Node-element hypergraph message passing for mesh-based simulations

Rui Gao, Indu Kant Deo
Department of Mechanical Engineering
University of British Columbia
Vancouver, BC
{garrygao,indukant}@mail.ubc.ca

Abstract

A recent surge of research works apply graph neural networks for mesh-based continuum mechanics simulations. Most of these works operate on graphs in which each edge connects two nodes. A message-passing network defined on such a graph mimics the finite volume method in computational mechanics. Inspired by the data connectivity of another popular computational mechanics approach – the finite element method, we connect the nodes by elements rather than edges, effectively forming a hypergraph. A message-passing network is implemented on such a node-element hypergraph, and applied to the modeling of fluid systems of flow around a cylinder and flow around an airfoil. The experimental results show that such a message-passing network defined on the node-element hypergraph is able to generate more stable and accurate temporal roll-out predictions compared to the baseline defined on a normal graph. Along with adaptations in activation function and training loss, we expect this work to set a new strong baseline for future explorations of mesh-based simulations with graph neural networks.

1 Introduction

Deep neural networks defined on a graph data structure are becoming increasingly popular in a wide range of applications. Embedded with the relational inductive bias between inter-connected entities [1], graph neural networks (GNN) are well-suited for learning physics-driven dynamics. Recently, GNNs have been introduced to the field of continuum mechanics [2, 3]. Existing works include the modeling of solid/structural systems [4, 5], fluid systems [6, 7, 8], as well as the interactions between them [9].

Traditional computational mechanics approaches of continuum systems, including computational fluid dynamics (CFD) and computational solid mechanics (CSM), usually discretize the simulation domain into a mesh. It happens that such a mesh (like the one sketched in Fig. 1a) can be converted to a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ connected by edges $\mathcal{E}$ rather easily. To the best knowledge of the authors, two approaches of conversion exist. The first approach (Fig. 1b) converts each vertex of the mesh to a node, and each cell boundary between two vertices to two directed edges. The second approach (Fig. 1c) converts each cell within the mesh into a node, while each border between two neighboring cells is converted to two directed edges.

With the system states originally attached to the mesh converted and re-attached to the graph as the node and edge features, a graph neural network can be applied to the converted graph to learn the temporal evolution of these features, effectively serving as a surrogate to the traditional CFD/CSM model of the system. For a graph neural network that fits into the generalized graph message-passing [1, 10] framework, each message-passing layer/step can be written as the combination of an edge

Preprint. Under review.

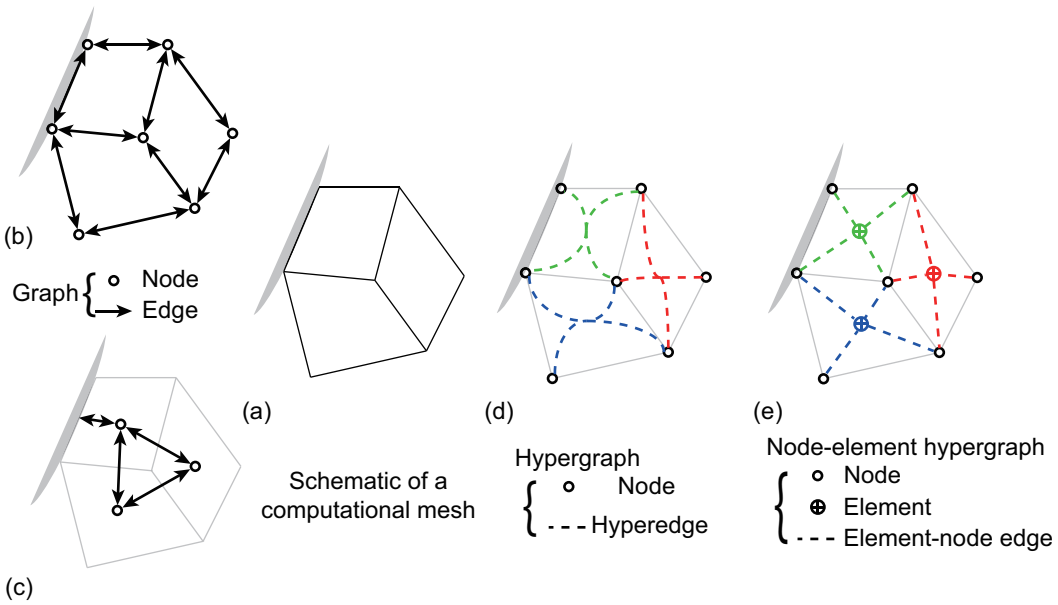


Figure 1: Different approaches to convert from a computational mesh to a graph. (a) Schematic of a computational mesh. (b) One approach to convert the mesh to a graph. (c) Another approach to convert the mesh to a graph. (d) Conversion to an undirected hypergraph (e) Conversion to an undirected node-element hypergraph, adopted in this work.

