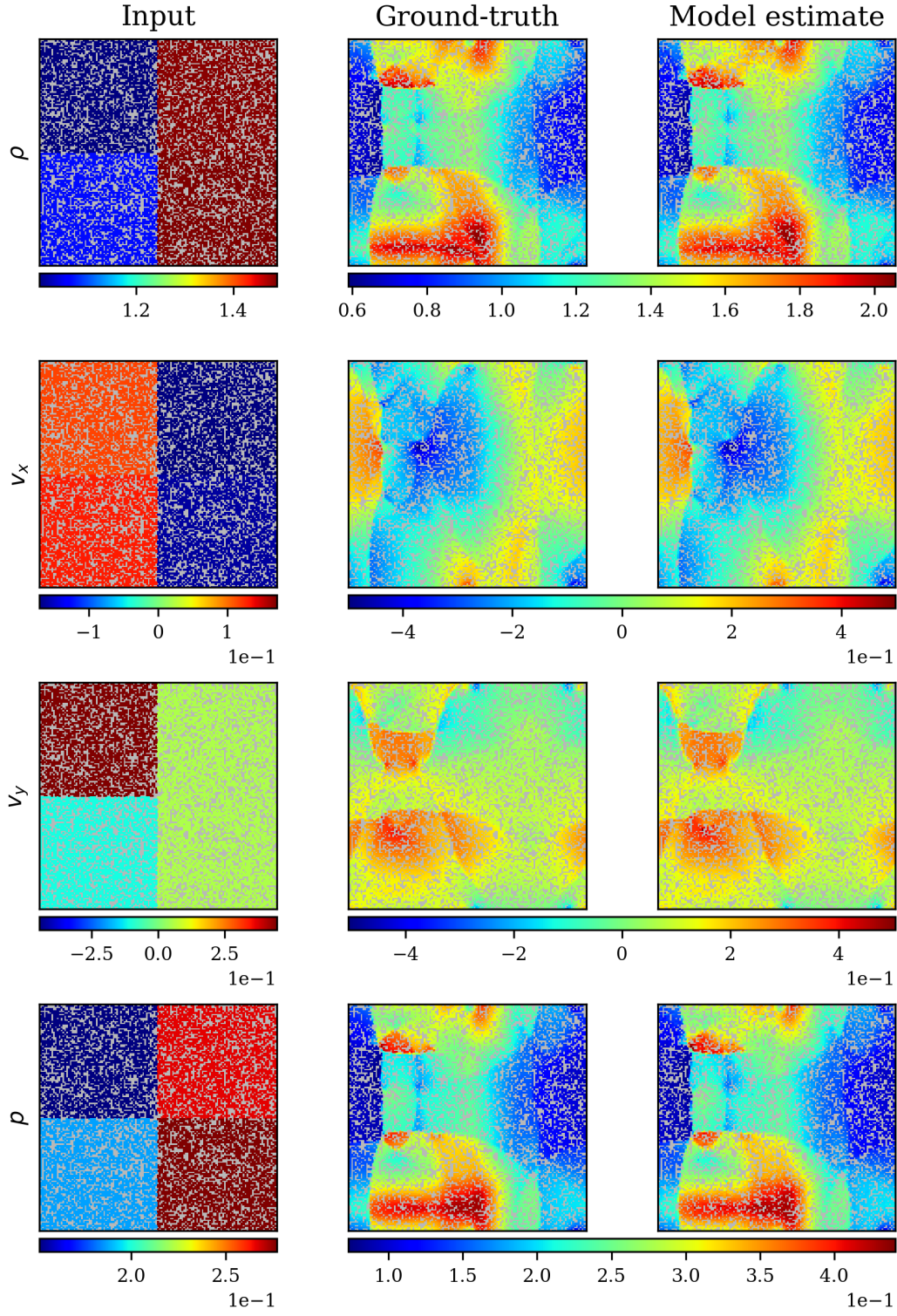


Figure G.15: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured CE-RP dataset.

