# REPORT ON THE PAPER "GEOMETRY AWARE OPERATOR TRANSFORMER AS AN EFFICIENT AND ACCURATE NEURAL SURROGATE FOR PDES ON ARBITRARY DOMAINS"

DEVANSH TRIPATHI[1]
ETH Zürich

Abstract. In this paper [10], the authors address the pressing issue in the realm of operator learning that many of the accurate models are not computaionally efficient and vice versa. They propose a geometry aware operator transformer (GAOT)(pronounced goat) for learning PDEs on arbitrary domains which is computaionally efficient as well as accurate. GAOT combines novel multiscale attentional graph neural operator encoders and decoders, together with geometry embeddings and (vision) transformer processors. With multiple innovations in the implementations of GAOT to ensure computational efficieny and scalability, GAOT achieves state of the art performance on a large 3D industrial CFD dataset.

## 1. Contributions of the paper

Authors proposes a *Geometry Aware Operator Transformer* (GAOT, pronounced goat) as a neural surrogate for PDEs on arbitrary domains. While being based on the well-established *encode-process-decode* paradigm, GAOT includes several novel features that are designed to ensure both computational efficieny and accuracy, such as

- Our proposed *multiscale attentional graph neural operator* (MAGNO) as the encoder between inputs on an arbitrary point cloud and a *coarser* latent grid, designed to enhance accuracy through its multiscale information processing and attention modules.
- Novel *Geometry embeddings* in the encoder (and decoder) that provides the model with access to information about the (local) domain geometry, greatly increasing accuracy.
- A *transformer processor* that utilizes patching (as in ViT [3]) for computational efficieny.
- A MAGNO decoder, able to generate *neural fields*, with the ability to approximate the underlying solution at *any query point* in the domain.
- A set of implementation strategies to ensure that the computational realization of GAOT is efficient and highly scalable.

The demonstration of these capabilities is by,

- Extensively testing GAOT on 24 challenging benchmarks for both timw-independent and time-dependent PDEs of various types, ranging from regular grids to random point clouds to highly instructured adapted grids, and comparing it with widely-used baselines to show that GAOT is both highly accurate as well as computationally efficient and scalable.
- The efficiency and scalability of GAOT is further showcased by it achieving state of the art (SOTA) performance on the large scale 3D industrial benchmark of *DrivAerNet++* dataset for automobile aerodynamics [4].

[1]Seminar für Angewandte Mathematik, HG E 62.2, Rämistrasse 101, 8092 Zürich, Switzerland
devansh.tripathi@sam.math.ethz.ch.

- Through extensive ablations, we also highlight how the novel elements in the design of GAOT such as multiscale attentional encoders and geometry embeddings crucially contribute to the overall performance of our model.

## 2. Methods

**Problem Formulation.** We start with a generic *time-independent* PDE,

$$\mathcal{D}(c,u) = f, \qquad \forall x \in D \subset \mathbb{R}^d, \qquad \mathcal{B}(u) = u_b, \qquad x \in \partial D, \tag{2.1}$$

with $u : D \to \mathbb{R}^m$, the PDE solution, $c$ is the coefficient (PDE parameters), $f$ is the forcing term, $u_b$ are the boundary values and $\mathcal{D}$ and $\mathcal{B}$ are the underlying differential and boundary operators, respectively. Denoting as $\chi_D$, a function (e.g. indicator or signed distance) parameterizing the domain $D$, we combine all the inputs to the PDE 2.1 together into $a = (c, f, u_b, \chi_D)$, then the *solution operator* $\mathcal{S}$ maps inputs into PDE solution with $u = \mathcal{S}_a$. The corresponding *operator learning task* is to learn the solution operator $\mathcal{S}$ from data. To this end, let $\mu$ be an underlying *data distribution*. We sample i.i.d inputs $a^{(i)} \sim \mu$, for $1 \leq i \leq M$ and assume that we have access to *data pairs* $(s^{(i)}, u^{(i)})$ with $u^{(i)} = \mathcal{S}a^{(i)}$, thus the operator learning task is to approximate the distribution $\mathcal{S}_{\#\mu}$ from these data pairs.

