Imperial College London

Department of Earth Science and Engineering

MSc in Applied Computational Science and Engineering

Independent Research Project
Project Plan

# Dimensionality reduction with Graph Networks and Space-Filling Curve Autoencoders for fluid flow problems

by

Andrea Pozzetti

ap2920@imperial.ac.uk
GitHub login: acse-ap2920

Supervisors:

Dr. Claire Heaney

Prof. Christopher Pain
Morgan Kerhouant

September 2021

## Acknowledgements

I would like to thank my supervisors Dr. Claire Heaney and Prof. Christopher Pain for their support, availability, kindness and most of all patience. I want to thank Yu Jin for developing the excellent SFC-CAE library and all of my colleagues in the autoencoders group for their ideas and help. Thank you to Morgan Kerhouant for helping me with the college HPC. Thank you to my roomate Jacopo Iollo for letting me use his Colab account in parallel with mine.

Lastly I would like to express my gratitude to Dr Gareth Roberts, for his kindness.

GitHub repository: https://github.com/acse-jy220/SFC-CAE-Ready-to-use/tree/adaptive-mesh-final

## Abstract

Convolutional Neural Networks are preferred over traditional Proper Orthogonal Decomposition for Reduced Order Modelling approaches for several fluid flow problems in computational physics [1].

They have been proven to be better than traditional approaches like Proper Orthogonal Decomposition at capturing non-linear activation functions and compressing advection-dominated flows [21], but it is hard to apply them to unstructured grids, which are however used widely in Computational Fluid Dynamics.

Two different approaches for using convolutional neural networks on unstructured meshes (namely Graph Convolutional Networks and space filling curves) will be explored in this project.

In particular a tool will built from the MeshCNN library for the compression GCAE. Convolutional operators are hard to define for pooling and unpooling operations [46].

Later the application of Space Filling Curve Convolutional Autoencoders (SFC-CAE) to simulations done on Adaptive Meshes will be explored. Since the approach uses 1D convolutional filters, it is hoped that it would present itself quite well to adaptivity.

# Contents

# 1 Introduction and Objectives

Reduced order modelling is a technique used to simplify or reduce complex models of physical systems and capture their features into a lower-dimensional space. It is widely used within Computational Physics to reduce the computational complexity of simulations [7].

Autoencoders are increasingly being used in Reduced-Order Modelling (ROM) as a tool for dimensionality reduction [22] [2].

Traditionally dimensionality reduction has been done through Linear dimension reduction methods such as Proper Orthogonal Decomposition (POD), which has worked well for many applications [5], however Autoencoders have been shown to be better than POD at capturing complex features of flows more efficiently, especially for advection-dominated flows [19][1]. Autoencoders can be thought of as a nonlinear generalisation of POD [4], and they enable the use of a smaller reduced space for most problems.

2D Convolutional Autoencoders in particular have been successfully applied to standard fluid flow problems which use structured meshes to store data for their solution [18, 24].

Convolutional neural networks excel at this kind of compression because of locality properties and shared weights being applied to the whole dataset. In simulations on grids, it is very often the case that local groups of values are highly correlated and emergent features can easily be tackled using a local approach.[11]

Within fluid flow problems in particular, some simulations are computed and stored on unstructured meshes, which have varying "resolution" to capture finer details in boundaries or zones of higher interest (Fig.1)

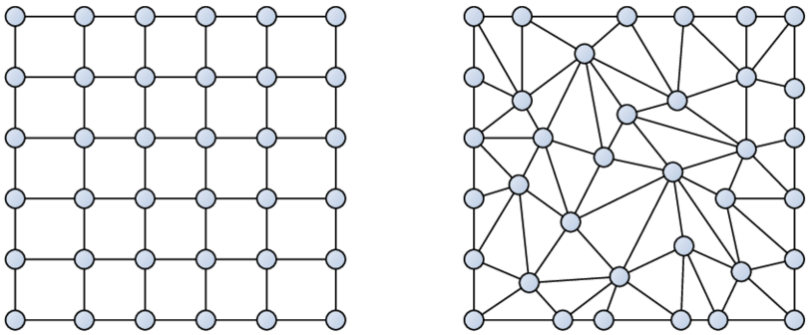


Figure 1: Structured Mesh (left) and Unstructured mesh (right)

This causes an issue for the application of standard Convolutional Autoencoders (or CAEs) to these problems, as the filters used in convolutional layers are designed for structured grids.

The order in which vertices, edges or faces of unstructured meshes are stored can be permuted while still describing the same physical system, which makes it hard to apply convolutional filters effectively.

Even defining convolutional filters on the manifold is quite complex because each vertex will have a different number of connections.

Nonetheless, several approaches have been proposed for tackling this [42] [3] [45] [15]

Graph convolutional networks (and several types of graph convolutional filters, though mainly for classification purposes) as well several other methods for dealing with unstructured meshes have been developed and are available for application.

During the course of this project two of them in particular were explored.

Graph Convolutional Networks have been developed quite recently (after 2016 [8]) and they have not yet been applied very successfully for this kind of compression yet. On the other hand, they have seen quite a lot of success in forward physical simulations on unstructured meshes [43] and the compression of 3D meshes in particular [47] [25].

During the course of this project a modification of a library called MeshCNN [13] has been developed specifically for 2D Fluid Flow Mesh compression applications. Several changes were made to the source code to obtain a Graph Convolutional Autoencoder (GCAE).

The largest advantage of Graph Convolutional Networks (or GCNs) over other methods is that they maintain locality directly. One disadvantage is that it is very hard to define transpose convolutional filters and unpooling operations that are invariant to rotations, translations and isotropic scaling. For MeshCNN in particular, these pooling/unpooling operations are also very slow and memory consuming (at least during training).

A promising and novel approach, Space Filling Curves [15], was also explored.

Space Filling Curves (or SFCs) are lines that fill a 2D or a 3D space (Fig.5) and can be used to order the data on the unstructured mesh such that 1D Convolutional Neural Networks (or CNNs) can be applied to the data meaningfully.

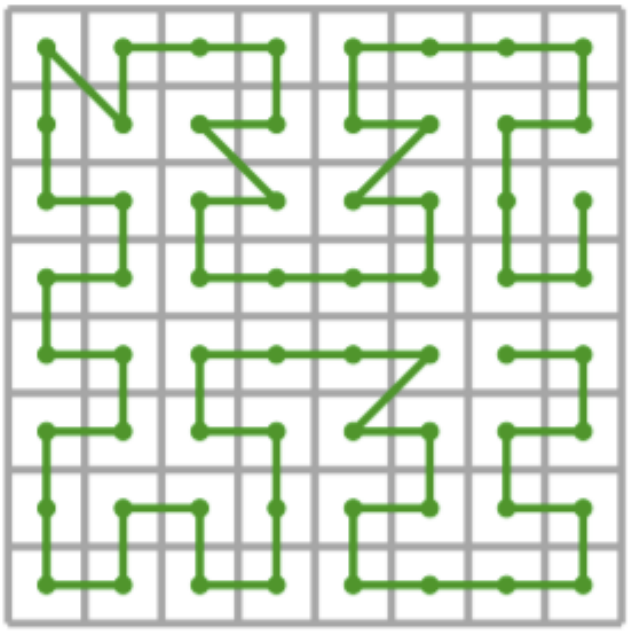


Figure 2: An example of a Hilbert Space Filling Curve

A couple major advantages of this approach are that the 1D ordered data maintains a great degree of spatial proximity and that it offers a data structure where coarsening is easy and immediately meaningful on the unstructured mesh as well [15]. This approach has also shown some strong results quite recently [16].

