Imperial College
London

Department of Earth Science and Engineering

MSc in Applied Computational Science and Engineering

Independent Research Project
Final Report

Dimensionality reduction with Space-filling curve autoencoders for fluid flow problems solved on unstructured meshes

by

Jin Yu

jin.yu20@imperial.ac.uk
GitHub login: acse-jy220

Supervisors:

Dr. Claire Heaney
Prof. Christopher Pain

27th August 2021

GitHub repository: https://github.com/acse-2020/acse2020-acse9-finalreport-acse-jy220

Contents

Acknowledgements

First of all, I would like to express my great thankfulness to my supervisors: Dr. Claire Heaney and Prof. Christopher Pain. Without their support, encouragement and advise, I can not complete this project successfully. Furthermore, I would like to thank Dr. Marijan Beg for providing me lots of useful information through the Study Group sessions. Also I want to thank Mr. Adrian D'Alessandro who offered me help with the college HPC through RCS clinic. I want to thank my colleagues in the autoencoder group for their ideas and help. Lastly, I would like to thank my family for supporting me during the project.

Abstract. Convolutional Neural Networks are powerful tools for extracting features from data in high-dimensional space, however, they usually handle the data on a structured grid. Created by domain decomposition techniques, Space-filling curves could provide sub-optimal mappings from multi-dimensional space to one dimensional space, to which standard 1D Convolutional autoencoders could be applied. The combination of multiple orthogonal Space-filling curves could also improve the performance of the network. In this project, a python module using Pytorch has been developed to produce an automatic generation of this type of autoencoder for data on 2D and 3D unstructured meshes. The module has a high degree of flexibility and can compress any number of scalar components at the same time. A simple variational autoencoder was also included to explore a better-behaved latent space.

Keywords — **Dimension Reduction**, **Space-Filling Curve**, **Convolutional Neural Network**, **Autoencoder**

1 Introduction and Objectives

Having been developed over several decades, Reduced Order Modeling (ROM) techniques play an important role in computational physics. The computational cost for large physical systems is usually high, and constructing such reduced-order models (ROMs) that can precisely capture the characteristics of large system with a much lower computational cost (Lucia et al. 2004) could be of great interest. Linear dimension reduction methods such as Singular Value Decomposition (SVD) and Proper Orthogonal Decomposition (POD) are popular ROM methods. However, for systems with strong nonlinearities and advection-dominated systems, the linear ROM methods perform poorly (Lee & Carlberg 2019). As a deep neural network, Convolutional Autoencoders (CAEs) have been demonstrated to have stronger ability to tackle such systems (Maulik et al. 2021).

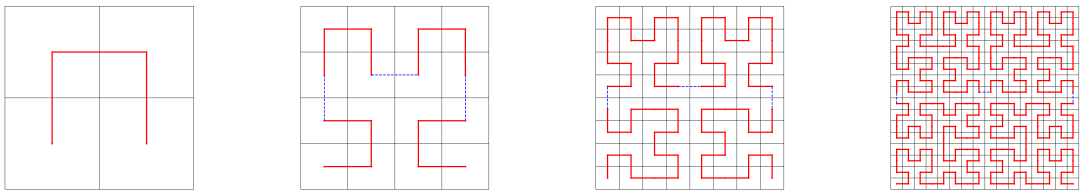
Based on Convolutional Neural Networks (CNNs), the CAEs are generally used on structured grids, for example an rectangular image consist of pixels. However, a large amount of computational fluid dynamics (CFD) models are consturcted over unstructured meshes to capture accurate enough flow patterns around boundary layers/diffusers (Tu et al. 2018). Direct application of CAEs on unstructured meshes is not feasible, alternative approaches involves domain-decomposition (Mack et al. 2020) and re-ordering data on unstructured meshes. The former one built a finer structured mesh, interpolate the data from the unstructured mesh to it, and then apply a classical CNN. However, errors will be introduced in the interpolation process. The latter one strongly depends on the quality of the ordering, if the ordering does not enable the convolutional filters to detect spatial correlations in the data, modelling the dynamic of the original system will be infeasible. Graph CNNs (Wu et al. 2021) are designed for data with irregular structures. The unstructured mesh, like the discontinuous Galerkin stencil could be abstracted as a graph, for which reason Graph-Based methods are appropriate for the problem.

Last year, the Applied Modelling and Computation Group (AMCG) at Imperial College London developed an approach of applying space-filling curves (Heaney et al. 2021) combining with a one-dimensional CAE, which successfully compressed the fluid data on a unstructured grid with low reconsturction error. This project aims to extend that piece of work, generalizing the SFC-CAE so that it can be applied to data on three dimensional unstructured meshes and also implementing a highly-automatic module which could generate an appropriate neural network based on the mesh input.

2 Generation and visualisation of Space-Filling Curves

2.1 Space-Filling Curves

In 1890, Giuseppe Peano (Peano 1890) discovered a countinuous curve which passes through every point into a unit square in the 2D space. The curve is now called *Peano curve*. A year later, David Hilbert (Hilbert 1891) found a continuous fractal space-filling curve, which is a variant of *Peano curve*. *Hilbert curve* maps points from 2D space into 1D space as well as preserving the locality, meaning that the neighbouring points in the Hilbert curve are also physically close in the origin dimension, however, the inverse is not always true. Because the *Hilbert curve* is a fractal curve, it could be expressed by a Lindenmayer system (Lindenmayer 1968). A Pseudo Hilbert curve can be easily constructed by recursion, as it cannot be infinitely magnified, of which the top-level one is a 'U'-like curve traverse through a 2×2 square grid. We take it as the first order PHC (Pseudo Hilbert curve) in the 2D space. As shown in Figure 1, by copying 4 PHCs of order n, rotate, and concatenate them together, we could get a PHC of order $n + 1$.



(a) 1st order Hilbert curve (b) 2nd order Hilbert curve (c) 3rd order Hilbert curve (d) 4th order Hilbert curve

Figure 1: The Hilbert curves on a 2-dimensional structured square grid of size 2^n

There are a couple of space-filling curves developed since then, such as the Moore curve (Moore 1900) and the Sierpiński curve (Platzman & Bartholdi 1989). The Space-filling curves (by recursive construction) are ideal to transform multidimensional data into one-dimensional space as it visits every cell exactly once and has the properties:
(1) If two data points are close to each other in the high dimensional space, they are also close to each other in the one-dimensional space-filling curve.
(2) If we take out a continuous segment of the space-filling curve, we will likely to have a convex subdomain in the original multidimensional domain.
(3) It is parameterised by the volume in the multidimensional space (Bader 2013) , the volume of a partition in the multidimensional domain is controlled by the interval in the space-filling curve, which defines Hölder continuity:

$$\|f(x) - f(y)\|_2 \leq C \sqrt[d]{V(f[x, y])} \ . \tag{1}$$

2.1.1 Space-Filling Curves on Unstructured meshes

Studies have shown that classical space-filling curves have good performance at preserving locality and data coherency on regular grids (Anjum et al. 2019, Zhou et al. 2020), but for unstructured meshes, most of the classical space-filling curves cannot be utilised directly. Generating high quality space-filling curves on unstructured meshes is sometimes nontrivial, which requires theory and methodology from subjects like Graph Theory. The information on a unstrcutured mesh can be abstracted to a graph, and to generate nice space-filling curves, appropriate graph partitioning method should be used to divide the graph into subdomains with almost equal numbers of vertices, as well as to minimise the communication between the subdomains. (Buluç et al. 2013) is a comprehensive survey for graph partitioning methods, including Global Algorithms like Spectral Partitioning, Geometric Partitioning, Streaming Graph Partitioning as well as Iterative Improvement Heuristics like Lin-Kernighan Local Search, Tabu Search, Bubble Framework and Random Walks. In our project, we will

use a local search method called MFT-RNN, which is a Recurrent Neural Network (Pain et al. 1999) approach for graph partitioning based on the Mean Field Theorem, we also inherit the implementation of the MFT-RNN over unstructured mesh from last year's work (Heaney et al. 2021) of the department.
The main objective function that the MFT-RNN (Pain et al. 1999) propose to minimize is

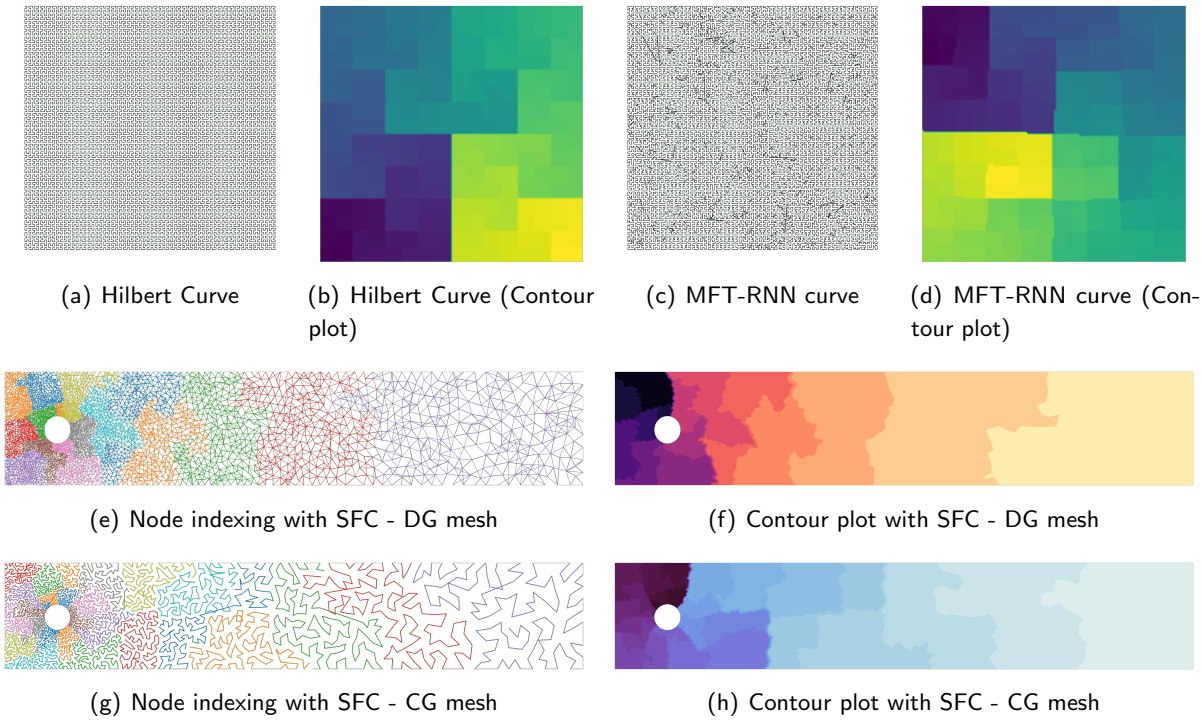
$$F = \frac{1}{2}\left[x^T \boldsymbol{K} x + \frac{1}{2}\alpha x^T \boldsymbol{C} x\right] \tag{2}$$

where $\boldsymbol{K}$ is a higher-order matrix which associated with the interpatition communications of the graph, while $\boldsymbol{C}$ is to balance the number of Nodes in each partition. With a appropriate α chosen, minimizing the functional could achieve the following two objectives simultaneously:

1. The communications between partitions are minimized.
2. The vertices in each partitions are almost equal.

The above two preperties define a good space-filling curve in high dimensional space. As Figure 2 shows, Hilbert SFC is actually an 'optimal' SFC on the 128×128 square grid because it traverses every node with the shortest path while the MFT-RNN space-filling curve is a less optimal one.

Besides regular grids, MFT-RNNs are also able to generate SFCs on unstrcutured grids, Figure 2 shows the space-filling curve ordering for 20550 Nodes of a discontinuous Galerkin (DG) mesh as well as 3571 Nodes of a continuous Galerkin mesh of a "flow past cylinder" problem.



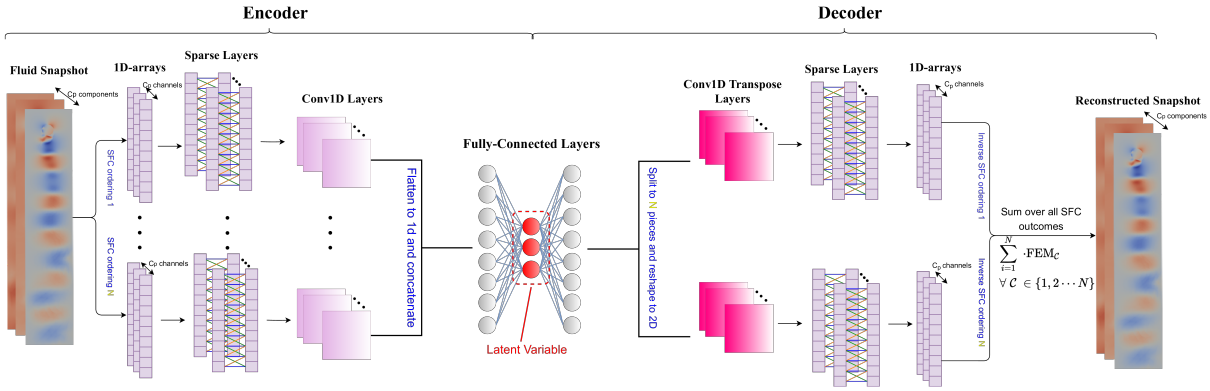
(a) Hilbert Curve (b) Hilbert Curve (Contour plot) (c) MFT-RNN curve (d) MFT-RNN curve (Contour plot)

(e) Node indexing with SFC - DG mesh (f) Contour plot with SFC - DG mesh

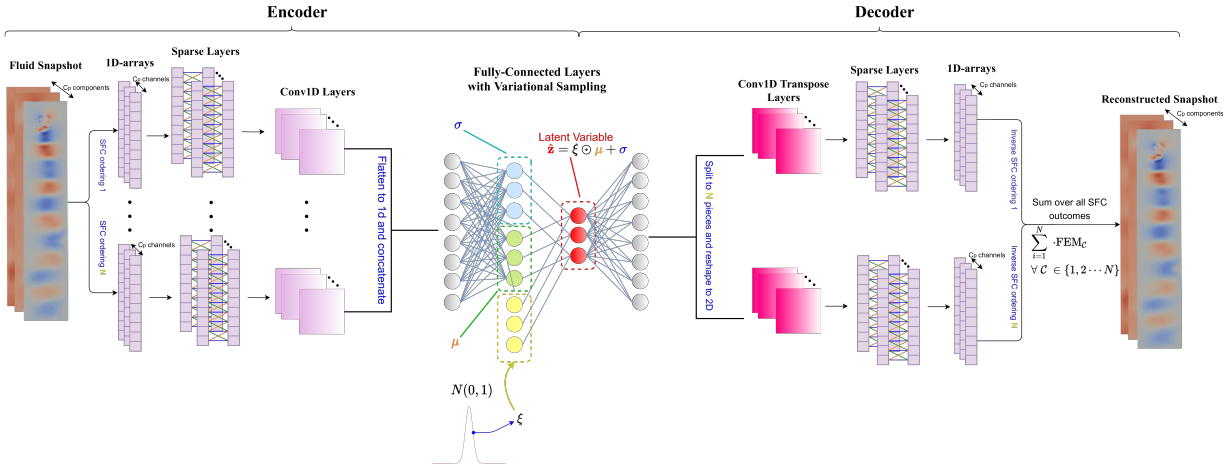
(g) Node indexing with SFC - CG mesh (h) Contour plot with SFC - CG mesh

Figure 2: SFC plots for 2D structured/unstructured meshes

3 Achitechture of the Autoencoder



(a) SFC-CAE



(b) SFC-VCAE

Figure 3: The Architecture of a Space-filling curve (Variational) Convolutional Autoencoder, the upper is a SFC-CAE and the lower is a SFC-VCAE.

