POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Project Report

Solving PDEs using a Graph Neural Network

Advanced Programming for Scientific Computing

Authors: Andrea Bonifacio, Sara Gazzoni

Advisors: Stefano Pagani

Co-advisor: Mattia Corti

Academic year: 2022-2023

1. Introduction

Full 3D blood flow models are important in the study of the cardiovascular system since they allow one to extract detailed quantities of interest, but their actual implementation is limited due to their high computational cost. For this reason, reduced order models are widely used in this field because of their efficiency [3]. An example is presented in [5], where a one-dimensional reduced order model is implemented to simulate the blood flow in the aorta using a graph neural network trained on three-dimensional simulations. In this work we propose a different application, where the graph neural network can be used to approximate the solution of different PDEs. In particular, we consider the heat equation as test case, but the goal of the project is to show the potential extension of this approach to solve more difficult problems with complex geometries, such as the simulations of proteins spreading in the neural system, which are at the basis of neurodegenerative diseases [1]. The main part of this project is the implementation of a library for data generation used to train a graph neural network, and the adaptation of the code[1] used in [5] to make it suitable for our specific test case. In the following sections, we first present the problem formulation and a detailed description of the code developed, then we show the results obtained and a discussion of the possible further developments and extensions.

2. Problem overview

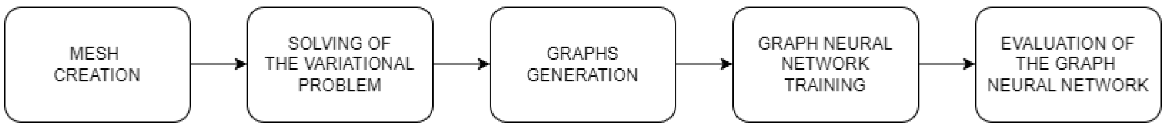

MESH CREATION
SOLVING OF THE VARIATIONAL PROBLEM
GRAPHS GENERATION
GRAPH NEURAL NETWORK TRAINING
EVALUATION OF THE GRAPH NEURAL NETWORK

Figure 1: Workflow of the project.

In this section we describe the problem formulation and the workflow of the project. The workflow is shown in Figure 1 and it is composed by five steps that will be described in detail.

The first step of the pipeline is the generation of the meshes. We consider a set of 2D meshes with different geometries, which will be used to solve the variational problem in the following step.

The second step consists in solving a variational problem in FEniCS [2], which is a Python platform for solving partial differential equations, particularly well suited for the finite element method. We consider a

[1]The code can be found here.

general time-dependent variational problem of the form:

$$\frac{\partial u}{\partial t} + Lu = f \quad \text{in } \Omega \subset \mathbb{R}^2,$$

with L a linear operator, f a source term and u the solution. Given a suitable set of initial and boundary conditions, a specific geometry Ω from the set of meshes generated in the previous phase, and a suitable time discretization, we obtain the solution u^n at each time step n by solving the problem using finite element method.

In the next step, we generate the graphs using the solutions obtained from the previous step. A graph describes the geometry and the variational problem: it is composed by nodes, located in the centerline of the mesh, and by edges connecting the nodes. The nodes and edges are characterized by different features, which are quantities of interest realted to the variational problem. Specifically, we have two types of features: time independent features, which are the same for all the time steps (e.g length of the edges, diffusivity constant,..) and time dependent features, which are computed at each time step (e.g. heat flux, temperature,..). The latters are the features that we refer to also as 'target features', since they are the values that the graph neural network has to predict.

Summarizing these first three steps, we start from the generation of a set of meshes, then we solve the variational problem N times, varying the geometry and the parameters of the problem, and finally we generate N graphs from the data obtained by solving the variational problem. In this way, we obtain a dataset of N graphs that can be used to train the graph neural network.

The final two steps of the workflow are related to the training and the testing of a graph neural network. The graph neural network considered in this project is the one proposed in [5], which is based on MeshGraphNet. MeshGraphNet is an innovative framework proposed by the Deepmind team to predict mesh-based simulation, in [6] it is used to predict cloth dynamics or structural deformation. In [5] the idea is modified to be applied to graph reduction of 3D blood vessels. In this work, we follow the second adaptation and modify it to be able to work on any kind of graph-based representation of a mesh.

We first focus on a brief description of how the graph neural network works, and then on the main characteristic of the training and testing phases. Graph neural networks (GNNs) are neural models that capture the dependence of graphs via message passing between the nodes of graphs. They can be used for a wide range of tasks, from traffic state prediction to protein interface prediction [7].

The GNN is applied iteratively: at each time step it takes as input the system state Θ^n, which is the set of the target features at that time step, and it predicts an update for the state variables (rollout phase). The prediction is combined with the previous time step to estimate Θ^{n+1}. A forward step of the GNN is composed by three stages:

1. Encode: a latent representation of the node and edge features is computed using a fully connected neural network.
2. Process: the process stage is composed by L identical blocks, each of them is applied in sequence to the output of the previous blocks, updating the node and edge features.
3. Decode: using a fully connected neural network, the node features are transformed from the latent space to the output space. The output of the GNN is a vector containing the update of the state variables $\delta\Theta^n$ at each node of the graph.

After this forward step, the state variables can be updated as $\Theta^{n+1} = \Theta^n + \delta\Theta^n$.

In the training phase, the GNN is trained to predict this update on the dataset of graphs generated in the previous steps. In particular, we consider a dataset composed of G graphs, each of them consisting of a set of nodes $n_1^g, ..., n_{N^g}^g$, and we define as M^g the final time step of the g-th graph. The neural network is trained on small strides s of consecutive timesteps since it has been proven to improve the accuracy of the predictions [5]. The loss function minimized during the training is defined as:

$$\mathcal{L} = \sum_{g \in \mathcal{T}} \frac{1}{|\mathcal{T}|(M^g - s)} \sum_{n=0}^{M^g - s} sMSE^{n,g,s}, \tag{1}$$

where $\mathcal{T}$ is the set of graphs used for training and $sMSE$ is the strided mean squared error between the prediction and the ground truth:

$$sMSE^{n,g,s} = \frac{1}{N^g} \sum_{l=1}^{s} \sum_{i=1}^{N^g} a_l (\hat{\Theta}^{k+l,g} - \Phi_l(\Theta^{k,g}|_i))^2. \tag{2}$$

In the above equation, $a_l = 1$ if $l = 1$ and $a_l = 0.5$ otherwise, $\hat{\Theta}^{k+l,g}$ is the exact value of the nodal state variable at time step $k + l$ of the g-th graph, and $\Phi_l(\Theta^{k,g}|_i)$ is the approximation of Θ at node i after l application of the GNN. Another quantity considered during the training is the metric, which is a strided mean absolute error defined, analogously to the loss function, as:

$$\mathcal{M} = \sum_{g\in\mathcal{T}} \frac{1}{|\mathcal{T}|(M^g - s)} \sum_{n=0}^{M^g-s} \frac{1}{N^g} \sum_{l=1}^{s} \sum_{i=1}^{N^g} a_l |\hat{\Theta}^{k+l,g} - \Phi_l(\Theta^{k,g}|_i)|. \tag{3}$$

where the notation is the same as in Equation (2).

