Imperial College London

Department of Earth Science and Engineering

MSc in Applied Computational Science and Engineering

Independent Research Project
Final Report

Solving Partial Differential Equations with AI Libraries

by

Chukwume Ijeh

Email: cgi21@ic.ac.uk
GitHub username: acse-cgi21
Repository: https://github.com/ese-msc-2021/irp-cgi21/

Supervisors:

Christopher Pain

Claire Heaney
Boyang Chen

September 2022

1 Abstract

Developing Computational Fluid Dynamics (CFD) models are extremely complex, and thus requires great expertise. CFD models are also computationally expensive to run, with some demanding computations taking days or weeks. With the presence of open-source artificial intelligence (AI) libraries, and the recent advent of AI computers capable of delivering the wall-clock compute performance of many tens to hundreds of graphics processing units (GPU), this thesis aims to address the problems of complexity, and computational costs associated with developing CFD models. To demonstrate this, we solve the scalar transport equation using a graph neural network (GNN) and a geometric multigrid solver. The GNN, which utilizes unstructured data in form of a graph, is shown to be successful in solving the 2D advection-diffusion equation (ADE), including the pure diffusion equation. The geometric multigrid solver which uses space filling curves (SFCs) to transform 2D unstructured data to 1D, is shown to robustly solve the pure diffusion case as well. We then compare the computational time taken for the GNN and multigrid solver to reach convergence, both solving the pure diffusion problem on a number of different mesh sizes. The multigrid method is shown to be up to 50 times faster than the GNN.

Keywords: Convolutional Neural Network, Graph Neural Network, Space Filling Curves, Scalar Transport Equation, Unstructured Data, Multigrid Method.

2 Introduction

The reach of AI today is almost immeasurable. The impact of AI is currently being felt in virtually all aspects of life, from pattern recognition, object detection, security analyses etc [1]. A very important aspect of AI is Machine Learning (ML) [2].

ML can be said to be responsible for most of the advancements that AI has made over the last decade [1] [2]. The bedrock of ML is the convolutional neural network (CNN), also known as convnets [3]. CNNs have done remarkably in the areas of computer vision, natural language processing [4], image classification [5], as well as fluid flow [6] and its motions [7].

Conventionally, CNNs are only applied to structured data. However, more recently, CNNs have been applied to unstructured data through the use of space filling curves (SFC) and graph-based methods. [3] uses SFCs to transform unstructured 2D data into 1D data, after which a 1D convolutional autoencoder (CAE) is successfully trained using the transformed data. This approach showed astonishing results. Graph-based methods have also been used to apply CNN on unstructured graph data. Because of the increasing number of applications where data are represented as graphs with complex relationships between objects, several graph-based methods have been developed. They include graph neural networks [8], recurrent graph neural networks, convolutional graph neural networks [9] [10], graph autoencoders, spatial-temporal graph neural networks [9] and graph attention networks [10].

All of the above are possible because of the availability of several open-source ML libraries aimed at simplifying the production of ML algorithms and methods. Some of these libraries include PyTorch [11], TensorFlow [12] and scikit-learn [13]. These libraries are built on the Python scripting language, allowing instant transferability of code between different hardware architectures, such as a central processing units (CPUs), graphics processing units (GPUs) and AI computers. The recent development of AI computers capable of delivering the wall-clock compute performance of many tens to hundreds of GPUs eg. the CS-2 developed by Cerebras, has spurred the prospect of using ML algorithms in many industries, especially in CFD, very tantalizing.

Currently, CFD models do not enjoy the benefits of using a scripting language, such as Python. Only a handful of expert developers that understand the ins and outs of fluid dynamics can adequately develop a CFD model. This is a big drawback, as it also hinders parallel scalability between developers. Also, due to the continuous change in technology, most of these codes have to continually be rewritten

to fit different hardware architectures, using some low-level languages such as FORTRAN/C/C++. In addition to being complex, CFD codes are also highly computational expensive, using up a lot of energy, time and resources.

By combining the use of AI, for the numerous benefits mentioned prior, with traditional methods of numerical modelling, this project aims to address the problems of complexity and computational cost associated with developing CFD models. This is to be achieved by increasing accessibility to develop CFD models; through the use of the Python scripting language, simplifying this development; through CNNs, while reducing computational cost and energy usage; through AI computers. Tackling these problems will therefore enable the relatively simple development of digital twins, as it makes use of the optimisation engine and sensitivities already embedded in AI software. These digital twins can thereafter be used to optimise systems, form error measures, assimilate data and quantify uncertainty in a relatively straight-forward manner.

To demonstrate the possibility of developing CFD models using AI libraries, this project aims at solving the scalar transport equation on unstructured meshes. Firstly, we implement a graph neural network (GNN), and then a 1D geometric multigrid solver. The GNN comprises of a neural network capable of taking in graph data as its input. SFCs are used in the multigrid solver to transform 2D data to 1D data. One benefit of this transformation is that 1D data are computationally less expensive to operate on than 2D data (as demonstrated in section 2. The multigrid solver also utilizes CNNs to carry out the processes of restriction and prolongation (more details in section 3.2.5).

The GNN is seen to robustly solve the 2D advection-diffusion equation (ADE), as well as the pure diffusion equation. An initial condition of a Gaussian distribution, coupled with a square distribution, are both seen to diagonally advect through the unstructured mesh in the same direction, while also experiencing a bit of diffusion. The geometric multigrid solver also successfully solves the pure diffusion equation. An initial rectangular distribution is seen to diffuse with increasing multigrid iterations, until it converges. Comparing the computational time taken for both methods to converge, while solving the same diffusion problem, show the multigrid solver to be about 50 times faster.

3 Methodology

This project consists of two approaches to solve the scalar transport equation (see equation 1) on unstructured meshes; the use of a GNN (see section 3.1) and a geometric multigrid solver (see section 3.2), using PyTorch.

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} + \sigma T + \kappa \nabla^2 T = s, \tag{1}$$

where T is a scalar concentration field, t is the time ($seconds$), u and v are the velocities (ms^{-1}) in the x and y directions respectively, σ is the absorption term, κ is the diffusivity (m^2s^{-1}) and s is the source.

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} + \kappa \nabla^2 T = 0, \tag{2}$$

Although there are other AI libraries which can be used, such as TensorFlow and Keras [14], PyTorch is used because of the following reasons:

- **Ease**: PyTorch is easy to use and models can be implemented in a more straightforward manner,
- **Flexibility**: Some CNNs used in this thesis, especially in the geometric multigrid method for restriction, prolongation and Jacobi iteration, can be used for data with different sizes. This

is unlike TensorFlow that requires different CNNs for different data sizes. This property of PyTorch models greatly increases readability, while avoiding long lines of code and complexity.