The general Architecture of the Followed SFC-CAE is demonstrated by figure 3, based on SFC-CAE built for the 2D cases in (Heaney et al. 2021), all possible layers in our networks (in sequence) are:

1. SFC Ordering
2. Sparse layer (optional)
3. 1D Convolutional Layers
4. Fully-connected Layers
5. 1D Transpose Convolutional Layers
6. Sparse layer (optional)
7. inverse SFC Ordering

3.1 Encoder part

3.1.1 Apply SFC Orderings

Suppose we have a snapshot from structured/unstructured mesh $\boldsymbol{x} \in \mathbb{R}^{n \times C_p}$, of which the number of nodes is n, and number of scalar components (channels) is C_p. As long as we have generated N_c space-filling curves

by techniques described in 2, by simply applying space-filling curves for all channels we have ${}^{sfc\mathcal{C}}\boldsymbol{x}, \quad \forall \mathcal{C} \in \{1, ..., N_c\}$.

Self-Concat (optional) In some cases we perform a Self-Concatenate operation before passing to the later filters, in which case the channels of input doubled to $2C_p$, i.e. ${}^{sfc\mathcal{C}}\boldsymbol{x} \in \mathbb{R}^{n \times 2C_p}$ after operation.

$$ {}^{sfc\mathcal{C}}\boldsymbol{x} = \text{concat2}({}^{sfc\mathcal{C}}\boldsymbol{x}) \tag{3} $$

the notation $\text{concat2}(\boldsymbol{a}) = (\boldsymbol{a}^T, \boldsymbol{a}^T)^T$, represents the concatenate of a vector with itself over the last dimension.

3.1.2 Sparse layer (optional)

The Sparse layers, also Known as Nearest Neighbouring, are introduced to add smoothy on both the feature inputs as well as the final results. For some component u of ${}^{sfc\mathcal{C}}\boldsymbol{x}$, say ${}^{sfc\mathcal{C}u}\boldsymbol{x}$, we have the upper neighbour of it is:

$$ {}^{sfc\mathcal{C}u}\boldsymbol{x}_i^+ = \begin{cases} {}^{sfc\mathcal{C}u}\boldsymbol{x}_{i+1}, & i < n \\ {}^{sfc\mathcal{C}u}\boldsymbol{x}_n, & i = n \end{cases} \tag{4} $$

as well as the lower neighbour:

$$ {}^{sfc\mathcal{C}u}\boldsymbol{x}_i^- = \begin{cases} {}^{sfc\mathcal{C}u}\boldsymbol{x}_{i-1}, & i > 1 \\ {}^{sfc\mathcal{C}u}\boldsymbol{x}_1, & i = 1 \end{cases} \tag{5} $$

If the number of the channels (components) $C_p > 1$, to make a better parallelism of the algorithm, we first reshape ${}^{sfc\mathcal{C}}\boldsymbol{x} \in \mathbb{R}^{n \times C_p}$ into 1-dimensionl ${}^{sfc\mathcal{C}}\boldsymbol{x} \in \mathbb{R}^{nC_p}$, and introduce the extend SFCs:

$$ {}^{Exsfc\mathcal{C}}\boldsymbol{x}^+_{u \cdot n+i} = {}^{sfc\mathcal{C}u}\boldsymbol{x}_i^+ + (u-1) \cdot n \quad \forall u \in \{1, ..., C_p\} \tag{6} $$

$$ {}^{Exsfc\mathcal{C}}\boldsymbol{x}^-_{u \cdot n+i} = {}^{sfc\mathcal{C}u}\boldsymbol{x}_i^- + (u-1) \cdot n \quad \forall u \in \{1, ..., C_p\} \tag{7} $$

We then initialize weights $\{{}^{Exsfc\mathcal{C}}\boldsymbol{w}, {}^{Exsfc\mathcal{C}}\boldsymbol{w}^+, {}^{Exsfc\mathcal{C}}\boldsymbol{w}^-\}$ and a bias term ${}^{Exsfc\mathcal{C}}\boldsymbol{b}$ for the sparse layers, they are all 1-dimensional vectors of shape nC_p.

With appropriate activation function h chosen, the main calculation step for the sparse layer is:

$$ {}^{sfc\mathcal{C}}\hat{\boldsymbol{x}} = h({}^{Exsfc\mathcal{C}}\boldsymbol{w} \odot {}^{Exsfc\mathcal{C}}\boldsymbol{x} + {}^{Exsfc\mathcal{C}}\boldsymbol{w}^+ \odot {}^{Exsfc\mathcal{C}}\boldsymbol{x}^+ + {}^{Exsfc\mathcal{C}}\boldsymbol{w}^- \odot {}^{Exsfc\mathcal{C}}\boldsymbol{x}^- + {}^{Exsfc\mathcal{C}}\boldsymbol{b}) \tag{8} $$

where $\odot$ is the Hardamard product indicating element-wise multiplication. Note that the outcome ${}^{Exsfc\mathcal{C}}\hat{\mathbf{x}}$ is still a 1-dimensional vector of length nC_p, so we need to reshape it back to $\mathbb{R}^{n \times C_p}$ to form ${}^{sfc\mathcal{C}}\hat{\mathbf{x}}$.

3.1.3 1D Convolutional Layers

1D Convolutional Layers, also known as 1D-CNN, acts in a similar way to 2D CNNs, but of which the filter itself as well as the input are both 1-dimensional. If we have a input of size ${n_H}^l$, with n_{c_1} channels, after applying a 1D-CNN of kernel size f, padding p, stride s with n_{c_2} channels, the output size ${n_H}^{l+1}$ would be:

$$ {n_H}^{l+1} = \frac{{n_H}^l + 2p - f}{s} + 1 \tag{9} $$

with n_{c_2} channels.

3.1.4 Down sampling fully-connected (FC) Layers

The size of neurons of each FC are defined by a intuiative algorithm described in subsection 3.3, in brief words, each level we divide the number of neurons by the stride s of the 1D Convolutional Layers:

$$h_{FC}{}^{l+1} = \lfloor \frac{h_{FC}^{l}}{s} \rfloor \tag{10}$$

until the number of neurons reach the dimension of Latent Variable n_ξ we defined.

3.2 Decoder part

The objective of a Decoder is to reconstruct the snapshot from the latent variable. Thus a natural thought is to build nearly symmetric layers of the Encoder, which performs an inverse operation of the Encoder at every step.

3.2.1 Up sampling fully-connected (FC) Layers

To keep symmetry of the whole network, the size $\hat{h}_{FC}^{l}$ of each layer of the up sampling fully-connected Layers are chosen to be exactly the same as the Down sampling fully-connected Layers defined in 3.1.4, but in an inverse sequence:

$$\hat{h}_{\mathrm{FC}}^{l} = \hat{h}_{\mathrm{FC}}^{n_{\mathrm{FC}}-l} \tag{11}$$

where n_{FC} is the number of down sampling fully-connected Layers.

3.2.2 1D Transpose Convolutional Layers

The kernel size f, padding p, stride s are chosen to be exactly the same as the 1D Convolutional Layers defined in 3.1.3, the number of channels of the filters are chosen to be the same as the 1D Convs in a inverse sequence, like we did for up sampling fully-connected Layers. However, due to an floor rounding of output size in Pytorch, to keep the symmetry at each 1D TransConv with respect to the corresponding 1D Conv layer, an extra output padding ${p_o}^l$ is needed for each TransConv layer:

$${p_o}^l = \hat{n}_H^{n_{\mathrm{Conv}}-l} \mod s \tag{12}$$

where n_{Conv} is the number of 1D Conv layers, and $\hat{n}_H$ the size of the training variable after each 1D Conv layer.

3.2.3 Sparse Layers in Decoder (Optional)

To keep the symmetry of the system, if the sparse layers have been added at the encoder, we should also have them in the decoder. Also notice if we have an addtional channel copying operation defined in 3.1.1, thus we should first split our training variable $\boldsymbol{x}$ into n_{concat} sub-variable corresponding to the channel copying operation, then build a sparse layer with $C_p \times n_{\mathrm{concat}}$ weights of size nC_p, as well as a bias term ${}^{\mathrm{Exsfc}\mathcal{C}}\boldsymbol{b}$:

$${}^{Exsfc\mathcal{C}}\hat{\boldsymbol{x}} = h\left(\sum_{i}^{n_{\mathrm{concat}}}\left({}^{Exsfc\mathcal{C}}\boldsymbol{w_i} \odot {}^{Exsfc\mathcal{C}}\boldsymbol{x_i} + {}^{Exsfc\mathcal{C}}\boldsymbol{w_i^+} \odot {}^{Exsfc\mathcal{C}}\boldsymbol{x_i^+} + {}^{Exsfc\mathcal{C}}\boldsymbol{w_i^-} \odot {}^{Exsfc\mathcal{C}}\boldsymbol{x_i^-}\right) + {}^{Exsfc\mathcal{C}}\boldsymbol{b}\right) \tag{13}$$

where h is the activation function, i represents one certain channel from the concatenation. Note that the outcome ${}^{Exsfc\mathcal{C}}\hat{\mathbf{x}}$ is still a 1-dimensional vector of length nC_p, so we need to reshape it back to $\mathbb{R}^{n\times C_p}$ to form ${}^{sfc\mathcal{C}}\hat{\mathbf{x}}$.

3.2.4 Apply inverse Orderings and Final output

Remember we applied applied the space-filling orderings to our input $\boldsymbol{x}$ to form ${}^{sfc\mathcal{C}}\boldsymbol{x}$, $\forall\mathcal{C} \in \{1, ..., N_c\}$ in 3.1.1, so before final output we need to apply inverse space-filling orderings to those ${}^{sfc\mathcal{C}}\boldsymbol{x}$ correspondingly.

The inverse space-filling orderings are just inverse maps for space-filling orderings in the permutation group S_n:

$$\mathrm{invsfc}\mathcal{C} \cdot \mathrm{sfc}\mathcal{C} = \mathrm{sfc}\mathcal{C} \cdot \mathrm{invsfc}\mathcal{C} = I, \qquad I \text{ is the Identity map in } S_n \tag{14}$$

where S_n represents the permutation group of dimension n, and the inverse map of an ordering $\tau = \{x1, x2 \dots xn\} \in S_n$ is just simply:

$$\tau_i^{-1} = \underset{x_i \in \tau}{\mathrm{argsort}}\ \tau = |\{x_j | x_j < x_i, x_i \in \tau\}| \tag{15}$$

Thus by applying $\mathrm{invsfc}\mathcal{C}$ to ${}^{sfc\mathcal{C}}\boldsymbol{x}$ respectively, we could revert the ordering of the data on the mesh before putting into the Encoder. We define those reordered data as ${}^{\mathrm{fem}\mathcal{C}}\boldsymbol{x}$, the our ANN (artificial neural network) output is computed by

$$\boldsymbol{x}_{\mathrm{out}} = \sum_{\mathcal{C}=1}^{N_{\mathcal{C}}} {}^{\mathrm{fem}\mathcal{C}}\boldsymbol{x} \tag{16}$$

of which the shape is $n \times C_p$, the same as the ANN input $\boldsymbol{x}$.

3.3 Finding size for 1D Conv (TransConv) layers and Fully-connected layers

Algorithm 1 is implemented to find the sizes of 1D Conv/TransConv layers and fully-connected layers for an unseen snapshot input $\mathbf{x} \in \mathbb{R}^{n \times C_p}$, the variables in the algorithms are:
Input:

- $\mathbf{n}$: number of Nodes in the snapshot.
- C_p: number of components we are compressing (initial channels).
- $\mathbf{n}_\xi$: size of the latent variable.
- $\mathbf{f}$: (constant) kernel size of the 1D convolutional layers.
- $\mathbf{p}$: (constant) padding of the 1D convolutional layers.
- $\mathbf{s}$: (constant) stride of the 1D convolutional layers.
- $N_{\mathcal{C}}$: number of space-filling curves generated.
- k: the increasing multiplication of channels in consecutive 1D Conv layers.
- C_M: the maximum number of channels that are allowed in the 1D Conv Layers.