During this project the application of SFC-CAEs Autoencoders to adaptive meshes was explored.

Adaptive mesh refinement is a technique through which meshes used for a simulation are updated (the position and number of their nodes changes), often to bring more resolution to a sensitive or turbulent region within a simulation [20] [26].

Building on a library for the application of SFC-CAEs mostly developed by another student, Yu Jin, I have created and applied these SFC-CAEs to adaptive data.

My first experiments have been with trying to apply the SFC-CAE to problems defined on fixed meshes and using several different Space Filling Curves to see if the Autoencoder would still be able to compress the data well with a different ordering of nodes.

Once that was confirmed, I have also applied SFC-CAEs to adaptive data where both the ordering and the number of nodes changed. I have attempted several strategies to control for the input-output size of the autoencoder, some of which have been more successful than others.

Hyperparameter optimisation and different architectures have also been explored both for fixed mesh and adaptive mesh problems.

# 2 Problems explored

During the course of this project, I have explored 3 different problems.

## 2.1 Advecting Block

The first is the 2D advection of a square wave over a square domain. The PDE for 2 dimensional advection can be written as:

$$\frac{\partial c}{\partial t} = -\boldsymbol{U}\ \frac{\partial c}{\partial x} - \boldsymbol{V}\ \frac{\partial c}{\partial y} \tag{1}$$

The initial condition

$$c\left(x, y, 0\right) = c^{0}\left(x, y\right) \tag{2}$$

was implemented as a simple square wave.

The data was generated with Python using the following analytical solution:

$$c^{0}\left(x, y\right) = \begin{cases} 1, \ (x, y) \in \left[x_0 - \frac{d}{2}, x_0 - \frac{d}{2}\right] \times \left[y_0 - \frac{d}{2}, y_0 - \frac{d}{2}\right] \\ 0, \ \text{otherwise} \end{cases} \tag{3}$$

## 2.2 Flow Past a Cylinder, fixed mesh and adaptive mesh

The second and third problems were both incompressible 2D Flow Past a Cylinder CFD simulations with Reynolds number 3900, defined by the following equations

$$\nabla \cdot \boldsymbol{u} = 0 \tag{4}$$

$$\frac{\partial}{\partial t}\left(\boldsymbol{u}\right) + \nabla \cdot \left(\boldsymbol{u} \otimes \boldsymbol{u}\right) + \nabla \cdot \tau = -\nabla p \tag{5}$$

These were solved by Dr Heaney on both adaptive and non adaptive Continuous Galerkin [6]

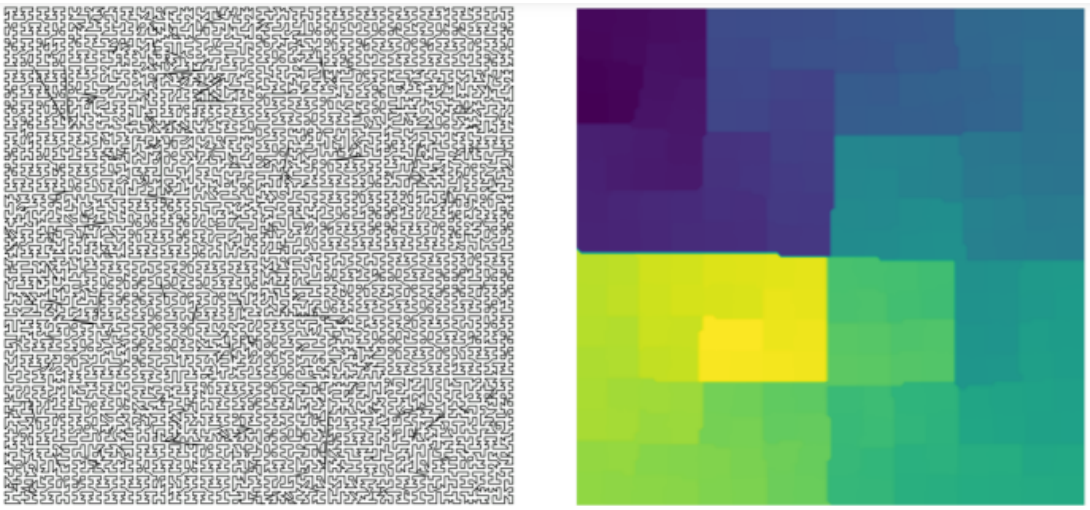
finite element meshes [15].

## 2.3 Generation of space filling curves

The Space Filling Curves were generated using a Recurrent Neural Network method called MFT RNN [23]

It was implemented within a FORTRAN script last year by the department [15] and I will be using it for my project.

Fig. 3 shows two generated Space Filling Curves on the 128x128 advection block problem domain and the flow past a cylinder 3571-vertices mesh.



(a) *SFC generated on square domain for the advection problem*

(b) *SFC generated for the FPC problem domain*

Figure 3: Space filling curves generated for the a) block advection and b) flow past a cylinder problems.

Since the fixed mesh problems were based on changing SFCs on the same mesh, a lot of care was taken to make sure that the Space Filling Curves were different from each other.

Sometimes when the MFT-RNN is asked to produce 500 curves, only about 200 of those that are generated are different. For optimal results, I advise asking for 4000 and then filtering them. Quite interestingly, the number of different curves it was able to produce for the advection block domain (128x128) was 512.

I have only generated 956 for the Flow Past a Cylinder problem.

# 3 Graph Convolutional Networks

## 3.1 Approach

Several libraries exist to facilitate the application of Graph Convolutional operations. Among the most well known and documented are PyTorch Geometric [10], Deep Learning Graph Library [9] and MeshCNN [13].

MeshCNN is the only one that offers the possibility to apply transpose convolutional graph operators and while the choice of pooling and convolutional operators is poorer than that offered by PyTorch Geometric, they are fully learnable and are more useful to our aim to build a convolutional unstructured mesh autoencoder (where save for a fully connected layer in the middle, both the encoder and the decoder are convolutional).

MeshCNN has been developed originally for 3D classification and segmentation purposes but I will be repurposing it to work as an Autoencoder for 2D problems as well.

## 3.2 Repurposing MeshCNN

### 3.2.1 Input features in MeshCNN

Input features in MeshCNN are defined on edges.

Starting from the CG data, I defined the feature vectors of the mesh edges as the average of their respective endpoints. This brought the total number of variables from 3571 (vertices) to 10421 (edges).

MeshCNN was created to compress 3D meshes, and it already defines a way to extract geometrical features from a simple mesh. Each edge's "extracted" features are 5: the dihedral angle, the two inner angles and two edge-length ratios for each face the edge is incident to. The edge ratio is defined as the ratio between the length of the edge being considered and the height of the triangular faces adjacent to it. These features are all invariant to translation, rotation and uniform scale.

In addition to these 5 features I concatenated my own two (x and y velocity) to form the input vector to the Graph Convolutional Autoencoder.

### 3.2.2 Graph Convolutional Operations

To define Graph Convolutional operators on unstructured meshes, MeshCNN assumes that they are manifold (possibly with boundary edges which are only incident to one face).

In such a mesh, each edge is adjacent to either two or four other edges. In the case it is adjacent to four edges, a convolution can be defined on it and it is ordered counterclockwise (within each face) (Fig.4).

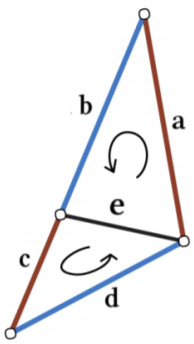



Figure 4: The ordering process for four edges adjacent to an edge

This first ordering operation still leaves some ambiguity (the 1-ring neighbors of the edge can be ordered as (a,b,c,d) or (c,d, a,b)), and to guarantee the invariance of the convolutional operators to similarity transformations, a further step is required. A simple symmetric function is applied on each pair of opposite edges (a and c, b and d in this case) to generate new, symmetrical features (example: replacing a and c by a+c and a+c).