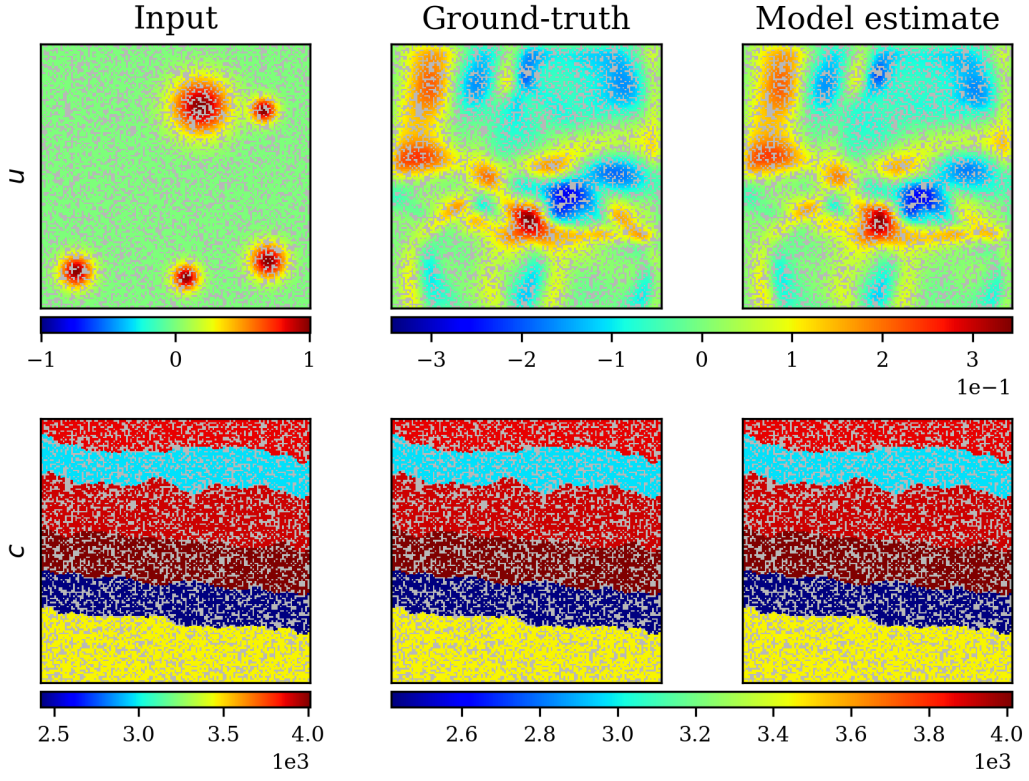


Figure G.16: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample unstructured Wave-Layer dataset.

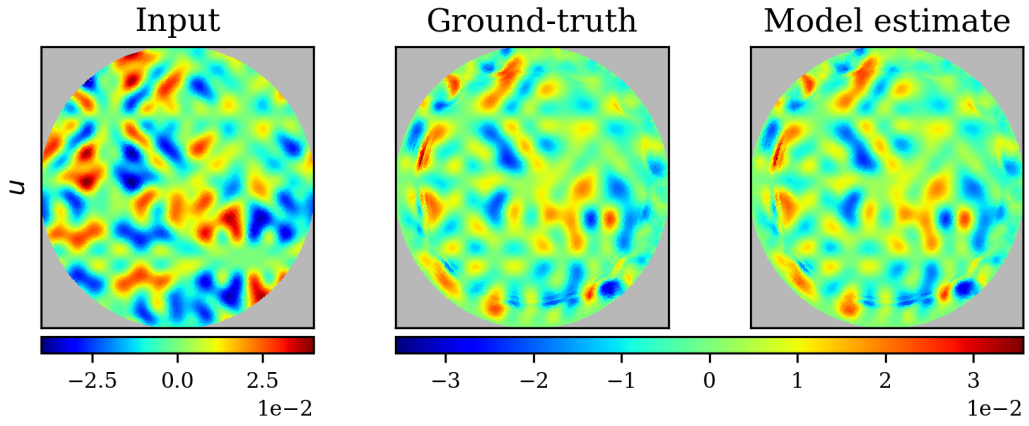


Figure G.17: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample Wave-C-Sines dataset.