In addition, during the training, the state variables are perturbed with random noise to simulate the error caused by the network's predictions, this is done to improve the robustness of the network, and we have hyperparameters to control the rate of the noise. It is important to make the graph neural network robust for rollouts of many time steps, since the error generated by the predictions is propagated at each time step. For this reason, also a rollout error is considered during the training, which is defined as:

$$e^g = \frac{\sum_{i=1}^{N^g} \sum_{k=11}^{M^g} (\hat{\Theta}_i^{k,g} - \Phi_l(\Theta^{k,g}|_i))^2}{\sum_{i=1}^{N_g} \sum_{k=1}^{M^g} (\hat{\Theta}_i^{k,g})^2}. \tag{4}$$

A more detailed desctiption of the training procedure can be found in Section 3.4.

The last block of the workflow is the testing of the trained network. In this phase, we consider as indicator of the accuracy of the network the rollour error defined in Equation (4). Moreover, the model is tested on the train and test set, but also on new graphs, which are generated using different meshes and different parameters of the variational problem.

2.1. Test case: heat equation

In this work, we consider the heat equation as test case. The mathematical formulation of the problem is the following:

$$\begin{cases} \frac{\partial u}{\partial t} = k\Delta u & \text{in } \Omega \subset \mathbb{R}^2, \\ k\frac{\partial u}{\partial n} = h & \text{on } \partial\Omega_{inlet}, \\ k\frac{\partial u}{\partial n} = 0 & \text{on } \partial\Omega_{outlet} \cup \partial\Omega_{walls}. \end{cases} \tag{5}$$

where u is the temperature, k is the thermal diffusivity, h is the Neumann condition at the inlet boundary. As domain Ω we consider different geometries such as the one shown in Figure 2: the 2D mesh is composed of 4 trapezoids where the interface between them have different lengths. The inlet boundary is the left one, while the outlet boundary is the right one. The other boundaries are the walls.

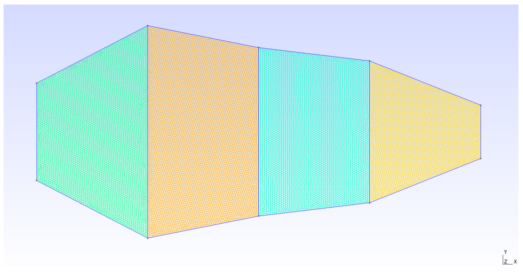


Figure 2: Example of a mesh used in the test case.

We generated 20 different mesh using gmsh. Then we solved the problem in FEniCS using Discontinuous Galerkin method and implicit Euler for time discretization, imposing as Neumann condition at inlet $h = 2e^{-(t-2.5)^2}$. From these solutions we generated a dataset of 277 graphs. Each graph has 5 nodes: an inlet node, an outlet node and 3 nodes in correspondence of the interfaces. As descriptor of the state of the system we consider the heat flux at each time step, which is computed as the integral of the normal derivative of the temperature on the interface. The other node features are the thermal diffusivity k, the interface length and the nodal type (inlet, outlet or branch node). As edge features we consider the area of the corresponding trapezoid and the distance between the nodes connected by the edge. The graph neural newtork is trained for 500 epochs on this dataset, a list of the hyperparameters used is shown in Table 1. For further details about the hyperparameters and the network architecture please refer to Section 3.4.

Parameter	Value
batch size	32
learning rate decay	0.001
learning rate	0.01
rate noise	5
rate noise features	10^{-5}
l2 regularization	10^{-5}
latent size gnn	16
latent size mlps	16
normalization type	1 (normal)
stride	5
n_out	1
bc_type	‘heat’
optimizer	‘adam’

Table 1: Hyperparameters used for the GNN training.

3. Code

The implementation of this work is in Python. The mesh generation is done using `gmsh`, while the variational problem is solved using `FEniCS`. The graph neural network is implemented using `PyTorch`. The code is available on GitHub[2].
The folders and files in the repository are organized as follows:

- **scripts**: this folder contains the scripts with all the classes and functions necessary to create the meshes, solve the variational problems and generate the graphs.
- **gNN**: this folder contains the code of the graph neural network.
- **notebooks**: this folder contains the jupyter notebooks that can be run to generate the dataset and to test the GNN.
- **models**: this folder contains the trained models.
- **data**: this folder contains the set of meshes used in the test case, the dataset of graphs used to train the GNN, and new graphs used to test the GNN.
- **README.md**: this file contains a brief description of the repository and the instructions to run the code.
- **requirements.txt**: this file contains the list of the packages needed to run the code.
- **documentaion.pdf**: this file contains the Doxygen documentation of the code.

In this section, we describe in detail the code implementation related to each block of the workflow presented in Section 2.

3.1. Mesh creation

The first step is the generation of the meshes, the code responsible for this part is the `MeshUtils.py` file in the `scripts` folder. It contains two classes whose structure in shown in Figure 3. The first one is `MeshCreator` which is used to create meshes using `gmsh`, while the second one is `MeshLoader` which is used to load the meshes from a `.xml` file and extract all the features that will be used to solve the variational problem.

[2]The repository can be found here.

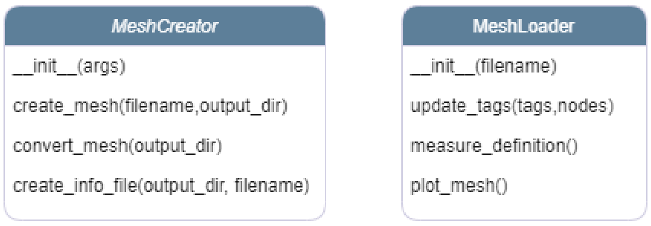


Figure 3: Classes and methods of `MeshUtils.py`.

The `MeshCreator` constructor takes as input command-line arguments, for this reason the file contains also a `main` function where we use the `argparse` Python module to handle command-line arguments. The possible arguments that can be modified by the user are summarized in Table 2.

Argument	Description
nmesh	number of meshes to create
interfaces	number of interfaces (including inlet and outlet)
lc	characteristic length of the mesh
hmax	maximum length of the interfaces
hmin	minimum length of the interfaces
seed	seed for random number generator
spacing	boolean variable, True if the interfaces are equispaced
wmax	maximum distance between the interfaces
wmin	minimum distance between the interfaces

Table 2: Arguments of `MeshCreation` constructor.

The method `create_mesh` generates nmesh new meshes using `gmsh` and the parameters passed to the constructor. The lengths of the intefaces are random numbers between hmin and hmax, while the distances between the interfaces are random numbers between wmin and wmax if the spacing argument is True, otherwise they are equispaced and the distance is set equal to wmax. Different tags are assigned to the walls, the inlet boundary, the outlet boundary, each interface and each face. In the end, the meshes are saved in the `output_dir` directory in `.msh` format. Then, the `convert_mesh` method is called to convert all the meshes in the `output_dir` from `.msh` to `.xml` format.