Similarly denoting a generic *time-dependent* PDE as,

$$u_t + \mathcal{D}(c,u) = 0, \quad \forall x \in D \subset \mathbb{R}^d, t \in [0,T] \quad u(0) = u_0, \quad x \in D \tag{2.2}$$

with, $u : D \times [0,T] \mapsto \mathbb{R}^m$, $c$ the PDE coefficient and $u_0$ the initial datum and the underlying (spatial) differential operator $\mathcal{D}$. Clubbing the *inputs* to the PDE 2.2 into $a = (c, u_0, \chi_D)$, the corresponding *solution operator* $\mathcal{S}_t$, with $u(t) = \mathcal{S}_t(a)$ for all $t \in [0,T]$, maps the input into trajectory of the solution. The *operator learning task* consists of approximating $(\mathcal{S}_t)_{\#\mu}$ from the data pairs $(a^{(i)}, u^{(i)}(t))$ for all $t \in [0, T^{(i)}]$ and $1 \leq i \leq M$ with samples $a_i$ drawn from the data distribution $\mu$. However, in practice, we only have access to data, sampled on a discrete set of spatial points per sample as well as only on discrete time snapshots $t_n^{(i)} \in [0, T^{(i)}]$ and have to learn the solution from them.

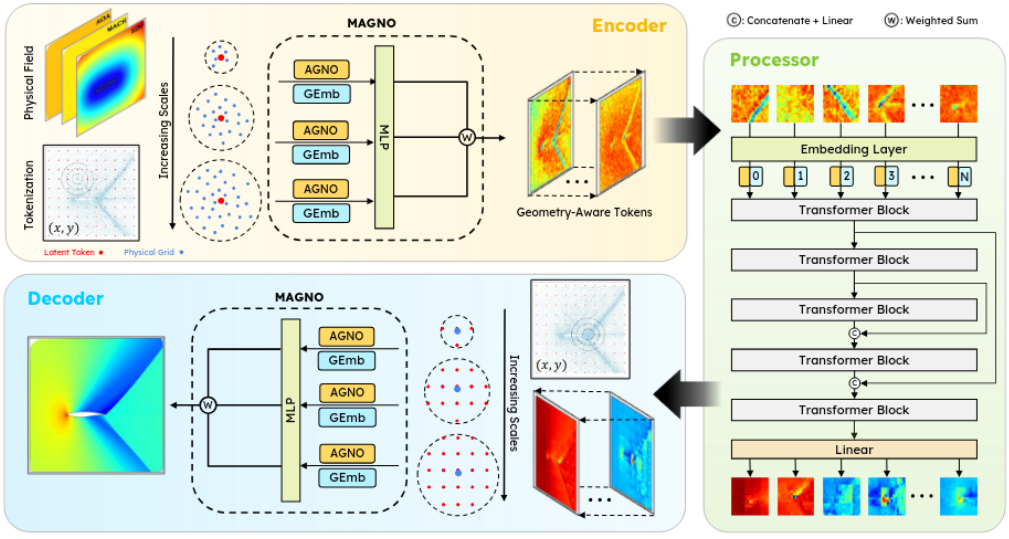



**Figure 2.** Schematic of the GAOT with an equispaced latent token grid. The encoder uses a multiscale attentional graph neural operator (MAGNO) to aggregate the input data into geometry-aware tokens. A vision transformer (ViT) block with residual connections processes tokens, enabling global exchange of information. A MAGNO decoder identifies the nearest tokens around a given query point to decode the final field.

FIGURE 1. Taken from [10]

**GAOT Model Architecture.** For simplicity, we start with the time-independent case, where given inputs $a(x_j)$ on the input cloud $D_\Delta = \{x_j\} \subset D$, for $1 \leq j \leq J$, GAOT provides an approximation to solution $u$ of the PDE 2.1 at any query point $x \in D$. In the first step, an *encoder* transforms the input on the underlying point cloud $D_\Delta$ to a *latent point cloud* $\mathcal{D} \subset \mathbb{R}^d$. The resulting *spatial tokens* are then processed by a processor module to learn useful representations and its output is remapped to the original domain $D$ via the *decoder*, which allows evaluation at any query point $x \in D$.