Output:

- L_{CONV}: the list of sizes of the output after each 1D Conv Layers.
- L_{CH}: the list of number of Channels of the output after each 1D Conv Layers.
- L_{FC}: the list of neurons of down-sampling fully-connected layers.
- L_{OP}: the list of output paddings of the 1D Transpose Conv layers.

To make the 1D SFC-CNN roughly analagouse to the 2D/3D classiacal CNN, we choose $f = 32$, $s = 4$ for 2D unstructured meshes and $f = 176$, $s = 8$ for 3D unstructured meshes, the padding p is chosen to be $p = \frac{1}{2}f$ for both 2D and 3D. Other combinations of kernel size, stride and padding could be tried as our algorithm are implemented for arbitary f, s, p inputs, however, by a number of experiments, the above parameters are the

most appropriate concerning both reconstruction accuracy and memory management.

Algorithm 1: Find Conv layers and FC layers

Data: $\mathbf{n}, \mathbf{n}_\xi, \mathbf{f}, \mathbf{p}, \mathbf{s}, N_\mathcal{C}, k, C_p, C_M$
Result: $L_{\mathrm{CONV}}, L_{\mathrm{CH}}, L_{\mathrm{FC}}, L_{\mathrm{OP}}$
Initialize $L_{\mathrm{CONV}}, L_{\mathrm{CH}}, L_{\mathrm{FC}}, L_{\mathrm{OP}}$ as empty lists ;
$n_H \leftarrow \mathbf{n}$;
if *self-concatenate* **then**
  $C \leftarrow 2C_p$;
else
  $C \leftarrow C_p$;
end
L_{CONV}.append(n_H) ;
L_{CH}.append(C) ;
—— Building convolutional layers ——
while $n_H \times C_M \times N_\mathcal{C} > 4000$ **do**
  $n_H \leftarrow \lfloor \frac{n_H + 2p - f}{s} \rfloor + 1$; `/* In Pytorch a floor rounding was automatically taken when the shape is not a integer */`
  L_{CONV}.append(n_H) ;
  L_{OP}.append($(n_H + 2p - f) \mod s$) ;
  if $C \times k < C_M$ **then**
    $C \leftarrow C \times k$;
    L_{CH}.append(C) ;
  else
    L_{CH}.append(C_M) ;
  end
end
—— Building fully-connected layers ——
$n_{\mathrm{FC}} \leftarrow n_H \times C_M \times N_\mathcal{C}$;
L_{FC}.append(n_{FC}) ;
while $\lfloor \frac{n_{\mathrm{FC}}}{s^{1.5}} \rfloor > \mathbf{n}_\xi$ **do**
  $n_{\mathrm{FC}} \leftarrow \lfloor \frac{n_{\mathrm{FC}}}{s} \rfloor$;
  if $n_{\mathrm{FC}} \times s < 100$ *and* $n_{\mathrm{FC}} < 50$ **then**
    break ;
  else
    pass;
  end
  L_{FC}.append(n_{FC}) ;
end
L_{FC}.append(n_ξ)

3.4 A built-in Variational Autoencoder

As shown in figure 3, at the last layer of down-sampling fully-connected layers, instead of directly pass it to the latent variable ξ, we pass the last FC layer to two vectors μ and σ of the same size of latent variable ξ, also we sample a random vector ϵ from standard normal distribution $N(0, 1)$, as the same size of latent variable ξ, the latent variable is then sampled by

$$\hat{\xi} = \epsilon \odot \mu + \sigma \tag{17}$$

This technique is called reparametrization trick, which makes the process of the back propagation easier.

4 Code Metadata

4.1 Environment for Development

The generation of space-filling curves on unstructured meshes is achieved with a fortran library, which contains the algorithm called MFT-RNN (Pain et al. 1999), developed by Prof. Christopher Pain. The main class of the space-filling curve autoencoder module is implemented mainly based on the popular deep learning library **Pytorch** (Paszke et al. 2019), besides other powerful libraries such as **meshio** (Schlömer 2021) / **vtktools** for reading/writing vtu files, **scipy** (Virtanen et al. 2020) and **numpy** (Paszke et al. 2019) are also important for producing simulations on structured grids. **ParaView** is used for visualising the result on unstructured meshes.

4.2 Module Overview

The directory tree for useful source scripts in the repository (have skipped the configuration files) are showed in the directory tree 4.2:

```
{home dir}
├── sfc_cae
│   ├── advection_block_analytical.py
│   ├── sfc_cae.py
│   ├── simple_hilbert.py
│   ├── structured.py
│   ├── training.py
│   └── utils.py
├── tests
│   └── tests.py
├── command_train.py
├── parameters.ini
├── space_filling_decomp.f90 (externel library)
└── vtktools.py (externel library)
```

The main functions are inside the **sfc_cae** folder, while the detailed function of each script are:

- advection_block_analytical.py provides simulation Code as well as the animation generation for pure advection problems on structured grids 5.4.1.
- sfc_cae.py contains the main class for the self-adjusting SFC-(V)CAE, inherits from a *Pytorch nn.module*.
- simple_hilbert.py is a simple implementation of creating Hilbert Curves (see Section 2) on a $2^n \times 2^n$ square grid.
- structured.py contains useful functions for finding sparse Matrix and creating MFT-RNN curves on a structured grid.
- training.py contains useful functions for loading, training (with visualization) and saving the model.
- utils.py contains useful functions for file I/O, data transformation, custom dataset and some custom layers for the SFC-(V)CAE module.

There are also some useful scripts outside the **sfc_cae** folder:

- command_train.py is used for training via command_line over a non-GUI operating system, e.g. the College HPC, used in combination with parameters.ini.
- parameters.ini defines the important parameters for initialize/training over the non-GUI operating system.

Tests are contained in tests/tests.py, an automated testing workflow is built on Github to test all the useful functions in the sfc_cae folder.

4.3 A Table Generator for viewing the network structure

The layers for a SFC_CAE class could be directly viewed via calling

```
autoencoder.parameters
```

in a Python environment. However,to have a better view of the network structure, an automatic **LaTeX** table generator is implemented within the SFC_CAE class, by calling

```
autoencoder.output_structure()
```

A txt file called 'LatexTable.txt' would be produced in the working directory, just copy and paste it into a LaTeX compiler, a nice table demonstrating the autoencoder architecture would be generated.

5 Result and Analysis

Four examples are used to measure the performance/demonstrate the results of the genralized SFC-(V)CAE, in sequence, they are:

1. An "advecting" block/gaussian, 2D
2. Flow past Cylinder, 2D
3. CO_2 in the room, 3D
4. Slugflow, 3D

5.1 Error Metric

The objective function used for evaluating the reconsturction error over standard SFC-CAE is the Mean Square Error (MSE):

$$\mathbb{MSE} = \frac{\sum_{i=1}^{T} (\hat{\mathbf{x}} - \mathbf{x}) \cdot (\hat{\mathbf{x}} - \mathbf{x})}{Tn}. \tag{18}$$

Where constant T is the total number of time levels (same to the number of snapshots), and the constant n is the number of Nodes in each snapshot as defined in Section 3.3, $\mathbf{x}$ is the orignial snapshot and $\hat{\mathbf{x}} = f^{AE}(\mathbf{x})$ is the reconstructed snapshot.

For a Variational Autoencoder (SFC-VCAE), an additional Kullback–Leibler divergence (KL divergence) is added to the MSE:

$$\mathbb{KL}\left(\mathcal{N}(\mu, \sigma) \parallel \mathcal{N}(0, 1)\right) = \sum_{i=1}^{n_\epsilon} \left(\sigma_i^2 + \mu_i^2 - \log \sigma_i - \frac{1}{2}\right) \tag{19}$$

Because we need to majorly focus on reducing the reconstruction error, an intuitive coefficient

$$\beta = \frac{1}{T \cdot \mathbf{n}} \tag{20}$$

for the KL divergence is introduced, thus the overall Loss for the SFC-VCAE is defined by:

$$Loss = \mathbb{MSE} + \beta \mathbb{KL} \tag{21}$$

5.2 Data Scaling

Before training, note that we are training multiple components/ channels of the same time, the average magnitude of them may have huge difference, for example, in the CO_2 dataset, the average value of **CO_2 - ppm** is over 1000, while the velocity components have a maximum value below 10. We want to balance

this effect during training, thus we rescale the data to an appropriate interval $[t_1, t_2]$ channel-wisely by linear maps:

$$\begin{aligned} &\forall i = 1 \dots C_p, \\ &\qquad \boldsymbol{k}_i = \frac{t_2 - t_1}{\max\limits_{n\in\{1\dots N\}} \mathbf{x}_{in} - \min\limits_{n\in\{1\dots N\}} \mathbf{x}_{in}}, \\ &\qquad \boldsymbol{b}_i = \frac{\left(\max\limits_{n\in\{1\dots N\}} \mathbf{x}_{in}\right) t_1 - \left(\min\limits_{n\in\{1\dots N\}} \mathbf{x}_{in}\right) t_2}{\max\limits_{n\in\{1\dots N\}} \mathbf{x}_{in} - \min\limits_{n\in\{1\dots N\}} \mathbf{x}_{in}}, \\ &\qquad \hat{\mathbf{x}_i} = \boldsymbol{k}_i \cdot \mathbf{x}_i + \boldsymbol{b}_i. \end{aligned} \tag{22}$$

where $\mathbf{x}_i \in \mathbb{R}^{T\times N}$ represents a certain component of the simulation over all timesteps, $\boldsymbol{k}_i$ and $\boldsymbol{b}_i$ are coefficients of length C_p, the number of components we are compressing. To fit the range of the non-linear activation function, we choose $[t_1, t_2] = [0, 1]$ for **ReLU** activation, and $[t_1, t_2] = [-1, 1]$ for **Tanh** activation.

When we reconstruct results, we revert this scaling operation:

$$\begin{aligned} &\forall i = 1 \dots C_p, \\ &\qquad \tilde{\mathbf{x}_i} = \frac{f^{\mathrm{AE}}(\hat{\mathbf{x}_i}) - \boldsymbol{b}_i}{\boldsymbol{k}_i}. \end{aligned} \tag{23}$$

where $\tilde{\mathbf{x}} \in \mathbb{R}^{T\times N\times C_p}$ is the reconstructed result.

5.3 Hyper-parameters

For each problem, the parameters of the Layers, such as kernel size, padding, stride have been introduced in Section 3.3. Other hyper-parameters associated with training such as batch size, optimizer type, learning rate, activation function, by lot of experiments made, we have found a nearly optimal combination of them for each problem, as showed in Table 1.

Problem	components	batch size	optimizer	learning rate	activation functions
Advection in 2D structured grid	1	64	Adam	1×10^{-4}	ReLU
Flow Past Cylinder	2	16	Adam	1×10^{-4}	Tanh
CO2 in the room	4	16	Adamax	1×10^{-3}	Tanh
Slugflow	4	16	Adamax	1×10^{-3}	Tanh

Table 1: The appropriate hyper-parameters used for training the SFC-CAEs.

5.4 Structured Grid

5.4.1 An advecting block in 2D

The Partial Differetial Equation (PDE) for two-dimensional pure advection problem can be written as:

$$\frac{\partial c}{\partial t} = -\boldsymbol{U}\frac{\partial c}{\partial x} - \boldsymbol{V}\frac{\partial c}{\partial y} \tag{24}$$

where c is the scalar concentration, $\boldsymbol{U}$ and $\boldsymbol{V}$ are constant advection velocities over the two dimension x and y respectively.

If we suppose the initial condition for our time-dependent concentration $c(x, y, t)$ is

$$c(x, y, 0) = c^0(x, y) \tag{25}$$

Then the analytical solution for the two-dimensional pure advection problem is

$$c(x, y, t) = c^0(x - \boldsymbol{U}t, y - \boldsymbol{V}t) \tag{26}$$

For the "advecting" block problem we implement the Square wave initial condition:

$$c^0(x,y) = \begin{cases} 1, (x,y) \in \left[x_0 - \frac{d}{2}, x_0 + \frac{d}{2}\right] \times \left[y_0 - \frac{d}{2}, y_0 + \frac{d}{2}\right] \\ 0, \text{otherwise} \end{cases} \tag{27}$$

where (x_0, y_0) the initial location of the center of the block, and d the length of the block.

We use the Hilbert/MFT-RNN space-filling curve generated in Section 2.1.1, built a classical 2D-CAE as the same architecture described in Section 3.2 of (Heaney et al. 2021), compressing data of size 128×128 to 16 latent variables. 20200 snapshots were devided to Train set: Valid set: Test set = 8: 1: 1, we use the mean square error (MSE) (Köksoy 2006) to measure the error, the data is trained with the hyper-parameters defined in 5.3 with 2000 epochs, the results are showed in table 4.

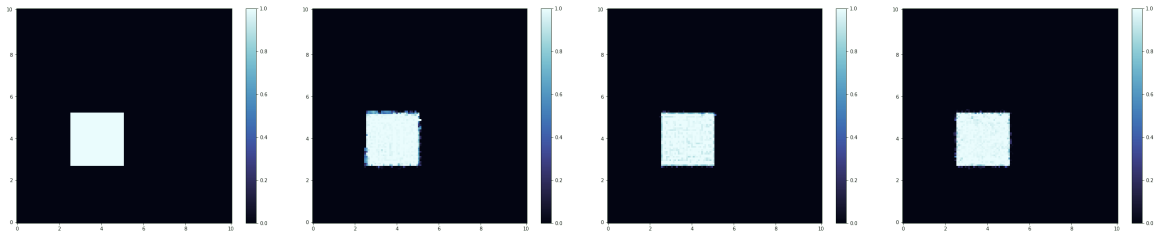
5.4.2 An advecting gaussian in 2D

Because the advecting block problem has a huge discontinuity at the block's boundary, to better test the performance of the neural network, a smooth Gaussian initial condition is also used:

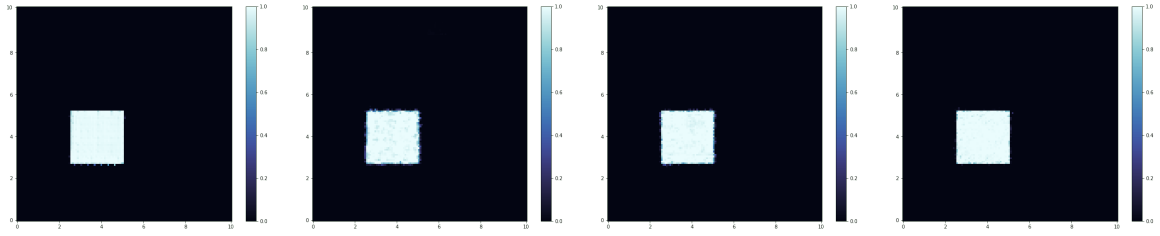
$$c^0(x,y) = exp\left(-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}\right)$$

the amount of snapshots, spliting of the dataset, dimension of latent space is same to the block problem, also trained with the hyper-parameters defined in 5.3 with 2000 epochs, MSE are showed in table 4, reconstructed snapshots are shown in Figure 4. Also the performance of the 2-SFC-CAE-NN on different number of latent variables are evaluated, the Losses are showed in Table 4.