The convolution is then applied on the new features which are symmetric and similarity transformation invariant.

### 3.2.3 Graph Convolutional Pooling

Pooling operators work through the collapse of edges.

When an edge is chosen, it is collapsed to a point (its midpoint) and consequently the edges of each face adjacent to the edge collapse into one. Five edges are transformed into two. With reference to Fig.4, edges a and b are collapsed together and so are edges c and d

The pooling operator works edge by edge, collapsing them in increasing order of their edge features norm. The pooling operator is therefore a learnable operator: the network learns to determine which edges it can collapse and which are more important to the end result.

This is advantageous to performance, but unfortunately having to remember the history of each edge collapse within each pooling operation is incredibly expensive memory-wise and though it is only needed for training (as the edge collapse procedure can then be fixed to a certain routine, as we are using a fixed mesh), it effectively limiting my batch size to 8 snapshots for training (for a train set of 1600 2x10421 float tensors) and took extremely long times to train to its maximum performance (up to 20 hours on a Google Colab GPU and up to 8 on a K80 offered by the Imperial College HPC)

### 3.2.4 Transpose Graph Convolutional Operations

Transpose Graph Convolutional operators are defined similarly to the aforementioned Graph Convolutional Operators.

### 3.2.5 Unpooling Operations

Unpooling operations are the exact reverse of the pooling operations: if edges a and b were averaged to form another edge c in a pooling operation, they are then set to equal a weighted combination of the pooled edge c features (where the weighting was stored previously).

This is typically combined with transpose convolutions to recover the lost resolution.

### 3.2.6 Problem with collapse

Aside from the obvious disadvantage of speed offered by MeshCNN's edge pooling approach (edge by edge collapse and memory storage of edge collapse history), it also sometimes imposes a very hard limit on the number of edges that one can collapse to, especially for 2D meshes.

Some edge collapses will inevitably lead to non-manifold vertices and eventually the algorithm runs out of edges with two adjacent faces to collapse and compression can go no further. [12]

The fact that the mesh is non-manifold would also seem to threaten the property of locality we are striving so hard to achieve.

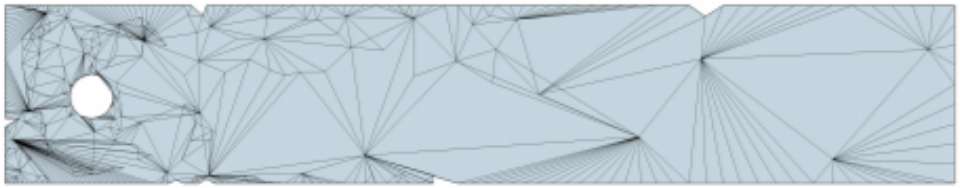
MeshCNN has a function that attempts to correct and remove these non-manifold vertices [14] whenever it is possible at the end of the edge collapse operations. Unfortunately that did not help me very much as I was only able to consistently compress down to 2200 edges (from 10421 edges

from our 3571 vertices) which I thought was not a terribly great performance as far as compression goes.

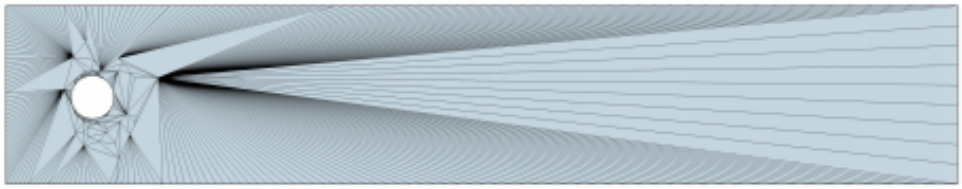
What I did was preventing the collapse of edges that would lead to non manifold vertices being created in the first place. I introduced a "checking" function to verify that after the collapse the edges connected to each of the vertices of the collapsed edge would not change ordering by angle.

By that I mean that for any edge collapse being considered, I calculated the angle to the x axis of the possible new edges (which will all share a vertex: the collapsed edge's midpoint) and if they have changed their (cyclically permuted) ordering then the collapse is prevented.[40]

This allowed me to collapse my mesh quite safely down to 500 variables, though I opted to only collapse down to 1024 in the end because of the excessive number of pooling layers I had to employ (if not done gradually, it would quite likely crash).



(a) *A badly compressed mesh with several non-manifold edges*



(b) *A mesh compressed to 500 edges after implementing the checking algorithm*

Figure 5: a) The compressed mesh (to 1200 variables) before the introduction of the checking algorithm and b) the compressed mesh to 500 variables after the introduction of the checking algorithm

Collapse beyond that point is most likely possible with PyTorch Geometric, though an arbitrary predefined edges-pooled edges correspondence would need to be established (possibly manually). This avenue was not explored during this project.

## 3.3 Architecture of GCAE

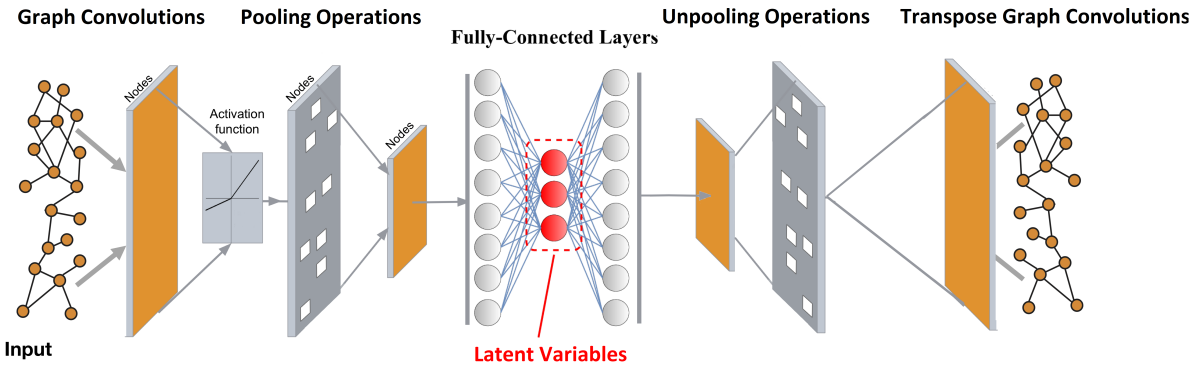



Figure 6: The structure of the GCAE: the convolutions-pooling and the unpooling-transpose convolutions steps are repeated several times

The final library (notebook) used to produce the results presented within this report is available here [40]

The final structure(Fig. 6) used to produce the results presented within this report is the following:

| **Operations** | **Size** | **Channels** |
|---|---|---|
| Input | 10421 | 7 (5 extracted features + x and y velocities) |
| 7 to 8 channel convolutions | 10421 | 8 |
| 10421–>4096 | 4096 | 8 |
| 8 to 8 channel convolutions | 4096 | 8 |
| 4096–>2048 | 2048 | 8 |
| 8 to 4 channel convolution | 2048 | 4 |
| 2048–>1024 | 1024 | 4 |
| 4 to 2 channel convolution | 1024 | 2 |
| Fully connected (2048 to 256) layer | 256 | 1 |
| nn.Tanh() | 256 | 1 |
| Fully connected (256 to 2048) layer | 2048 | 1 |
| nn.Tanh() | 2048 | 1 |
| 2 to 4 channel convolution | 1024 | 4 |
| 1024–>2048 | 2048 | 4 |
| 4 to 8 channel convolution | 2048 | 8 |
| 2048–>4096 | 4096 | 8 |
| 8 to 8 channel convolution | 4096 | 8 |
| 4096–>10421 | 10421 | 8 |
| 8 to 2 channel convolution | 10421 | 2 |

The number of latent variables is 256.