update stage

$$\mathrm{e}_{ij} \leftarrow \phi^{\mathrm{e}}(\mathrm{e}_{ij}, \mathrm{v}_i, \mathrm{v}_j), \tag{1a}$$

and a node update stage,

$$\mathrm{v}_i \leftarrow \phi^{\mathrm{v}}(\mathrm{v}_i, \mathrm{AGG}_j \mathrm{e}'_{ji}), \tag{1b}$$

in which v_i and v_j denotes the node features attached to the nodes i and j respectively, and e_{ij} denotes the edge feature attached to the directed edge pointing from node i to node j. The edge processor ϕ^{e} and the node processor ϕ^{v} are some functions, e.g., multi-layer perceptrons. The function AGG is an aggregation function that aggregates the information from all the edges pointing to each node i.

From a computational mechanics point of view, such a message-passing network defined on the graph $\mathcal{G}$ mimics the finite volume method. Interpreting each node as a control volume, and each edge as the boundary between neighboring control volumes, the edge update stage can be seen as calculating the "flux" of information between the two cells, and the node update stage can be seen as updating the information carried in each cell by all the incoming "flux". With a residual link for node and edge features, each layer/step within the message-passing network can be explained as one iteration in the iterative approximation to the ground truth flux and cell information updates.

Besides the finite volume method, many other approaches are available in computational mechanics. Another popular approach, the finite element method, treats each cell within the mesh as an *element* connecting all its vertices. During the computation, the vertices connected by each element interact with each other. Inspired by this data connectivity, we propose to connect the nodes $\mathcal{V}$ by elements $\mathcal{E}^+$, effectively forming a hypergraph, illustrated in Fig. 1d. Further modifications of the hypergraph (detailed in Sec. 3.1) lead to a node-element hypergraph (Fig. 1e).

We implement a message-passing network that is defined on the node-element hypergraph converted from mesh. The results from such a network are compared with the results of generalized message-passing network [1, 3] defined on a normal graph converted from the same mesh. Experiments on fluid systems of flow around a cylinder and flow around an airfoil show that such a message-passing network defined on the node-element hypergraph can generate more stable and accurate predictions than the baseline defined on the normal graph. With additional adjustments in the activation function and loss function, we set up a new, strong baseline for future explorations.

The remaining part of this report is organized as follows: Section 2 contains the discussions on the existing works that are closely related to this work. We then present the definitions and formulations

of the node-element graph and node-element graph message-passing networks in Sec. 3. The experiments and results are reported and discussed in Sec. 4. We conclude the work in Sec. 5.

2 Related works

GNN for physics simulations Graph neural network has been applied to the simulations of systems governed by physics laws for several years. The first applications focus on Lagrangian systems, like mass-spring systems [11] or fluid systems in which the continuous fluid flow is abstracted as moving particles [12]. More recently, it is introduced to the field of continuum mechanics [2, 3], inspiring a surge of works in the past two years. In particular, the encode-process-decode architecture adopted by Pfaff et al. [3] with generalized graph message-passing layers [1, 10] see popularity in application. Various techniques have been built above such an architecture, including field super-resolution [6], multi-grid methods [7, 8, 13, 14], physical invariances and/or equivariances [7], among many more.

FEM-Inspired GNN applications Several existing works have already taken some inspiration from finite element methods. Alet et al. [15], for example, constructs encoders and decoders that interpolate between data at random points within the field and data on the nodes of the node-edge graph, mimicking the behavior of shape function-based interpolation within the element in finite element methods. More recently, a few works start to use finite-element concepts to calculate the loss during the training process. Gao et al. [16], for example, proposed to calculate the loss by integrating the prediction error over the simulation domain using Gaussian quadrature integration with high-order shape functions instead of the traditional mean-squared loss on nodes.

Perhaps the most closely-related work to this research is the recent work by Lienen and Günnemann [17], who use a single hypergraph message-passing step to estimate the time derivatives of features at different time instants, which are then sent to an ODE solver to generate continuous predictions of the features. Different from their approach, we choose to follow an approach more similar to that of Pfaff et al. [3]: We stick to discrete time stepping with fixed intervals, the model is trained with one-step supervision only, and the model directly predict the difference of system states between neighboring time steps in a feed-forward manner rather than relying on an ODE solver. We also target generating stable and accurate predictions for a very long period of time – up to more than a thousand time steps, while the results reported by Lienen and Günnemann focus on predictions within a short period of future – 60 time steps at most.

3 Methodology

In this section, we describe the node-element hypergraph and the message-passing network defined on it. As the experiments reported in Sec. 4 are all 2D cases, we assume a 2D domain for simplicity.