From observing the final losses and the reconstructed snapshots, the reconstruction snapshots utilizing a single MFT-RNN curve are blotchy while the Hilbert SFC are more smooth. As (b), (f) of Figure 4 shows, adding sparse layers would improve the smoothness of both MFT-RNN/ Hilbert SFC-CAEs. Moreover, combining two MFT-RNN curves would greatly improve the quality of reconstruction, of which the reconstruction error is even less than a standard 2D CAE (showed in Table 4).

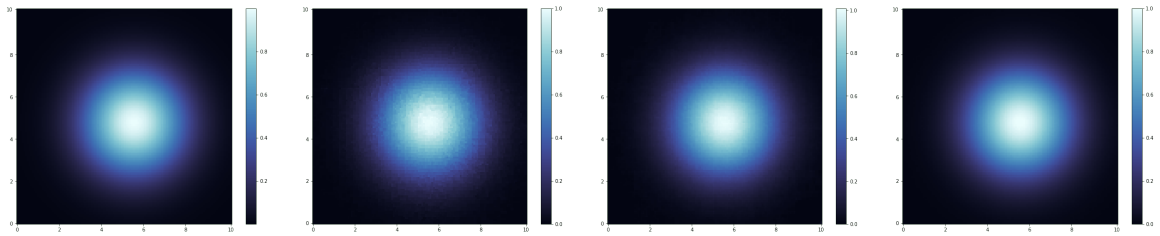


(a) Original snapshot (b) MFT-RNN SFC-CAE (c) Hilbert SFC-CAE (d) MFT-RNN-2-SFC-CAE

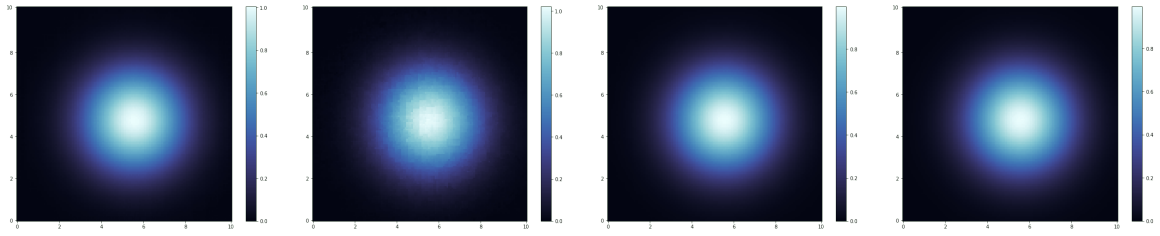


(e) 2D CAE (f) MFT-RNN SFC-CAE-NN (g) Hilbert SFC-CAE-NN (h) MFT-RNN-2-SFC-CAE-NN

(A) Snapshots of a square wave advecting in a 128×128 grid. The original figure (top left) is also placed here for comparison.



(i) Original snapshot (j) MFT-RNN SFC-CAE (k) Hilbert SFC-CAE (l) MFT-RNN-2-SFC-CAE



(m) 2D CAE (n) MFT-RNN SFC-CAE-NN (o) Hilbert SFC-CAE-NN (p) MFT-RNN-2-SFC-CAE-NN

(B) Snapshots of a Gaussian wave advecting in a 128×128 grid. The original figure (top left) is also placed here for comparison.

Figure 4: Reconstrcution snapshots on 128×128 square grid, Part (A) is the advection of a block wave, Part (B) is the advection of a gaussian wave. The compression level for all autoencoders is 16.

5.5 2D Flow Past Cylinder

The 2D Flow Past Cylinder datasets are generated by simulations of incompressible fiuld in 2D (Heaney et al. 2021):

$$\nabla \cdot \mathbf{u} \quad = \quad 0, \tag{28}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) + \nabla \cdot \tau \quad = \quad -\nabla p \tag{29}$$

The operation $\otimes$ represents tensor product, ρ is constant density, τ contains stress and viscous term, $\mathbf{u}$ is the velocity vector in 2D, t is time, p is pressure and gradient operator $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})^T$. The Reynolds number Re for this problem is 3900.

5.5.1 Discontunous/Continuous Galerkin Mesh

The velocity data for FPC-DG is based on a discontinuous Galerkin (DG) finite element mesh, where each elements has three neighbours on that mesh. The velocity data for FPC-CG is based on a Continuous Galerkin (CG) finite element mesh, the elements could have common Nodes. The discontinuous Galerkin (DG) mesh has $n = 20550$ Nodes and the continuous Galerkin (CG) mesh has 5137 Nodes, both datasets have $T = 2000$ snapshots, with velocity $\mathbf{u}$ in 2 dimension, e.g.$C_p = 2$. The snapshots for both datasets were split to Train set: Valid set: Test set $=$ 8: 1: 1 for training.

We created space-filling curves with MFT-RNN (As shown in Figure 2) on two meshes, built SFC-CAEs with 2 space-filling curves to concatenate and train the x and y component of the 2-D velocity, which compressed the snapshot to 16 latent variables. The data was trained with the hyper-parameters defined in 5.3 with 2000 epochs, final losses are showed in table 2. The reconstructed snapshots of the Autoencoders are showed in Figure 5.

From the results, we could see that the SFC-(V)CAE with sparse layers would always outperform those without sparse layers, and at same number of epoches, the SFC-VCAE will always be bigger MSE loss than a SFC-CAE as it has to make a compromise to reduce the KL loss at the same time. Moreover, just looking at the final loss of the 2-SFC-CAE-NN in Table 2 over different compression levels, we can see that our autoencoder achieved a very good performace at 16 or 8 latent varibles, but start to have overfitting when the latent variable is less than 4. And when the latent variable is set to 1, the accuracy of the autoencoder is not as good as other compression levels.

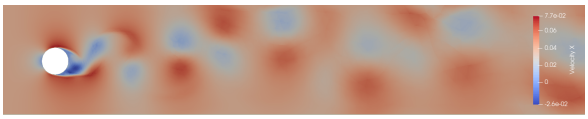
(a) Flow Past Cylinder - Discontinuous Gelarkin

	2-SFC-CAE	2-SFC-CAE-NN	2-SFC-VCAE		2-SFC-VCAE-NN		Latent
			MSE	KL	MSE	KL	
Train set	1.01×10^{-4}	5.17×10^{-5}	1.43×10^{-4}	3.49×10^{-4}	1.49×10^{-4}	3.04×10^{-4}	
Valid set	1.16×10^{-4}	8.40×10^{-5}	2.26×10^{-4}	3.55×10^{-4}	1.94×10^{-4}	3.15×10^{-4}	16
Test set	1.11×10^{-4}	6.47×10^{-5}	2.33×10^{-4}	3.54×10^{-4}	2.77×10^{-4}	3.14×10^{-4}	
Train set	9.66×10^{-5}	9.03×10^{-5}	1.88×10^{-4}	2.86×10^{-4}	1.39×10^{-4}	2.82×10^{-4}	
Valid set	1.57×10^{-4}	1.16×10^{-4}	3.16×10^{-4}	2.85×10^{-4}	1.97×10^{-4}	2.82×10^{-4}	8
Test set	2.08×10^{-4}	1.02×10^{-4}	2.51×10^{-4}	2.84×10^{-4}	2.80×10^{-4}	2.78×10^{-4}	
Train set	1.07×10^{-4}	6.68×10^{-5}	2.17×10^{-4}	2.42×10^{-4}	1.40×10^{-4}	2.39×10^{-4}	
Valid set	1.81×10^{-4}	1.14×10^{-4}	4.84×10^{-4}	2.45×10^{-4}	2.56×10^{-4}	2.38×10^{-4}	4
Test set	1.75×10^{-4}	9.50×10^{-5}	3.55×10^{-4}	2.55×10^{-4}	3.09×10^{-4}	2.37×10^{-4}	
Train set	2.28×10^{-4}	1.71×10^{-4}	2.60×10^{-4}	2.02×10^{-4}	2.14×10^{-4}	2.01×10^{-4}	
Valid set	8.16×10^{-4}	7.30×10^{-4}	7.25×10^{-4}	2.01×10^{-4}	8.09×10^{-4}	2.02×10^{-4}	2
Test set	8.24×10^{-4}	8.04×10^{-4}	6.90×10^{-4}	2.03×10^{-4}	8.07×10^{-4}	2.01×10^{-4}	
Train set	1.93×10^{-3}	1.32×10^{-3}	2.54×10^{-3}	1.98×10^{-4}	1.75×10^{-3}	1.88×10^{-4}	
Valid set	2.10×10^{-3}	1.97×10^{-3}	3.08×10^{-3}	1.97×10^{-4}	2.19×10^{-3}	1.87×10^{-4}	1
Test set	2.13×10^{-3}	1.92×10^{-3}	3.17×10^{-3}	2.01×10^{-4}	1.94×10^{-3}	1.89×10^{-4}	

(b) Flow Past Cylinder - Continuous Gelarkin

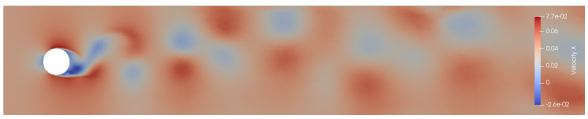
	2-SFC-CAE	2-SFC-CAE-NN	2-SFC-VCAE		2-SFC-VCAE-NN		Latent
			MSE	KL	MSE	KL	
Train set	6.79×10^{-5}	4.06×10^{-5}	4.07×10^{-4}	1.60×10^{-3}	4.10×10^{-4}	1.60×10^{-3}	
Valid set	7.72×10^{-5}	6.04×10^{-5}	5.64×10^{-4}	1.60×10^{-3}	5.63×10^{-4}	1.63×10^{-3}	16
Test set	7.76×10^{-5}	6.35×10^{-5}	5.78×10^{-4}	1.61×10^{-3}	5.35×10^{-4}	1.62×10^{-3}	
Train set	9.59×10^{-5}	7.36×10^{-5}	1.95×10^{-3}	1.15×10^{-3}	5.06×10^{-4}	9.09×10^{-4}	
Valid set	1.54×10^{-4}	8.18×10^{-5}	1.96×10^{-3}	1.14×10^{-3}	8.53×10^{-4}	9.00×10^{-4}	8
Test set	1.34×10^{-4}	8.50×10^{-5}	1.85×10^{-3}	1.16×10^{-3}	1.08×10^{-3}	9.03×10^{-4}	
Train set	8.64×10^{-5}	6.59×10^{-5}	4.94×10^{-4}	9.61×10^{-4}	2.20×10^{-4}	2.01×10^{-4}	
Valid set	1.46×10^{-4}	8.80×10^{-5}	8.24×10^{-4}	9.51×10^{-4}	7.00×10^{-4}	2.03×10^{-4}	4
Test set	1.50×10^{-4}	1.08×10^{-4}	9.49×10^{-4}	9.69×10^{-4}	8.12×10^{-4}	2.02×10^{-4}	
Train set	1.03×10^{-4}	9.71×10^{-5}	4.39×10^{-4}	1.48×10^{-3}	2.88×10^{-4}	9.95×10^{-4}	
Valid set	9.14×10^{-4}	8.21×10^{-4}	8.21×10^{-4}	1.49×10^{-3}	6.93×10^{-4}	9.98×10^{-4}	2
Test set	5.27×10^{-4}	4.13×10^{-4}	7.56×10^{-4}	1.47×10^{-3}	7.95×10^{-4}	9.91×10^{-4}	
Train set	1.93×10^{-3}	1.80×10^{-3}	2.16×10^{-3}	6.34×10^{-4}	2.01×10^{-3}	5.51×10^{-4}	
Valid set	2.27×10^{-3}	2.10×10^{-3}	2.32×10^{-3}	6.33×10^{-4}	2.23×10^{-3}	5.46×10^{-4}	1
Test set	2.35×10^{-3}	2.06×10^{-3}	2.52×10^{-3}	6.30×10^{-4}	2.52×10^{-3}	5.48×10^{-4}	

Table 2: The losses for SFC-CAE on the FPC data, compressed to 16, 8, 4, 2, 1 latent variables, trained with the Hyper-parameters defined in 5.3, we can see 2-SFC-CAE-NN outperforms other types of autoencoder with 2 SFCs in MSE loss. Moreover, results for compression level of 16, 8, 4 (in red) are more reliable than 2 or 1 (in blue).