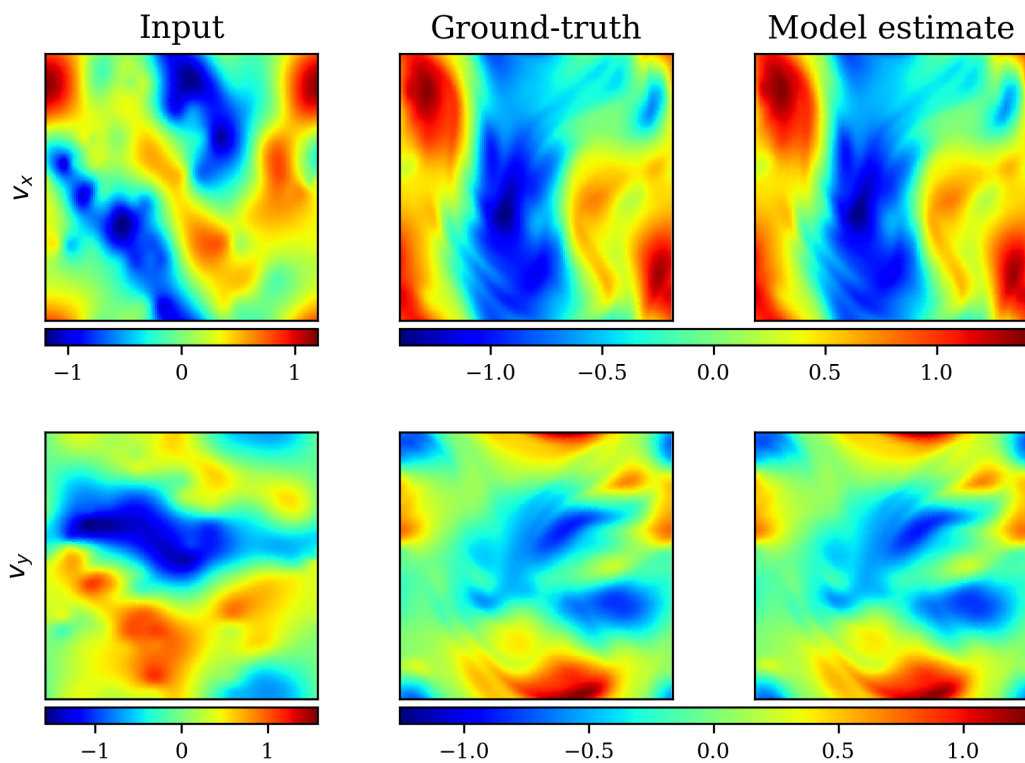


Figure G.18: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample NS-Gauss dataset.