This thesis is an extension of the work done by Chen Boyang. Through the discretization of the ADE (see equation 2) to determine the weights for a CNN filter, he uses the aforementioned CNN to solve the equation on structured data, and shows the results through time. He specifically uses the upwind differencing scheme for advection and the central differencing scheme for diffusion, It is also an extension of the work done by Christopher Pain. He produces the $shape\_functions$ and $space\_filling\_curves$ modules used in this thesis. In the $shape\_functions$ module,

- $get\_fe\_matrix\_eqn$ is used to discretize, using finite element method (FEM), the unstructured mesh and get the FEM A matrix and b vector that solve the equation: $Ax = b$, where x is the unknown scalar field. This method also produces the lumped mass matrix, M_L, of the A matrix.
- $best\_sfc\_mapping\_to\_sfc\_matrix\_n$ is used to map A, b and M_L produced by the $get\_fe\_matrix\_eqn$ to SFC ordering.

The $space\_filling\_curves$ module is used to get the SFC ordering of the 2D unstructured mesh, using the SFCs seen in figure 3b and c.

3.1 Graph Neural Network

CNNs are used to transverse n-dimensional data, containing 'cells' or 'nodes', using another but smaller n-dimensional array called a kernel/filter, containing values called weights. In a typical CNN, the filter size and weights remain the same throughout the data. This is particularly true for structured data. However, for unstructured data, this is not the case. In unstructured data, which are usually produced from the finite element (FE) discretization of a partial differential equation (PDE), the self weights, neighboring weights, as well as the number of neighbors may differ, as you move from one node to another. Because of this property of unstructured data, a typical CNN cannot be employed. This is where the concept of graph data comes in.

Graph data are unstructured data which have differing or complex connections or relationships between the nodes [15]. Merging the concept of CNNs with graph data leads to the development of a GNN. With a simple GNN, we ensure that as we transverse through the n-dimensional data, the filter weights, as well as its size, can change. The GNN employed in this thesis requires no training, and therefore requires no geometric convolutional layers from the PyTorch Geometric [16] library. However, it does require the PyTorch $Data$ object.

The graph data utilized in this method is assumed to have the following properties:

1. Each node is connected to its neighbors through its edges. These connections are stored in the $edge\_index$ list of the PyTorch $Data$ object.
2. Every node is connected to itself.
3. Weights associated with edge connections, i.e. edge weights, are stored in the $edge\_attributes$ list of the PyTorch $Data$ object.
4. The scalar field is stored in the x list of the PyTorch $Data$ object. It is worthy to note that although it is a 2D data, because it is an unstructured mesh, the scalar field in each node is stored as a 1D array/list in an ordered, consecutive manner.

3.1.1 GNN Architecture

To account for the differing weights of the GNN leads to the development of an *adjacency_matrix* [17]. With an n number of nodes, an adjacency matrix is a square matrix of size $n \times n$ that clearly shows

the connections between nodes, by identifying each row as a node and each column as a connection. For example, this matrix: $\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$ represents a graph data where the 1's represent a connection. From that matrix, it can be deduced that node 1 is connected to node 2 and itself, node 2 is connected to nodes 1, 3 and itself, and so on. This adjacency matrix forms the weights matrix where the non-zeros (nnzs) are replaced with the actual weights, gotten from the FEM discretization of the problem. Figure 1 shows a depiction of the workings of the GNN.

3.1.2 Mesh Weights

The $get\_fe\_matrix\_eqn$ method from the $shape\_functions$ module uses the finite element method (FEM) [18] to discretize the unstructured mesh to produce the matrix A and vector b that solve the equation $Ax = b$, where x is the scalar field and is not needed. This method also produces the lumped mass matrix M_L of the matrix A. The matrix A is stored as a sparse matrix, specifically using the compressed row storage (CSR) [19] method, where:

- A contains the values of the matrix,
- the column indices of the values in A are stored in a variable called $cola$. The length of $cola$ is the number of nnzs in the A matrix, and
- the locations in A that start a row are stored in a variable called $fina$. The length of $fina$ is $nnzs + 1$.

Both the $cola$ and $fina$ variables are used as inputs to the $get\_fe\_matrix\_eqn$ method.

The weights of the GNN are then calculated, using A and M_L in the following way:

$$M_L \frac{T^{n+1} - T^n}{\Delta t} + AT^n = 0, \tag{3}$$

$$M_L(T^{n+1} - T^n) + AT^n \Delta t = 0, \tag{4}$$

$$M_L T^{n+1} - M_L T^n + AT^n \Delta t = 0, \tag{5}$$

$$T^{n+1} = M_L^{-1}(M_L - A\Delta t) \times T^n, \tag{6}$$

where T^n is the scalar field at the current time-step, T^{n+1} is the scalar field at the next time-step, A is the same FEM matrix and M_L is the lumped mass matrix, both gotten from the $shape_functions$ module.

As seen in item 4 of the list above, the scalar field of the mesh is stored as a list or tensor, while the weights are stored as a matrix. Thus, to time-step, a matrix-vector multiplication is initiated (see figure 1). To enforce a Dirichlet boundary condition of 0, after every time-step, the values in the boundary nodes are equated to 0.

3.1.3 Testing

The GNN is tested on a 128x128 structured quadrilateral mesh, as well as a 64x64 unstructured triangular mesh, both seen in figure 4a and figure 5a, respectively. The meshes are generated using the GMSH [20] software.

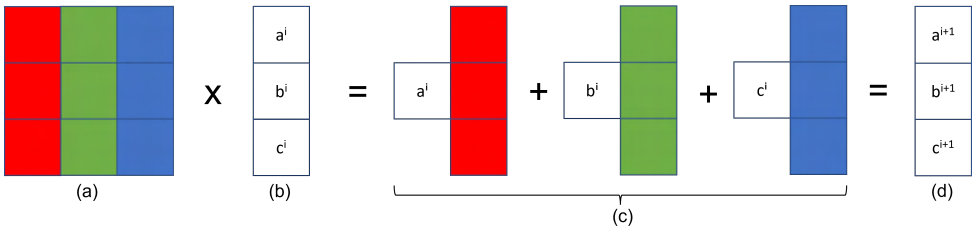

Figure 1: Simple architecture of the GNN, where (a) is the adjacency matrix containing weights that multiplies the (b) scalar field at the current time step to give (c) the respective multiplications and additions and finally (d) the scalar field at the next time step

Gaussian and square distributions were investigated on the structured mesh, while a somewhat rectangular distribution was investigated on the unstructured mesh, using the following variables:

- uniform velocity in the x and y directions, $(u, v) = (1, 1)ms^{-}1$
- diffusivity, $\kappa = 0.2m^2s^{-1}$
- source, $s = 0$
- absorption term $\sigma = 0$
- change in x and y, $dx = dy = 1$
- time-step, $dt = 0.1$

with the scalar transport equation seen in equation 1.

The Gaussian distribution is given by:

$$T^0(x, y) = exp\left(-\left(\frac{(x - x_0)^2}{2\gamma^2} + \frac{(y - y_0)^2}{2\gamma^2}\right)\right), \tag{7}$$

and the square distribution by:

$$T^0(x, y) = \begin{cases} 1, & x \in [75m, 95m], y \in [70m, 90m] \\ 0, & \text{otherwise} \end{cases}$$

where T_0 represents the initial condition of the scalar field T, and the coefficient representing the width of the distribution is $\gamma = 40$.