The last method of the class is `create_info_file` which is used to save a json file in the `output_dir` containing the parameters used to generate the meshes.

The `MeshLoader` class is used to load a mesh from a `.xml` file. The constructor takes as input the name of the file and stores the following attributes:

- `mesh`: FEniCS mesh object
- `bounds`: FEniCS `MeshFunction` associated to the boundaries of the mesh
- `face`: FEniCS `MeshFunction` associated to the faces of the mesh
- `n`: outward-pointing normal vector to the mesh boundaries
- `h`: minimun element size, used as characteristic length of the mesh

Another method of the class is `update_tags(self,tags,nodes)`: this method can take as input a dictionary containing the tags of the mesh and the number of nodes of the graph (which is the number of interface plus the number of inlets and outlets). If the dictionary is not provided, the method computes automatically the tags dictionary based on the `nodes` input. The dictionary has the following structure: the keys are the strings 'walls', 'inlet', 'outlet', 'interface' and 'faces' and the values are lists containing the corresponding tags. If the number of nodes is not provided, the dictionary must be provided and it must have the same structure explained above.

Then the class has the `measure_definition(self)` method: it return three `Mesure` FEniCS objects, which are used for integration over external boundaries, internal boundaries (interfaces) and faces.

Finally, the mesh can be plotted using the `plot_mesh(self)` method.

3.2. Solving the variational problem

This part of the code refers to the second step of the pipeline, which consists in solving the variational problem and save the solution at each time step. To this purpose, we implemented a `Solver` class, which is located in the `DataGenerator.py` file in the `scripts` folder. The structure of the class is shown in Figure 4.

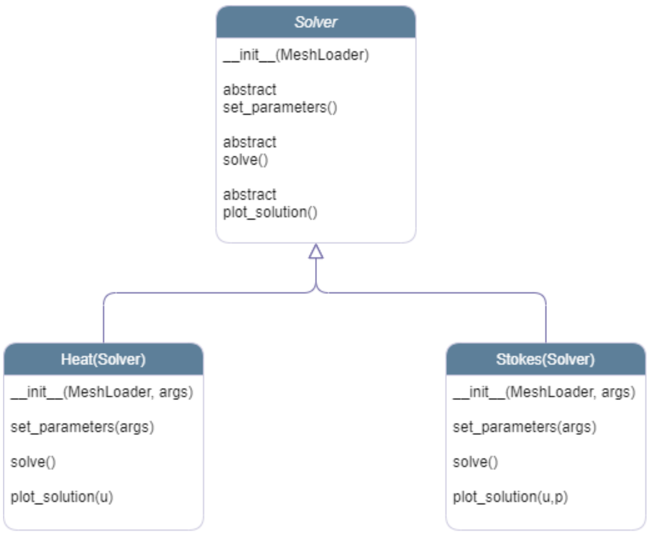


Figure 4: `Solver` class structure and methods.

As we can notice from the scheme, there is an abstract base class `Solver` which has two child classes: `Heat` and `Stokes`, that are used to solve the heat equation and the Stokes equation respectively. In the code 1 we show the definition of the parent class, were we can observe a constructor and three abstract methods. In particular, the constructor takes as input a `MeshLoader` object, which is useful to load the mesh considered in the problem to solve. The abstract methods are used to set the parameters of the problem, to solve it and to plot the solution, and they are all implemented in the child classes.

```python
class Solver(ABC):

    def __init__(self,mesh):
        self.mesh = mesh

    @abstractmethod
    def set_parameters(self):
        pass

    @abstractmethod
    def solve(self):
        pass

    @abstractmethod
    def plot_solution(self):
        pass
```

Code 1: Solver abstract class.

The choice of a parent abstract class for the solver is useful because it allows to use the same code for different problems, implementing child classes that solve different equations, but with the same structure. We focus on the description of the `Heat` class, since it is the one used in the test case, but the `Stokes` class is implemented analogously.

The `Heat` class contains the following methods overridden from the parent class:

- `__init__(self,mesh, V, k, f, u0, dt, T, g, doplot=False)`: constructor which uses the `super()` function to inherit the base class constructor. The other problem parameters passed to the constructor are the function space, the diffusivity constant, the source term, the initial condition, the time step, the final step, the Neumann boundary condition at the inlet and a boolean variable to decide if the solution at each time step has to be plotted or not.
- `set_parameters(self,V,k,f,u0,dt,T,g)`: function to set different problem parameters.
- `solve(self)`: this method solves the Heat equation using Discontinuous Galerkin method, imposing a non-homogeneous and time-dependent Neumann condition at the inlet boundary and no-flux condition at the walls and at the outlet. The temperature at each time step is stored as attribute of the object in a numpy array, as well as the time instants.
- `plot_solution(self,u)`: it takes as input the solution at a specific time step and it plots it.

3.3. Graphs generation

When the variational problem is solved and the solution are saved, the next step of the pipeline is to store the quantities of interest of the problem and to generate a graph from these. To this purpose, we implemented the DataGenerator class, which is also located in the DataGenerator.py file in the scripts folder. The structure of the class can be obsserved in Figure 5.

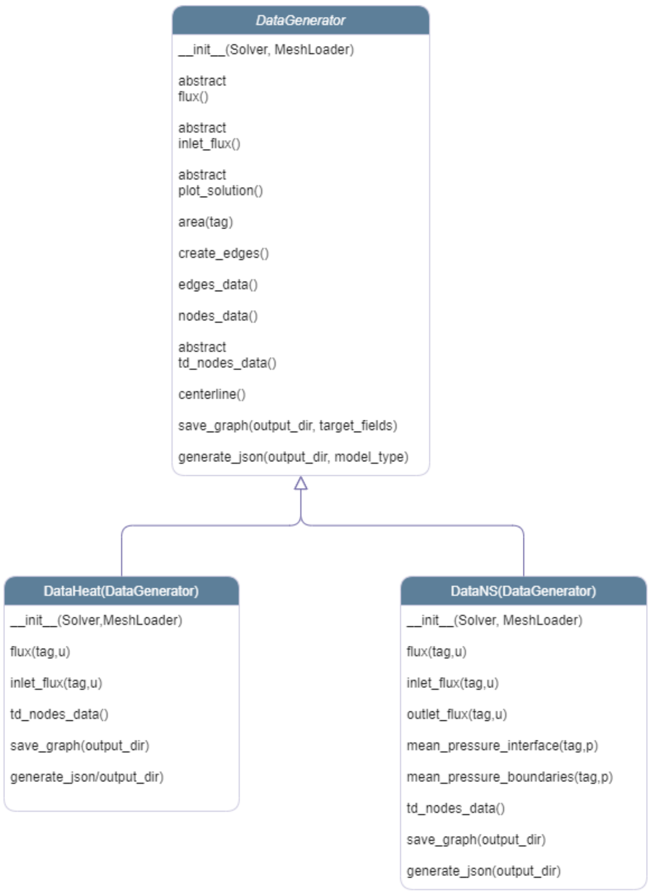


Figure 5: DataGenerator class structure and methods.