3.1 Node-element hypergraph

Consider a bounded spatial domain that is meshed. The mesh can be converted to an undirected hypergraph (Fig. 1d) by converting each vertex within the mesh to a node, and each cell within the mesh to an undirected hyperedge connecting all vertices of the cell. Further instantiating each hyperedge as an (undirected) "element", and explicitly defining the connection between each element and each of the nodes it connects as an (undirected) element-node edge, we arrive at a node-element hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}_v)$, in which $\mathcal{V}$ is the set of all nodes, $\mathcal{E}^+$ is the set of all elements, and $\mathcal{E}_v$ is the set of all element-node edges.

It should be mentioned that it is possible to adopt a directed hypergraph rather than an undirected version. Ma et al. [18], for example, used a directed hypergraph for the simulation of particulate suspensions. However, to achieve permutation invariance, each undirected hyperedge connecting k nodes have to be converted to $k!$ directed hyperedges. This means that the computational cost can be prohibitively high when each hyperedge is connecting more than three nodes (e.g., hypergraphs converted from quadrilateral or hexagonal meshes in 2D, or hypergraphs converted from meshes in 3D). We, therefore, choose to stick to an undirected hypergraph in this work.

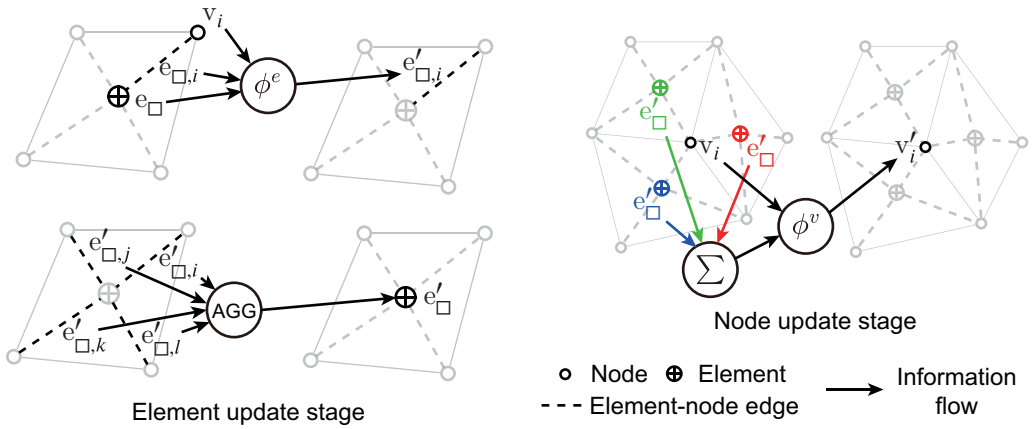


Figure 2: Schematic of the element and node update stages within each node-element hypergraph message-passing layer. (a) The element update stage. (b) The node update stage.

3.2 Node-element hypergraph message-passing

With the node-element hypergraph defined, we proceed to define the message-passing network on such a hypergraph. Similar to the generalized message-passing defined on a normal graph, one can define each message-passing layer/step on a node-element hypergraph as two consecutive stages, namely the element update stage and the node update stage. For the sake of consistency and simplicity in symbols, we write the equations in this subsection assuming a node-element hypergraph converted from the 2D quadrilateral mesh (with the conversion illustrated in Fig. 1).

For an element connecting four nodes with node features v_i, v_j, v_k, and v_l respectively, we can write the element update stage in a reasonably general form,

$$e_\square \leftarrow \phi^{e,e}\left(e_\square, \mathrm{AGG}_q^e\left(a_q \phi^{e,v}(v_q, e_\square, e_{\square,q})\right)\right), \tag{2a}$$

in which $e_\square$ is the element feature, $e_{\square,q}$ is the feature carried by the element-node edge between element $\square$ and node q, $q = i, j, k, l$, a_q is the (optional) attention weight, AGG^e denotes a permutation-invariant node aggregation function, and $\phi^{e,v}$ and $\phi^{e,e}$ are two functions that are preferably non-linear.

The subsequent node updating stage, in a similar level of generality, can be defined as

$$v_i \leftarrow \phi^{v,v}\left(v_i, \mathrm{AGG}_\square^v\left(a_i \phi^{v,e}(v_i, e_{i\square}, e_{i\square,i})\right)\right), \tag{2b}$$

in which the subscript $e_{i\square}$ denotes the element feature of any element that connects node i with some other nodes, $e_{i\square,i}$ denotes the corresponding element-node feature for node i within such an element, and $\phi^{v,e}$ and $\phi^{v,v}$ are two functions that are preferably non-linear. Similar to the element update step, the element aggregation function AGG^v should also be permutation-invariant.

In practice, these two update stages can be simplified significantly. In the experiments described in Sec. 4, we adopt simplified element and node update stages

$$e'_\square = \mathrm{AGG}_q\left(\phi^e(v_q, e_\square, e_{\square,q})\right), \tag{3a}$$

$$v'_i = \phi^v(v_i, \sum_\square e_{i\square}), \tag{3b}$$

which reduces the computational overhead to a level comparable to the generalized message-passing network defined in Eq. 1, and also rather easy to implement using a gather-scatter scheme similar to that in Pytorch Geometric [19].