3.2 Multigrid Solver using Space Filling Curves

Multigrid solvers have been widely employed in FE and CFD problems [21] [22] [23], because of its advantage of defining low frequency errors as high frequency on coarser grids [24] (see appendix A: figure A.1). Although multigrid solvers can be employed using 2D data, we utilize 1D data in this thesis because 1D data is computational cheaper to compute.

We transform our 2D unstructured data to 1D using Space-Filling Curves (SFCs). SFCs have a magic property of being able to transverse every node in a multidimensional mesh, visiting each node/cell just once and ensuring nodes that are close together in the multidimensional mesh are also close together in the curve [25] [26]. SFcs are, in summary, used to map multidimensional data (eg. 2D) to 1D. An example of using SFCs to map unstructured data to 1D is shown in figure 2.

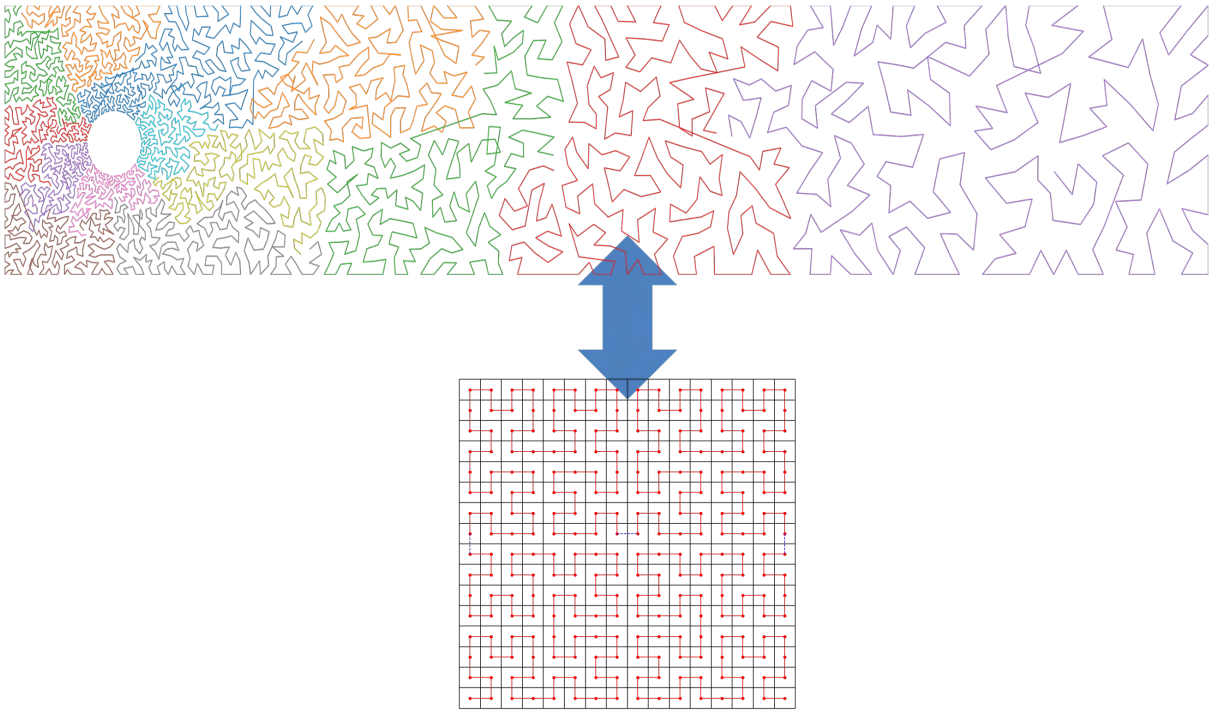

Figure 2: The space filling curve (SFC) ordering of nodes in an unstructured mesh for flow past a cylinder. Top plot shows the lines drawn between nodes with consecutive ordering with the SFC ordering gotten from the SFC in the bottom plot

3.2.1 Finite Element Discretization

The same ADE shown in equation 2 is solved using this solver. This multigrid approach first involves the FE discretization of the 2D data to generate the FE matrix, A and bias vector, b, that solve the $Ax = b$ equation, where the unknown scalar field, x, is not needed. The FE discretization also produces the lumped mass matrix, M_L.

To solve the equation using zero boundary conditions, the contributions from the boundary nodes are excluded from the matrix A, vectors b and M_L. This is also to ensure that the boundary nodes are not considered when putting the 2D mesh in SFC ordering. These boundary nodes are also excluded from the initial condition.

3.2.2 Mapping to Space Filling Curves

After removing the boundary nodes, the resulting mesh, and its corresponding A, b and M_L values, are then mapped to two different 1D orderings using two orthogonal SFCs, shown in figure 3b and c. The use of two orthogonal SFCs have been shown to give great results when applied to CFD problems [3]. The mapping of the 2D mesh uses an n-Point stencil along the SFC trajectory. For example, in a 3-Point (3P) stencil, for a given node in position *k* along the SFC trajectory and multigrid iteration *i*, $node_k^i$, the nodal values from its left and right neighbors, $node_{k-1}^i$ and $node_{k+1}^i$, are used to calculate the nodal value in $node_k^{i+1}$. Increasing the stencil, eg. to a 9P, is expected to give more accurate results. This increase in accuracy is attributed to the collection of more neighborhood information of the concerned node.

3.2.3 Agglomeration of Reordered 1D Data

As stated in [23], there are two ways coarse grids can be formed from an FE mesh; algebraic and geometric. In this thesis, we use geometric agglomeration to generate coarse grids from the 1D

reordered finer data to the coarsest grid of size 1. The reordered data is agglomerated to size 1 to aid convergence. The *best_sfc_mapping_to_sfc_matrix_n* method of the *shape_functions* module successfully does this. The method produces the A matrix (in banded sparse format), the b and M_L vectors of each coarse grid. It stores these in the same variable, where the first *n* values are for the finest grid of length *n*, the next *m* values are for the second finest grid of length *m*, and so on.

The *best_sfc_mapping_to_sfc_matrix_n* method uses two important variables: **relax_keep_off** and **nfilt_size_sfc**. With the latter determining the size of the stencil, e.g. 3P or 9P, the latter determines how much of a node's neighbors, outside this stencil, is added to that node's self weight. For more contrast, the higher the *nfilt_size_sfc*, the lower the *relax_keep_off* should be.

3.2.4 Scaling the FEM Matrices

The FEM A matrices and b vectors generated by the *shape_functions* module for the original 2D mesh and the agglomerated grids are all scaled using their corresponding M_L values. This is done to also aid convergence of the solver.

3.2.5 Employing Convolutional Neural Networks

In this multigrid solver, CNNs are employed in three ways:

- Restriction
- Prolongation
- Jacobi Matrices

Restriction is the process of interpolating the error from a finer grid to a coarser one, while prolongation is the opposite of restriction (see appendix A: figure A.3 and figure 3).

CNNs are used because of its simplicity. Numerous and bogus lines of CFD code are easily replaced using CNNs. CNNs also have the ability to automatically apply requirements such as the constant multiplicator (filter/kernel values), the size of the multiplicator (kernel size), and other requirements, such as a stride and a padding (if required). The properties of the restriction and prolongation CNNs used in this solver are shown in table 1.