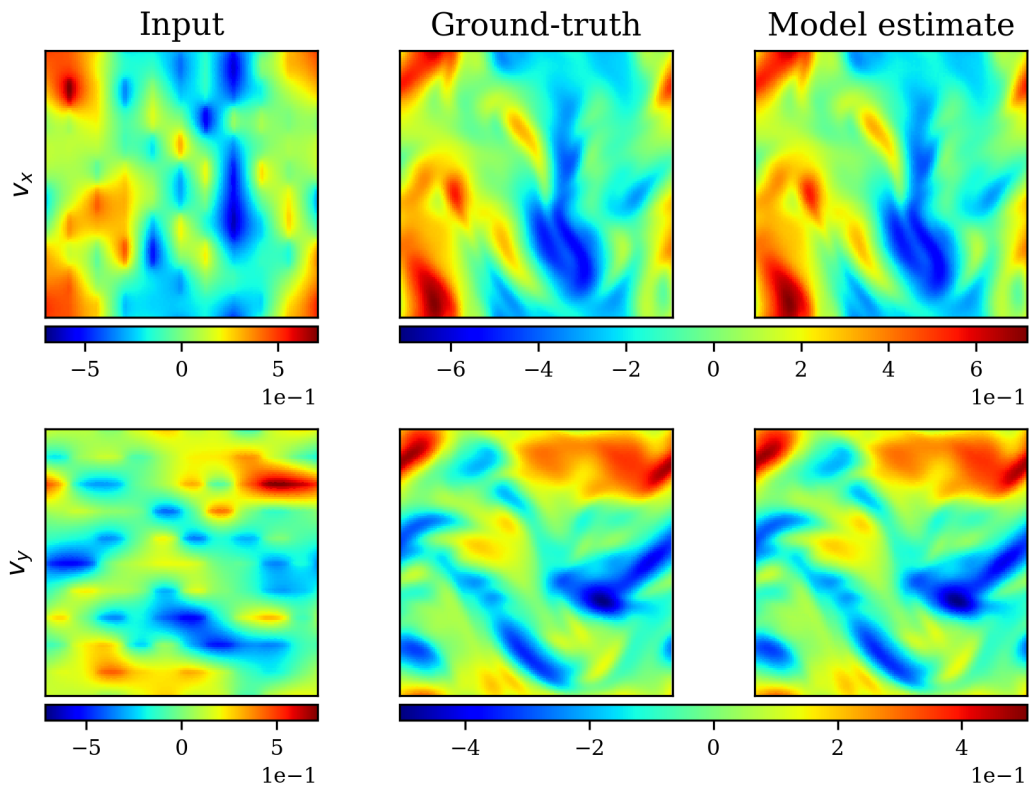


Figure G.19: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample NS-PwC dataset.

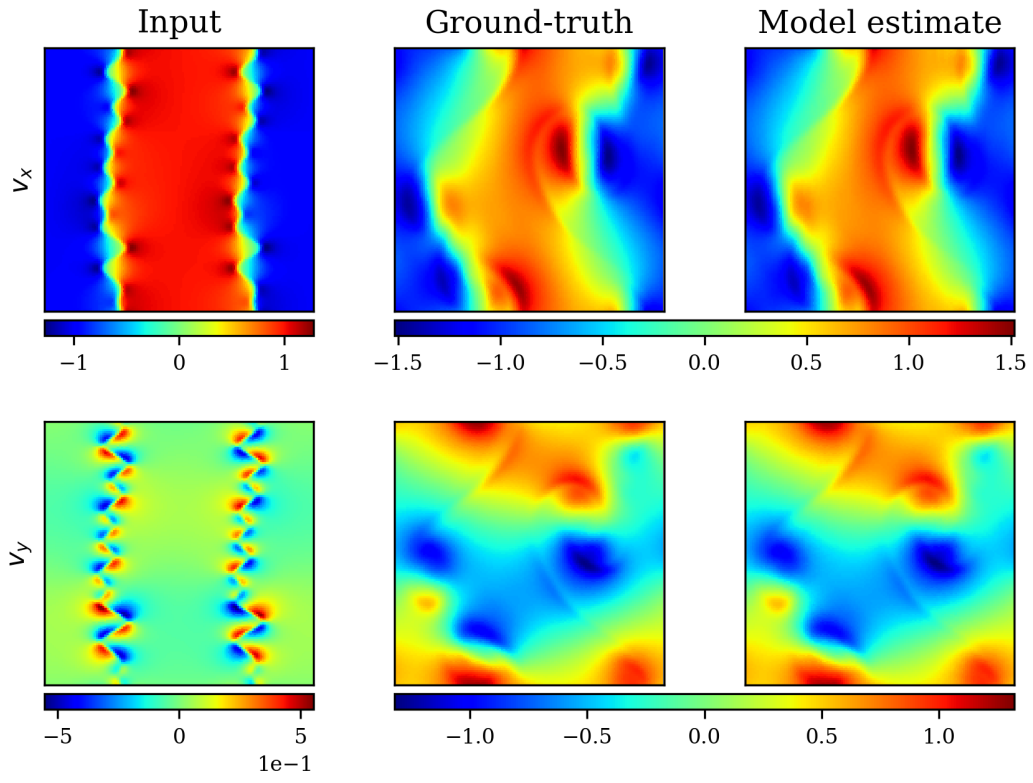


Figure G.20: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample NS-SL dataset.