Also in this case, we have an abstract base class DataGenerator which has two child classes: DataHeat and DataNS, that are used to generate and store the data for the heat equation and the Stokes equation respectively. The DataGenerator base class contains some abstract method that will be overridden in the child classes, their definition is shown in the code 2, together with the class constructor. We can observe that the constructor takes as input a Solver object and a MeshLoader object and a NNodes attribute which is set as the sum of the number of inlets, outlets and interfaces of the mesh. The abstract method are used to compute some quantities of interest of the problem and to store the time dependent features, but they will be described in detail later.

```
    class DataGenerator(ABC):

        def __init__(self, solver, mesh):
            self.solver = solver
            self.mesh = mesh
            self.NNodes = len(self.mesh.tags['interface']) + \
            len(self.mesh.tags['inlet']) + len(self.mesh.tags['outlet'])

        @abstractmethod
        def flux(self):
            pass

        @abstractmethod
        def inlet_flux(self,tag, u):
            pass
```

```python

    @abstractmethod
    def td_nodes_data(self):
        pass
```

Code 2: DataGenerator abstract class.

This class contains also some concrete methods, in particular the followings are useful to store the node and edge features necessary to generate the graphs:

- `nodes_data(self)`: this function saves as attribute of the object a dictionary containing the time independent nodes features. The keys of the dictionary are strings with the name of the features and the values are numpy arrays containing the values at each node. The features considered for each node are the thermal diffusivity, the interface length where the node is located, the nodal type (inlet, outlet or branch node) and a global node ID.
- `create_edges(self)`: it is used to save as attributes of the object two numpy arrays (`self.edges1` and `self.edges2`) containing respectively the node IDs of the source nodes of every edge and the node IDs of the destination nodes.
- `edges_data(self)`: it stores as attributes of the object a dictionary containing the edge feature. The dictionary structure is analogous to the one of the node features. The features saved are the distance between the nodes connected by the edge and the area of the trapezoid corresponding to the edge.

Then we have two other concrete methods that are used to compute some time independent features of the graph related to the geometry of the mesh:

- `area(self,tag)`: concrete method that computes the area of the trapezoid corresponding to the tag passed as input
- `centerline(self)`: this function computes the coordinates of the graph nodes, which are the coordinates of the centerline of the mesh at the interfaces. These coordinates are stored in a numpy array of size $NNodes \times 2$.

These two function are called inside the `nodes_data` and `edges_data` methods to save the corresponding features.

Moreover, we have a concrete methods useful to generate the graph and to save it in a file which is called `save_graph(self, output_dir,fields_names)`: this method takes as input the name of the time dependent features of the graphs and the output directory where the graph has to be saved. It returns a dgl graph which is generated calling the functions defined in `generate_graphs.py`, that will be described later in this section.

The last method of the class is `generate_json(output_dir,model_type)`: this function is used when a set of graphs is already saved in a directory. It is used to generate and save in the output directory (that must be the directory containing the graphs) a json file containing information about the dataset. This file is necessary to make the dataset compatible to the gNN code.

To summarize, this abstract class takes as input a `MeshLoader` object that contains the information about the mesh and a `Solver` object that contains the solution of the variational problem. Then, using the methods described above, all the necessary features are computed and stored in dictionaries. Finally, from these saved quantities a graph is generated and saved in file.

Now we can focus on the child classes and on the abstract methods. As for the `Solver` class, the abstract methods are overridden in the two child classes `DataNS` and `DataHeat`. As above, we focus on the description of the `DataHeat` class, since it is the one used in the test case, but the `DataNS` class is implemented analogously.

First, the constructor of this child class is inherited from the parent class and two additional attributes are introduced: `model_type` and `target_fields`. The first one is a string containing the name of the equation considered, which is 'heat' in this case. The second one is a list containing the name of the target features of the graph (which are the features that the GNN has to predict). In this case the list contains only the string 'flux'. The constructor definition is shown in Code 3.

```python
def __init__(self, solver, mesh):
    super().__init__(solver, mesh)
    self.model_type = "heat"
    self.target_fields = ['flux']
```

Code 3: Solver class.

Then we can describe the implementation of the abstract methods whose definition was shown in the code 2. Firstly, the `flux(self,tag,u)` and `inlet_flux(self,tag,u)` are two methods that compute the heat flux at the interfaces and at the inlet respectively, given the solution u at a specific time step and the tag of the interface or the inlet boundary. Notice that the flux at the outlet boundary is not computed, this is due to the fact that the flux at the outlet is always zero, since the no-flux condition is imposed.

Then, there is the `td_nodes_data(self)` method: it stores as attribute of the object the time dependent nodes features in a dictionary. In this case the only time dependent feature is the heat flux: the dictionary has as keys the time instants and as value for each key a numpy array containing the heat flux at that time instant at each node.

Finally, in this class we define, as shown in Code 4, the `save_graph(self, output_dir,fields_names)` method and the `generate_json(output_dir,model_type)` method. The first one invokes the corresponding method of the parent class using the `super()` function, but the input `fields_names` is the `target_fields` attribute of the object. In the same way, the second method invokes the corresponding method of the parent class and the `model_type` input is the homonymous attribute of the object.

```python

def save_graph(self, output_dir):

    return super().save_graph(output_dir,self.target_fields)

def generate_json(self,output_dir):

    return super().generate_json(output_dir,self.model_type)
```

Code 4: Definition of the `save_graph` and `generate_json` methods of the `DataHeat` class.

The `DataNS` is build analogously, with the only difference that the time dependent features are the flow rate and the pressure instead of the heat flux. In this case, the `flux` and `inlet_flux` method computes the flow rate at the interfaces and at the inlet respectively. In addition, the class has a method `outlet_flux` that computes the flow rate at the outlet and two other methods, `mean_pressure_interface` and `mean_pressure_boundaries`, that compute the mean pressure at the interfaces and at the inlet and outlet boundaries respectively.

Now we can describe the functions defined in `generate_graphs.py`, included in the `scripts` folder. This file contains useful functions to generate the graphs from the features saved by the `DataGenerator` class. The functions are the following:

- `generate_graph(point_data, points, edges_data, edges1, edges2)`: this function takes as input a dictionary containing the time independent node features, a numpy array with the node coordinates, a dictionary containing the edge features and two numpy arrays containing the source and destination nodes of the edges. It return a dgl graph with the node and edge features stored as pytorch tensors.
- `add_fields(graph, field, field_name, offset=0)`: function to add a time dependent feature to the dgl graph. It take as input the dgl graph, a dictionary containing the field values at each time step, the name of the field to insert and an offset with the number of time steps to skip. It returns the dgl graph with the new field added.
- `save(filename, output_dir)`: function to save the dgl graph in a file in the output directory.

All this function are invoked by the `save_graph` method of the `DataGenerator` class, in this way a graph with all the necessary features is generated and saved in the output directory.

To conclude, all the classes and function described in this section are useful to save the quantities of interest of a variational problem and to generate a graph from these. This process, from solving a problem to generating a graph, can be repeated a proper number of times to generate a dataset of graphs. So, thanks to the implementation explained above, a dataset can be generate and used to train a graph neural network.