CNN		Parameters	Settings
Restrictor	Grid agglomeration	Kernel size	2
		Filter	0.5
		Padding	0
		Stride	2
Prolongator	With an intended size of j and current size of i	Scale factor	j/i

Table 1: CNN Parameters for the Restrictor and Prolongator CNNs

In order to apply the restriction CNN to multigrid layers containing grids with an odd number of cells, that grid is padded with 0 of size 1, on the right end of the grid.

For the case of the Jacobi matrices, the CNN model is used in a similar way as the GNN, where just the weights, the curve to be considered (0 for first curve and 1 for second curve), and the current scalar field are required inputs. Convolutional layers or typical convolutional variables, such as filter, padding etc., do not need to be passed or specified in the model. In unstructured meshes, the weights associated with each node and its neighbors vary. These weights are the A matrices, for each multigrid layer, after scaling. The model therefore, uses the desired SFC, the scaled A matrices and corresponding corrections, $x_(k)$ for each multigrid layer to get $x_(k+1)$ and D_1 as seen in equation 10.

3.2.6 The Multigrid Solver

Finally, to the multigrid solver. A rough schematic of the solver is depicted in figure 3a alongside the SFCs alternatively used in the solver shown in figure 3b and c.

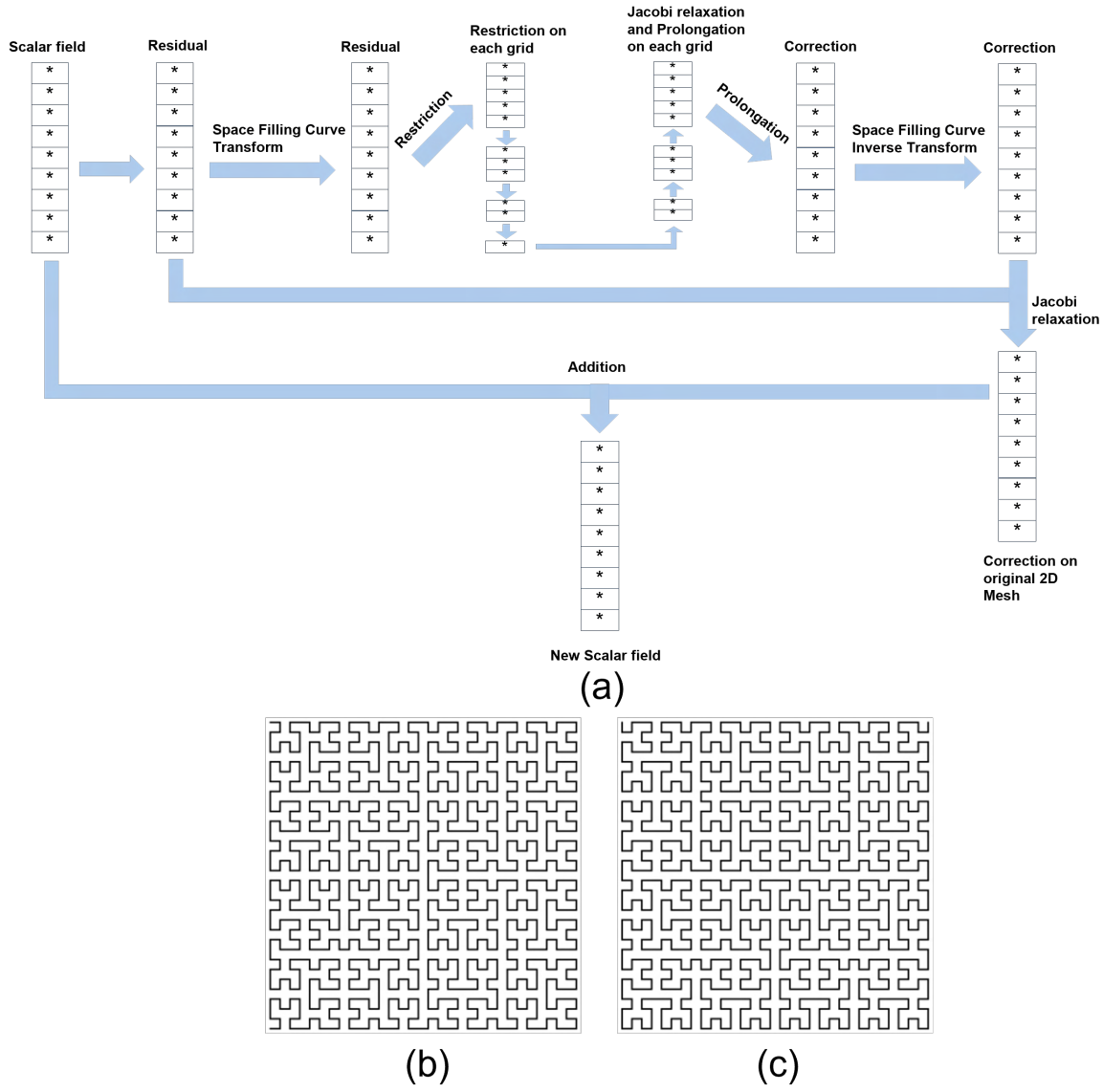

Figure 3: (a) Rough schematic of the geometric multigrid solver showing the use of restriction, prolongation, the Jacobi relaxation method and the Space Filling Curves in alternating manner seen in (b) and (c)

For every multigrid iteration, the solver involves:

- Initializing the correction to 0,
- The consecutive use of two SFCs,
- Calculating the residual based on the original 2D mesh, using equation 8,
- Reordering the residual based on the current SFC,
- Interpolating the residual down through the multigrid layers, as seen in appendix A: figure A.3a and figure 3a,

- Performing a Jacobi relaxation on the correction, using equation 12, and prolongation starting from the coarsest grid to the second finest grid, as seen in appendix A: figure A.3b and figure 3a,
- Reordering the correction and the residual on finest grid back to its original order,
- Performing a Jacobi relaxation on the original 2D mesh using the correction and residual in original order,
- Adding the correction to the current state of the system/scalar field.

From the FEM discretization, in section 3.2.1, which solves the equation $Ax = b$, the residual for the 2D mesh can thus be calculated in the following way:

$$r = b - Ax, \tag{8}$$

As stated earlier, grid corrections are computed using the Jacobi iteration method [27] (i.e smoothing), seen in equation 10. A relaxation coefficient is included in the Jacobi iteration method (see equation 12) to form a Jacobi relaxation method. This is also done to aid convergence of the solver.