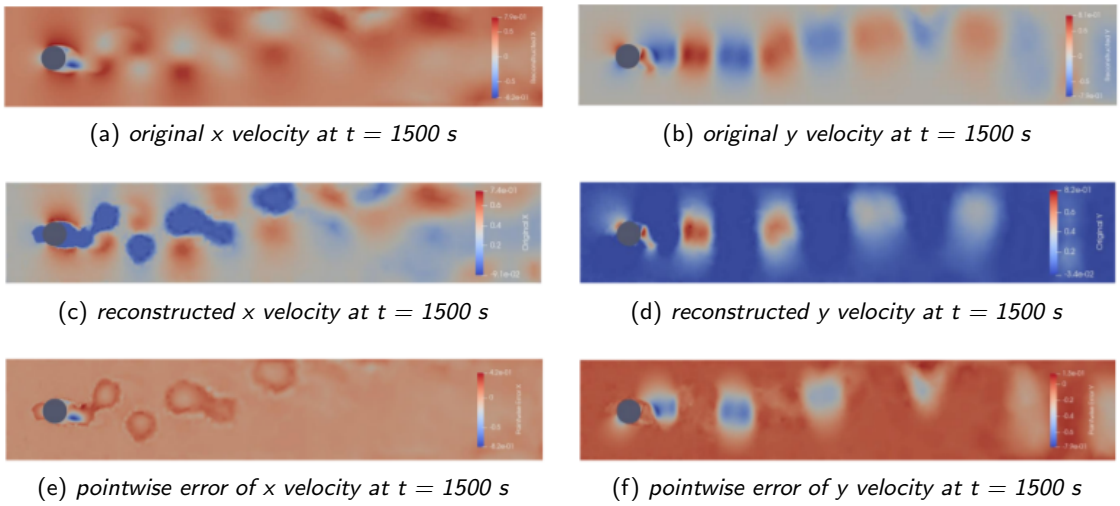
The activation function used after each fully connected layers was Tanh() because I have found it speeds up the improvement of the collapsing order somewhat.

The data was also normalised before being inputted to the GCAE.

## 3.4 Performance on FPC CG data

Performance reached was not very high.

On normalised vectors of Flow Past a Cylinder CG data, the GCAE never achieved anything better than a relative MSE loss of 0.20 7.



(a) *original x velocity at t = 1500 s*

(b) *original y velocity at t = 1500 s*

(c) *reconstructed x velocity at t = 1500 s*

(d) *reconstructed y velocity at t = 1500 s*

(e) *pointwise error of x velocity at t = 1500 s*

(f) *pointwise error of y velocity at t = 1500 s*

Figure 7: Results from the GCAE:Unfortunately I forgot to change the coordinate scale.

# 4 Adaptive SFC-CAE

## 4.1 Approach

Building on code developed doing a past ACSE project last year for the application of SFC-CAEs, one of my colleagues from the Autoencoder group, Yu Jin, has developed a very nifty library for their application.

Over the course of this project we have shared a repository, joined our code a few times and given each other tips on what functions to implement, my "adaptive" repository is a branch of his [17]

The latest iteration of my work is built from the latest version of Yu Jin's excellent code.

Components of the architecture of the SFC-CAE autoencoder (SFC ordering layer, sparse layers, convolutional layers, fully connected layers) are as described in Yu Jin's report [16].

There were a few additions: one can now choose the number of fully connected layers, can specify parameters and a verbosity option.

### 4.1.1 Smoothing Layers

Smoothing layers are simply stride 1 convolutional layers (normally with a high number of channels) that are placed either before the start of the stride 4 (or higher) convolutions in the encoder or after the decoder.

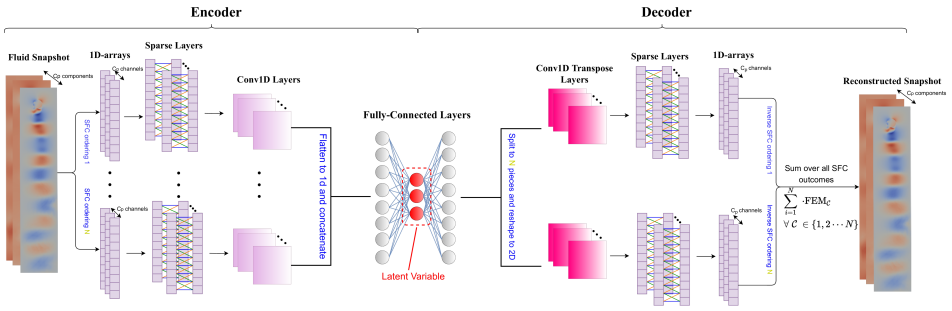
### 4.1.2 Coordinate feeding to layers

A "coordinate feeding" option was implemented, so that the user could choose whether to concatenate the (SFC-ordered) coordinates of the points to the data within any hidden layer (convolutional or fully-connected).

In hidden layers where the length of the data is smaller than the input size, the coordinates are "coarsened" (if the hidden layer's data has length 1000 and the input size is 4000, only one in four coordinates is picked) to provide an accurate representation of the spatial relationships between vertices at that level.

There are two options: coption 1 and coption 2. When coption == 1, the coordinates are directly concatenated to the data as two additional channels (x and y).

When coption == 2, the coordinates are manipulated to find instead the difference between each node and its two neighbours on the SFC. A zero is appended to the tensor to remedy the absence of neighbours on either side of the SFC. This "difference" or "distance" representation of coordinates is then concatenated to the data as four additional channels (x and y distance from previous node, x and y distance from the successive node).




(a) SFC-CAE

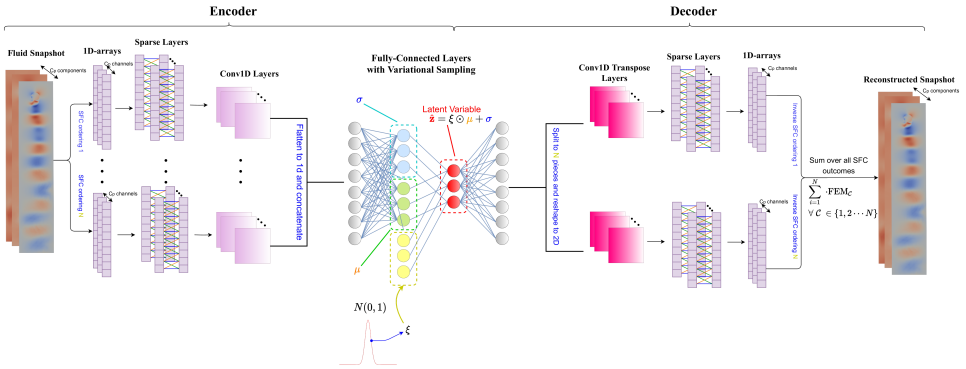



Figure 8: Architecture of Space Filling Curve Convolutional Autoencoder (here with 2 SFCs).

# 5 Fixed mesh, shuffled curves problems

As mentioned in the introduction, I first applied SFC-CAEs to data from fixed meshes ordered by different space-filling curves.

The idea is that if the 1D convolutional filters are able to generalise to unseen space-filling curves, then they are more likely to be able to generalise on unseen meshes for adaptive simulations

## 5.1 Advecting block problem - Fixed mesh

I started with the simplest structure and the simplest problem possible, one that I already had done for the GCAE exploration.