**Choice of Latent Domain.** As shown in the figure in appendix B.1, the latent domain $\mathcal{D}$ (to which the encoder maps) can be chosen in three different ways, i) a regular (structured) grid stencil, consisting of equispaced points on a Cartesian domain ii) randomly downsampling the underlying point cloud $D_\Delta$ or iii) a projected low-dimensional representation, where a high-dimensional domain is projected to a lower dimension (for instance using tri-plane embeddings in 3D [2]) and a regular grid is used in the lower-dimension domain. GAOT is a general framework where any of these point cloud choices can be employed for $\mathcal{D}$.

**Encoder.** Given input values $a(x_j)$ on the underlying point cloud $D_\Delta$, the encoder aims to transforms it into latent features $w_e(y)$ at any point $y \in \mathcal{D}$ on the latent point cloud. Using a graph-neural operator (GNO) encoder as in GINO [6] would lead to,

$$\tilde{w}_e(y) = \sum_{k=1}^{n_y} \alpha_k K(y, x_k, a(x_k))\varphi(a(x_k)), \tag{2.3}$$

with MLPs $K$ and $\varphi$ and the sum above taken over all the $n_y$ points $x_k \in D_\Delta$ such that $|y - x_k| \leq r$ for some hyperparameter $r > 0$, where $\alpha_k$ are some given quadrature weights. In other words, GNO accumulates information from all the points in the original point cloud that lie inside a ball of radius $r$, centered at the given point $y$ in the latent point cloud, and processes them through a kernel integral.

Authors propose a mechanism to integrate *multiscale* information into the encoder. To this end, we choose $r_m = s_m r_0$, for some base radius $r_0$ and scale factors $s_m$, for $m = 1, \ldots, \bar{m}$ to modify GNO 2.3, by,

$$\tilde{w}_e^m(y) = \sum_{k=1}^{n_y^m} \alpha_k^m K^m(y, x_k, a(x_k))\varphi(a(x_k)), \tag{2.4}$$

for any fixed scale $m$ and with MLPs $K^m.\varphi$. The above sum is taken over all the $n_y^m$ points $x_k \in D_\Delta$ such that $|y - x_k| \leq r_m$. To choose the quadrature weights $\alpha_k^m$, we propose an *attention based* choice,

$$\alpha_k^m = \frac{\exp(e_k^m)}{\sum_{k'=1}^{n_y^m} \exp(e_{k'}^m)}, \qquad e_k^m = \frac{\langle \mathbf{W}_q^m y, \mathbf{W}_\kappa^m x_k \rangle}{\sqrt{\bar{d}}}, \tag{2.5}$$

with $\mathbf{W}_q^m, \mathbf{W}_\kappa^m \in \mathbb{R}^{\bar{d} \times m}$ are query and key matrices respectively, completing the description of the *attentional graph neural operator* (AGNO) at each scale $m$.

**Geometry Embeddings.** The only geometric information in the afore-described encoder is provided by the coordinates of the underlying points. The authors argue that this alone does not convey the rich geometric information about the domain that can affect the solution of the underlying PDE 2.2. In the literature, the geometry information is usually provided either by appending them as node and edge features on the underlying graphs or by encoding a signed distance function. The authors propose to use a novel *geometry embeddings* to encode this information. To this end as described in appendix B.3, we can rely on *local statistical embeddings* for each point $y \in \mathcal{D}$ as all the neighbouring points $x_k$ in $D_\Delta$ with $|y - x_k| \leq r_m$ have already been computed in the AGNO encoder. From these points, we can readily compute statistical descriptors such as

(1) number of neighbors $x_k \in D_\Delta$, in the ball $B_{r_m}(y)$

(2) the *average distance* $D_{avg} = \frac{1}{n_y^m} \sum_{k=1}^{n_y^m} |y - x_k|$

(3) the variance of this distance $D_{var}$, with respect to the average $D_{avg}$

(4) the *centroid offset vector* $\Delta_y = \frac{1}{n_y^m} \sum_{k=1}^{n_x^y} (x_k - y)$ and

(5) a few principal component (PCA) features of the covariance matrix of $y - x_k$ to calculate the *local shape anisotropy*. More about this in appendix B.3.

These statistical descriptors, for each scale $m$ and each point $y \in \mathcal{D}$ are then concatenated into a single vector $z_y$, normalised across components to yield zero mean and unit variance and fed into an MLP to provide the embedding $g^m(y)$. Alternatively, geometry embedding using emphPointNet models [1] can also be considered.