Jacobi iteration solves the square system of *n* linear equations, $Ax = b$, where A contains the coefficients of the unknowns x and b is the solution vector, in the following way:

$$x^{(k+1)} = D^{-1}(b - (L + U)x^{(k)}, \tag{9}$$

Assuming,

$$A = D + L + U$$

equation 9 becomes:

$$x^{(k+1)} = x^{(k)} - AD^{-1}x^{(k)} + D^{-1}b, \tag{10}$$

where $x^{(k+1)}$ is the next solution of the equation, x^k is the current solution, D, L and U are the diagonal, lower triangular matrix and upper triangular matrix of the matrix A respectively. Using a relaxation in the Jacobi iteration, equation 9 becomes:

$$x^{(k+1)} = \alpha D^{-1}(b - (L + U)x^{(k)} + (1 - \alpha)x^{(k)}, \tag{11}$$

which can be simplified to:

$$x^{(k+1)} = x^{(k)} - \alpha AD^{-1}x^{(k)} + \alpha D^{-1}b, \tag{12}$$

3.2.7 Testing

The multigrid solver is used to solve the pure diffusion equation (see equation 13) on a 64x64 unstructured mesh with 4095 triangular FE cells. The pure diffusion equation is gotten from equation 1), by equating $u = v = 0$, and $\sigma = 0$, to give:

$$\frac{\partial T}{\partial t} + \kappa \nabla^2 T = s, \tag{13}$$

The unstructured mesh is generated using the GMSH software and can be seen in figure 5a. The diffusion equation is solved with the following parameters:

- diffusivity, $\kappa = 0.2$
- relaxation, $\alpha = 1/3$
- amount of non-neighboring values to add to self-weight/diagonal, $relax\_keep\_off = 0.5$
- filter size, $nfilt\_size\_sfc = 3$

,.

4 Code Metadata

The code associated with this thesis is predominantly written using Python, PyTorch and FORTRAN, with its current code version number as $v1.0.0$.

With smaller problems, such as problems involving 32 x 32 or 64 x 64 2D meshes, a Jupyter Notebook [28] can adequately be used to run the associated thesis code. However, with bigger problems, such as a 128 x 128 2D mesh, more memory is required. Thus, a Google Colaboratory notebook [29] can suffice. For laptops running on Windows Operating System (OS), the Windows Subsystem for Linux (WSL) [30] is advised, so as to effortlessly compile and run the FORTRAN scripts vital for this thesis.

Some of the code dependencies of this project and their versions are:

- Python [31] $==$ 3.8.10
- PyTorch [11] $==$ 1.12.0+cu102
- PyTorch Geometric [16] $==$ 2.0.4
- PyTorch Scatter $==$ 2.0.9
- PyTorch Sparse $==$ 0.6.14
- Jupyter notebook [28] $==$ 6.4.12
- NumPy [32] $==$ 1.23.1
- toughio [33] $==$ 1.11.0
- PyVista [34] $==$ 0.36.1
- matplotlib [35] $==$ 3.5.2
- GMSH [20] $==$ 4.10.5
- math
- time
- shape_functions
- space_filling_curves

All dependencies can be found in the 'Requirements.txt' file in the GitHub repository containing this thesis's associated code, found here: GitHub Repoaitory: Solving PDEs with AI librariess.

The *shape_functions.f90* and *space_filling_decomp_new.f90* FORTRAN files are crucial to successfully run the code associated with this thesis and can be compiled with the following command:

$$python3 \ -m \ numpy.f2py \ -c \ x.f90 \ -m \ u2r$$

replacing $x.f90$ with the appropriate path to the required files, e.g $'shape\_functions.f90'$ and $u2r$ with desired module name, e.g $'shape_functions'$, and $python3$ with $python2$ if using Python version 2.

Basic hardware requirements to run the associated thesis code include:

- A laptop with a Windows 64-bit operating system, x64-based processor with 8.0GB RAM

5 Results

5.1 Graph Neural Network

The GNN implemented here was tested on structured data with quadrilateral cells and unstructured data with triangular cells (refer to section 3.1.3). The GNN showed to successfully solve the 2D ADE, as seen in equation 2 and the pure diffusion equation (see equation 13). Results from solving the pure diffusion equation is compared with that from the multigrid method and can be seen in table 2.

Figure 4 shows the structured mesh and the corresponding results gotten after every 200 time steps. With the equation variables mentioned in section 3.1, the initial Gaussian and square concentrations are both seen to diagonally advect with a uniform velocity, while also diffusing.

For a pure advection case, the GNN suffered from Gibb's oscillations. Gibb's oscillations are numerical errors produced from a central differencing scheme applied on advection terms. However, with the introduction of sufficient diffusion, the oscillations experienced by the mesh were significantly reduced (also see figure 4).

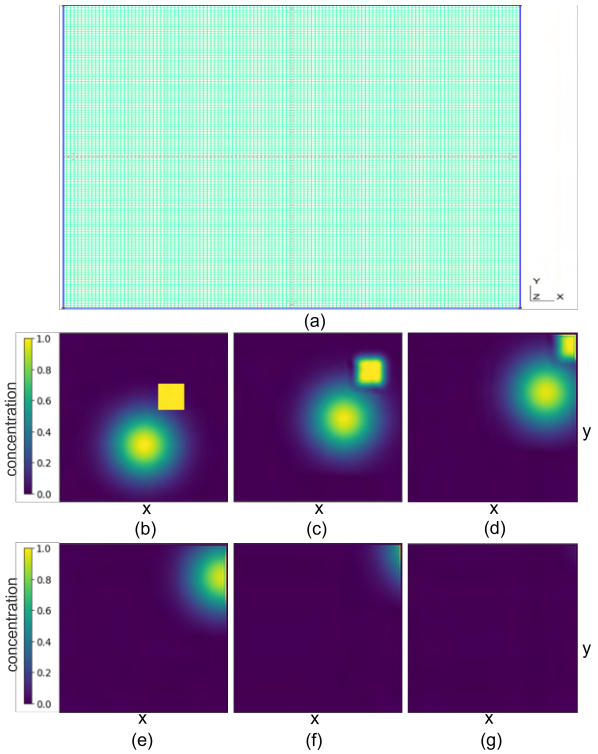

Figure 4: Advection-diffusion results on (a) a 128x128 structured mesh with results shown after every 200 timesteps in (b) (c) (d) (e) (f) (g)

With the same variables mentioned in section 3.1.3, the 2D ADE was solved on a 64x64 unstructured triangular mesh. The initial rectangular concentration is also seen to successfully advect diagonally while significantly diffusing over time. The results can be seen in figure 5.