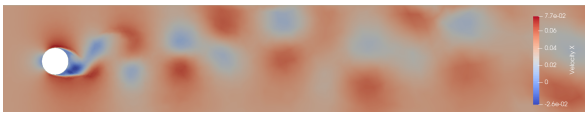


(a) original x velocity at t = 1270 second

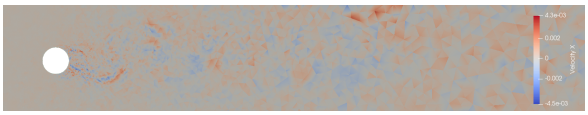
(b) original x velocity at t = 1675 second



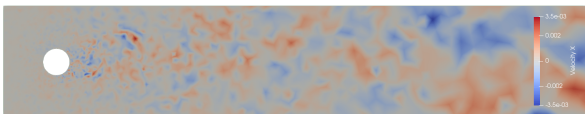
(c) reconstructed x velocity at t = 1270 second



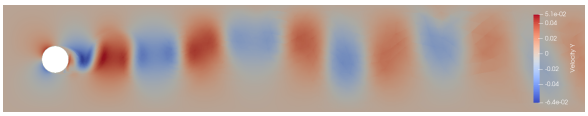
(d) reconstructed x velocity at t = 1675 second



(e) pointwise error of x velocity at t = 1270 second

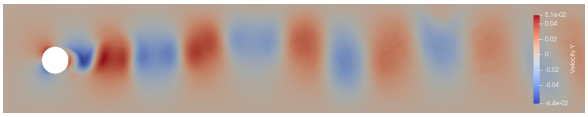


(f) pointwise error of x velocity at t = 1675 second

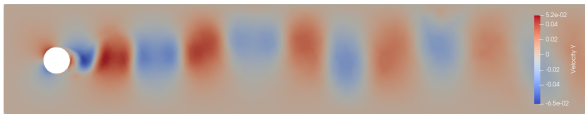


(g) original y velocity at t = 1270 second

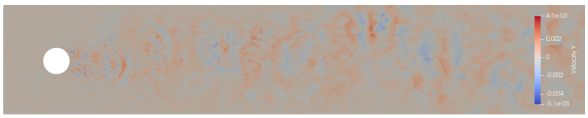
(h) original y velocity at t = 1675 second



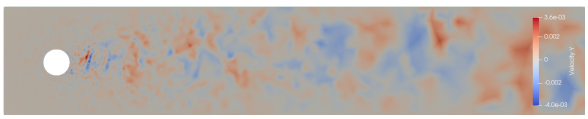
(i) reconstructed y velocity at t = 1270 second



(j) reconstructed y velocity at t = 1675 second



(k) pointwise error of y velocity at t = 1270 second



(l) pointwise error of y velocity at t = 1675 second

Figure 5: Reconstructed snapshots for SFC-CAE on FPC-DG and FPC-CG meshes. Results in the left column are reconstrcuted by a 2-SFC-CAE-NN compressed to 16 latent variables. Results in the right column are reconstrcuted by a 2-SFC-CAE-NN compressed to 4 latent variables.

5.6 3D unstructured mesh

For 3D unstructured mesh, we will modify the kernel size and stride as introduced in 3.3, using the the Hyper-parameters listed in 5.3.

5.6.1 CO_2 in the Room

The CO_2 dataset is based on a 3-dimensional Continuous Galerkin (CG) mesh. The mesh has $n = 148906$ Nodes, with 3 velocity components in x ,y, z axis and an additional scalar component **CO_2 - ppm**, e.g.$C_p = 4$. The whole simulation $X \in \mathbb{R}^{455\times148906\times4}$ was split to Train set: Valid set: Test set = 8: 1: 1 for training.

We created 3 Space-filling curves on this mesh as showed in 9, and trained the data with a 3-SFC-CAE-NN and 3-SFC-VCAE-NN with latent variable of $n_\xi = 16, 8$ and 4 with 2000 epoches. Final Losses are shown in the table 3.

5.6.2 Slugflow

The PDE equations for slugflow simulation in 3D are quite similar to those in 5.5, however a additional momentum source $\mathbf{s}_u$ is added. Thus equation 29 would become

$$\frac{\partial}{\partial t}(\rho\mathbf{u}) + \nabla\cdot(\rho\mathbf{u}\otimes\mathbf{u}) + \nabla\cdot\tau \quad = \quad \mathbf{s}_u, \tag{30}$$

and the continuity equation

$$\frac{\partial}{\partial t}(c) + \nabla\cdot(\mathbf{u}c) \quad = \quad \mathbf{0}, \tag{31}$$

is introduced here to solve the volume fraction of the water c. Note here $\mathbf{u}$ is a 3-dimensional vector, so the gradient operator $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})^T$.

The mesh has $n = 1342756$ Nodes, with 3 velocity components in x ,y, z axis and an additional scalar component **Volume Fraction**, e.g.$C_p = 4$. The whole simulation $X \in \mathbb{R}^{1706\times1342756\times4}$ was split to Train set: Valid set: Test set = 1506: 100: 100 for training.

We created 3 Space-filling curves on this mesh as showed in 9, and trained the data with a 3-SFC-CAE-NN and with latent variable of $n_\xi = 256, 128$ and 64 with 1500 epoches. As the dataset is quite big, the model was trained with 4 RTX6000 GPUs on the HPC using DataParallel technique implemented in $\mathrm{Pytorch}$. The Final Losses for the 3D cases are showed in the table 3, and the reconstruction snapshots are in Figure 6.

From the results, we observed that the autoenoders generally had a great performance on the 3D problems, and especially for the CO_2 dataset, the MSE loss is quite low for even 4 latent variables. The MSE loss for the slugflow data is not that good comparing to other datasets.Within the limited time given in this project, tuning hyper-parameters for this large size of the dataset becomes a difficulty, as a single run of 100 epoches takes over 12 hours. However, some possible approaches are offered to find a preciser model on this dataset in the future. One is to change the type of optimizer, we have found that the AdaMax optimizer (Kingma & Ba 2017) performs better for training the CO_2 data than Adam optimizer (Kingma & Ba 2017). Perheps other types of algorithm, e.g. Nadam (Zhang et al. 2015) could handle the slugflow data better. Other strategies can be replacing the activation function, doing more data augmentations before training etc.

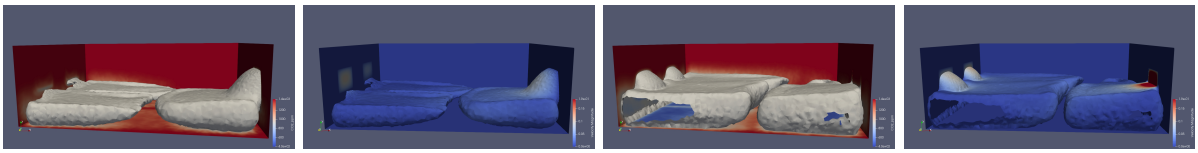
(a) CO_2

	3-SFC-CAE-NN	3-SFC-VCAE-NN		Latent
		MSE	KL	
Train set	1.30×10^{-4}	1.48×10^{-4}	7.99×10^{-5}	
Valid set	1.53×10^{-4}	1.98×10^{-4}	7.95×10^{-5}	16
Test set	2.09×10^{-4}	2.51×10^{-4}	7.76×10^{-5}	
Train set	1.34×10^{-4}	1.42×10^{-4}	6.18×10^{-5}	
Valid set	1.93×10^{-4}	5.49×10^{-4}	6.12×10^{-5}	8
Test set	2.31×10^{-4}	5.17×10^{-4}	6.07×10^{-5}	
Train set	1.18×10^{-4}	1.24×10^{-4}	3.65×10^{-5}	
Valid set	2.17×10^{-4}	4.75×10^{-4}	3.59×10^{-5}	4
Test set	2.25×10^{-4}	4.79×10^{-4}	3.62×10^{-5}	

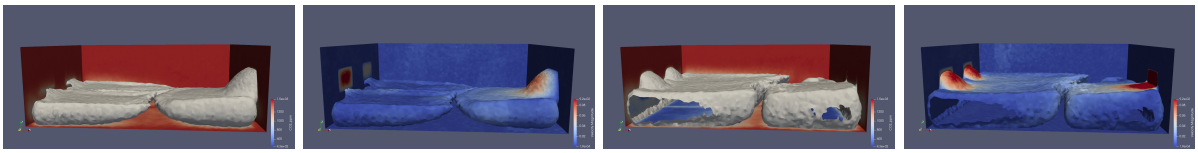
(b) Slugflow

	3-SFC-CAE-NN	Latent
Train set	1.32×10^{-3}	
Valid set	1.38×10^{-3}	256
Test set	1.42×10^{-3}	
Train set	1.34×10^{-3}	
Valid set	1.32×10^{-3}	128
Test set	1.36×10^{-3}	
Train set	1.31×10^{-3}	
Valid set	1.34×10^{-3}	64
Test set	1.34×10^{-3}	

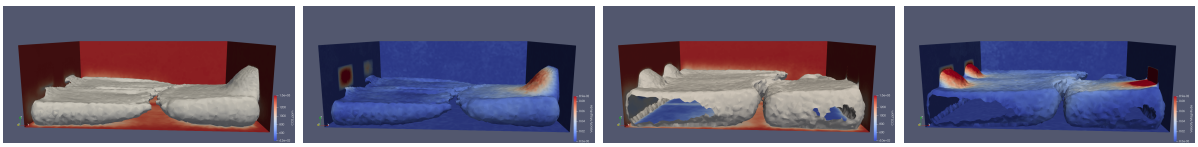
Table 3: The MSE/KL for 3-SFC-(V)CAE-NN on the 3D datasets



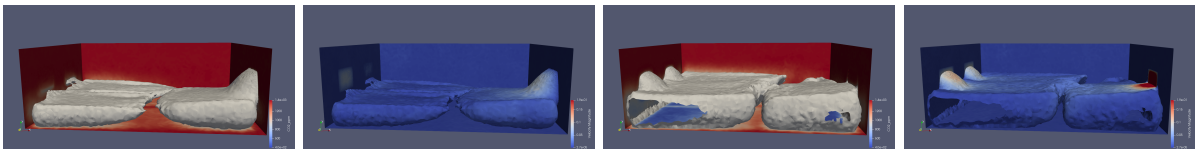
(a) original simulation



(b) Reconstructed by 16 Latent Variables

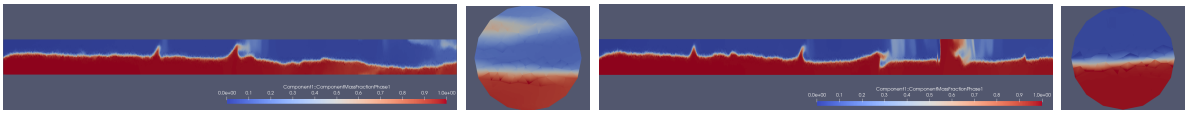


(c) Reconstructed by 8 Latent Variables

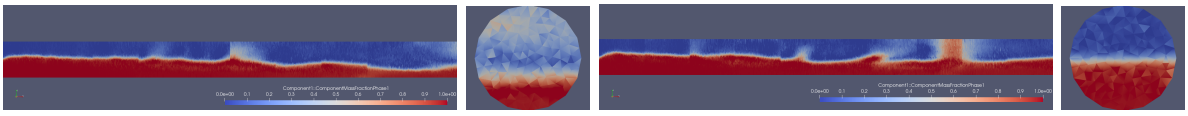


(d) Reconstructed by 4 Latent Variables

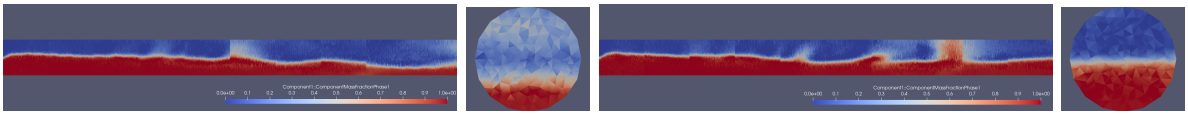
(A) Results for 3-SFC-CAE-NN on 3D CG mesh for the CO_2 data. From left to right, the first and second column represents CO_2 ppm and Velocity Magnitude at time level $t = 607$ second, and the third and fourth column represents CO_2 ppm and Velocity Magnitude at time level $t = 1181$ second



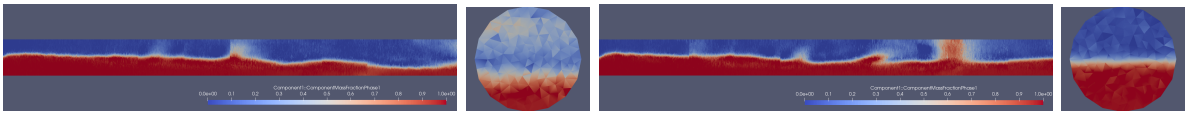
(e) original Volume Fraction



(f) Reconstructed Volume Fraction by 256 Latent Variables



(g) Reconstructed Volume Fraction by 128 Latent Variables



(h) Reconstructed Volume Fraction by 64 Latent Variables

(B) Results on 3D slugflow datasets, with a 3-SFC-CAE-NN compressed to 256, 128 and 64 Latent Variables, the left two columns front face/ right size face projections of the Volume Fraction inside the pipe at t = 492 second, the right two columns are the same features at t = 780 second.

Figure 6: Original/Reconstruction snapshots on 3D unstructured meshes

5.7 Comparing the latent space generated by CAE and VCAE

The advantage of a Variational Autoencoder over the normal one is that it has the ability to generate a well-behaved, mathematically completed latent space. If we generate some virtual latent variable in the latent space which is closed to the real latent variables measured by a metric, we will decode something which is very realistic to the real physics. To better illustrate this, the following two experiments are made:

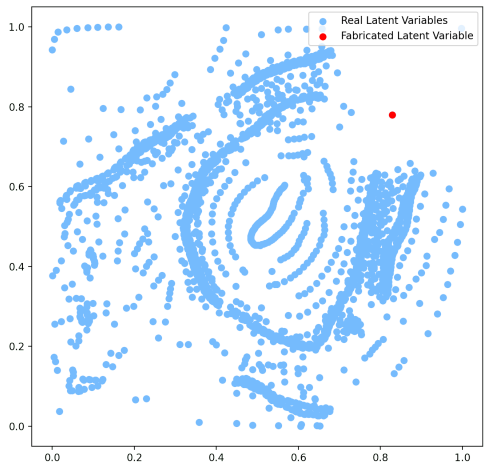
(A) We compressed the whole simulation of the DG-FPC $X \in \mathbb{R}^{2000\times 20550\times 2}$ to $L_{\mathrm{AE}} \in \mathbb{R}^{2000\times 2}$ using our trained 2-SFC-CAE-NN and 2-SFC-VCAE-NN model from 5.5. We then picked two points $[0.15, 0.67]$ and $[0.85, 0.75]$ directly in the normalized latent space (as the red points marked in Figure 7 (a), then we decoded them with the trained decoder of 2-SFC-CAE-NN and 2-SFC-VCAE-NN seprately, the reconstructed snapshots are showed in (c) and (d) of Figure 7.