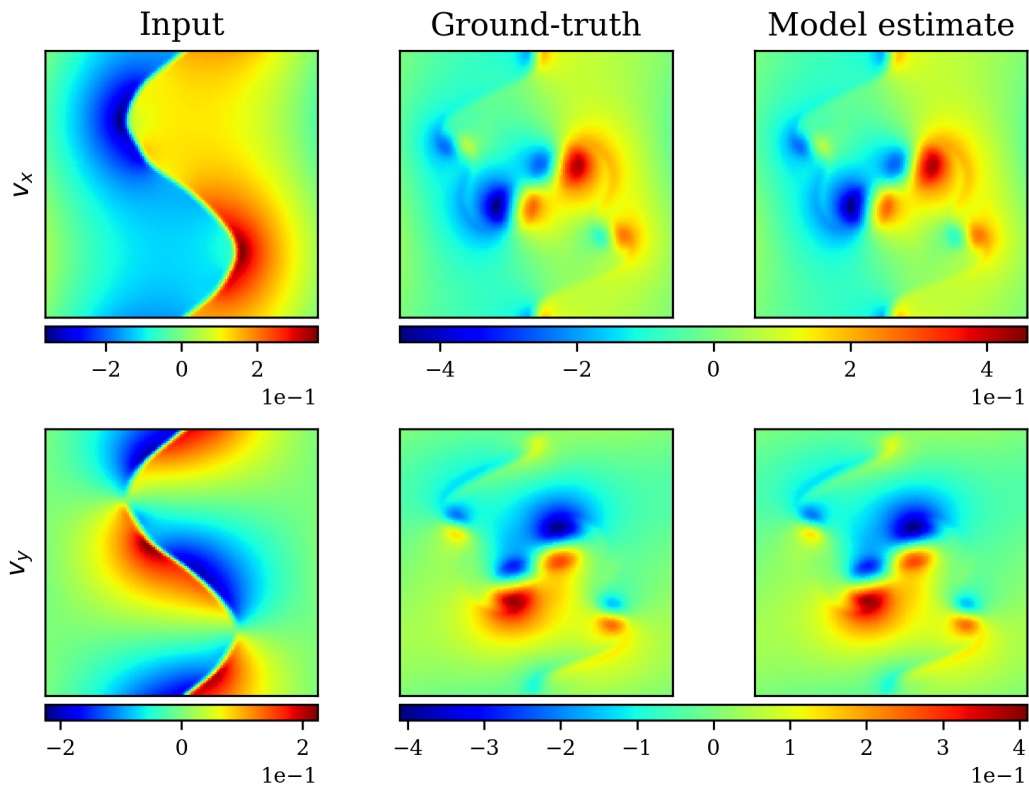


Figure G.21: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample NS-SVS dataset.