An illustration of the two hypergraph message-passing stages described in Eq. 3 is attached in Fig. 2.

3.3 Model architecture

In general, we follow the same encode-process-decode architecture and forward Euler time stepping in the MeshGraphNet baseline [3]. At roll-out time step t_n, the neural network output $\hat{\psi}$ is added to

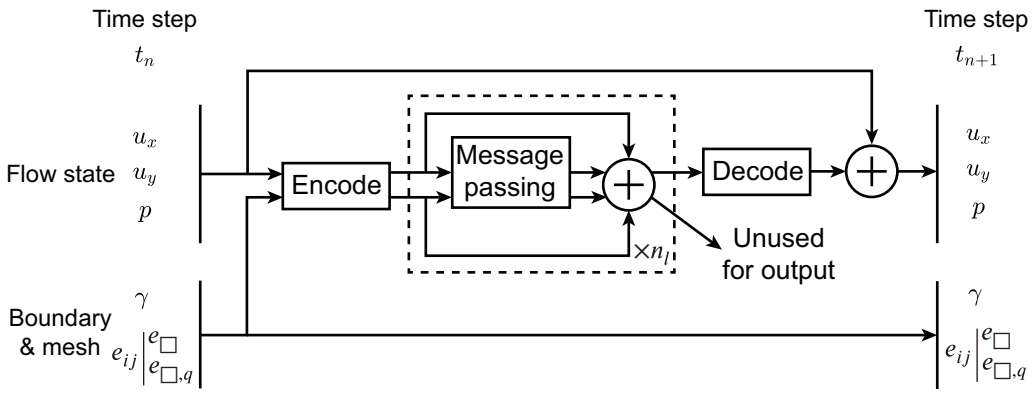


Figure 3: Schematic of the model architecture. The inputs are first encoded, then pass through $n_l = 15$ message-passing layers, and then the node features are decoded to generate the neural network output. Residual links are used for the node and edge/element features for the message-passing layers.

the state parameters $\boldsymbol{s}_n$ to generate the predicted state parameters $\hat{\boldsymbol{s}}_{n+1}$ for time step t_{n+1},

$$\boldsymbol{s}_{n+1} \approx \hat{\boldsymbol{s}}_{n+1} = \boldsymbol{s}_n + \hat{\psi} \tag{4}$$

The neural network itself includes an encoder, a series of message-passing layers, and a decoder, stacked together in a feedforward fashion. The encoder, decoder, and the processors in the message-passing layers are selected to be multi-layer perceptrons in this work for the sake of consistency with the baseline, but it should be noted that other choices are also possible.

The model architecture is summarized in Fig. 3.

3.4 Choice of activation function

Most of the existing works discussed in Sec. 2 use Rectified linear Units (ReLU) or ReLU-like activation functions. We notice that the use of the sinusoidal activation function [20] improves the accuracy and smoothness of the captured details compared to ReLU in shape representation tasks. It is therefore reasonable to hypothesize that it should also help in capturing smooth and accurate system dynamics when used as the activation function for the encoders, decoders, and processors in the graph neural network. In Sec. 4, we report the results for both when ReLU/ReLU-alike activation functions are used, and when sinusoidal activation functions are used.

3.5 Adaptive smooth L1 loss

Most of the existing works discussed in Sec. 2 adopt mean squared error (MSE) as the training loss. When training the node-element hypergraph with sinusoidal activation functions, however, we found that the use of MSE loss leads to instability in the training process. We therefore seek an alternative training loss. Another typical loss, the L1 loss, might not be preferable in this case since it is not smooth at zero, and we therefore adopt a smooth-L1 loss [21] function, which states that for ground truth ψ and its neural network prediction $\hat{\psi}$, the loss

$$L_i(\psi_i, \hat{\psi}_i) = \begin{cases} \frac{(\psi_i - \hat{\psi}_i)^2}{2\beta}, & \text{if} \quad |\psi_i - \hat{\psi}_i| < \beta \\ \left|\psi_i - \hat{\psi}_i\right| - \frac{\beta}{2}, & \text{if} \quad |\psi_i - \hat{\psi}_i| \geq \beta \end{cases} \tag{5}$$

in which β is a non-negative parameter that controls the transition point between the L2 loss region and the L1 loss region. The subscript i here denotes entry-by-entry calculation. A fixed β value is not preferable, since the loss function is not very different from the L1 loss when β is too small and will converge to a scaled L2 loss during training when β is too large. For the present case, the instability in training occurs when the training MSE error is relatively low, so an adaptive scheme for β is preferred. Assuming that the distribution of error during the training (approximately) follows a symmetric distribution centered at zero, the target is to make sure that the two tails of the distribution fall into the L1 loss region so that they do not lead to instability. More complex on-the-fly β adaptation algorithms