**MAGNO.** As shown in figure 1, the scale-dependent AGNO $\tilde{w}_e^m$ (2.4) and the geometry embeddings $g^m$, at each scale $m$, can be concatenated together and passed through another MLP to yield a scale-specific latent features functions $\hat{w}^m(y)$. Next, we need to integrate these features across all $m$ scales. Instead of naively summing these scales contributions, we observe that different scales might contribute differently for every latent token to the encoding. TO ascertain this relative contribution, we introduce a (small) MLP $\psi_m$ and weigh the relative contribution with a *softmax* and combine them into the *multiscale attentional graph neural operator* or MAGNO encoder by setting,

$$w_e(y) = \sum_{m=1}^{\bar{m}} \beta_m(y) \hat{w}^m(y), \qquad \forall y \in \mathcal{D}, \qquad \beta_m(y) = \frac{\exp(\psi_m(y))}{\sum_{m'=1}^{M} \exp(\psi_{m'}(y))} \tag{2.6}$$

**Transformer Processor.** The encoder provides a set of *geometry aware tokens* $w_e(y_l)$, for all points $y_l \in \mathcal{D}$, with $1 \leq l \leq L$, in the latent point cloud. These tokens are further transformed by a processor, a suitable transformer based processor. Details of the processor architecture can be found in B.4 while the choices are summarized here. If the latent points $\{y_l\}$ lie on a regular grid (either through a structured stencil or a projected low-dimensional one), we use a patch-based *vision transformer* or ViT [3] for computational efficiency. The equispaced latent points are combined into patches and the tokens in each patch are flattened into a single token embeddings which serves as the input for a multi-head attention block, followed by a feed forward block. RMS normalisation is applied to the tokens before processing. Either sinusoidal absolute position embeddings or rotary relative position embeddings are used to encode token positions. If the latent points $y_l$ are randomly downsampled from the original point cloud, there is no obvious way to patch them together. Hence, a standard transformer [9], but with RMS normalisation, can be used. Additionally, we employ multiple skip connections across transformer blocks. The transformer processor transforms the token $w_e(y_l)$ into processed tokens, that we denote by $w_p(y_l)$, for all $1 \leq l \leq L$.

**Decoder.** Given any query point $x \in D$ in the original domain, the task of the decoder in GAOT is to provide $w(x)$, which approximates the solution $u$ of the PDE 2.1 at that point. To this end, we simply employ the MAGNO architecture in reverse. By choosing a base radius $\hat{r}_0$ and scale factors $\hat{s}_m$, a set of increasing radii $\hat{r}_m = \hat{s}_m \hat{r}_0$ are selected to define a set of increasing balls $B_{\hat{r}_m}(x)$ around the query point $x$ (see Fig.1). A corresponding AGNO model is defined by replacing $y \to x$, $x_k \to y_\ell$ and $a \to w_p$ in 2.4, with corresponding attentional weights computed via 2.5. In parallel, geometry embeddings over each ball $B_{\hat{r}_m}(x)$ are computed to provide statistical information about how the latent points $y_\ell$ are distributed in the neighborhood of the query point $x$. These AGNO features and geometry embeddings are concatenated and passed through an MLP to provide $w(x)$, which has the desired dimensions of the solution $u$ of the PDE 2.1. We denote the GAOT model as $S_\theta$ with the output $w = S_\theta(a)$ for the inputs $a$ to the PDE 2.1. It is trained to minimize the mismatch with the underlying operator $S$, i.e., the parameters $\theta$ are determined to minimize a loss $\mathcal{L}(S(a), S_\theta(a))$ over all input samples $a_i$, with $\mathcal{L}$ being either the absolute or mean-square error.