(B) Another experiment is made on the CG mesh FPC problem. After we compressed the whole simulation $X \in \mathbb{R}^{2000\times 3571\times 2}$ to $L_{\mathrm{AE}} \in \mathbb{R}^{2000\times 16}$ and $L_{\mathrm{VAE}} \in \mathbb{R}^{2000\times 16}$ using the trained 2-SFC-CAE-NN and 2-SFC-VCAE-NN model from 5.5 separately, by picking a certain time level $t = 217$, we have the latent variables $L_{\mathrm{VAE}}(217)$ and $L_{\mathrm{AE}}(217)$, both of length 16. Then we create an artificial Gaussian Noise $\boldsymbol{\lambda} \sim N(0, 1)$ of length 16, by adding this noise to the above two latent spaces we will get the fabricated latent variables:

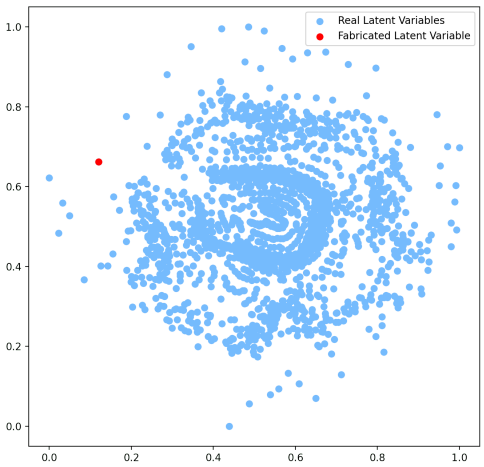
$$\begin{aligned} \tilde{L}_{\mathrm{AE}}(217) &= L_{\mathrm{AE}}(217) + \boldsymbol{\lambda} \\ \tilde{L}_{\mathrm{VAE}}(217) &= L_{\mathrm{VAE}}(217) + \boldsymbol{\lambda} \end{aligned} \tag{32}$$

by decoding from those two 'fake' latent variables with the trained decoder, we obtain the reconstruction of the simulation $\tilde{X}_{\mathrm{AE}}(217)$, $\tilde{X}_{\mathrm{VAE}}(217)$ (showed in (g) and (h) in Figure 7) We could further reduce the dimensionality of the latent space from 16 to 2 using t-Distributed Stochastic Neighbor Embedding (t-SNE) (Kadeethum et al. 2021), then we can visulize the position of those fake latents in 2D.

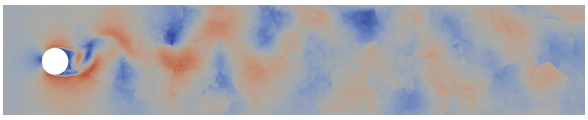
By observing the reconstructed snapshots in Figure 7, we could see that if fabricate some latent variables which are closed to the real latent variables of the simulation in the latent space, the decoded snapshot of a SFC-VCAE is more realistic than the decoded snapshot of a SFC-CAE.



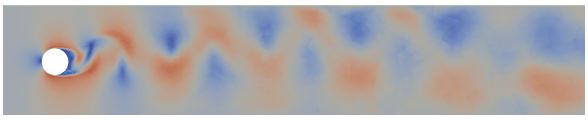
(a) plot for AE latent space plus the fake latent



(b) plot for VAE latent space plus the fake latent

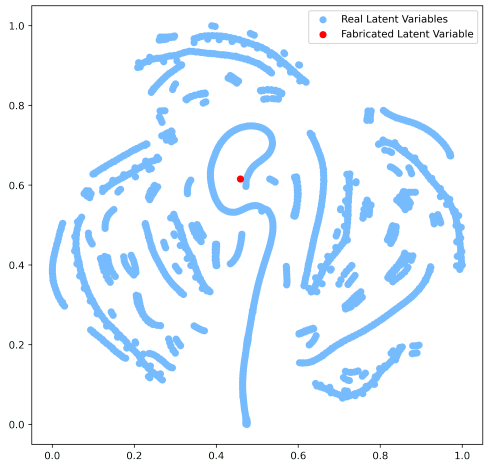


(c) Decoded snapshot from the fake latent variable by AE

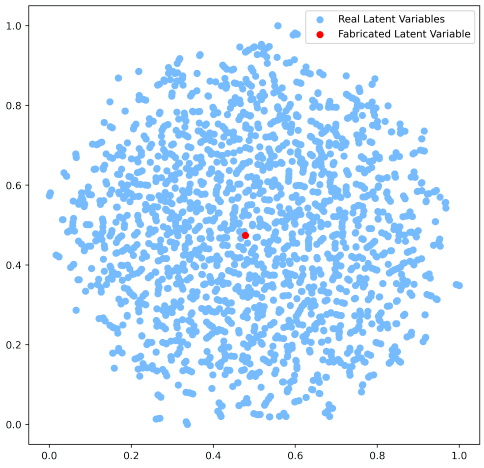


(d) Decoded snapshot from the fake latent variable by VAE

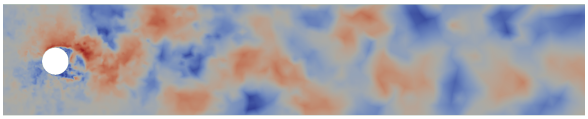
(A) 2-SFC-(V)CAE-NN compressing DG mesh FPC to 2 latent variables



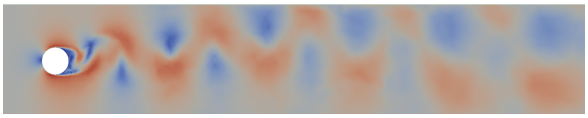
(e) t-SNE plot for AE latent space plus the fake latent



(f) t-SNE plot for VAE latent space plus the fake latent



(g) Decoded snapshot from the fake latent by AE



(h) Decoded snapshot from the fake latent by VAE

(B) 2-SFC-(V)CAE-NN compressing CG mesh FPC to 16 latent variables
(visualizing latent space in 2-dimenisonal use t-SNE.)

Figure 7: The above two figures are the reconstructed FLow Past Cylinder snapshots by VAE/AE from artificial latent variables, as well as their location in the latent space. The left column are for SFC-CAE, the right column are for SFC-VCAE. t-SNE are used to visulize high dimensional latent space.

6 Conclusions and Future Work

With the implementation of automatic generation of SFC-CAEs, users and researchers could directly apply this type of neural network to unseen unstructured datasets directly. A high efficiency of compressing could be achieved by its ability to compressing multiple scalar fields at a same time. Furthermore, by altering parameters of the networks, we extend the application of this type of autoencoder to compressing 3D unstructured datasets. As experiments showed in this paper, the newly developed autoencoder could compress a complex 3D dataset (CO_2 in the room 5.6.1) of over $670,000$ data into 4 latent variables with a low reconstruction error.

As the SFC-CAE has already showed great performance on unstructured meshes, other SFC-based models could be explored, for example, add an adversarial part (SFC-CAAE) to increase the sharpness of the output. Another big challenge is to make the SFC-based models work on an adaptive mesh (Kampitsis et al. 2020), a possible approach would be inputting coordinate values from meshes during training. Moreover, the current model is neither temporally-aware or physically-informed, more advanced methodology could be applied to make it could predict in time.

References

Anjum, M. M., Tahmid, I. A. & Rahman, M. S. (2019), 'Cnn model with hilbert curve representation of dna sequence for enhancer prediction', *bioRxiv* .
URL: *https://www.biorxiv.org/content/early/2019/02/27/552141*

Bader, M. (2013), *Locality Properties of Space-Filling Curves*, Springer Berlin Heidelberg, Berlin, Heidelberg, p. 179.
URL: *https://doi.org/10.1007/978-3-642-31046-1*$_1$1

Buluç, A., Meyerhenke, H., Safro, I., Sanders, P. & Schulz, C. (2013), 'Recent advances in graph partitioning', *CoRR* **abs/1311.3144**.
URL: *http://arxiv.org/abs/1311.3144*

Heaney, C. E., Li, Y., Matar, O. K. & Pain, C. C. (2021), 'Applying Convolutional Neural Networks to Data on Unstructured Meshes with Space-Filling Curves'.

Hilbert, D. (1891), 'Über die stetige abbildung einer linie auf ein flächenstück', *Mathematische Annalen* **38**, 459–460.
URL: *http://www.digizeitschriften.de/dms/img/?PID=GDZPPN002253135*

Kadeethum, T., Ballarin, F., Choi, Y., O'Malley, D., Yoon, H. & Bouklas, N. (2021), 'Non-intrusive reduced order modeling of natural convection in porous media using convolutional autoencoders: comparison with linear subspace techniques'.

Kampitsis, A., Adam, A., Salinas, P., Pain, C. C., Muggeridge, A. H. & Jackson, M. D. (2020), 'Dynamic adaptive mesh optimisation for immiscible viscous fingering', *Computational Geosciences* **24**, 1221–1237.

Kingma, D. P. & Ba, J. (2017), 'Adam: A method for stochastic optimization'.

Köksoy, O. (2006), 'Multiresponse robust design: Mean square error (MSE) criterion', *Applied Mathematics and Computation* **175**(2), 1716–1729.
URL: *https://www.sciencedirect.com/science/article/pii/S0096300305007332*

Lee, K. & Carlberg, K. T. (2019), 'Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders', *Journal of Computational Physics* **404**, 108973.

Lindenmayer, A. (1968), 'Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs', *Journal of Theoretical Biology* **18**(3), 300–315.
URL: *https://www.sciencedirect.com/science/article/pii/0022519368900805*

Lucia, D. J., Beran, P. S. & Silva, W. A. (2004), 'Reduced-order modeling: new approaches for computational physics', *Progress in Aerospace Sciences* **40**(1), 51–117.
URL: *https://www.sciencedirect.com/science/article/pii/S0376042103001131*

Mack, J., Arcucci, R., Molina-Solana, M. & Guo, Y.-K. (2020), 'Attention-based Convolutional Autoencoders for 3D-Variational Data Assimilation', *Computer Methods in Applied Mechanics and Engineering* **372**, 113291.
URL: *https://www.sciencedirect.com/science/article/pii/S004578252030476X*

Maulik, R., Lusch, B. & Balaprakash, P. (2021), 'Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders', *Physics of Fluids* **33**(3), 037106.
URL: *https://doi.org/10.1063/5.0039986*

Moore, E. H. (1900), 'On certain crinkly curves', *Transactions of the American Mathematical Society* **1**(1), 72–90.
URL: *http://www.jstor.org/stable/1986405*

Pain, C. C., Oliveira, C. R. E. D. & Goddard, A. J. H. (1999), 'A neural network graph partitioning procedure for grid-based domain decomposition', *International Journal for Numerical Methods in Engineering* **44**.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), Pytorch: An imperative style, high-performance deep learning library, *in* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox & R. Garnett, eds, 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.
URL: *http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf*

Peano, G. (1890), 'Sur une courbe, qui remplit toute une aire plane', *Mathematische Annalen* **36**(1), 157–160.

Platzman, L. K. & Bartholdi, J. J. (1989), 'Spacefilling curves and the planar travelling salesman problem', *J. ACM* **36**(4), 719–737.
URL: *https://doi.org/10.1145/76359.76361*

Schlömer, N. (2021), 'meshio'. If you use this software, please cite it as below.
URL: *https://doi.org/10.5281/zenodo.5167106*

Tu, J., Yeoh, G.-H. & Liu, C. (2018), Chapter e1 - CFD Case Studies, *in* J. Tu, G.-H. Yeoh & C. Liu, eds, 'Computational Fluid Dynamics (Third Edition)', third edition edn, Butterworth-Heinemann, pp. e1–e122.
URL: *https://www.sciencedirect.com/science/article/pii/B9780081011270000209*

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P. & SciPy 1.0 Contributors (2020), 'SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python', *Nature Methods* **17**, 261–272.

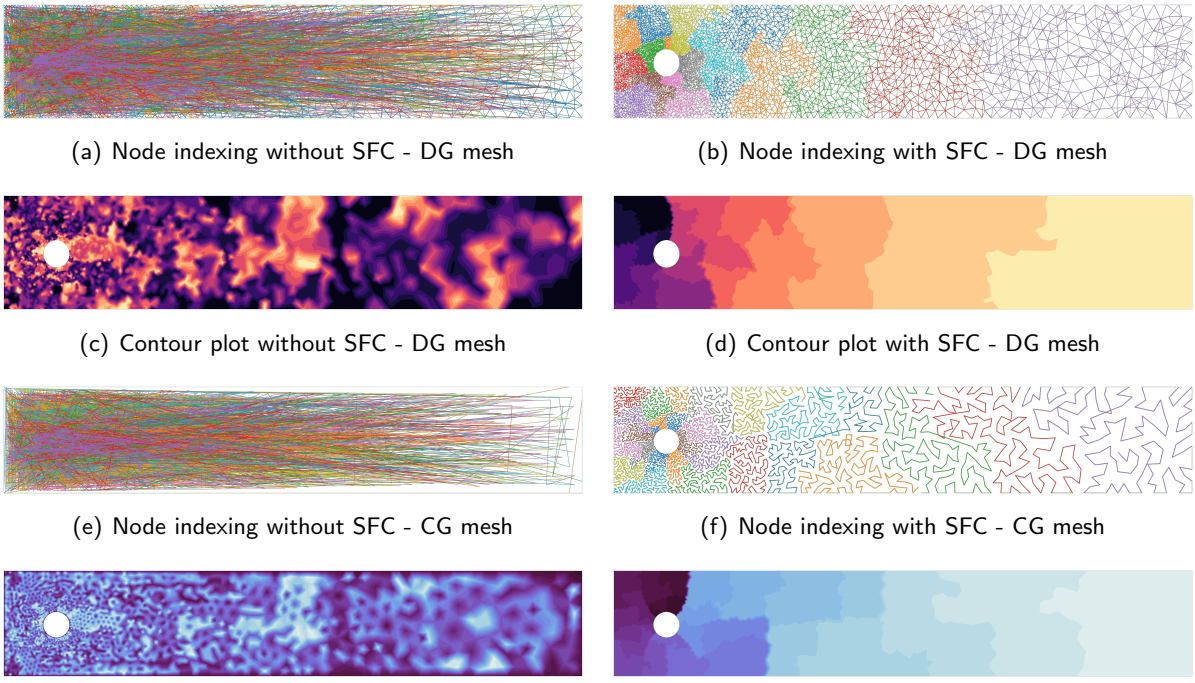
Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. & Yu, P. S. (2021), 'A Comprehensive Survey on Graph Neural Networks', *IEEE Transactions on Neural Networks and Learning Systems* **32**(1), 4–24.