like references [22, 23, 24] exist, but we choose to control it in this work using a simpler approach by setting β^2 as the variance of the model prediction error

$$\beta^2 = \mathrm{Var}(\psi - \hat{\psi}) \approx \mathrm{MSE}(\psi_{train} - \hat{\psi}_{train}) \tag{6}$$

based on the idea that at least part of the two tails of any zero-centered symmetric distribution would reside more than one standard deviation away from zero. The variance of prediction error is approximated by computing the MSE error between the predicted and ground truth value of the whole training set, which can be further approximated on-the-fly by an exponential moving average

$$\beta^2 \leftarrow (1 - \frac{1}{N_b})\beta^2 + \frac{1}{N_b}\,\mathrm{MSE}(\psi_{batch} - \hat{\psi}_{batch}) \tag{7}$$

in which N_b is the total number of training steps within each epoch. We further stabilize the process by preventing β from increasing, i.e.,

$$\beta^2 \leftarrow (1 - \frac{1}{N_b})\beta^2 + \frac{1}{N_b}\,\min\{\beta^2, \mathrm{MSE}(\psi_{batch} - \hat{\psi}_{batch})\} \tag{8}$$

for each training step.

4 Experiments

In this section, we apply the network described in Eq. 3 to the modeling and prediction of two fluid systems. In the subsequent subsections, we will first briefly describe the data sets used, then present the details on the setup of the proposed and baseline models, and finally demonstrate the results. All experiments are performed with all random seeds fixed at 1.

4.1 Experimental setup

Data sets We choose two typical fluid flow systems for the experiments: The flow around a circular cylinder and the flow around an airfoil. The flow around the cylinder is simulated at Reynolds number $Re = 200$, and used to test the capability of the neural network in learning a certain dynamic without overfitting to it. The flow around the airfoil is simulated at multiple Reynolds numbers (with the flow at each Reynolds number forming a separate "trajectory") within the range $Re \in [1000, 4000]$, and used to test the capability of the network to interpolate within a range of dynamics and extrapolate out of the range. Both flow data sets are generated via a finite element solver written in Matlab. The simulated flow data are interpolated onto a coarser mesh before subsequent conversion to graph/hypergraph. More details on the data sets used are provided in appendix A.

Models & implementation We choose the state-of-the-art MeshGraphNet [3] as our baseline model. Apart from the baseline, the performance of three other models are evaluated and reported: The alternative MeshGraphNet with activation function changed to a sine function (abbreviated as *Node-edge-sin*), the node-element hypergraph message-passing network with Relu-like activation function (*Node-elem*), and the node-element hypergraph message-passing network with sinusoidal activation function (*Node-elem-sin*). All models are implemented in PyTorch [25]. The message-passing layers are implemented through a gather-scatter scheme similar to that in PyTorch Geometric [19]. Different from the practice in the original MeshGraphNet, we follow the adaptation suggested by Lino et al. [7] to temporally evolve the pressure along with the velocity.

Training We train all models with an Adam [26] optimizer for a total of 207 epochs with batch size 4. Mean-squared loss is used when ReLU-alike activation functions are used following the practice in the MeshGraphNet baseline, while adaptive smooth L1 loss described in Sec. 3.5 is used when the sinusoidal activation function is used. It should be emphasized that we choose to NOT use any training noise for all the cases reported, as it is too computationally expensive to optimize this hyperparameter for a fair comparison.

More implementation and training details are attached in appendix B.

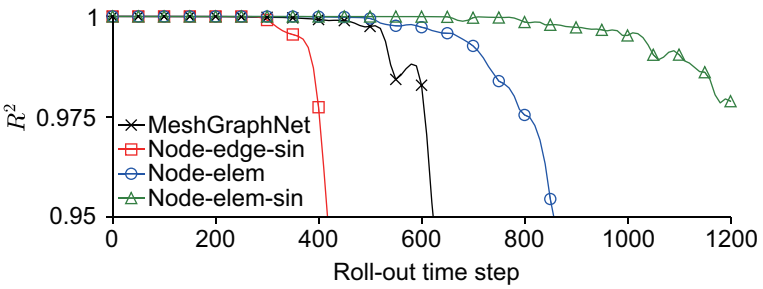


Figure 4: Coefficient of determination R^2 of the predicted pressure field over 1200 steps for the flow around cylinder data set.