The problem is defined as a simple advecting square wave on a structured grid.

I created a "shuffle" option within the training function and provided it with several space filling curves to shuffle through. A number N of indices between 0 and the length of this sfcstoshuffle list, where N is the number of space filling curves needed for the autoencoder, are picked randomly for every batch.

Since the general idea of this project is that we want it to generalise to unseen space-filling curves, I have also added a parameter within the training function that enables the user to provide validation space filling curves to shuffle for validation.

This provides us with a live update of how well the model generalises on unseen space filling curves. At the end of training, the test set (unseen data) is then compressed using one of these unseen curves and invariably it will return a similar MSE loss to the one found for the validation.

I had access to 512 space filling curves, and the best results I have obtained were on a 500/12 split between training and validation.

| | **Advecting block, 3 fcl** | **Advecting Block, 0 fcl** | **Latent** |
|---|---|---|---|
| Train set | $2.55 \times 10^{-1}$ | 2.907 | |
| Valid set | $3.03 \times 10^{-1}$ | 2.957 | 16 |
| Test set | $3.06 \times 10^{-1}$ | 2.985 | |
| Train set | $1.45 \times 10^{-1}$ | $1.74 \times 10^{-1}$ | |
| Valid set | $2.06 \times 10^{-1}$ | $2.52 \times 10^{-1}$ | 32 |
| Test set | $2.06 \times 10^{-1}$ | $2.61 \times 10^{-1}$ | |
| Train set | $1.41 \times 10^{-1}$ | $1.85 \times 10^{-1}$ | |
| Valid set | $1.94 \times 10^{-1}$ | $2.38 \times 10^{-1}$ | 64 |
| Test set | $1.99 \times 10^{-1}$ | $2.41 \times 10^{-1}$ | |

Table 1: Relative MSE loss for SFC-CAE on the advecting block problem, with 3 or 0 fully connected layers in the middle [27]

The problem did generalise quite decently 2, for a high number of curves, both with and without fully connected layers in the middle.

Without fully connected layers, training took three or four times longer to show signs of convergence and it would not converge for latent variables $= 16$ (or maybe it would eventually converge with more training). For a number of latent variables $>= 32$, the autoencoder's relative MSE reached very similar relative validation MSE loss.

Even though the network works when fully convolutional, fully connected layers help quite a bit. As an alternative to fully connected layers, smoothing layers (especially after the decoder) and coordinate feeding (mostly in combination, better when coption $== 2$) also seems to be helpful.

The sparse layers are also quite useful and I have used them for my best architectures for all three problems.

### 5.1.1 Hyperparameter optimisation

I have ran a couple Bayesian search projects for hyperparameter optimisation on weights and biases [44] for 64 latent variables for the advecting block problem.

For the following parameters:[28]

- increase_multi

- kernel_size

- num_final_channels

- stride

What has emerged is that the parameters that we had guessed (4,32,16,4) were not too far off from their "optimal" configuration (8,36,32,4), and that the number of maximum channels is the most important parameter out of all these and it is highly correlated with the performance of the autoencoder itself.

The second Bayesian search I have ran was for:[29]

- nearest_neighbouring (the parameter which controls the application of sparse layers)

- nfclayers (number of fully connected layers)

- number of curves the autoencoder takes as an input.

What has emerged is that sparse layers are still effective with changing curves, that the more fully connected layers the better and, quite confusingly, that one space filling curve only is better than two or more, although the image produced looks a lot less like a square.

## 5.2 Flow past a cylinder problem - Fixed Mesh

I followed a similar approach for the Flow Past a Cylinder problem.

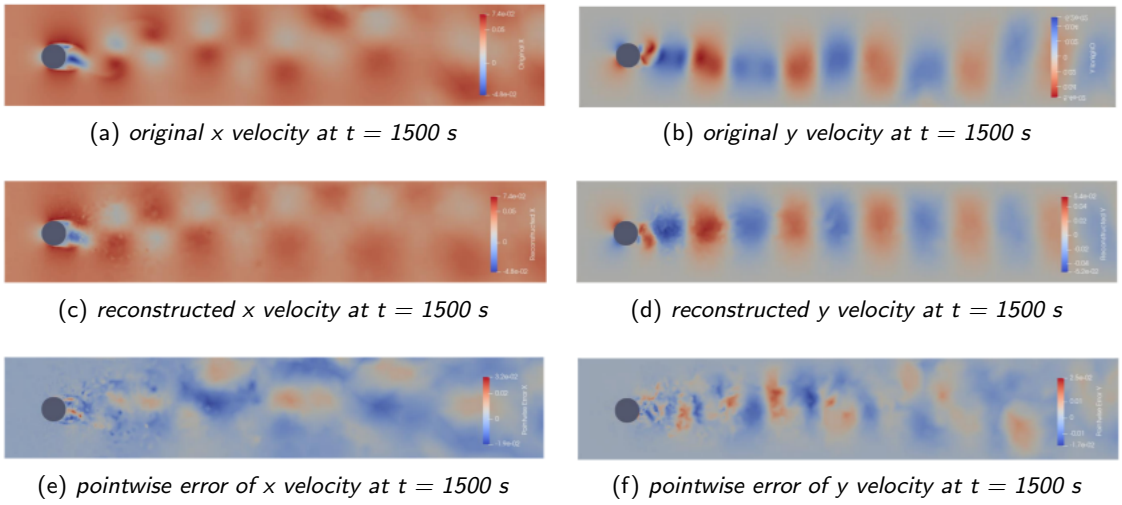
I used a shuffling function for both training and loading, and I made sure to provide the training with lots of different space filling curves.

The results from this latter fixed mesh problem have been much more encouraging than the ones from the advecting block: I have found that not only does it work a lot better for 64 latent dimensions, it also works immensely better for 16.

| | **FPC, 3 fcl, 500 epochs** | **FPC, 0 fcl, 3000 epochs** | **Latent** |
|---|---|---|---|
| Train set | $4.8 \times 10^{-2}$ | $3.05 \times 10^{-1}$ | |
| Valid set | $4.9 \times 10^{-2}$ | $3.14 \times 10^{-1}$ | 16 |
| Test set | $5.2 \times 10^{-2}$ | $3.29 \times 10^{-1}$ | |
| Train set | $4.7 \times 10^{-2}$ | $3.25 \times 10^{-1}$ | |
| Valid set | $4.7 \times 10^{-2}$ | $3.30 \times 10^{-1}$ | 32 |
| Test set | $5.0 \times 10^{-2}$ | $3.40 \times 10^{-1}$ | |
| Train set | $4.2 \times 10^{-2}$ | $3.0 \times 10^{-2}$ | |
| Valid set | $4.5 \times 10^{-2}$ | $3.2 \times 10^{-2}$ | 64 |
| Test set | $4.4 \times 10^{-2}$ | $3.2 \times 10^{-2}$ | |
| Train set | $3.9 \times 10^{-2}$ | $2.85 \times 10^{-2}$ | |
| Valid set | $4.2 \times 10^{-2}$ | $2.88 \times 10^{-2}$ | 128 |
| Test set | $4.4 \times 10^{-2}$ | $2.93 \times 10^{-2}$ | |

Table 2: Relative MSE loss for SFC-CAE on the flow past a cylinder problem, with 3 or 0 fully connected layers in the middle and 500 and 3000 epochs of training respectively [30]

When reading Table 2, it is important to keep in mind that the results for the non-fully connected autoencoder were obtained after 3000 epochs and the ones obtained for the fully connected one after 500. It is quite likely that the fully connected autoencoder would have surpassed those results if given the training time.