**Extension to time-dependent problems.** To learn the solution operator $\mathcal{S}_t$ of time-dependent PDE, we observe that the $\mathcal{S}_t$ can be used to update the solution forward in time, given that the

solution at any time point $u(t)$ by applying $u(t+\tau) = \mathcal{S}_\tau(u(t))$. Thus, for any time $t$, given the augmented input $a(t) = (c, u(t))$, with $c$ being the coefficient in the PDE 2.2, we need GAOT to output $u(t+\tau)$, for any $\tau \geq 0$. For this, we retain the architecture of GAOT, as described for the time-independent case above, and simply add the current time $t$ and the *lead-time* $\tau$ as further inputs to the model. More precisely, the time-dependent version of GAOT is of the form $\hat{S}_\theta(x, t, \tau, a(t))$, where $a(t)$ takes values at points sampled in $D$. Following [7], the map $\hat{\mathcal{S}}_\theta$ can be used to update an approximate solution of PDE 2.2 in time by follwing a very generic time-stepping strategy:

$$\mathcal{S}_\theta(t, \tau, a(t)) = \gamma u(t) + \delta \hat{\mathcal{S}}_\theta(x, t, \tau, a(t)). \tag{2.7}$$

Here, choosing the parameters $(\gamma, \delta)$ approximately leads to different strategies for time stepping: $\gamma = 0, \delta = 1$ directly approximates the *output* of the solution operator at time $t+\tau$; $\gamma = 1, \delta = 1$ yields the *residual* of the solution at the later time, with respect to the solution at current time; $\gamma = 1, \delta = \tau$ is equivalent to approximating the *time-derivative* of the solution. GAOT provides the flexibility to use any of these time stepping strategies. Authors also use the *all2all* training strategy [5] to leverage trajectory data for time-dependent PDEs.

**Efficient implementation.** Authors realized that the heaviest burden of the computation should fall on the processor. The encoder and decoder are often responsible for memory overheads as these modules entail sparse computations on graphs with far more edges than nodes, making the computations largely edge-based and leading to high (and inefficient) memory usage. Moreover, in many PDE learning tasks on arbitrary geometries, the underlying domain (and the resulting graph) varies significantly between data samples, making load balancing very difficult.

To address these computational challenges, they resorted to i) moving the graph construction outside the model evaluation by either storing the graph, representing the input point cloud, in memory for small graphs or on disk for large graphs and loading them during training with efficient data loaders ii) sequentially processing each input in a given batch for the encoder and decoder, while still batch processing in the transformer processor, allowing us to reduce memory usage while retaining efficiency and iii) if needed for very large-scale datasets, we use an edge-dropping strategy to further decrease the memory usage of the encoder and decoder.

## Appendix A. **Details of GAOT Architecture**

A.1. **General Form of PDEs.** The focus is on two broad class of PDEs: time-dependent and time-dependent.

**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 can be written as

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

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. source term);
- $\mathcal{D}$ is a (spatial) differential operator;
- $\mathcal{B}$ is a boundary operator acting on $\partial D$;
- $u_b(x)$ 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 $c \in \mathcal{Q} \subset L^p(D; \mathbb{R}^m)$ is taken from a parameter space.

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

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

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.,

$$\bar{u}(x) = \lim_{t \to \infty} u(t, x). \tag{A.3}$$

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

A.2. **Solution Operator for PDEs.** Let us denote the solution to the time-independent PDE A.1 by

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

where $\mathcal{S} : \mathcal{X}^0 \times \mathcal{Q} \times (0, T) \to \mathcal{X}$ 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 $t$.

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

$$\mathcal{S}^\dagger : \mathcal{X} \times \mathcal{Q} \times (0, T) \times \mathbb{R}^+ \to \mathcal{X}, \tag{A.5}$$

such that

$$\mathcal{S}^\dagger(u^t, c^t, t, \tau) = \mathcal{S}^t(u^t, c^t, \tau) = u^{t+\tau}. \tag{A.6}$$