4.2 Evaluation metric

The trained models are evaluated on the test data sets. It should be noted that no cross-validation data sets are needed as we do not tune the hyperparameters, but rather follow the choices of the MeshGraphNet baseline. As the main purpose of the neural network surrogate model is to generate accurate roll-out simulations, we evaluate the models by feeding the state of the system at a certain time step (the first time step within each of the test data sets) to the model and compare the predicted system states over the next hundreds of time steps with the ground truth. Specifically, we quantitatively compare the similarity between the ground truth non-dimensionalized pressure field $\boldsymbol{p}^*$ and the predicted pressure field $\hat{\boldsymbol{p}}^*$ over the predicted time steps by calculating the coefficient of determination

$$R^2 = 1 - \frac{\|\boldsymbol{p}^* - \hat{\boldsymbol{p}}^*\|_2^2}{\|\boldsymbol{p}^* - \overline{\boldsymbol{p}^*}\|_2^2}, \tag{9}$$

in which the overline $\overline{(\cdot)}$ denotes the mean operation. A higher coefficient of determination indicates a more accurate prediction, up to $R^2 = 1$ which means perfectly accurate predictions.

4.3 Results and discussions

Starting from the first time step in each of the test data sets, the models generate roll-out predictions of the states of the fluid systems over the future time steps. For the flow around cylinder data set, predictions for 1200 future time steps are generated. For the flow around airfoil data set, predictions for 800 future time steps are generated for each Reynolds number. In this subsection, we report these evaluation results, using the metric described in Sec. 4.2.

Learning a certain flow dynamic Figure 4 shows the coefficient of determination of the predicted non-dimensionalized pressure field $\hat{\boldsymbol{p}}^*$ for the flow around cylinder data set. It is clear that the node-element hypergraph message-passing networks are able to produce stable and accurate predictions for a longer period of time compared to the baseline MeshGraphNet. The use of the sinusoidal activation function proves to be helpful for the node-element hypergraph message-passing network, verifying the hypothesis in Sec. 3.4. In the meantime, we notice that the opposite is true for the generalized message-passing network on a normal graph. As the training mean-squared error for both architectures reaches about 4×10^{-9} when the sinusoidal activation function is used, we conclude that the message-passing network on a normal graph is more prone to overfitting when used to learn a certain flow dynamic.

Interpolation & extrapolation Figure 5 shows the coefficient of determination of the predicted non-dimensionalized pressure field $\hat{p}^*$ for the flow around airfoil data sets at different Reynolds numbers. Note that the flow at each testing Reynolds number (i.e., each testing "trajectory") is tested separately, using the system state of the first time step of each testing "trajectory" as the input to the model. The Reynolds number range of the training 'trajectories' is marked out. Within the interpolation range, we observe that the node-element hypergraph message-passing networks are able to generate accurate roll-out predictions for a longer period of time.

In the meantime, all models do not perform well in extrapolating out of the training range. This is within expectation, especially for MeshGraphNet, since all of its components are linear or piecewise

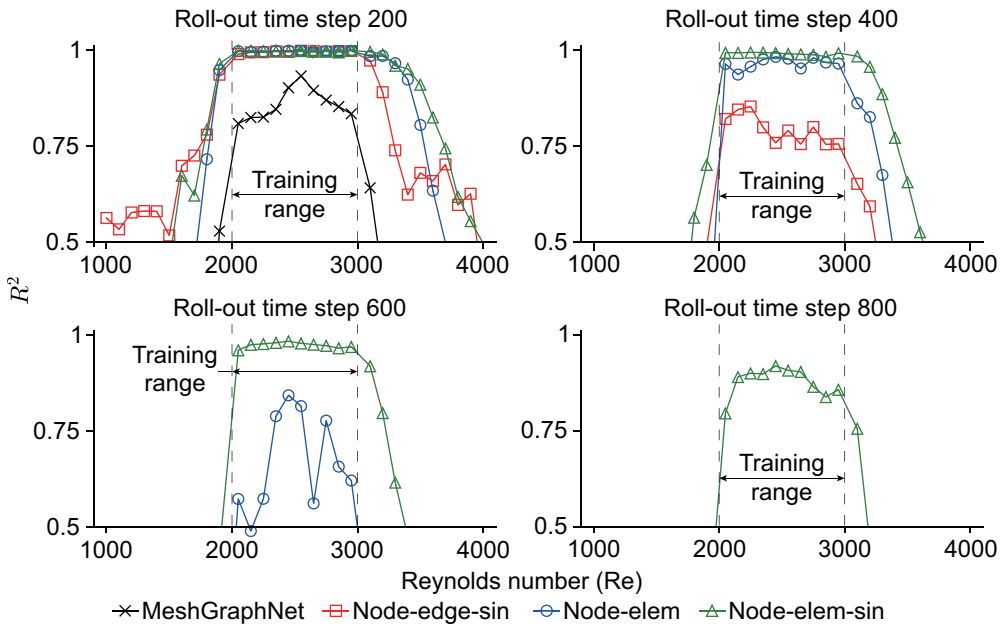


Figure 5: Coefficient of determination R^2 of the predicted pressure field over the prediction roll-out for the flow around airfoil data set for all testing Reynolds numbers, sampled at roll-out time step 200, 400, 600, and 800.

linear functions, which means that the network output $\hat{\psi}$ is a piecewise linear function of its inputs. Empirical results [27] show that this kind of network extrapolates linearly outside of the training range. Since the dynamics of a fluid flow system, governed by the Navier-Stokes equations, is not linear, the network is not expected to be able to extrapolate outside of the training Reynolds number range. The same conclusion is also approximately (and only approximately as GeLU activation function is used) true for *node-elem* model. For models with sinusoidal activation functions, the extrapolation pattern is not completely clear, but the results in Fig. 5 show that they are also not able to extrapolate well.