3.4. Graph Neural Network

In this section we describe the code of the graph neural network, located in the `gNN` folder. This part of the code refers to the last two blocks of the our workflow, which are the training and the testing of the network. Since the code was already implemented, and we just made some modifications, we will not describe it in detail, but we will focus on the changes we made and on the more relevant parts of the code, which are inside the `network1D` folder.

Code Structure

To better understand the behavior of the network, it is necessary to look at its inner mechanisms. Starting from the `MLP` class, we can see that it is a simple multi-layer perceptron, with a number of hidden layers specified by the input parameter `n_h_layers`. This is the basic mechanism of all the network, every time that we want to process a node or an edge, we use an `MLP` to do so. The `MLP` class has a constructor that takes as input the number of input features, the number of output features, the size of the latent space and the number of hidden layers and use those inputs to create the input and output layers and the hidden layers. The input and output layers are `Linear` layers, while the hidden layers are `Linear` layers followed by a `LeakyReLU` activation function.

The MLP class also has a forward method that computes the forward step of the network. It takes as input the input tensor and returns the result of the forward step. This is what will be meant when talking about encoding or decoding, feeding those data to an MLP.

The EncodeProcessDecodeNetwork class is the core of the network, it is the one that actually processes the graph. It is composed by three main parts: the encoder, the processor and the decoder. The encoder is composed by two MLPs, one for the nodes and one for the edges. The processor is made up by a number of MLPs equal to the number of iterations specified by the input parameter process_iterations. The decoder is built using a single MLP that has the same number of neuron in its last layer as the output.

The EncodeProcessDecodeNetwork class constructor takes as input a dictionary of hyperparameters and uses it to initialize the encoder, the processor and the decoder. The EncodeProcessDecodeNetwork class also has five methods, one that encodes the edges (encode_edges) and the other are for processing and decoding both edges and nodes (process_edges, process_nodes, decode_edges, decode_nodes). The attentive reader will surely notice that the method needed for encoding the nodes is missing, but this will become clear in the next paragraph.

The MeshGraphNet class is a subclass of EncodeProcessDecodeNetwork, it is the one that actually computes the pressure and flowrate updates given the previous system state. Its built using the same constructor as its father class.

The method inside this class are the following:

- encode_nodes: this method takes as input the node features and returns the result of the encoding step. It is done inside this class to avoid losing generality in its parent class since nodes could have different features depending on the problem (e.g. inlet_mask is quite specific to the fluid dynamics problem that this code originally solved).
- continuity_loss: this method is not used in this test case, since we do not have any junction nodes. It is used to compute loss at junction nodes and it is left there to future proof the code.
- forward: this method is the one that actually computes the forward step of the network. It encodes both nodes and edges to the latent space, then perform a loop of processing iterations and finally decodes both nodes and edges to the output space. It returns the predicted update of the system state.

Training

Here we describe the training process more in detail. The process is done using the training.py file and the rollout.py file. The first one is used to perform the steps necessary to train the network, like backpropagation and loss computation, while the second one is used to perform the forward step of the network and compute the predictions.

The training of the network starts with the definition of its hyperparameters described in Table 3. In particular, the first four parameters are the ones that define the network architecture, as explained in the previous section. The rate_noise and rate_noise_features parameters are related to the noise that is added to the system state during the training process, to make the model more robust. The first parameters listed in this table are the one from the original implementation, but we added three more to make the network more flexible: a n_nout parameter that defines the number of outputs of the network (and it must be equal to the number of target features), a bc_type which is a string that defines the type of boundary condition that has to be imposed (e.g. 'heat' in our test case) and the optimizer parameter which allows the user to choose the optimizer to use during the training process. The default optimizer is Adam [4], which we found to be the best for our problem.

Parameter	**Description**
latent_size_gnn	Size of the latent space of the Graph Neural Network
latent_size_mlp	Size of the latent space of each MLP
process_iterations	Number of iterations of the process step
number_hidden_layers_mlp	Number of hidden layers of each MLP
learning_rate	Learning rate of the optimizer
batch_size	Batch size
lr_decay	Learning rate decay
nepochs	Number of epochs
weight_decay	Weight decay for the optimizer
rate_noise	Rate of noise
rate_noise_features	Rate of noise for the features
stride	Stride
n_out	Number of outputs of the network
bc_type	Type of boundary condition
optimizer	Optimizer

Table 3: Hyperparameters of the network.

With all the hyperparameters set, we can start the training process. Since this part of the code was only modified by us, we will not go into much detail, but we will only describe the main steps. The training starts by taking a dataset composed of `DGL` graphs, which are split into a train set and a test set. Then are defined a loss (MSE) and a metric (MAE). The dataset is created in such a way that to train the network on all the graphs a map is needed so that when a single time-step is evaluated, the network is able to map that single time-step to the correct graph. The forward step is performed using the function `perform_timestep` from the `rollout.py` file. This function takes as input the model, the graph and the previous time-step and predict the value of the quantity of interest at the next step in time. It also takes as input an array containing the boundary conditions and a boolean variable that indicates if the boundary conditions have to be imposed: if the boolean variable is True, the boundary conditions are imposed according to the `bc_type` parameter. This function is used also when evaluating the model since when we use it while training we compute the loss between the model's prediction and the ground truth, updating the weights of the network accordingly, while when we compute the rollout errors we use the model in inference mode and then compute the error between the prediction and the ground truth.

While the original implementation was meant to work only with blood flow data, we modified the original code so that, given a suitable dataset, it can be used to solve any problem. The main changes that we made are the following:

- We modified the data generation part of the code to make it suitable for any kind of problem. In particular, we made sure that during the dataset creation, the N Quantities of Interest (QoI) are stored in the first N position of the dictionary containing the node features. This is done to make sure that the network can access the QoI easily.
- We modified the dataset generation part in a way such that the program is able to create N deltas, one for each QoI. In this way we obtain our true update of the system state, which is the difference between the QoI at the current time-step and the QoI at the previous time-step. These deltas will be used to compute the loss of the network.
- We modified the `training.py` file to make it able to take as input the number of outputs needed by the network. In this way we have completely generalized our library to be able to receive as input any suitable dataset.

With these modifications, the network is able to predict a variable number of QoI, which have to be specified by the user in the `main` function of the `training.py` file, in a list called `target_features`. The limitation that persists despite of our changes, is the imposition of the boundary conditions. In fact, the network is able to impose only two types of boundary conditions that we called 'heat' and 'stokes', accordingly to the type of problems that we are solving in this work. Specifically, the 'heat' boundary condition that we used in the test case imposes the exact heat flux at the inlet node (Neumann boundary condition) and the zero heat flux at the outlet node (homogeneous Neumann boundary condition). If a new problem is considered, the user has to modify the `perform_timestep` function in the `rollout.py` file in order to impose the correct boundary conditions.