(a) *original x velocity at t = 1500 s*

(b) *original y velocity at t = 1500 s*

(c) *reconstructed x velocity at t = 1500 s*

(d) *reconstructed y velocity at t = 1500 s*

(e) *pointwise error of x velocity at t = 1500 s*

(f) *pointwise error of y velocity at t = 1500 s*

Figure 9: Results of SFC-CAE on flow past a cylinder fixed mesh problem

### 5.2.1 Hyperparameter optimisation and experiments

I have ran a Bayesian search projects for hyperparameter optimisation on weights and biases for 64 latent variables for the flow past a cylinder problem too.

Unfortunately I have deleted it by accident as I was tidying my projects up but the results were similar to those that have emerged from the Advecting Block problem as well: optimal stride, kernel size and factor of increase of channels are within the "standard" range, and the number of maximum channels is instead strongly correlated with the performance.

I have also ran a Bayesian search for number of fully connected layers, sparse layers on/off and number of curves as input to the autoencoder.[31]

What has emerged is that more curves are better, as opposed to the block. Otherwise it's very similar in its results: having sparse layers and 3 fully connected layers is the optimal solution.

As for the block, it converges well even without any fully connected layers, it just takes a lot (4 to 6 times) longer.

I have also ran several more experiments on the FPC data that I assume would extend to other problems as well.

I have briefly investigated the possibility of updating the Autoencoder's space-filling curve more frequently (up to a maximum of using a different space-filling curve for every single snapshot in the training set), but after several very long runs there is very little difference for the validation relative MSE achieved in the same process time by different batch sizes in the range 16-64. It looks like the optimal batch size is 32, and only extreme values (1 or 128) display a significantly worse performance.[32]

I have also investigated the possibility of coordinate scaling. Especially for coption == 2 (feeding the distance to other points), coordinates are scaled very differently to the data.[33]

What has emerged is that coordinate scaling does have an impact on the performance of the autoencoder, and just like for batch_size, there is a "sweet spot".

Perhaps what could be explored in the future is a standardisation function being applied at each (possibly coarsened) level's coordinate difference.

Similarly to the advection block problem, I have found that coordinate and smoothing layers are

quite helpful without fully connected layers, [34] but otherwise have relatively little impact [35] [36], especially for the Flow Past a Cylinder problem. With 3 fully connected layers, the only noticeable difference is made by a 64 channel kernel size around 17 smoothing layer after the decoder.

I have also investigated other splits and how well the model generalises as a function of the number of space filling curves [37]

As one can clearly see, 20 curves aren't enough for the autoencoder to be able to generalise to 36 unseen ones and the validation loss never converges to below 0.015. It generalises increasingly better as we increase the number of training curves from 100 to 500 to 920. Training MSE does increase a little bit from 100 to 500 curves, but not much from 500 to 920.

For the autoencoder to generalise equally well to unseen Space Filling Curves, one needs to provide it with as many Space Filling Curves as possible. I suspect I could have achieved an even lower validation MSE with 2000 curves.

Alternative attempts to prevent what could be seen partly as "overfitting", such as regularisation through very small weight decay or dropout within the fully connected layers didn't work well at all.

I also ran a couple more experiments on the fpc.

The first one was trying to reconstruct the coordinates of the snapshot as well as its velocity components. This was ideally meant to force the autoencoder to include a representation of the coordinates within its hidden layers in the hope it could possibly improve the loss on the velocity components. What emerged (with various coordinate scaling attempts) is that this is not a good idea (at least for a SFC-CAE with 3 fully connected layers on an FPC problem) [38]

(I realise now that the loss shown here is computed on the entire output vector and not just the first two channels, but I did also check for that)

The final experiment I have ran on the Flow Past a Cylinder was using the samefilter option with twice the number of channels. [39]

It seems that using two filters is the optimal solution, though those are only two runs and it is quite possible that they would be equivalent if I did more.

# 6 Adaptive Flow Past a Cylinder

For the Adaptive Flow Past a Cylinder, I used the adaptive Flow Past a Cylinder CG simulation data created by Dr Heaney. Each of the snapshots has a different number of nodes with different coordinates, so I have used the MFT-RNN to generate 10 space-filling curves for each.

Using adaptive data required some adjustments to be made to the structure of the autoencoder itself: differently sized inputs might well produce the same number of latent variables after several convolutions, but even if they do the decoder will take those latent variables and produce an output of a set size which will almost always not be the size of the original snapshot (if a 3560x2 tensor is fed to an SFC-CAE created with input size 3571, it will return a 3571x2 tensor, which is not ideal for calculating the loss or ordering it with a 3560-long Space Filling Curve)

I have explored three different options to remedy the problem.

## 6.1 Adaptive flow past a cylinder - Direct padding

My first approach was to pad the data very directly. I have attempted to pad the tensors being fed to the autoencoder with zeroes, extending them to the maximum length within the set of adaptive snapshots (4072).

The space filling curves generated on those snapshots (previous to their expansion) were extended with "dummy" indices that reorder each point to its own location.

The tensors were then fed to an otherwise completely regular SFC-CAE and the curves provided to the training and validation functions.

Unfortunately the approach does not seem to work very well, and the validation MSE relative loss has never dropped below 0.3. I have therefore opted not to generate any visualisations or weights and biases runs.

On the suggestion of Dr Pain I have tried using no bias in the decoder's convolutional layers. I have also (both separately from and together with Dr Pain's suggestion) attempted to feed the coordinates (-50 in x and -50 in y, a very extreme value that I hoped would be "noticed") directly to the fully connected layers.

Both attempts were without success and results were as mentioned not worthy of being displayed.

## 6.2 Adaptive flow past a cylinder - Decoder padding method

Another option that was explored was to use the same filters as before, but controlling for latent variable size and increasing the output padding in the output layers.

I implemented this within a new class sfc_cae_adaptive.

The size and stride of the convolutional filters is the same in every layer of the autoencoder, and so by using the function findsizeconv to find the right number of layers and outputpaddings, I introduced it as a control within the forward() function.

Whenever the forward() function of the autoencoder is called on a tensor, it automatically finds the right number of layers and the right output paddings to output a tensor of the same size.

The output paddings and the number of layers the tensor goes through (it didn't vary much, it was either 4 or 5) are then dynamically updated even though the same filters are applied on data of different sizes.

This has been the most promising out of all approaches, and it is a shame that I cannot continue to pursue it. I have only been able to make this work for a single curve (for reasons obscure to me) and without any fully connected layers, which no doubt would have helped.

Nevertheless, the results with 128 latent variables are quite encouraging.