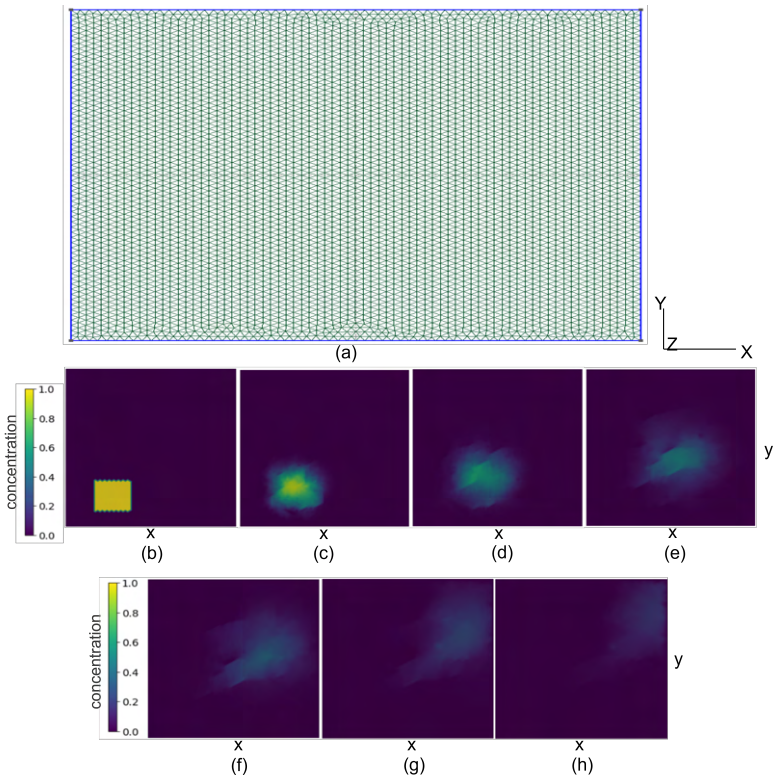

Figure 5: (a) The 64x64 unstructured triangular mesh used to solve the advection-diffusion equation from (b) the initial condition, and its state after (c) 200 time steps, (d) (e) (f) (g) (h) 500 time steps each

5.2 Multigrid Solver

Using variables mentioned in section 3.2.7, the multigrid solver was tested on structured and unstructured meshes.

The solver was successful in solving the pure diffusion equation, as seen in equation 13.

The solver was however unable to successfully solve the ADE. This can be attributed to the following reasons:

- The presence of zeros (0s) in the diagonal of the A matrices produced for each grid of the agglomerated mesh,
- The use of the Jacobi iteration to compute the corrections on each grid.

5.2.1 Multigrid Solver on Structured Mesh

Testing the solver on the same 128x128 structured quadrilateral mesh (seen in figure 4a), an initial Gaussian scalar field is seen to diffuse over time, as seen in figure 7.

5.2.2 Multigrid Solver on Unstructured Mesh

Testing the solver on the same 64x64 unstructured triangular mesh (seen in figure 5a), an initial somewhat rectangular scalar field is seen to diffuse over time, as seen in figure 7.

With a $relax\_keep\_off$ value of 0.5, there is a compromise between stability and accuracy. If 100%, i.e. $relax\_keep\_off = 1$, of a node's neighbors from the original mesh are considered, i.e. added to its self weight, there is a higher accuracy, while risking stability. However, without any

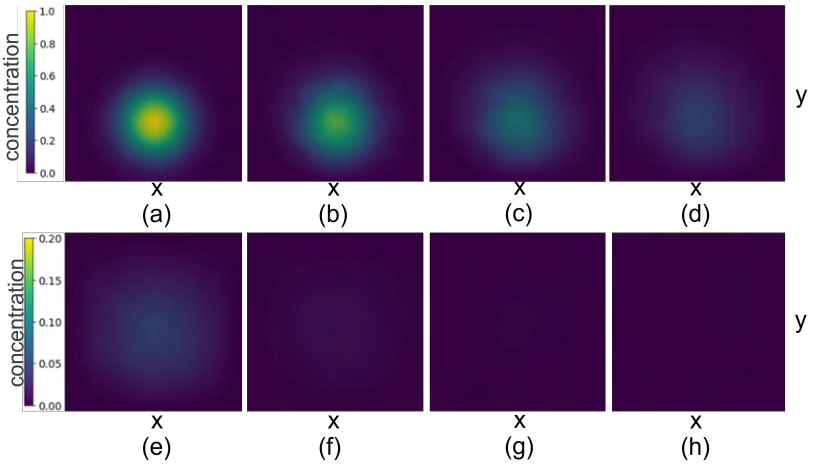

Figure 6: Solving the diffusion equation on a 128x128 structured rectangular mesh using a $relax\_keep\_off$ value of 0.5 with (a) initial condition, and results after (b) 5, (c) 15, (d) 30 multigrid iterations on a concentration interval between $0 - 1$, (e) 80, (f) 130, (g) 180 and (h) 230 multigrid iterations on a concentration interval between $0 - 0.2$, where the system state reaches a precision of 10^{-4}

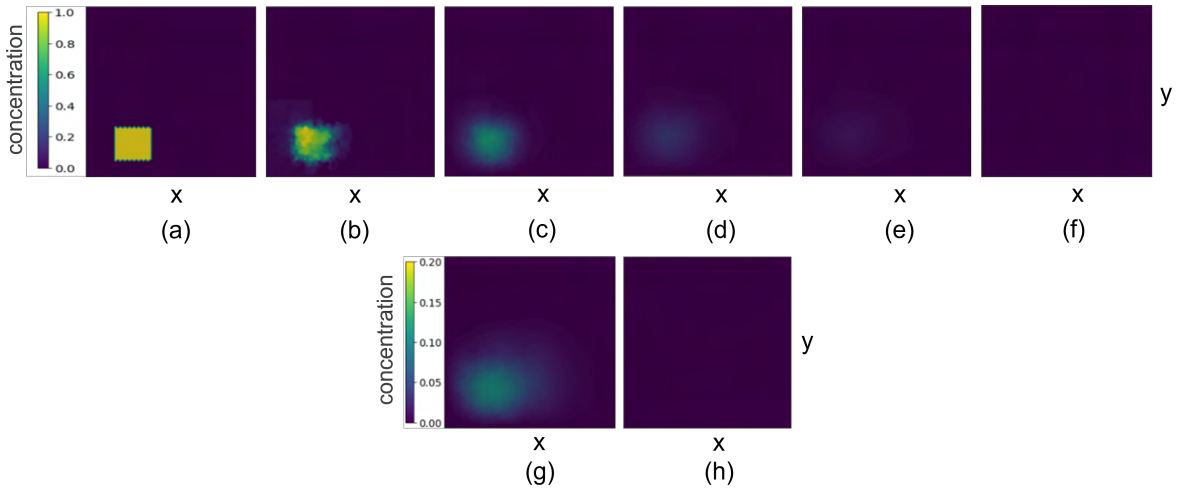

Figure 7: Solving the diffusion equation on a 64x64 unstructured triangular mesh using a $relax\_keep\_off$ value of 0.5 with (a) initial condition, and results after (b) one (1) multigrid iteration (c) five (5), (d) ten (10), (e) twenty (20) and (f) eighty (80) multigrid iterations, where the solution reaches a precision of 10^{-4}. Figures (g) and (h) show the result after 20 and 80 multigrid iterations, using a scale between 0-0.2 to clearly show the difference in the system state

addition from neighbors outside of the 3P stencil used, the solver is more stable but less accurate. With a $relax\_keep\_off = 1$ and just ten (10) multigrid iterations, the solution is already seen to diverge. However, with a $relax\_keep\_off = 0$, as seen in appendix A: figure A.2, the system continues to converge to the solution. However, with the same number of multigrid iterations as a $relax\_keep\_off = 0.5$, its precision is reduced by a factor of 10.