Tester

The `tester.py` file is used to test the trained model on graphs from the training and test set and on new graphs. The file is composed of the following already implemented functions:

- `get_dataset_and_gnn`: loads the dataset and the trained GNN given a path to a saved model folder
- `get_gnn_and_graphs`: loads the trained GNN and the graphs given a path to a saved model folder
- `evaluate_all_method`: computes the rollout phase for all the graphs in the dataset ('train' or 'test' is specified by the user), given a trained model. This function returns the average rollout error.

In this file, we implemented two additional functions:

- `plot_predictions(dataset, split_name, gnn_model, params, graph_idx=-1)`: it computes the rollout phase for a single graph and it plots the model's predictions compared to the real data, given a trained model and the dataset on which it was trained. It takes as input the dataset, the name of the split ('train' or 'test'), the trained model, the parameters of the model and the index of the graph to plot. If the index is not specified, the graph is randomly chosen. The function returns the rollout error between the model's predictions and the real data.
- `evaluate_new_graph(path, graph_name, graphs_folder, data_location)`: it computes the rollout phase and it plots the model's predictions and the real data for a new graph given a trained model. It takes as input the path to the model folder, the name of the graph to evaluate, the folder where the graph is stored and the path to the folder where the data are stored. As above, the function returns the average error of the rollout phase. Inside this method, the function `generate_normalized_graph` is called since we need to store the new graph features in order to evaluate the model.

3.5. Notebooks

To conclude the description of the code, we briefly explain the content of the notebooks in the `notebooks` folder. In this folder, there are some useful notebooks that can be used to reproduce the results of this work or to generate new datasets easily.

Firstly, there is the `HeatDatasetGen.ipynb` notebook, which is used to generate a dataset based on the Heat equation, as our test case. Inside this one, a random mesh is loaded from a given folder and the variational problem is solved using the `Heat` class, given a random diffusivity constant and the other parameters of the problem. Then, a graph is generated using the `DataHeat` class and it is saved in the output directory. This process is repeated for a given number of times, in order to generate a dataset of graphs. Finally, the json file containing the information about the dataset is generated and saved in the same output directory.

Then, there is the `StokesDatasetGen.ipynb` notebook, which is similar to the previous one, but the problem considered is the Stokes equation, so the corresponding classes are used. We did not use this notebook in our work and we did not focus on this part of the code, so the parameters of the variational problem in this notebook are just an example to show the possibility to consider a different problem with respect to the one considered in the test case. So, the purpose of this notebook is jut to show that the code is able to generate a graph also for a different problem. To actually use the code to generate a dataset of graphs for the Stokes equation, it would be necessary to set the parameters of the variational problem properly and choose which parameters could change during the dataset generation.

Finally, there is the `ModelTester.ipynb` which can be used to evaluate a trained model. In this notebook, it is necessary to specify the path to the model folder and the path to the proper dataset folder. In this way, the model and the dataset are both loaded and it is possible to visualize the predictions of the model on the graphs of the dataset and on new graphs from a different folder. It is also possible to display the average rollout errors on the train and test sets. All of this is done using the functions implemented in the `tester.py` file.

4. Running instructions

To run the code and reproduce the results presented in Section 5, related to the testing of the GNN, it is necessary to run the `ModelTester.ipynb` notebook. The path to our trained model and to the dataset that we used are already specified in the notebook, so it is not necessary to change them. It is possible to plot the predictions of the model on different graphs from the training and the test set just changing the index of the graph to plot. Then, the model's predictions on new graphs can be plotted just specifying the filename of a new graph and the folder where it is stored, some examples that can be tested are inside the `graphs_new` folder.

Another possibility is to generate new meshes and new graphs, using the `MeshUtils.py` file and the `HeatDatasetGen.ipynb` notebook.

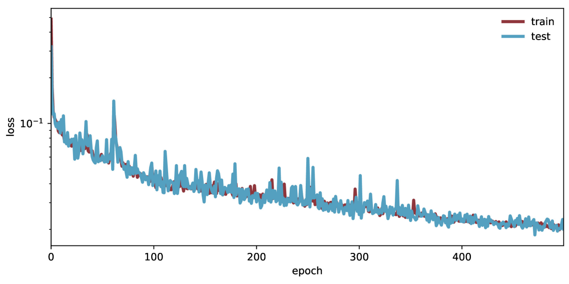
First, new meshes can be created running the `main` function of the `MeshUtils.py` file from command-line. All the parameters that can be modified are listed in Table 2 and they can be specified as command-line arguments. The directory where the meshes are saved and the mesh names can be modified in the `main` function.

Then, the `HeatDatasetGen.ipynb` notebook can be used to generate a new dataset of graphs. It is necessary to specify the path to the folder containing the meshes, the path to the folder where the graphs have to be saved and the number of graphs to generate. Also, all the parameters of the variational problem can be modified in the notebook. Following these instructions, it is possible to evaluate the trained model on the new graphs generated using the `ModelTester.ipynb` notebook and specifying the name of the new graphs folder and the name of the graph to evaluate. Notice that it is necessary that the new graphs directory is inside the `data` folder.

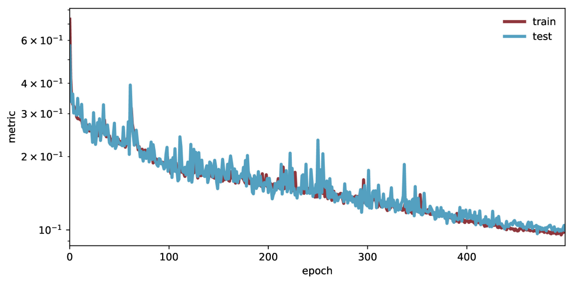
For a more detailed description of the running instructions and the training of the model, please refer to the `README.md` file in the repository: here we describe the instructions to run the whole code from data generation and testing of the network.

5. Results

In this section we show the results on the test case described in the section 2.1. It is important to keep in mind that every result presented in this section is obtained using a dataset in which we decided to restrict k to the range $[1, 100]$. We made this choice during our development since we saw that, in our test case, for values of k greater than 100, the heat flux behavior is not so dissimilar from the case in which $k = 100$. This choice was made to make the dataset more compact and to make the training process faster. In Figure 6 the performance of the model is shown in terms of loss and metric during the training. The loss function considered is the one defined in equation (2), while the metric is the mean absolute error between the model's predictions and the real data of Equation (3). Moreover, the rollout error's behavior is shown in Figure 7 and it is computed as in Equation (4) on a graph randomly selected in the dataset and at some random epochs during the training. We can observe that the loss and the metric decreases during the training as expected, while the rollout error is not monotonically decreasing. This is due to the fact that the rollout error is computed on a single graph, so it is not a good indicator of the model's performance. In Table 4 we show the final rollout errors on the training and the test set, which are computed as the average of the rollout errors of all the graphs in the dataset.



(a) Decay of loss function over 500 epochs.



(b) Decay of metric function over 500 epochs.

Figure 6: Loss and metric during the training.