Zhang, S., Choromanska, A. & LeCun, Y. (2015), 'Deep learning with elastic averaging sgd'.

Zhou, L., Johnson, C. R. & Weiskopf, D. (2020), 'Data-Driven Space-Filling Curves'.

Appendices

Figure 8 shows the original index ordering / contour plot for the FPC-DG and FPC-CG mesh, placed here as comparison to 2.



(a) Node indexing without SFC - DG mesh

(b) Node indexing with SFC - DG mesh

(c) Contour plot without SFC - DG mesh

(d) Contour plot with SFC - DG mesh

(e) Node indexing without SFC - CG mesh

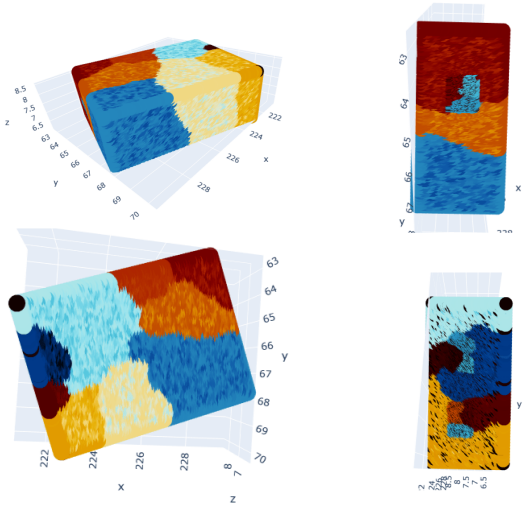
(f) Node indexing with SFC - CG mesh

(g) Contour plot without SFC - CG mesh

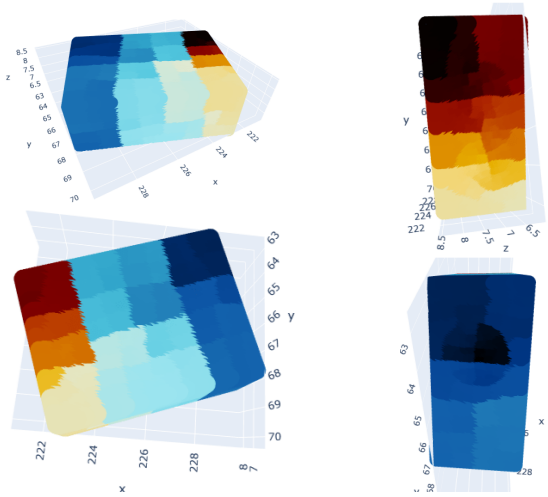
(h) Contour plot with SFC - CG mesh

Figure 8: SFC plots for 2D structured/unstructured meshes

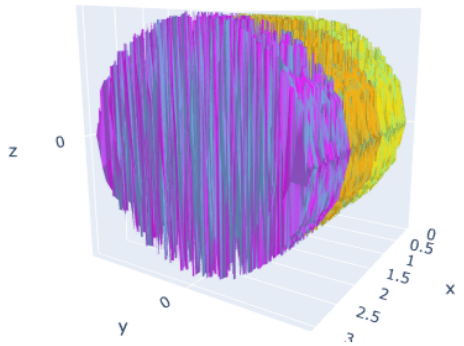
Figure 9 shows the contour plots of SFC orderings on a 3D unstructured meshes. We could see after applying SFC orderings, the whole meshes were divided into nearly perfect partitions with respect to index ordeings.



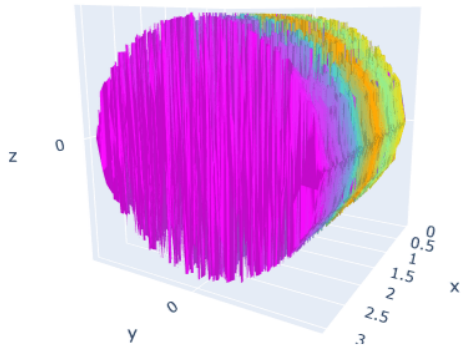
(a) Contour plot for original ordering for mesh of CO2 data



(b) Contour plot for SFC ordering for mesh of CO2 data



(c) Contour plot for original ordering for mesh of slugflow data ($\frac{1}{4}$ partition)



(d) Contour plot for SFC ordering for mesh of slugflow data ($\frac{1}{4}$ partition)

Figure 9: Contour Plots of original/space-filling orderings on 3D unstructured meshes, plots supported by Ploty.

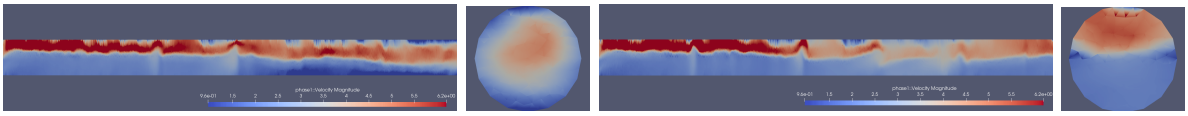
(a) The MSE for autoencoders compressing square wave to 16 latent variables

	Classical 2D CAE	Hilbert-SFC-CAE	Hilbert-SFC-CAE-NN	MFT-RNN-SFC-CAE	MFT-RNN-SFC-CAE-NN	MFT-RNN-2-SFC-CAE	MFT-RNN-2-SFC-CAE-NN	Initial Condition
Train set	2.56×10^{-4}	5.51×10^{-4}	4.56×10^{-4}	1.97×10^{-3}	1.17×10^{-3}	5.45×10^{-4}	1.06×10^{-4}	
Valid set	2.47×10^{-4}	6.12×10^{-4}	5.41×10^{-4}	2.09×10^{-3}	1.27×10^{-3}	6.22×10^{-4}	2.07×10^{-4}	Block
Test set	2.00×10^{-4}	5.84×10^{-4}	4.95×10^{-4}	1.99×10^{-3}	1.15×10^{-3}	5.47×10^{-4}	1.26×10^{-4}	
Train set	2.02×10^{-6}	8.45×10^{-6}	2.82×10^{-6}	1.64×10^{-4}	1.17×10^{-5}	2.72×10^{-6}	2.13×10^{-6}	
Valid set	3.20×10^{-6}	8.21×10^{-6}	2.98×10^{-6}	1.57×10^{-4}	1.10×10^{-5}	5.59×10^{-6}	1.18×10^{-6}	Gaussian
Test set	3.04×10^{-6}	8.18×10^{-6}	2.87×10^{-6}	1.56×10^{-4}	1.05×10^{-5}	5.29×10^{-6}	1.16×10^{-6}	

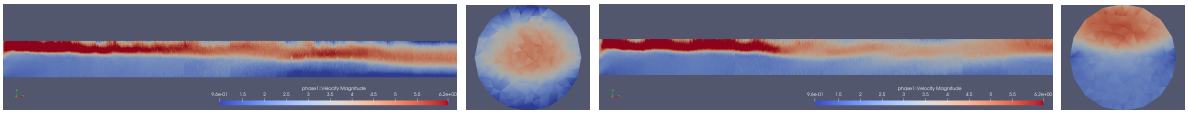
(b) The MSE for a 2-SFC-CAE-NN compressing gaussian wave to 16, 8, 4, 2, 1 latent variables

	Number of latent variables				
	16	8	4	2	1
Train set	2.13×10^{-6}	4.60×10^{-6}	2.87×10^{-6}	2.46×10^{-6}	2.64×10^{-4}
Valid set	1.18×10^{-6}	3.85×10^{-6}	3.51×10^{-6}	2.31×10^{-6}	2.84×10^{-4}
Test set	1.16×10^{-6}	3.71×10^{-6}	3.67×10^{-6}	2.33×10^{-6}	2.64×10^{-4}

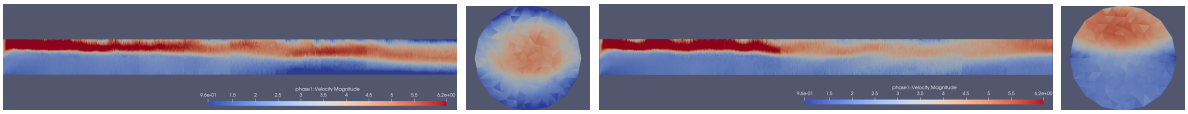
Table 4: The final MSE loss for different types of AE, all AEs are trained with 2000 epoches.



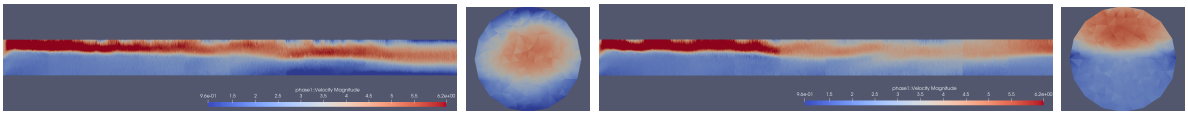
(a) Original Velocity Magnitude



(b) Reconstructed Velocity Magnitude by 256 Latent Variables



(c) Reconstructed Velocity Magnitude by 128 Latent Variables



(d) Reconstructed Velocity Magnitude by 64 Latent Variables

Figure 10: Results on 3D slugflow datasets, with a 3-SFC-CAE-NN compressed to 256, 128 and 64 Latent Variables, the left column are Velocity Magnitude at t = 492 second, the right column are Velocity Magnitude at t = 780 second.

The following tables are template SFC-(V)CAE network structures for the 5 datasets, generated by the Automatic Table generator introduced in 4.3:

layers	input size & ordering	kernel size	channels	stride	padding	output padding	output size & ordering	activation
1-Grid	(16384, 1, Grid)	1 Identity	1	1	0	0	(16384, 1, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2\}$	Identity
Encoder								
Copying channels and concatenate at the 2nd dimension to form (16384, 2, SFC$\mathcal{C}$), flatten it to 1D.								
2-ExpandNN-SFC$\mathcal{C}$	(32768, 1, SFC$\mathcal{C}$)	3 Variable (3 × 32768)	1	1	0	0	(32768, 1, SFC$\mathcal{C}$)	ReLU
Reshape (32768, 1, SFC$\mathcal{C}$) to form (16384, 2, SFC$\mathcal{C}$)								
3-Conv1d-SFC$\mathcal{C}$	(16384, 2, SFC$\mathcal{C}$)	32	4	4	16	0	(4097, 4, SFC$\mathcal{C}$)	ReLU
4-Conv1d-SFC$\mathcal{C}$	(4097, 4, SFC$\mathcal{C}$)	32	8	4	16	0	(1025, 8, SFC$\mathcal{C}$)	ReLU
5-Conv1d-SFC$\mathcal{C}$	(1025, 8, SFC$\mathcal{C}$)	32	16	4	16	0	(257, 16, SFC$\mathcal{C}$)	ReLU
6-Conv1d-SFC$\mathcal{C}$	(257, 16, SFC$\mathcal{C}$)	32	16	4	16	0	(65, 16, SFC$\mathcal{C}$)	ReLU
7-FC	2080 ($= 65 \times 16 \times 2$)						520	ReLU
8-FC	520						130	ReLU
8-FC	130						16	ReLU
Decoder								
10-FC	16						130	ReLU
11-FC	130						520	ReLU
12-FC	520						2080	ReLU
Split the data into 2 sequences as the input of layer 13-TransConv1d-SFC$\mathcal{C}$ $\forall\, \mathcal{C} \in \{1,2\}$, convert from 1040 to (65, 16)								
13-TransConv1d-SFC$\mathcal{C}$	(65, 16, SFC$\mathcal{C}$)	32	16	4	16	1	(257, 16, SFC$\mathcal{C}$)	ReLU
14-TransConv1d-SFC$\mathcal{C}$	(257, 16, SFC$\mathcal{C}$)	32	16	4	16	1	(1025, 8, SFC$\mathcal{C}$)	ReLU
15-TransConv1d-SFC$\mathcal{C}$	(1025, 8, SFC$\mathcal{C}$)	32	8	4	16	1	(4097, 4, SFC$\mathcal{C}$)	ReLU
16-TransConv1d-SFC$\mathcal{C}$	(4097, 4, SFC$\mathcal{C}$)	32	4	4	16	0	(16384, 2, SFC$\mathcal{C}$)	ReLU
Apply inverse SFC orderings to (16384, 1, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2\}$, flatten into (32768, 1, Grid$\mathcal{C}$)								
17-ExpandNN-Grid$\mathcal{C}$	(32768, 1, Grid$\mathcal{C}$)	3 Variable (3 × 32768)	1	1	0	0	(32768, 1, Grid$\mathcal{C}$)	ReLU
Reshape (32768, 1, Grid$\mathcal{C}$), and separate self-concat channels, to form (16384, 2, Grid$\mathcal{C}$) × 2								
18-Grid	(16384, 1, Grid$\mathcal{C}$) × 4	$\sum_{i=1}^{4}$ Grid$i$	1	1	0	0	(16384, 1, Grid$\mathcal{C}$)	ReLU