Figure 8 shows the performance of the geometric multigrid method with different problem sizes: 32 $\times$ 32, 64 $\times$ 64 and 128 $\times$ 128 2D meshes. These results were gotten whilst solving a pure diffusion problem, with the following parameters:

- amount of non-neighboring values to add to self-weight/diagonal, $relax\_keep\_off = 0$

- filter size, $nfilt\_size\_sfc = 129$
- relaxation, $alpha = 1$
- diffusivity, $\kappa = 1ms$
- source, $s = 0$, with a strength of unity in the middle of the domain

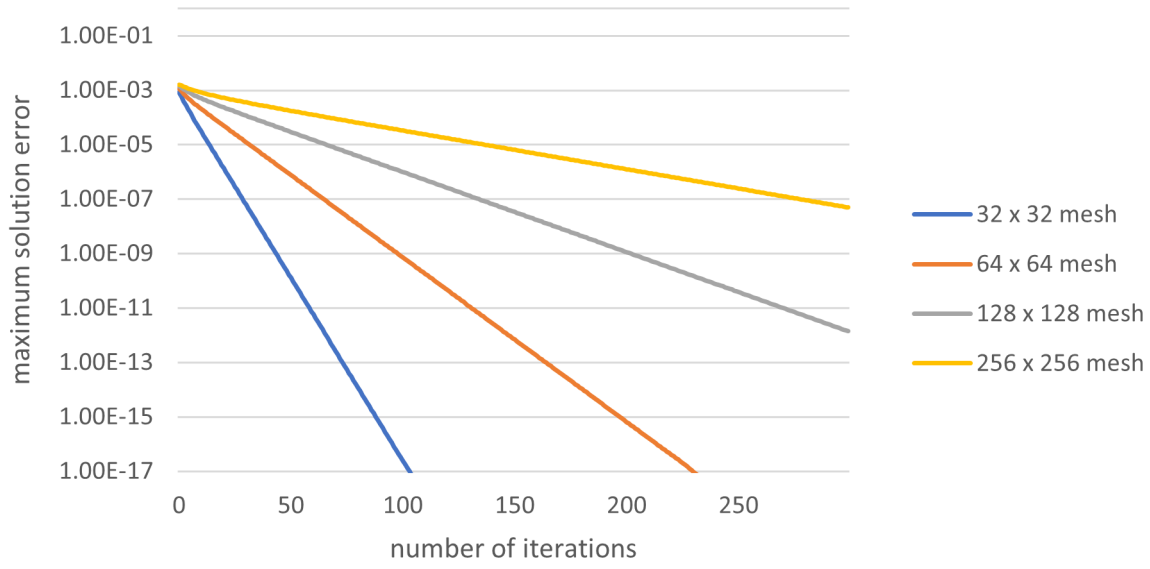

Figure 8: Graph showing the maximum error in each iteration for different problem sizes: 32 x 32, 64 x 64 and 128 x 128 2D meshes

5.3 Comparison between the Graph Neural Network and Multigrid Solver

From prior results, it can be seen that the GNN might have a bit of superiority over the multigrid solver, due to the fact that it is capable of successfully solving the ADE.

However, when comparing the computational time of both methods on a pure diffusion case, the multigrid solver takes the upper hand. Table 2 shows the performance of the GNN and the multigrid solver on different sizes of unstructured triangular meshes.

S/N	Grid Size	Number of nodes	Time in seconds (s) to reach precision of 10^-3	
			Methods	
			Graph Neural Network	Multigrid Solver
1.	128x128	16641	>100000	2045
2.	64x64	4095	1534	31.2
3.	32x32	1296	199	4.42

Table 2: Comparison of Computational Time between the Graph Neural Network and Multigrid Solver on different Mesh sizes

6 Discussions and Conclusions

6.1 Discussions

The production of correct A matrix, b and ml stalled correct testing of the GNN model on structured and unstructured meshes. However, with continuous debugging, these issues were resolved and

implementation proceeded seamlessly.

The multigrid solver was seen to be more technical to implement than the GNN, due to reasons such as: excluding boundary nodes from SFC generation and reordering nodes using the SFCs and back to its original order after every multigrid iteration. However, this extra complexity is not in vain, as the multigrid solver can be seen to do excellently in terms of speed and convergence of both larger and smaller problems (see table 2). be preferred for much larger problems. From figure 8, it can be seen that the number of iterations to meet the same degree of accuracy is not much for the 64×64 and 128×128 2D meshes.

Both the GNN and multigrid methods struggled to solve a pure advection problem, with the GNN however, solving the 2D advection-diffusion problem. This is attributed to the GNN suffering from Gibb's oscillations for a pure advection case, and the multigrid solver having zeros in the diagonals of its agglomerated meshes.

One of the strengths of this solution is the use of PyTorch. PyTorch uses the Python scripting language, enabling this code and potential CFD codes using this approach to be run on any CPU, GPU and AI computer, without the need for the codes to be rewritten. The use of an AI library to solve CFD problems involving unstructured meshes has never been done before. This, therefore, promises to be a game-changer in the world of CFD modelling.

The code associated with this thesis was not tested on real-life, large CFD problems. The efficiency and computational costs associated with using this code on a real-life CFD problem, can therefore not be determined. With this in mind, it is advised that to move this thesis further, solving real-life CFD problems with this thesis's associated code should be carried out.

6.2 Conclusions

From the results, the graph neural network (GNN) and the geometric multigrid solver show great promise in replacing some demanding and complex steps associated with writing CFD code, mostly through the use of custom convolutional neural networks (CNN). With this, we can ensure that future CFD codes do not have to be rewritten to suit different hardware architectures, thereby increasing simplicity, reducing complexity and ensuring that CFD codes can exploit the power of 'AI' computers.

We show this by solving the 2D advection-diffusion equation and/or diffusion equation on unstructured meshes using both methods. Although the GNN is way slower than the multigrid solver to converge, it can adequately be used to solve CFD problems involving scalar transport. We also show that it is beneficial to use two space filling curves and a larger filter size to help increase the accuracy and the convergence of the multigrid method.

Finally, the production of digital twins needed for various modelling purposes can be made simpler, increasing productivity and efficiency of teams.

References

[1] Devanshi Dhall, Ravinder Kaur, and Mamta Juneja. Machine learning: a review of the algorithms and its applications. *Proceedings of ICRIC 2019*, pages 47–63, 2020.

[2] Darrell M West and John R Allen. How artificial intelligence is transforming the world. *Report. April*, 24:2018, 2018.

[3] Claire E Heaney, Yuling Li, Omar K Matar, and Christopher C Pain. Applying convolutional neural networks to data on unstructured meshes with space-filling curves. *arXiv preprint arXiv:2011.14820*, 2020.

[4] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.

[5] Mahbub Hussain, Jordan J Bird, and Diego R Faria. A study on cnn transfer learning for image classification. In *UK Workshop on computational Intelligence*, pages 191–202. Springer, 2018.

[6] Francisco J Gonzalez and Maciej Balajewicz. Deep convolutional recurrent autoencoders for learning low-dimensional feature dynamics of fluid systems. *arXiv preprint arXiv:1808.01346*, 2018.

[7] Kookjin Lee and Kevin T Carlberg. Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders. *Journal of Computational Physics*, 404:108973, 2020.