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

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

$$\overline{\mathcal{S}} : \mathcal{Q} \to \mathcal{X} \tag{A.7}$$

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

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

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

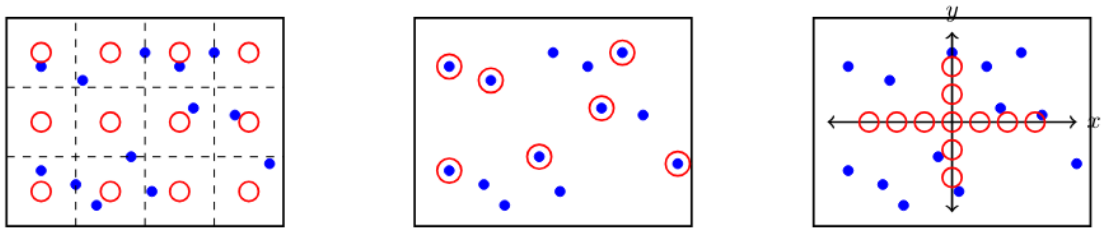
A.3. **Discretizations.** We start by describing these discretizations for the time-independent PDE A.2. To this end, fix the $i$-th sample and let $D_{\Delta^{(i)}} = \{x_j^{(i)} \in D^{(i)}\}$, for $1 \leq jleqJ^{(i)}$ denote a set of *sampling points* on the underlying domain $D^{(i)}$. Observe that the underlying domain itself can be an input to the solution operator $\overline{S}$ of A.2. We assume access to the functions $(c^{(i)}(x_j), f^{(i)}(x_j), u^{(i)}(x_j))$ and the corresponding discretized boundary values. Denoting these discritized inputs and outputs as $a^{(i)}_{\Delta^{(i)}}$ (where $a = (c, f, u_b)$) and $u^{(i)}_{\Delta^{(i)}}$, respectively, the underlying learning task boils down to approximating $\overline{\mathcal{S}}_{\#\mu}$ from the discretized data-pairs $\left(a^{(i)}_{\Delta^{(i)}}, u^{(i)}_{\Delta^{(i)}}\right)$. Note that although the data is given in a discretized form, we still require that our operator learning algorithm can provide values os the output functions $u$ at any *query point* $x \in D$.

For the time-dependent PDE A.1, in addition to the spatial discretization $D_{\Delta^{(i)}} = \{x_j^{(i)} \in D^{(i)}\}$, 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 time-dependent operator learning task consists of inputs $(c^{(i)}(x_j), u_0^{(i)}(x_j))$ and outputs $u(x_j^{(i)}, t_n^{(i)})$, from which the space- and time-continuous solution operator $\mathcal{S}_t$ has to be learned at every query point $x \in D$ and time point $t \in [0, T]$.

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

## Appendix B. **Details of GAOT Architecture**

B.1. **Choice of Latent Grid.** Given the input point cloud, denoted by $D_\Delta$ above, the first step in our design is to select a *latent* point cloud, consisting of points at which our spatial tokens are going to be specified. Here, we explore three distinct ways of choosing spatial tokens, each offering different trade-offs in terms of computational cost, geometric coverage, and ease of patching for efficient attention.




**(a)** Strategy I **(b)** Strategy II **(c)** Strategy III

**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.

Figure 2. Taken from [10]

**Strategy Stencil Grid (Strategy I).** In this approach, we overlay a structured grid of tokens across $D_\Delta$. For 2D domains, this may be a uniform mesh of cells; for 3D domains, an analogous dense grid can be used.

- *Advantages.* This grid can be quite fine if needed, ensuring adequate coverage. Moreover, we can group tokens into patches before feeding them into the transformer processor, effectively reducing the token count (number of latent points). Authors empirically show that patch size has a negligible effect on performance; hence large patch sizes can be used to speed up training.
- *Limitations.* The main drawback is that token count grows exponentially with the dimension; for 3D, the number of grid cells can be prohibitively large. Also, if the input data lie on a low-dimensional manifold embedded in $D_\Delta$, some tokens may remain underutilized. Nonetheless, we find in practice that even empty tokens (those with no neighboring input points) can still contribute to better global encoding and improved convergence.

**Downsampled Unstructured Points (Strategy II).** This method directly downsamples the input unstructured point cloud and treat each sampled point as a token, RIGNO [7] is a typical example based on this strategy. If the data are denser in some regions, naturally more tokens appear there.

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

**Projected Low-Dimensional Grid (Strategy III).** One can project the 3D domain (or higher-dimensional space) into a lower-dimensional representation and then place a structured stencil grid in the reduced coordinates for example, using triplane embeddings in 3D [2].

- *Advantages.* Such a projection drastically reduces the token count in 3D, and avoids the purely *low-dimensional manifold* disadvantage of Strategy I. Moreover, one can still apply patching on the structured plane.
- *Limitations.* Decomposing $d$-dimensional coordinates into disjoint projections (e.g. splitting $x, y, z$ axes) can introduce additional approximation errors. Some local neighborhood information is inevitably lost during projection. This trade-off can degrade the final accuracy compared to direct methods (Strategy I or II).