Additional results on the inference speed of the models are reported in Appendix C.

5 Conclusion

Targeting Eulerian mesh-based simulations with GNN, we propose to convert the computational mesh to a node-element hypergraph rather than a normal graph. We implement a message-passing network on such a hypergraph and showed that the network is able to generate stable and accurate roll-out predictions for a longer period of time compared with the baseline defined on a normal graph. We further demonstrate that the use of sinusoidal activation functions is preferable compared with ReLU-like activation functions.

As the proposed network architecture only changes the graph connectivity and message-passing strategy at individual graph levels, it should be compatible (after minor modifications) with most existing techniques like multi-grid methods and physical invariance/equivariance that were originally built upon graph neural networks. This means that we have established a new, stronger baseline for future research efforts in mesh-based simulations with GNN.

It should be noted that the current model only takes inspiration from the graph connectivity of the finite element method, but would it be possible to construct a network architecture that is strictly a neural network version of the finite element method (i.e., a step further from this work and the work by Lienen and Günnemann [17])? In our future work, we plan to explore such possibility.

Distribution of work

The group shares the workload of the project. Rui Gao generated the initial idea, and finalized the methodology in the discussion with Indu Kant Deo. Rui Gao coded the experiments. Indu Kant Deo set up the Compute Canada environment for running the cases. The group collaborated on making the presentation slides. Rui Gao wrote the initial versions of the project proposal and the final report, which are later revised by Indu Kant Deo.

Acknowledgments and Disclosure of Funding

This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Seaspan Shipyards, and performed under the supervision of Dr. Rajeev Jaiman. The training and evaluation of the neural network models were supported in part by the computational resources and services provided by Advanced Research Computing at the University of British Columbia. Dr. Renjie Liao and Xiaoyu Mao provided numerous suggestions to the work, their help is hereby gratefully acknowledged.

References

[1] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[2] Filipe De Avila Belbute-Peres, Thomas Economon, and Zico Kolter. Combining differentiable pde solvers and graph neural networks for fluid flow prediction. In *international conference on machine learning*, pages 2402–2411. PMLR, 2020.

[3] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

[4] Roberto Perera, Davide Guzzetti, and Vinamra Agrawal. Graph neural networks for simulating crack coalescence and propagation in brittle materials. *Computer Methods in Applied Mechanics and Engineering*, 395:115021, 2022.

[5] Tianju Xue, Sigrid Adriaenssens, and Sheng Mao. Learning the nonlinear dynamics of soft mechanical metamaterials with graph networks. *arXiv preprint arXiv:2202.13775*, 2022.

[6] Jiayang Xu, Aniruddhe Pradhan, and Karthikeyan Duraisamy. Conditionally parameterized, discretization-aware neural networks for mesh-based modeling of physical systems. *Advances in Neural Information Processing Systems*, 34:1634–1645, 2021.

[7] Mario Lino, Stathi Fotiadis, Anil A Bharath, and Chris D Cantwell. Multi-scale rotation-equivariant graph neural networks for unsteady eulerian fluid dynamics. *Physics of Fluids*, 34(8):087110, 2022.

[8] Zhishuang Yang, Yidao Dong, Xiaogang Deng, and Laiping Zhang. Amgnet: multi-scale graph neural networks for flow field prediction. *Connection Science*, 34(1):2500–2519, 2022.

[9] Rui Gao and Rajeev K Jaiman. Quasi-monolithic graph neural network for fluid-structure interaction. *arXiv preprint arXiv:2210.04193*, 2022.

[10] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pages 4470–4479. PMLR, 2018.

[11] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29, 2016.

[12] Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B Tenenbaum, and Antonio Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv preprint arXiv:1810.01566*, 2018.

[13] Meire Fortunato, Tobias Pfaff, Peter Wirnsberger, Alexander Pritzel, and Peter Battaglia. Multiscale meshgraphnets. *arXiv preprint arXiv:2210.00612*, 2022.

[14] Yadi Cao, Menglei Chai, Minchen Li, and Chenfanfu Jiang. Bi-stride multi-scale graph neural network for mesh-based physical simulation. *arXiv preprint arXiv:2210.02573*, 2022.