A relative MSE loss of 0.11 was achieved on unseen snapshots, and the reconstruction of the flow is quite visually accurate (Fig. 10 ) ([41])

## 6.3 Adaptive flow past a cylinder - Interpolation method

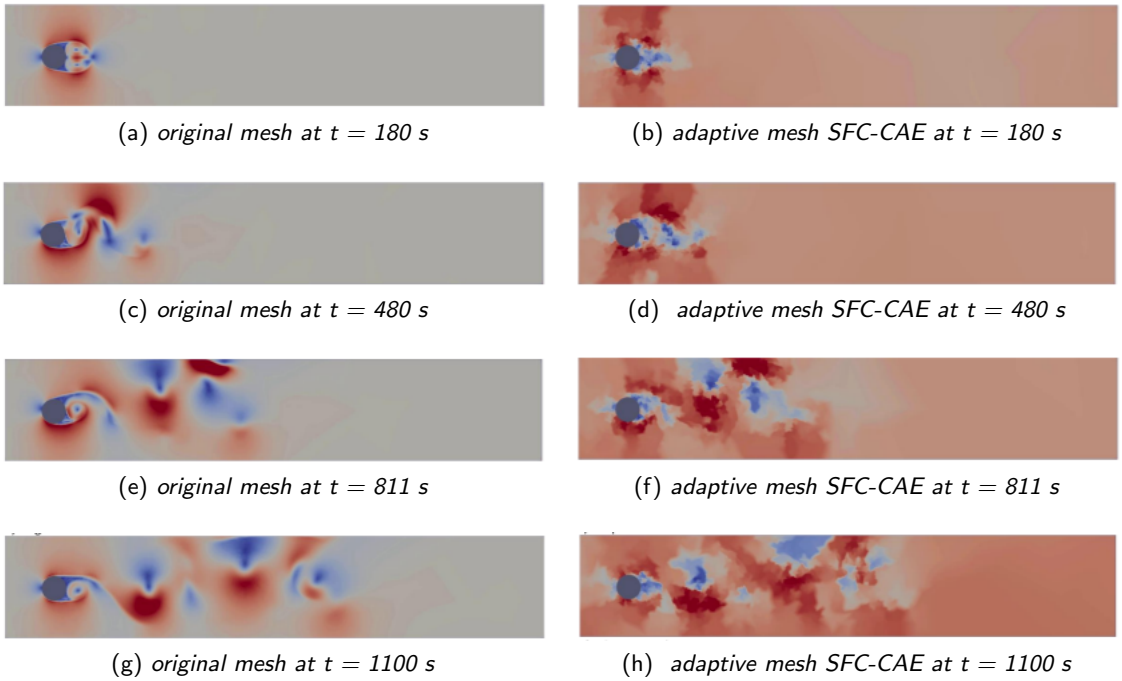
The other idea I have attempted had was to linearly interpolate each curve (and ordered points) so that all curves and snapshots would have the same length (even though with different coordinates). A linear interpolation method was developed by Dr Pain and implemented within a fortran script.

The Interpolating Adaptive SFC-CAE is implemented itself within a separate class sfc_cae_interpol.

I ordered each snapshots in the 10 different ways I had pre-generated, I interpolated each and fed both the velocity components and the coordinates as a 4 channel input to the autoencoder (which then would return 2 channels only).

I calculted the loss (MSE error) on the velocity components.

Although the autoencoder did reach quite low losses (such as a relative MSE loss of 0.14 on an unseen snapshot), I have had no time to properly produce the findings and report them here.



(a) *original mesh at $t = 180$ s* (b) *adaptive mesh SFC-CAE at $t = 180$ s*

(c) *original mesh at $t = 480$ s* (d) *adaptive mesh SFC-CAE at $t = 480$ s*

(e) *original mesh at $t = 811$ s* (f) *adaptive mesh SFC-CAE at $t = 811$ s*

(g) *original mesh at $t = 1100$ s* (h) *adaptive mesh SFC-CAE at $t = 1100$ s*

Figure 10: Adaptive mesh reconstruction of flow past a cylinder with 128 latent variables.

An interesting note is that the convergence only started to appear after I started training it on the full set of snapshots ordered in 10 different ways. Before that I had prototyped it with a single space filling ordering per snapshot. This suggests that just like the SFC-CAE for the flow past a cylinder fixed mesh problem generalised better the more curves it was trained on, the adaptive flow past a cylinder SFC-CAE could possibly generalise better if it was trained on more Space Filling Curves.

I also forgot to attempt to feed it the "distance" vector (like the one generated by option 2) instead of the plain coordinates.

# 7 Conclusion

I think that even though the Adaptive Mesh SFC-CAEs I have created did not work exceedingly well, the fact the fixed mesh SFC-CAEs worked so well with different curves is quite encouraging as to the possible applications of SFC-CAEs to adaptive meshes. As mentioned before, convergence with these architectures is really tricky and requires a lot of data, a lot of curves and a lot of patience in trying out different architectures. I am sure that much better results can be reached.

## References


[1] Ahmed, S. E., Rahman, S. M., San, O., Rasheed, A. and Navon, I. M. [2019], 'Memory embedded non-intrusive reduced order modeling of non-ergodic flows', *Physics of Fluids* .
**URL:** *https://doi.org/10.1063/1.5128374*

[2] Ahmed, S. E., San, O., Bistrian, D. A. and Navon, I. M. [2020], 'Sampling and resolution characteristics in reduced order models of shallow water equations: Intrusive vs nonintrusive'.
**URL:** *https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.4815*

[3] Bouritsas, G., Bokhnyak, S., Ploumpis1, S., Bronstein, M. and Zafeiriou1, S. [2019], 'Neural 3d morphable models: Spiral convolutional networks for 3d shape representation learning and generation'.
**URL:** *https://arxiv.org/pdf/1905.02876.pdf*

[4] Brunton, S. L., Noack, B. R. and Koumoutsakos, P. [2020], 'Machine learning for fluid mechanics', *Annual Review of Fluid Mechanics* .
**URL:** *https://www.annualreviews.org/doi/full/10.1146/annurev-fluid-010719-060214*

[5] Cammilleri, A., Guéniat, F., Carlier, J., Pastur, L., Mémin, E., Lusseyran, F. and Artana, G. [2013], 'Pod-spectral decomposition for fluid flow analysis and model reduction'.
**URL:** *https://hal.inria.fr/hal-00793380/document*

[6] Cangiani, A., Chapman, J., Georgoulis, E. and Jensen, M. [n.d.], 'Implementation of the continuous-discontinuous galerkin finite element method'.
**URL:** *https://www2.le.ac.uk/departments/mathematics/extranet/staff-material/staff-profiles/eg64/papers/ENUMATH_cdG_final.pdf*

[7] David J.Lucia, Philip S.Beran, W. A. [2004], 'Reduced-order modeling: new approaches for computational physics'.
**URL:** *https://www.sciencedirect.com/science/article/abs/pii/S0376042103001131*

[8] Defferrard, M., Bresson, X. and Vandergheynst, P. [2016], 'Convolutional neural networks on graphs with fast localized spectral filtering', *arXiv* .
**URL:** *https://arxiv.org/abs/1606.09375*

[9] DGLTeam [2018], '"A blitz introduction to DGL"'.
**URL:** *https://docs.dgl.ai/tutorials/blitz/index.html*

[10] Fey, M. [2021], 'Pytorch geometric documentation'.
**URL:** *https://pytorch-geometric.readthedocs.io/en/latest/*

[11] Gonzalez, F. J. and Balajewicz, M. [2018], 'Deep convolutional recurrent autoencoders for learning low-dimensional feature dynamics of fluid systems'.
**URL:** *https://arxiv.org/pdf/1808.01346.pdf*

[12] Hanocka, R., Hertz, A., Fish, N., Giryes, R., Fleishman, S. and Cohen-Or, D. [2019*a*], 'Code crashing under certain examples'.
**URL:** *https://github.com/ranahanocka/MeshCNN/*

[13] Hanocka, R., Hertz, A., Fish, N., Giryes, R., Fleishman, S. and Cohen-Or, D. [2019*b*], 'MeshCNN: A network with an edge'.
**URL:** *https://arxiv.org/abs/1809.05910*

[14] Hanocka, R., Hertz, A., Fish, N., Giryes, R., Fleishman, S. and Cohen-Or2, D. [2019].
**URL:** *https://github.com/ranahanocka/MeshCNN/issues/43*

[15] Heaney, C. E., Li, Y., Matar, O. K. and Pain, C. C. [2020], 'Applying convolutional neural networks to data on unstructured meshes with space-filling curves'. last revised 2021.
**URL:** *https://arxiv.org/abs/2011.14820*