B.2. **Multiscale Attentional Graph Neural Operator.** Both the encoder and decoder in GAOT 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 construct 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 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 $D_\Delta$ (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 2.3 from main text:

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

where the sum is over $n_y$ points $x_k$ in the original point cloud $D_\Delta$ such that $|y - x_k| \leq r$. $K$ and $\varphi$ are MLPs, $a(x_k)$ is the input features at point $x_k$, and $\alpha_k$ are given quadrature weights. This form can be seen a discrete approximation of an integal operator:

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

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 $(m = 1, \ldots, \bar{m})$. For each scale $m$, we gather points $x_k$ from the original point cloud $D_\Delta$ 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 by eq 2.4 from the main text:

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

Here, $n_y^m$ is the number of neighbors $x_k$ within radius $r_m$. The MLP $K_m$ and $\varphi$ 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 Integation (AGNO).** In the main text, authors propose an attention-based choice for the quadrature weights $\alpha_k^m$, as given by eq 2.5:

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

where $\mathbf{W_q^m}, \mathbf{W_q^m} \in \mathbb{R}^{\bar{\mathbf{d}} \times \mathbf{d}}$ (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 scale $m$.

B.2.2. **Attentional Fusion of Multiscale Features.** After computing the AGNO features $\tilde{w}_e^m(y)$ for each scale (which is then fused with geometry embeddings, detailed in Sec. B.3, to form $\hat{w}^m(y)$), we need to integrate this information from different scales. Instead of simple addition or concatenation, authors introduce a small MLP $\psi_m$ to learn the relative contribution of each scale to the final encoded feature $w_e(y)$:

$$w_e(y) = \sum_{m=1}^{\bar{m}} \beta_m(y) \hat{w}^m(y), \qquad \forall y \in \mathcal{D}, \qquad \beta_m(y) = \frac{\exp(\psi_m(y))}{\sum_{m'=1}^{M} \exp(\psi_{m'}(y))} \tag{B.5}$$

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

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

B.3. **Geometry Embeddings.** While the multiscale Attentional GNO already leverages geometric structure via local neighborhoods, one often needs to incorporate more explicit shape or domain information in practical PDE scenarios when the geometry of the domain itself is play an important role in solution operator. Authors introduce geometry embeddings to enhance the model's geometric awareness. These embeddings work in tandem with the MAGNO providing a rich geometric description for each token and its neighborhood at various scales $m$.

Prior work on including geometric information in neural PDE solvers resorts to two major approaches: (i) appending geometry features directly into node/edge attributes [7], or (ii) using a signed distance function (SDF) [6]. However, authors argue that:

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

Authors proposed two more direct and flexible mechanisms for extracting geometries descriptors: local *Statistical embedding* and *PointNet-based embeding*:

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

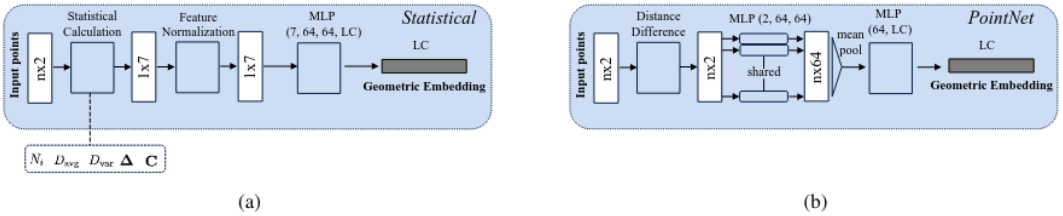



**Figure B.2.** Schematic of the Geometric Embedding for Statistical Embedding (a) and PointNet-based Embedding (b). LC denotes the lifting channels for MAGNO output.

FIGURE 3. Taken from [10]

we describe the calculation of PCA features and the other statistics are described in the main text 2.

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

$$\mathbf{C}_y^m = \frac{1}{n_y^m}\sum_{k=1}^{n_y^m}(x_k - \bar{x}^m_{nbrs,y})(x_k - \bar{x}^m_{nbrs,y})^\top \tag{B.6}$$

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

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

$$g^m(y) = \text{MLP}_{\text{geo}}(\text{Normalize}(z_y^m)) \tag{B.7}$$

This $\text{MLP}_{\text{geo}}$ is typically shared across all points and scales.