[8] Francis Ogoke, Kazem Meidani, Amirreza Hashemi, and Amir Barati Farimani. Graph convolutional networks applied to unstructured flow field data. *Machine Learning: Science and Technology*, 2(4):045020, 2021.

[9] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[10] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[13] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[14] Nikhil Ketkar. Introduction to keras. In *Deep learning with Python*, pages 97–111. Springer, 2017.

[15] Renzo Angles and Claudio Gutierrez. An introduction to graph data management. In *Graph Data Management*, pages 1–32. Springer, 2018.

[16] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[17] Eric W Weisstein. Adjacency matrix. *https://mathworld.wolfram.com/*, 2007.

[18] Olek C Zienkiewicz, Robert Leroy Taylor, and Jian Z Zhu. *The finite element method: its basis and fundamentals*. Elsevier, 2005.

[19] Fethulah Smailbegovic, Georgi Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format, 01 2005.

[20] Christophe Geuzaine and J-F Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre-and post-processing facilities, 2008.

[21] Nils Margenberg, Dirk Hartmann, Christian Lessig, and Thomas Richter. A neural network multigrid solver for the navier-stokes equations. *Journal of Computational Physics*, 460:110983, 2022.

[22] Mark Adams and James W Demmel. Parallel multigrid solver for 3d unstructured finite element problems. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 27–es, 1999.

[23] Pieter Wesseling and Cornelis W Oosterlee. Geometric multigrid with applications to computational fluid dynamics. *Journal of computational and applied mathematics*, 128(1-2):311–334, 2001.

[24] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[25] Mohamed F Mokbel and Walid G Aref. Space-filling curves, 2009.

[26] Daniela Lera, Mikhail Posypkin, and Yaroslav D Sergeyev. Space-filling curves for numerical approximation and visualization of solutions to systems of nonlinear inequalities with applications in robotics. *Applied Mathematics and Computation*, 390:125660, 2021.

[27] Eric W Weisstein. Jacobi method. *https://mathworld. wolfram. com/*, 2002.

[28] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows., 2016.

[29] Praveen Gujjar and Naveen Kumar. Google colaboratory: Tool for deep learning and machine learning applications. *Indian Journal of Computer Science*, 6(3-4):23–26, 2021.

[30] Hayden Barnes. *Pro Windows Subsystem for Linux (WSL)*. Springer, 2021.

[31] Guido Van Rossum and Fred L Drake. *Python 3 reference manual*. CreateSpace, 2009.

[32] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[33] Keurfon Luu. toughio: Pre-and post-processing python library for tough. *Journal of Open Source Software*, 5(51):2412, 2020.

[34] C. Bane Sullivan and Alexander Kaszynski. PyVista: 3d plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK). *Journal of Open Source Software*, 4(37):1450, may 2019.

[35] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95, 2007.

Appendices

A

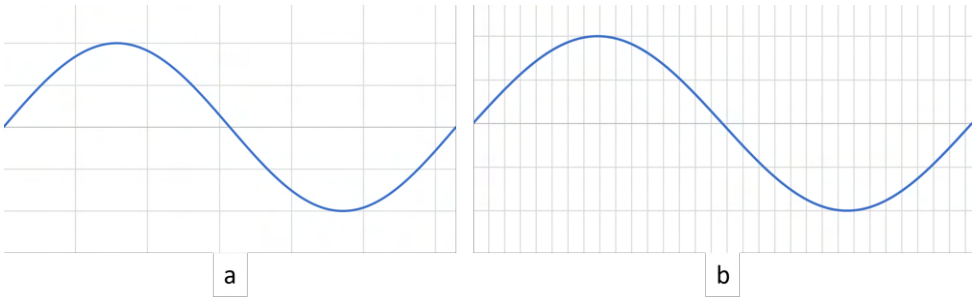

Figure A.1: Error as seen on (a) a coarser grid versus (b) a finer grid. It can be seen that the same error in (a) is assumed to be much bigger as opposed to when it is in (b)

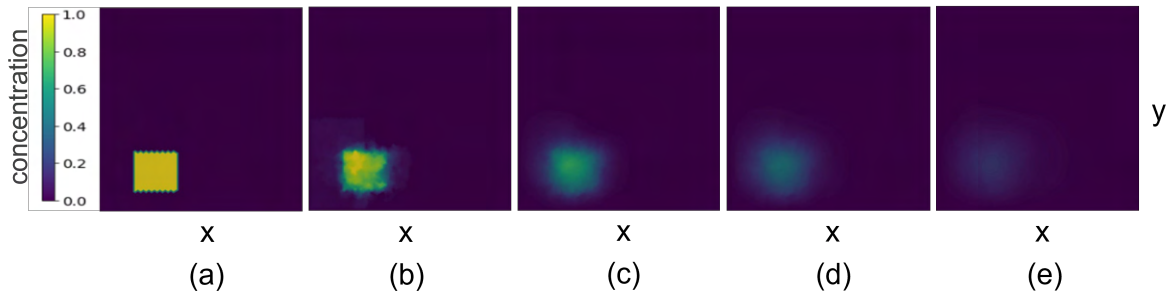

Figure A.2: Solving the diffusion equation on a 64x64 unstructured triangular mesh using $relax\_keep\_off = 0$ with (a) initial condition, and results after (b) 1 multigrid iteration (c) 5, (d) 10, (e) 20 and (f) 80 multigrid iterations, where the solution reaches a precision of 10^{-3}

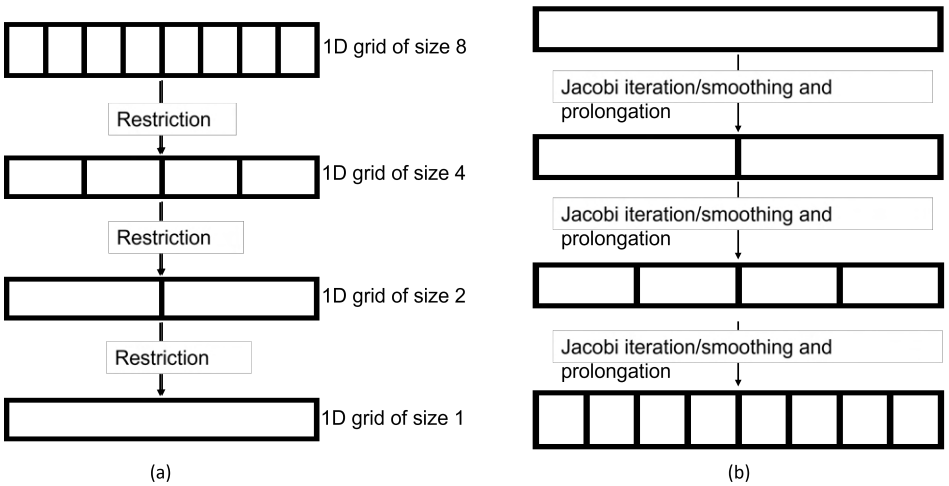

Figure A.3: Restriction and prolongation where (a) shows restriction from a 1D grid of size 8 to a 1D grid of size 1, and (b) shows prolongation of a 1D grid of size 1 to a 1D grid of size 8. As the solution correction is corrected from one grid to another, a Jacobi iteration is performed