POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Project Report

Title

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 cardiovascular system since they allow 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 fields because of their efficiency. An example is presented in [2], where a one-dimensionalreduceced 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 is used to approximate the solution of different problems. In particular, we consider theheat equation as test case, but the goal of the project is to show the potentialextension of this approach to solve more difficult problems with complex geometries,such as the simulations of proteins spreading in the neural system, which are atthe basis of neurodegenerative diseases [1].The main part of this project is the implementation of a library for data generation used to train the graph neural network and the adaptation of the code [di Luca non so come citarlo] 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

We consider a general time-dependent variational problem of the form:

$$Lu = f$$

with L a linear operator, f a source term and u the solution. Given a specific geometry Ω and using the finite element method implemented in Fenics, we can solve this problem and obtain the solution u^n at each time step n. From this,we can generate a graph that decribes the geometry of the problem and the solution, storing some values of interest as features of the nodes and the edges.By solving the problem for different geometries and different values of the parameters(e.g. the diffusivity constant) we can generate a dataset that will be used to train the graph neural network.

The graph neural network used in this project is the one presented in [2], which is an adaptation of the MeshGraphNet implementation [3], which we modified to make it suitable for our test case and extendable to other problems. The GNN is applied iteratively: at each time step it takes as input the system state Θ^n, which is the set of all the nodes and edges features at that time step, and it predicts an update for the state variables. 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 blocs, updating the node and edge features.
3. Decode: using a fully connected neural network, the node features are transformend 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 $\Theta^{n+1} = \Theta^n + \delta\Theta^n$.

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:

$$\begin{cases} \frac{\partial u}{\partial t} = k\Delta u \quad \text{in } \Omega \subset \mathbb{R}^2, \\ \frac{\partial u}{\partial n} = h \quad \text{on } \partial\Omega_{inlet}, \\ \frac{\partial u}{\partial n} = 0 \quad \text{on } \partial\Omega_{outlet} \cup \partial\Omega_{walls}. \end{cases} \tag{1}$$

where u is the temperature, k is the diffusivity constant, h is the Neumann condition at the inlet boundary. As domain Ω we consider different geometries such as the one shown in Figure : the 2D mesh is composed of 4 trapezoids where the interface between them have different lengths.

We generated 20 different mesh using gmsh. Then we solved the problem in Fenics using Discontinuous Galerkin method and implicitEuler 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 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 solution 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 thecorresponding trapezoid and the distance between the nodes connected by the edge.

3. Code

3.1. Mesh creation

3.2. Data generation

`GenerateData.py` contains two abstract classes: `Solver` and `DataGenerator`. The first one is used to solve the variational problem, while the second one is designed to store all the quantities of interest whichwill be used to build the graphs. Each of these parent classes has two child classes: we start describing the solver one.

The abstract base class `Solver` contains the following methods:

- `__init__(self,mesh)`: costructor that takes as input a `MeshLoader` object
- `set_parameters(self)`: abstract method
- `solve(self)`: abstract method
- `plot_solution(self)`: abstract method

All the abstract method are overriden in the child classes `Heat` and `Stokes`. The choice of a parent abstract class for the solver is useful because it allows to use the same code for different problems, implementingchild 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 overriden 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 plot the solution at each time step.
- `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 and imposing a Neumann conditiona at the inlet boundary. The solution at each time step is stored in a list, 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.

The second abstract base class `DataGenerator` contains the following methods:

- `__init__(self,solver,mesh)`: constructor that takes as input a `Solver` object and a `MeshLoader` object

- `flux(self)`: abstract method
- `inlet_flux(self)`: abstract method
- `area(self,tag)`: concrete method that computes the area of the trapezoid corresponding to the tag passed as input
- `nodes_data(self)`: this function save 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 list (numpy array) containing the values at each node.
- `td_nodes_data(self)`: abstract method
- `create_edges(self)`: it stores as attributes of the object two lists (`self.edges1` and `self.edges2`) containing respectively the nodes ID of the source nodes of every edge and the node ID 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.
- `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 ?
- `save_graph(self, fields_names, output_dir)`: 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 saves. It returns s dgl graph which is generated calling some functions defined in `generate_graphs.py`, that will be described in the next section.

As for the `Solver` class, the abstract methods are overriden in the two child classes `DataNS` and `DataHeat`. The `DataHeat` class includes these methods:

- `__init__(self,solver,mesh)`: constructor inherited from the parent class.
- `flux(self,tag,u)`: method overriden from the parent class, it computes the heat flux of the solution u at the interface corresponding to the tag passed as input.
- `inlet_flux(self,tag,u)`: method overriden from the parent class, it computes the heat flux at the inlet. The solution u and the tag of the inlet boundary are passe as input.
- `td_nodes_data(self)`: methos overriden from the parent class, 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 key the time instant and as value a numpy array containg the heat flux at that time instant at each node.
- `save_graphs`: method inherited from the parent class using the `super()` function, but the input `fields_names` is a list containing only the string `'flux'`.

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 computes the mean pressure at the interfaces and at the inlet and outlet boundaries respectively.

3.3. Graph generation

In this section we describe the functions defined in `generate_graphs.py` that are used to generate a dgl graph from the data obtained from the `DataGenerator` class. This file contains three functions:

- `generate_graph(point_data, points, edges_data, edges1, edges2)`: this function takes as input a dictionary containing the time independent node features, a list (o numpy array) with the node coordinates, a dictionary containing the edge features and two lists 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 and an offset with the number of time steps to skip. It returns the dgl graph with the new field added.
- `save_graph(filename, output_dir)`: function to save the dgl graph in a file in the output directory.

3.4. Graph Neural Network

4. Results

5. Further work

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] 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.

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