**Point-based Embedding.** As an alternative, we can train a PointNet-style network [8] to derive a compact geometric descriptor from each token's neighborhood. Classical PointNet architecture typically include:

- Input Transformer: aligns input points to a canonical space (optional),
- Shared MLP: processes each point individually,
- Symmetric Pooling: aggregates per-point features into a global decriptor, ensuring permutation invariance.

In our PDE setting, we do not necessarily need an input transformer; the local coordinates can directly serve as input features. We replace the typical max-poooling TODO

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 B.1, we can choose the following strategies, respectively:

- Regular Grid (Strategy I or III): If the latent points $\{y_l \in \mathcal{D}\}$ 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 transformer (ViTs) [3]. The latent points are grouped into non-overlapping "patches". All tokens 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 $\{y_l\}$ are randomly downsampled from the original point cloud $D_\Delta$, they lack a regular grid structure. In this case, there is no obvious "patching" method, and each latent token $w_e(y)$ 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 rejected. In GAOT, we use the Rotary Positional Embeddings (RoPE), which is a method that integrates relative positional information direclty into the self-attention mechanism. It achieves this by applying rotations, dependent on their relative positions, to the Query and Key vector. This has shown strong performance in many Transformer models.

**Transformer Block Structure.** For the transformer blocks, we adopt an RMS norm $\mathrm{RMSNorm}(\cdot)$ at the beginning of attention and feedforward layer:

$$\mathbf{z} = \mathrm{RMSNorm}(\mathbf{x}), \qquad \mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\mathrm{mean}(\mathbf{x}^2)}} \odot \alpha \tag{B.8}$$

where $\alpha$ 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 [9]. Each block has the structure,

$$\mathbf{Z}_{\text{attn}} = \mathbf{X} + \mathrm{MultiHeadAttn}(\mathrm{RMSNorm}(\mathbf{X})), \qquad \mathbf{Z}_{\text{ffn}} = \mathbf{Z}_{\text{ffn}} + \mathrm{FFN}(\mathrm{RMSNorm}(\mathbf{Z}_{\text{attn}})). \tag{B.9}$$

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. 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 $w_e(y_l)$ from the encoder into processed tokens $w_p(y_l)$ 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.

## References

[1] R. Qi Charles, Hao Su, Mo Kaichun, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, 2017. doi: 10.1109/CVPR.2017.16.

[2] Qian Chen, Mohamed Elrefaie, Angela Dai, and Faez Ahmed. Tripnet: Learning large-scale high-fidelity 3d car aerodynamics with triplane networks, 2025. URL `https://arxiv.org/abs/2503.17400`.

[3] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL `https://arxiv.org/abs/2010.11929`.

[4] Mohamed Elrefaie, Florin Morar, Angela Dai, and Faez Ahmed. DrivAerNet++: A large-scale multimodal car dataset with computational fluid dynamics simulations and deep learning benchmarks, 2025. URL https://arxiv.org/abs/2406.09624.

[5] Maximilian Herde, Bogdan Raonić, Tobias Rohner, Roger Käppeli, Roberto Molinaro, Emmanuel de Bézenac, and Siddhartha Mishra. Poseidon: Efficient foundation models for pdes, 2024. URL https://arxiv.org/abs/2405.19101.

[6] Zongyi Li, Nikola Borislavov Kovachki, Chris Choy, Boyi Li, Jean Kossaifi, Shourya Prakash Otta, Mohammad Amin Nabian, Maximilian Stadler, Christian Hundt, Kamyar Azizzadenesheli, and Anima Anandkumar. Geometry-informed neural operator for large-scale 3d pdes. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[7] Sepehr Mousavi, Shizheng Wen, Levi Lingsch, Maximilian Herde, Bogdan Raonić, and Siddhartha Mishra. Rigno: A graph-based framework for robust and accurate operator learning for pdes on arbitrary domains, 2025. URL https://arxiv.org/abs/2501.19205.

[8] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation, 2017. URL https://arxiv.org/abs/1612.00593.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[10] Shizheng Wen, Arsh Kumbhat, Levi Lingsch, Sepehr Mousavi, Yizhou Zhao, Praveen Chandrashekar, and Siddhartha Mishra. Geometry aware operator transformer as an efficient and accurate neural surrogate for PDEs on arbitrary domains, 2025. URL https://arxiv.org/abs/2505.18781.