Table 5: The neural network of a 2-SFC-CAE-NN compressing the advection wave (20200 snapshots, 16384 Nodes, 1 component) to 16 latent variables.

layers	input size & ordering	kernel size	channels	stride	padding	output padding	output size & ordering	activation
1-FEM (ANN input)	(20550, 2, FEM)	1 Identity	2	1	0	0	(20550, 2, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2\}$	Identity
Encoder								
Copying channels and concatenate at the 2nd dimension to form (20550, 4, SFC$\mathcal{C}$), flatten it to 1D.								
2-ExpandNN-SFC$\mathcal{C}$	(82200, 1, SFC$\mathcal{C}$)	3 Variable (3 × 82200)	1	1	0	0	(82200, 1, SFC$\mathcal{C}$)	Tanh
Reshape (82200, 1, SFC$\mathcal{C}$) to form (20550, 4, SFC$\mathcal{C}$)								
3-Conv1d-SFC$\mathcal{C}$	(20550, 4, SFC$\mathcal{C}$)	32	8	4	16	0	(5138, 8, SFC$\mathcal{C}$)	Tanh
4-Conv1d-SFC$\mathcal{C}$	(5138, 8, SFC$\mathcal{C}$)	32	16	4	16	0	(1285, 16, SFC$\mathcal{C}$)	Tanh
5-Conv1d-SFC$\mathcal{C}$	(1285, 16, SFC$\mathcal{C}$)	32	16	4	16	0	(322, 16, SFC$\mathcal{C}$)	Tanh
6-Conv1d-SFC$\mathcal{C}$	(322, 16, SFC$\mathcal{C}$)	32	16	4	16	0	(81, 16, SFC$\mathcal{C}$)	Tanh
7-FC	2592 ($= 81 \times 16 \times 2$)						648	Tanh
8-FC	648						162	Tanh
9-FC	162						40	Tanh
9-FC	40						2	Tanh
Decoder								
11-FC	2						40	Tanh
12-FC	40						162	Tanh
13-FC	162						648	Tanh
14-FC	648						2592	Tanh
Split the data into 2 sequences as the input of layer 15-TransConv1d-SFC$\mathcal{C}$ $\forall\, \mathcal{C} \in \{1,2\}$, convert from 1296 to (81, 16)								
15-TransConv1d-SFC$\mathcal{C}$	(81, 16, SFC$\mathcal{C}$)	32	16	4	16	2	(322, 16, SFC$\mathcal{C}$)	Tanh
16-TransConv1d-SFC$\mathcal{C}$	(322, 16, SFC$\mathcal{C}$)	32	16	4	16	1	(1285, 16, SFC$\mathcal{C}$)	Tanh
17-TransConv1d-SFC$\mathcal{C}$	(1285, 16, SFC$\mathcal{C}$)	32	16	4	16	2	(5138, 8, SFC$\mathcal{C}$)	Tanh
18-TransConv1d-SFC$\mathcal{C}$	(5138, 8, SFC$\mathcal{C}$)	32	8	4	16	2	(20550, 4, SFC$\mathcal{C}$)	Tanh
Apply inverse SFC orderings to (20550, 2, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2\}$, flatten into (82200, 1, FEM$\mathcal{C}$)								
19-ExpandNN-FEM$\mathcal{C}$	(82200, 1, FEM$\mathcal{C}$)	3 Variable (3 × 82200)	1	1	0	0	(82200, 1, FEM$\mathcal{C}$)	Tanh
Reshape (82200, 1, FEM$\mathcal{C}$), and separate self-concat channels, to form (20550, 4, FEM$\mathcal{C}$) × 2								
20-FEM (ANN output)	(20550, 2, FEM$\mathcal{C}$) × 4	$\sum_{i=1}^{4}$ FEMi	2	1	0	0	(20550, 2, FEM$\mathcal{C}$)	Tanh

Table 6: The neural network of a 2-SFC-CAE-NN compressing the FPC-DG Data (2000 snapshots, 20550 Nodes, 2 components) to 2 latent variables.

layers	input size & ordering	kernel size	channels	stride	padding	output padding	output size & ordering	activation
1-FEM (ANN input)	(3571, 2, FEM)	1 Identity	2	1	0	0	(3571, 2, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2\}$	Identity
Encoder								
Copying channels and concatenate at the 2nd dimension to form (3571, 4, SFC$\mathcal{C}$)								
2-Conv1d-SFC$\mathcal{C}$	(3571, 4, SFC$\mathcal{C}$)	32	8	4	16	0	(893, 8, SFC$\mathcal{C}$)	Tanh
3-Conv1d-SFC$\mathcal{C}$	(893, 8, SFC$\mathcal{C}$)	32	16	4	16	0	(224, 16, SFC$\mathcal{C}$)	Tanh
4-Conv1d-SFC$\mathcal{C}$	(224, 16, SFC$\mathcal{C}$)	32	16	4	16	0	(57, 16, SFC$\mathcal{C}$)	Tanh
5-FC	1824 ($= 57 \times 16 \times 2$)						456	Tanh
6-FC	456						114	Tanh
7-FC	114						28	Tanh
7-FC	28						4	Tanh
Decoder								
9-FC	4						28	Tanh
10-FC	28						114	Tanh
11-FC	114						456	Tanh
12-FC	456						1824	Tanh
Split the data into 2 sequences as the input of layer 13-TransConv1d-SFC$\mathcal{C}$ $\forall\, \mathcal{C} \in \{1,2\}$, convert from 912 to (57, 16)								
13-TransConv1d-SFC$\mathcal{C}$	(57, 16, SFC$\mathcal{C}$)	32	16	4	16	0	(224, 16, SFC$\mathcal{C}$)	Tanh
14-TransConv1d-SFC$\mathcal{C}$	(224, 16, SFC$\mathcal{C}$)	32	16	4	16	1	(893, 8, SFC$\mathcal{C}$)	Tanh
15-TransConv1d-SFC$\mathcal{C}$	(893, 8, SFC$\mathcal{C}$)	32	8	4	16	3	(3571, 4, SFC$\mathcal{C}$)	Tanh
Apply inverse SFC orderings to (3571, 4, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2\}$, and separate self-concat channels, to form (3571, 2, FEM$\mathcal{C}$) $\times$ 2								
16-FEM (ANN output)	(3571, 2, FEM$\mathcal{C}$) $\times$ 4	$\sum_{i=1}^{4}$ FEMi	2	1	0	0	(3571, 2, FEM$\mathcal{C}$)	Tanh

Table 7: The neural network of a 2-SFC-CAE compressing the FPC-CG Data (2000 snapshots, 3571 Nodes, 2 components) to 4 latent variables.

layers	input size & ordering	kernel size	channels	stride	padding	output padding	output size & ordering	activation
1-FEM (ANN input)	(148906, 4, FEM)	1 Identity	4	1	0	0	(148906, 4, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2,3\}$	Identity
Encoder								
Copying channels and concatenate at the 2nd dimension to form (148906, 8, SFC$\mathcal{C}$), flatten it to 1D.								
2-ExpandNN-SFC$\mathcal{C}$	(1191248, 1, SFC$\mathcal{C}$)	3 Variable (3 $\times$ 1191248)	1	1	0	0	(1191248, 1, SFC$\mathcal{C}$)	Tanh
Reshape (1191248, 1, SFC$\mathcal{C}$) to form (148906, 8, SFC$\mathcal{C}$)								
3-Conv1d-SFC$\mathcal{C}$	(148906, 8, SFC$\mathcal{C}$)	176	16	8	88	0	(18614, 16, SFC$\mathcal{C}$)	Tanh
4-Conv1d-SFC$\mathcal{C}$	(18614, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(2327, 16, SFC$\mathcal{C}$)	Tanh
5-Conv1d-SFC$\mathcal{C}$	(2327, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(291, 16, SFC$\mathcal{C}$)	Tanh
6-Conv1d-SFC$\mathcal{C}$	(291, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(37, 16, SFC$\mathcal{C}$)	Tanh
7-FC	1776 ($= 37 \times 16 \times 3$)						222	Tanh
Variational Reparametrization								
7-FC-σ	222						16	Tanh
7-FC-μ	222						16	Tanh
8-Sampling	16	2 Variable (2 $\times$16)	1	1	0	0	16	Identity
Decoder								
10-FC	16						222	Tanh
11-FC	222						1776	Tanh
Split the data into 3 sequences as the input of layer 12-TransConv1d-SFC$\mathcal{C}$ $\forall\, \mathcal{C} \in \{1,2,3\}$, convert from 592 to (37, 16)								
12-TransConv1d-SFC$\mathcal{C}$	(37, 16, SFC$\mathcal{C}$)	176	16	8	88	3	(291, 16, SFC$\mathcal{C}$)	Tanh
13-TransConv1d-SFC$\mathcal{C}$	(291, 16, SFC$\mathcal{C}$)	176	16	8	88	7	(2327, 16, SFC$\mathcal{C}$)	Tanh
14-TransConv1d-SFC$\mathcal{C}$	(2327, 16, SFC$\mathcal{C}$)	176	16	8	88	6	(18614, 16, SFC$\mathcal{C}$)	Tanh
15-TransConv1d-SFC$\mathcal{C}$	(18614, 16, SFC$\mathcal{C}$)	176	16	8	88	2	(148906, 8, SFC$\mathcal{C}$)	Tanh
Apply inverse SFC orderings to (148906, 4, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1,2,3\}$, flatten into (1191248, 1, FEM$\mathcal{C}$)								
16-ExpandNN-FEM$\mathcal{C}$	(1191248, 1, FEM$\mathcal{C}$)	3 Variable (3 $\times$ 1191248)	1	1	0	0	(1191248, 1, FEM$\mathcal{C}$)	Tanh
Reshape (1191248, 1, FEM$\mathcal{C}$), and separate self-concat channels, to form (148906, 8, FEM$\mathcal{C}$) $\times$ 2								
17-FEM (ANN input)	(148906, 4, FEM$\mathcal{C}$) $\times$ 6	$\sum_{i=1}^{6}$ FEMi	4	1	0	0	(148906, 4, FEM$\mathcal{C}$)	Tanh

Table 8: The neural network of a 3-SFC-VCAE-NN compressing the CO2 Data (455 snapshots, 148906 Nodes, 4 components) to 4 latent variables.

layers	input size & ordering	kernel size	channels	stride	padding	output padding	output size & ordering	activation
1-FEM (ANN input)	(1342756, 4, FEM)	1 Identity	4	1	0	0	(1342756, 4, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1, 2, 3\}$	Identity
Encoder								
Copying channels and concatenate at the 2nd dimension to form (1342756, 8, SFC$\mathcal{C}$), flatten it to 1D.								
2-ExpandNN-SFC$\mathcal{C}$	(10742048, 1, SFC$\mathcal{C}$)	3 Variable (3 $\times$ 10742048)	1	1	0	0	(10742048, 1, SFC$\mathcal{C}$)	Tanh
Reshape (10742048, 1, SFC$\mathcal{C}$) to form (1342756, 8, SFC$\mathcal{C}$)								
3-Conv1d-SFC$\mathcal{C}$	(1342756, 8, SFC$\mathcal{C}$)	176	16	8	88	0	(167845, 16, SFC$\mathcal{C}$)	Tanh
4-Conv1d-SFC$\mathcal{C}$	(167845, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(20981, 16, SFC$\mathcal{C}$)	Tanh
5-Conv1d-SFC$\mathcal{C}$	(20981, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(2623, 16, SFC$\mathcal{C}$)	Tanh
6-Conv1d-SFC$\mathcal{C}$	(2623, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(328, 16, SFC$\mathcal{C}$)	Tanh
7-Conv1d-SFC$\mathcal{C}$	(328, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(42, 16, SFC$\mathcal{C}$)	Tanh
7-FC	2016						128	Tanh
Decoder								
9-FC	128						2016	Tanh
Split the data into 3 sequences as the input of layer 10-TransConv1d-SFC$\mathcal{C}$ $\forall\, \mathcal{C} \in \{1, 2, 3\}$, convert from 672 to (42, 16)								
10-TransConv1d-SFC$\mathcal{C}$	(42, 16, SFC$\mathcal{C}$)	176	16	8	88	0	(328, 16, SFC$\mathcal{C}$)	Tanh
11-TransConv1d-SFC$\mathcal{C}$	(328, 16, SFC$\mathcal{C}$)	176	16	8	88	7	(2623, 16, SFC$\mathcal{C}$)	Tanh
12-TransConv1d-SFC$\mathcal{C}$	(2623, 16, SFC$\mathcal{C}$)	176	16	8	88	5	(20981, 16, SFC$\mathcal{C}$)	Tanh
13-TransConv1d-SFC$\mathcal{C}$	(20981, 16, SFC$\mathcal{C}$)	176	16	8	88	5	(167845, 16, SFC$\mathcal{C}$)	Tanh
14-TransConv1d-SFC$\mathcal{C}$	(167845, 16, SFC$\mathcal{C}$)	176	16	8	88	4	(1342756, 8, SFC$\mathcal{C}$)	Tanh
Apply inverse SFC orderings to (1342756, 4, SFC$\mathcal{C}$) $\forall\, \mathcal{C} \in \{1, 2, 3\}$, flatten into (10742048, 1, FEM$\mathcal{C}$)								
15-ExpandNN-FEM$\mathcal{C}$	(10742048, 1, FEM$\mathcal{C}$)	3 Variable (3 $\times$ 10742048)	1	1	0	0	(10742048, 1, FEM$\mathcal{C}$)	Tanh
Reshape (10742048, 1, FEM$\mathcal{C}$), and separate self-concat channels, to form (1342756, 8, FEM$\mathcal{C}$) $\times$ 2								
16-FEM (ANN output)	(1342756, 4, FEM$\mathcal{C}$) $\times$ 6	$\sum_{i=1}^{6}$ FEMi	4	1	0	0	(1342756, 4, FEM$\mathcal{C}$)	Tanh

Table 9: The neural network of a 3-SFC-CAE-NN compressing the slugflow Data (1706 snapshots, 1342756 Nodes, 4 components) to 128 latent variables.