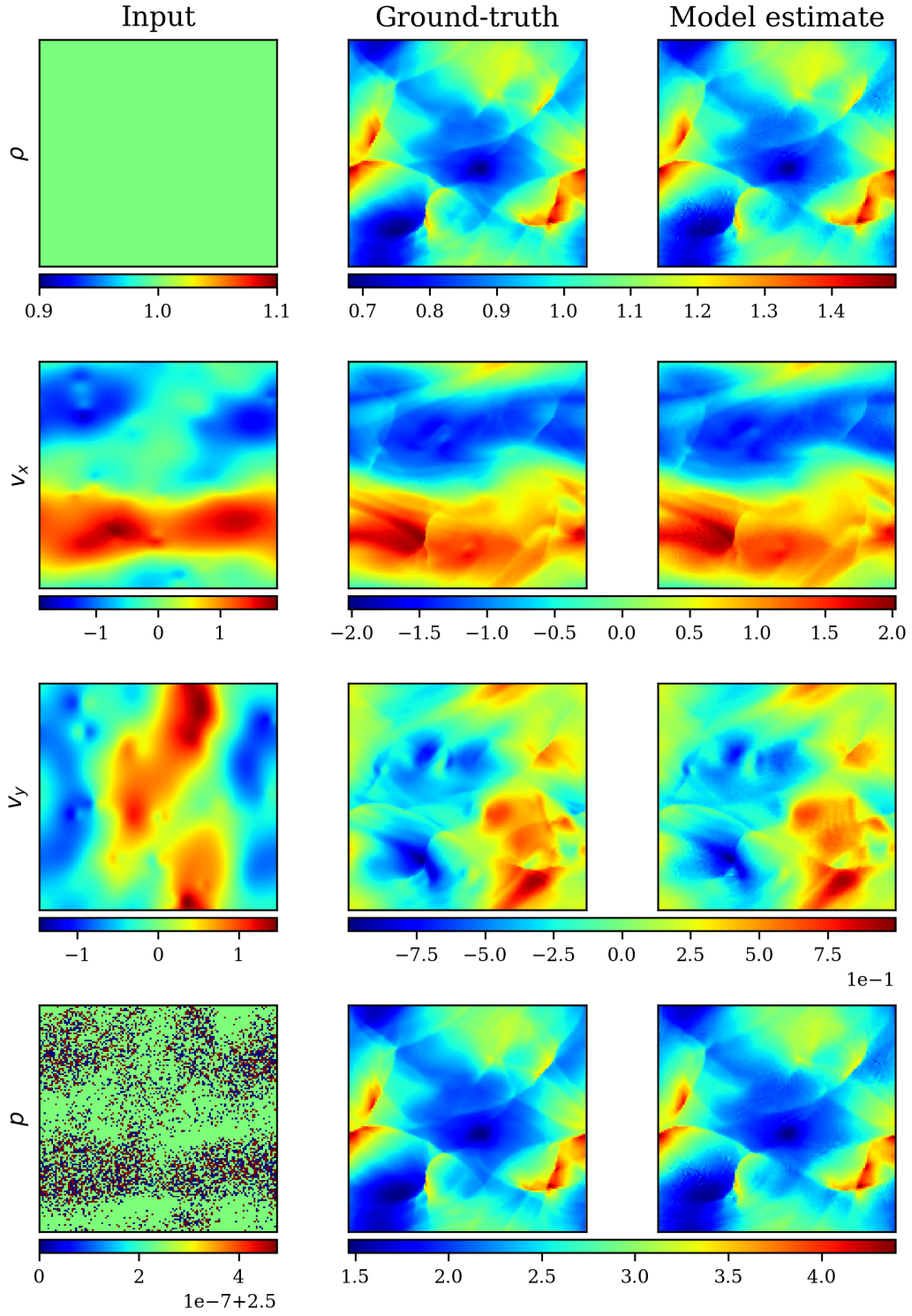


Figure G.22: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample CE-Gauss dataset.