[16] Ju, Y. [2021*a*], 'Dimensionality reduction with Space- filling curve autoencoders for fluid flow problems solved on unstructured meshes'.
**URL:** *https://github.com/acse-jy220/SFC-CAE-Ready-to-use/blob/main/JinYu_ACSE9_FinalReport.pdf*

[17] Ju, Y. [2021*b*], 'Yin ju repository'.
**URL:** *https://github.com/acse-jy220/SFC-CAE-Ready-to-use*

[18] Kim, Y., Choi, Y., Widemann, D. and Zohdi, T. [2020], 'A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder', *arXiv* .
**URL:** *https://arxiv.org/abs/2009.11990*

[19] Lee, K. and Carlberg, K. T. [2019], 'Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders'.
**URL:** *https://arxiv.org/abs/1812.08373*

[20] Löhner, R. [1995], 'Mesh adaptation in fluid mechanics', *Engineering Fracture Mechanics* .
**URL:** *https://www.sciencedirect.com/science/article/abs/pii/0013794494E0062L*

[21] Maulik1, R., Lusch1, B. and Balaprakash, P. [2021], 'Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders', *Physics of Fluids* .
**URL:** *https://aip.scitation.org/doi/10.1063/5.0039986*

[22] Milano, M. and Koumoutsakos, P. [2002], 'Neural network modeling for near wall turbulent flow using deep convolutional autoencoders'.
**URL:** *https://www.sciencedirect.com/science/article/pii/S0021999102971469*

[23] Pain, C. C., Oliveira, C. R. E. D. and Goddard, A. J. H. [1999], 'A neural network graph partitioning procedure for grid-based domain decomposition'.
**URL:** *https://onlinelibrary.wiley.com/doi/abs/10.1002/(SICI)1097-0207(19990220)44:5%3C593::AID-NME516%3E3.0.CO;2-0*

[24] Philips, T., Heaney, C., Smith, P. N. and Pain, C. [2020], 'An autoencoder-based reduced-order model for eigenvalue problems with application to neutron diffusion', *International Journal for Numerical Methods in Engineering* .
**URL:** *https://doi.org/10.1002/nme.6681*

[25] Piperakis, E. and Kumazawa, I. [2003], '3d polygon mesh compression with multi layer feed forward neural networks', *Journal of Systemics and Cybernetics and Informatics* .
**URL:** *http://www.iiisci.org/journal/PDV/sci/pdfs/001251.pdf*

[26] Powell, K. G., Roe, P. L. and Quirk, J. [1993], 'Adaptive-mesh algorithm,s for computational fluid dynamics', *Algorithmic Trends in Computational Fluid Dynamics* .
**URL:** *https://link.springer.com/chapter/10.1007/978-1-4612-2708-3_18*

[27] Pozzetti, A. [2021*a*].
**URL:** *https://wandb.ai/ap2920/Ndims%20for%20Advection%20Block/overview?workspace=*

[28] Pozzetti, A. [2021*b*].
**URL:** *https://wandb.ai/ap2920/Bayesian%20block%20param%20optimisation?workspace=user-ap2920*

[29] Pozzetti, A. [2021*c*].
**URL:** *https://wandb.ai/ap2920/Bayesian%20block%20optimisation,%20final/table?workspace=user-ap2920*

[30] Pozzetti, A. [2021*d*].
**URL:** *https://wandb.ai/ap2920/Ndims%20for%20fpc/overview?workspace=user-ap2920*

[31] Pozzetti, A. [2021*e*].
**URL:** *https://wandb.ai/ap2920/Bayesian%20fpc%20optimisation,%20final/table?workspace=user-ap2920*

[32] Pozzetti, A. [2021*f*].
**URL:** *https://wandb.ai/ap2920/Batch%20size%20investigation%20fpc?workspace=user-ap2920*

[33] Pozzetti, A. [2021*g*].
**URL:** *https://wandb.ai/ap2920/Coordscaling%20for%20fpc?workspace=user-ap2920*

[34] Pozzetti, A. [2021*h*].
**URL:** *https://wandb.ai/ap2920/Coordlayers%20for%20fpc%20with%20no%20fclayers/table?workspace=user-ap2920*

[35] Pozzetti, A. [2021*i*].
**URL:** *https://wandb.ai/ap2920/Coordlayers%20for%20fpc/table?workspace=user-ap2920*

[36] Pozzetti, A. [2021*j*].
**URL:** *https://wandb.ai/ap2920/Smoothinglayers%20for%20fpc/table?workspace=user-ap2920*

[37] Pozzetti, A. [2021*k*].
**URL:** *https://wandb.ai/ap2920/Ntrainingcurves,%20fpc%20with%202%20nfclayers%20and%202%20curves/tab*
*ap2920*

[38] Pozzetti, A. [2021*l*].
**URL:** *https://wandb.ai/ap2920/Reconstructing%20x%20and%20y%20on%20fpc/table?workspace=user-ap2920*

[39] Pozzetti, A. [2021*m*].
**URL:** *https://wandb.ai/ap2920/Samefilter?workspace=user-ap2920*

[40] Pozzetti, A. [2021*n*], 'Andrea pozzetti github repository'.
**URL:** *https://github.com/acse-ap2920/GCAE*

[41] Pozzetti, A. [2021*o*], 'Reconstructed flow past a cylinder on an adaptive mesh'.
**URL:** *https://www.youtube.com/watch?v=B0UlpYydlZ0*

[42] Sahbi, H. [2021], 'Learning Chebyshev basis in graph convolutional networks for skeleton-based action recognition'.
**URL:** *https://arxiv.org/abs/2104.05482*

[43] Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J. and Battaglia, P. W. [2020], 'Learning mesh-based simulation with graph networks'.
**URL:** *https://arxiv.org/pdf/2010.03409*

[44] *Weights and Biases* [n.d.].
**URL:** *https://gitbook-docs.wandb.ai/*

[45] Yang, S., Gong, Z., Ye, K., Wei, Y., Huang, Z. and Huang, Z. [2019], 'EdgeCNN: Convolutional neural network classification model with small inputs for edge computing'.
**URL:** *https://arxiv.org/abs/1909.13522*

[46] Zhang, S., Tong, H., Xu, J. and Maciejewski, R. [2019], 'Graph convolutional networks: a comprehensive review', *Computational Social Networks* .
**URL:** *https://computationalsocialnetworks.springeropen.com/articles/10.1186/s40649-019-0069-y*

[47] Zhou, Y., Wu, C., Li, Z., Cao, C., Ye, Y., Saragih, J., Li, H. and Sheikh", Y. [2020], 'Fully convolutional mesh autoencoder using efficient spatially varying kernels', *arXiv* .
**URL:** *https://arxiv.org/pdf/2006.04325.pdf*

# Reconstruction of Advection Block from shuffled curve SFC-CAE

## A Appendix

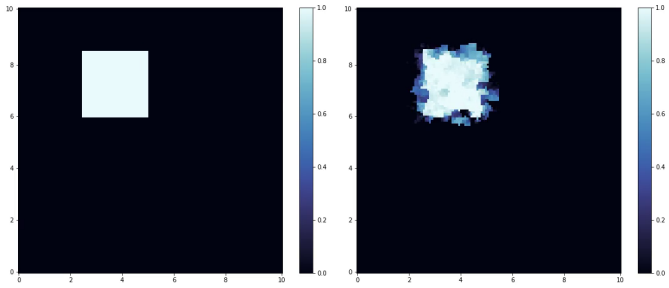


Figure 11: Advecting square wave reconstructed by SFC-CAE on unseen space filling curve on 64 variables.