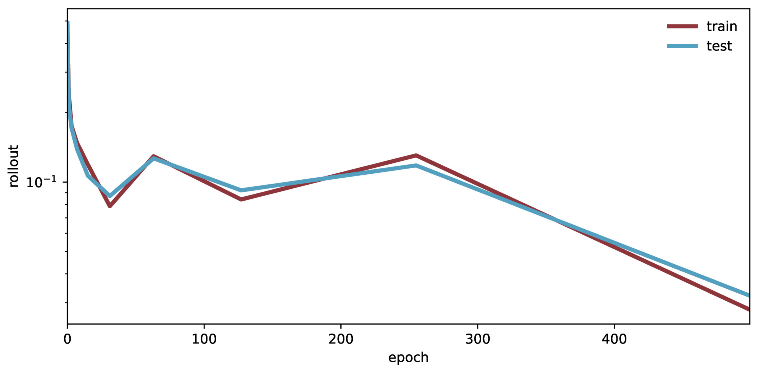
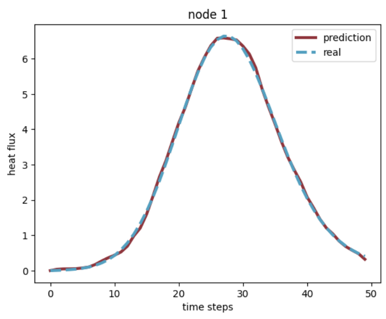


Figure 7: Rollout error over 500 epochs.

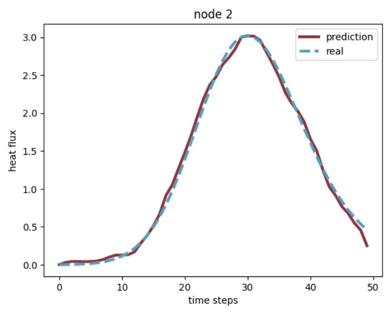
Dataset	Rollout error
Training	0.027
Test	0.029

Table 4: Average rollout error on the training and the test set.

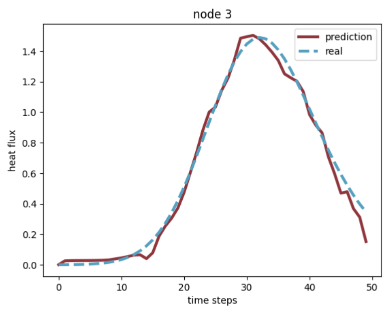
In Figures 8 and 9 we can see some example of the model's predictions on each internal node of a graph from the training and the test set respectively. We don't show the predictions on the inlet and the outlet nodes since they are imposed by the boundary conditions and not predicted by the model. In general, the model is able to reconstruct well the heat flux at the interfaces, according to the rollout error, which is always around $10^{-}2$ when tested on graphs from the training and the test set.



(a) Heat flux at interface 1.

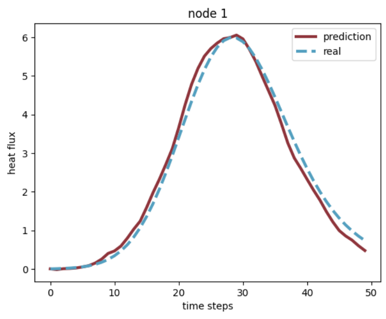


(b) Heat flux at interface 2.

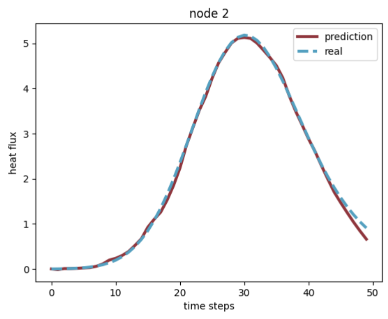


(c) Heat flux at interface 3.

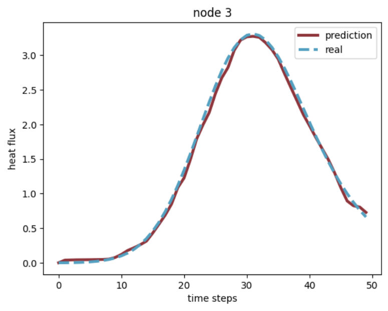
Figure 8: Real heat flux at the interfaces predicted by the model on a graph from the training set with diffusivity constant $k = 59.76$.



(a) Heat flux at interface 1.



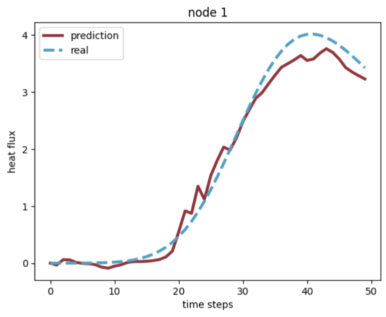
(b) Heat flux at interface 2.



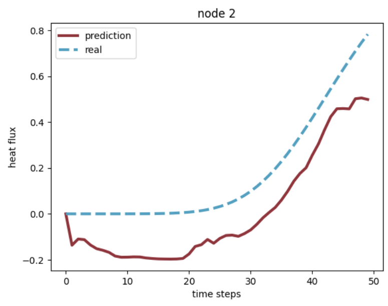
(c) Heat flux at interface 3.

Figure 9: Real heat flux at the interfaces predicted by the model on a graph from the test set with diffusivity constant $k = 99.66$.

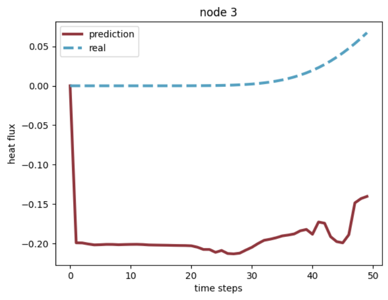
For a more accurate evaluation of the model's performance, we tested it on new graphs that were not included in the dataset. The predictions on these graphs compared to the real data are shown in Figures 10 and 11. These two graphs are generated solving the heat equations on new meshes with respect to the ones used to train the model and with random diffusivity constants in the range $[1, 100]$. We can observe that the predictions are more accurate on the nodes near to the inlet and less accurate on the nodes near to the outlet. Moreover, we have a slightly better performance on the graph with a higher k. This could be due to the fact that with a lower diffusivity constant the values of the heat flux are smaller and the model has more difficulties in predicting them. The same argument can be applied to the nodes near to the outlet, where the heat flux is smaller with respect to the inlet.



(a) Heat flux at interface 1.

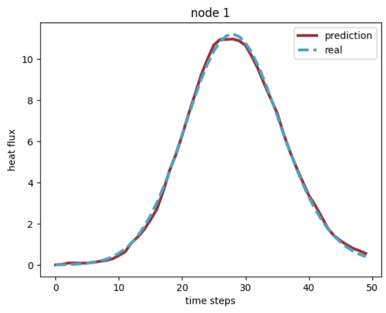


(b) Heat flux at interface 2.

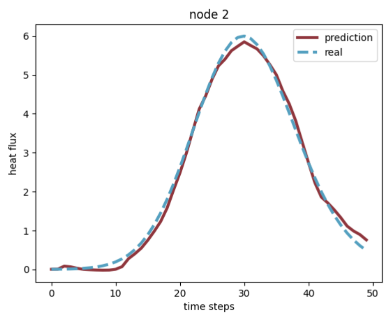


(c) Heat flux at interface 3.