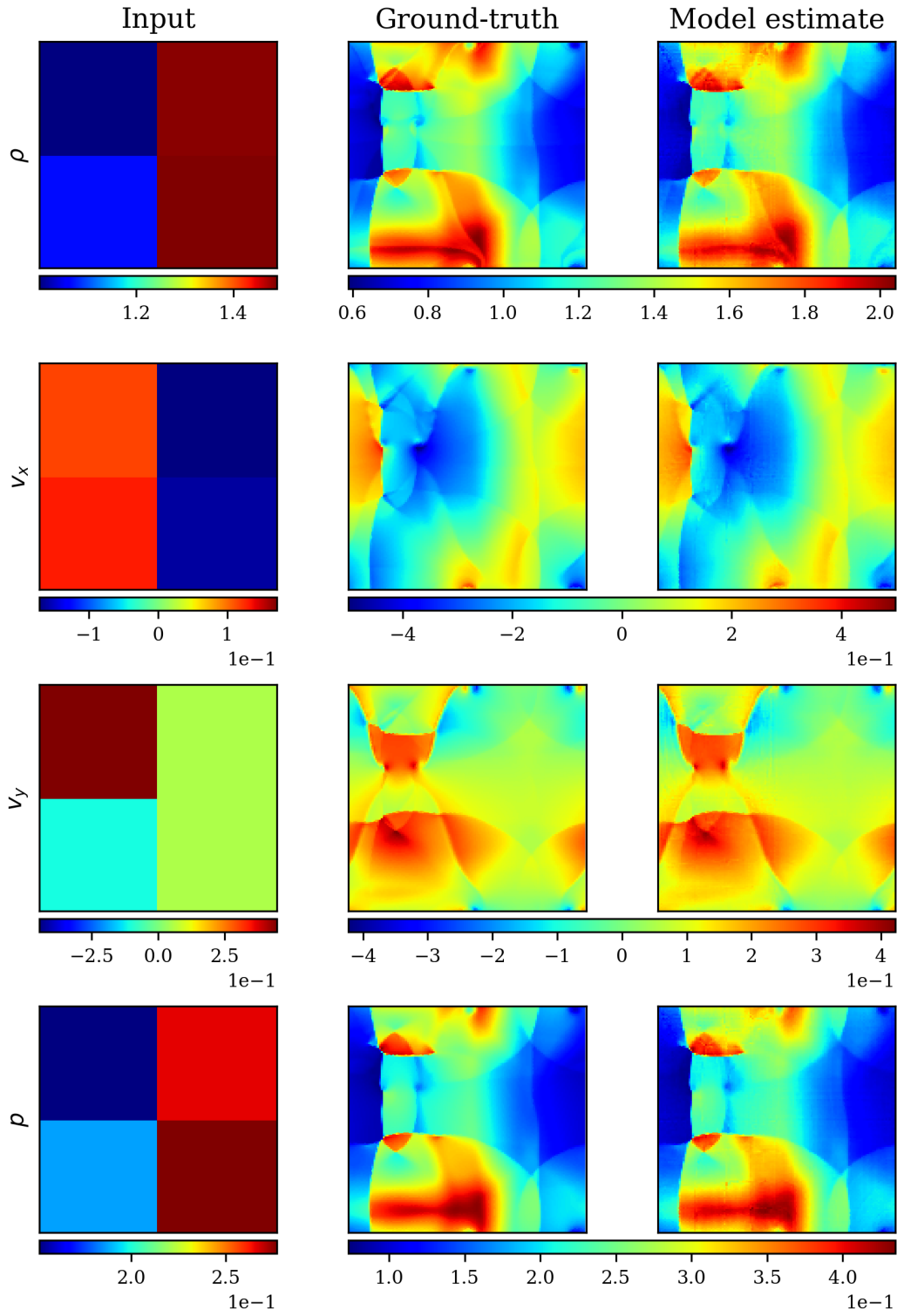


Figure G.23: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample CE-RP dataset.

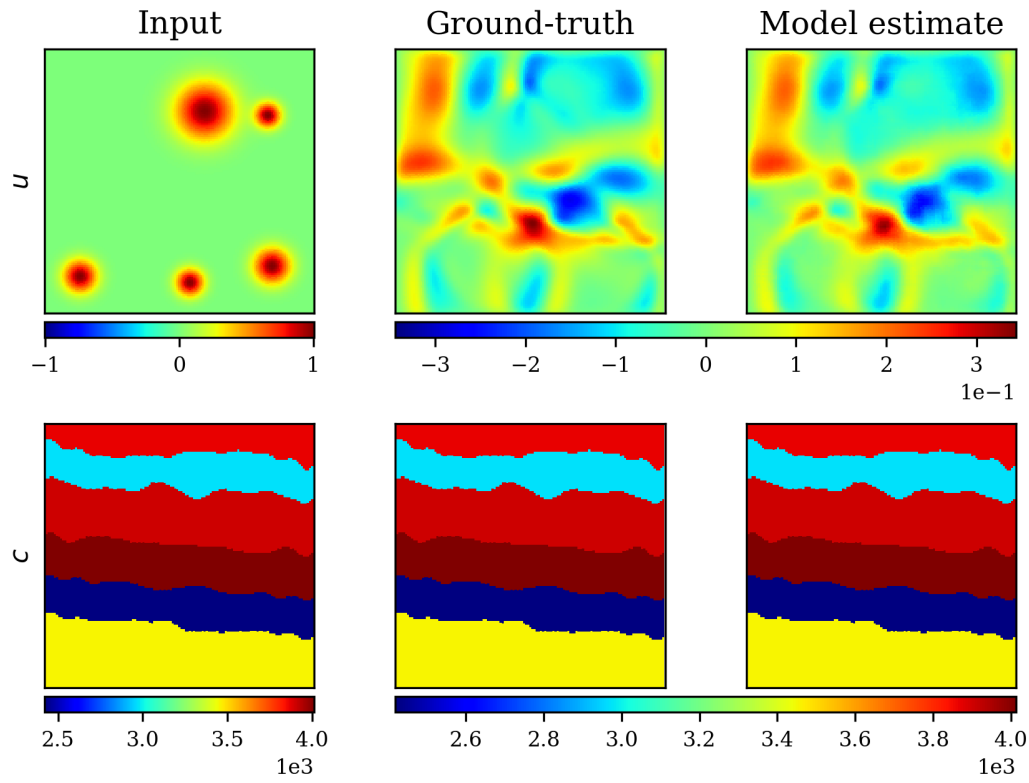


Figure G.24: Model input at $t = t_0$, ground-truth solution and model estimate at $t = t_{14}$ of a test sample Wave-Layer dataset.