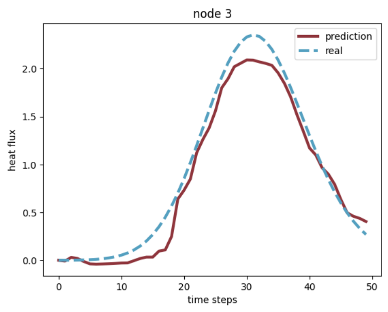
Figure 10: Real heat flux at the interfaces predicted by the model on a new graph with diffusivity constant $k = 3.05$.



(a) Heat flux at interface 1.



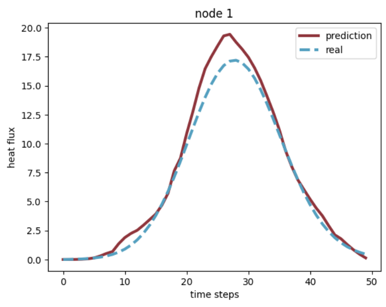
(b) Heat flux at interface 2.



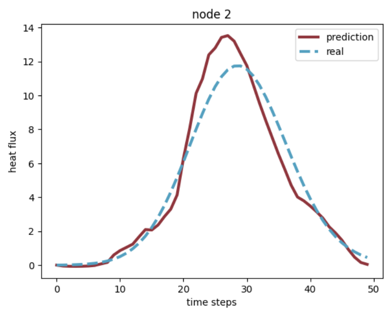
(c) Heat flux at interface 3.

Figure 11: Real heat flux at the interfaces predicted by the model on a new graph with diffusivity constant $k = 67.44$.

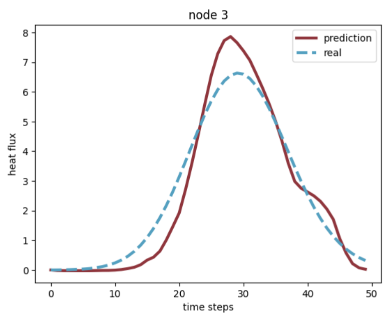
Finally, we tested the model on a new graph with a diffusivity constant $k = 197.44$, which is outside the range of the dataset. Also in this case the mesh is different from the ones used to train the model. The predictions are shown in Figure 12. We observe that the model is able to predict the heat flux behavior at the interfaces, even if the predictions are less accurate with respect to the previous cases, as expected. In fact, the average rollout error on this graph is 0.07, which is higher than the average rollout error on the test set.



(a) Heat flux at interface 1.



(b) Heat flux at interface 2.



(c) Heat flux at interface 3.

Figure 12: Real heat flux at the interfaces predicted by the model on a new graph with diffusivity constant $k = 197.44$.

6. Conclusions and further work

In this work, we focused on the implementation of an environment able to generate everything needed to train the Graph Neural Network and to test it on new graphs. One of the main modifications that we made to the original code is the possibility to train the network on different problems, such as the heat equation and the Stokes equation, but potentially any time dependent problem. However, there are still some limitations that could be fixed. One could be a deeper rewrite of the GNN code, which was built around a specific problem, and it is not very flexible. Also, our model is not very robust and couldn't be used to solve any real problem. One of its weaknesses is that we trained it on a very specific and simple problem, with a dataset that has little variability. This could be solved in a number of ways. First, in the dataset generation the variability of the dataset can be increased simply by changing the parameters of the mesh creation (for example considering not equally spaced interfaces or a higher number of interfaces) and also by changing other parameters of the problem in addition to the diffusivity constant (such as the inlet condition). In this way, the dataset will be more complex and the model will be more robust.

In fact, our mesh creation code is limited to the generation of a specific type of mesh, with a single inlet and a single outlet. It could be interesting to extend it to generate more complex meshes, with more inlets and outlets, and also to generate meshes with bifucations and multiple branches, since the graph neural network code is already able to handle this kind of meshes. In this case, it would be necessary to modify the `DataGenerator` class to make it able to handle more complex meshes. Another easier modification can be to consider a 3-dimensional problem, instead of a 2-dimensional one, in order to make the model more realistic. Apart from the mesh creation, it would be necessary to modify only the `centerline` function of the `DataGenerator` class to make it able to handle 3-dimensional meshes.

About the problems parameters of our test case, we considered only the diffusivity constant as a variable parameter, but it could be interesting to change also the other ones, such as the inlet condition or the source term, or even the time step and the final time.

Another important extension is the possibility to consider different problems, such as the Stokes equation that we already implemented. We did not generate a dataset for this problem, and we did not train the model on this problem for reasons of time, but it could be done with some modifications. To do so, it would be necessary to choose suitable problem parameters to generate a proper dataset. Once the dataset is generated, to train the graph neural network on the Stokes equation, it is necessary to modify the `main` function of the `training.py` file changing the target features to 'flowrate' and 'pressure', and also to modify the number of outputs of the network, which is a command-line argument. It is also required to modify the boundary conditions type in 'stokes', which is as well a command-line argument.

However, this is just an example to show how the code can be extended to other problems analogously, implementing a corresponding `Solver` class and a corresponding `DataGenerator` class, inherited from the parent classes, and modifying the `main` function of the `training.py` file with the proper target, nodes and edges features. In this case, it may be necessary to change how the boundary conditions are imposed accordingly to the problem considered (it can be done in the `perform_timestep` function of the `rollout.py` file). This is the main limitation in the GNN code that has to be fixed manually by the user when a new problem is considered.

To conclude, in this work we have managed to implement a fully working pipeline from mesh and dataset generation to training and testing of a graph neural network. In this way, we have shown that this approach can be used to solve a wide range of problems, from fluid dynamics to heat transfer. We believe that this could be a very powerful tool for simulations of complex problems, where the use of traditional numerical methods and of 3-dimensional models is not feasible. It has some drawbacks, clearly, such as the computational cost of the training phase and the need of a large dataset, but in the case of very complex problems, we think that this method could still be a good alternative to traditional ones.

References

[1] Mattia Corti, Francesca Bonizzoni, Luca Dede', Alfio M. Quarteroni, and Paola F. Antonietti. Discontinuous galerkin methods for fisher-kolmogorov equation with application to alpha-synuclein spreading in parkinson's disease. *Computer Methods in Applied Mechanics and Engineering*, 2023.

[2] FEniCS Team. FEniCS Documentation. `https://fenicsproject.org/olddocs/dolfin/2019.1.0/python/`, 2019. [Online; accessed 2-February-2024].

[3] L. Formaggia, A. Quarteroni, and A. Veneziani. *Cardiovascular Mathematics*. Springer Milan, Milano, 2009.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. (arXiv:1412.6980), January 2017. arXiv:1412.6980 [cs].

[5] Luca Pegolotti, Martin R. Pfaller, Natalia L. Rubio, Ke Ding, Rita Brugarolas Brufau, Eric Darve, and Alison L. Marsden. Learning reduced-order models for cardiovascular simulations with graph neural networks. *Computers in Biology and Medicine*, 2023.

[6] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. 2021.

[7] 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. (arXiv:1812.08434), October 2021.