[15] Ferran Alet, Adarsh Keshav Jeewajee, Maria Bauza Villalonga, Alberto Rodriguez, Tomas Lozano-Perez, and Leslie Kaelbling. Graph element networks: adaptive, structured computation and memory. In *International Conference on Machine Learning*, pages 212–222. PMLR, 2019.

[16] Han Gao, Matthew J Zahr, and Jian-Xun Wang. Physics-informed graph neural galerkin networks: A unified framework for solving pde-governed forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering*, 390:114502, 2022.

[17] Marten Lienen and Stephan Günnemann. Learning the dynamics of physical systems from sparse observations with finite element networks. *arXiv preprint arXiv:2203.08852*, 2022.

[18] Zhan Ma, Zisheng Ye, and Wenxiao Pan. Fast simulation of particulate suspensions enabled by graph neural network. *Computer Methods in Applied Mechanics and Engineering*, 400:115496, 2022.

[19] Matthias Fey and Jan E Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[20] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

[21] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[22] Cheng-Yang Fu, Mykhailo Shvets, and Alexander C Berg. Retinamask: Learning to predict masks improves state-of-the-art single-shot detection for free. *arXiv preprint arXiv:1901.03353*, 2019.

[23] Hongkai Zhang, Hong Chang, Bingpeng Ma, Naiyan Wang, and Xilin Chen. Dynamic r-cnn: Towards high quality object detection via dynamic training. In *European conference on computer vision*, pages 260–275. Springer, 2020.

[24] Arief Rachman Sutanto and Dae-Ki Kang. A novel diminish smooth l1 loss model with generative adversarial network. In *International Conference on Intelligent Human Computer Interaction*, pages 361–368. Springer, 2021.

[25] 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.

[26] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. *arXiv preprint arXiv:2009.11848*, 2020.

[28] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331, 2009.

[29] Rajeev K Jaiman, M Z Guan, and Tharindu P Miyanawala. Partitioned iterative and dynamic subgrid-scale methods for freely vibrating square-section structures at subcritical reynolds number. *Computers & Fluids*, 133:68–89, 2016.

[30] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J van der Walt, Matthew Brett, Joshua Wilson, K Jarrod Millman, Nikolay Mayorov, Andrew R J Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E A Quintero, Charles R Harris, Anne M Archibald, Antônio H Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[31] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[32] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

A Details of the data sets used

Two data sets are used for the evaluation of the model. In this appendix, we provide the detailed description of these data sets.

A.1 Data generation, train-test split

Both flow data sets are 2-D incompressible fluid flow simulated at laminar flow condition. The computational meshes are drawn with Gmsh [28]. The momentum equations are solved using a Petrov-Galerkin finite element solver with a semi-discrete time-stepping scheme [29] written in Matlab. The domain for the two sets of simulations are plotted in Fig. 6. The inlet features a uniform flow $u_x = U_\infty = 1, u_y = 0$ condition, the outlet Γ_{out} is set to be traction free, while top and bottom of the boundary conditions Γ_{top} and Γ_{bottom} are set to be slip-wall. The computational meshes are shown in Fig. 7a and Fig. 8a. For the flow around cylinder, a total of 6499 continuous time steps are sampled with non-dimensionalized time step $\Delta t^* = 0.04$ at Reynolds number $Re = 200$. For the flow around airfoil, flow data are sampled at Reynolds number $Re = \{1000, 1100, \ldots, 2000, 2033.33, 2050, 2066.67, 2100, 2133.33, \ldots, 3000, 3100, \ldots, 4000\}$. For each of the sampled Reynolds number, 4500 continuous time steps are sampled with non-dimensionalized time step $\Delta t^* = 0.0167$.

For the neural network, we use coarser meshes (also generated by Gmsh) for both cases, plotted in Fig. 7b and 8b. It should be noted that the meshes for the flow around cylinder are the same as the one used for the fluid-structure interaction between fluid flow and an elastically mounted cylinder in reference [9]. The statistics about the number of nodes, edges and elements for the two data sets are reported in Table 1. The simulated flow data on the dense CFD mesh are interpolated onto the coarse neural network mesh via a clough-tocher interpolator available in SciPy package [30]. The two data sets after interpolation are publically available at `https://drive.google.com/drive/folders/17sLVTbcDP5Y5-x4FcHumTbaBtR5xyxNj?usp=sharing`. The original CFD data sets are also available on request.

Table 1: Statistics of the converted graph/hypergraph for the data sets used

Data set	Nodes	Edges	Elements	Element-node edges
Cylinder	2204	8728	2160	8640
Airfoil	3653	14486	3590	14360

The interpolated data sets are then split into train and test data sets, as listed in Table 2. Separate cross-validation data sets are not necessary, since we do not fine-tune the hyperparameters of the model, but rather follow the choices in the MeshGraphNet baseline [3].

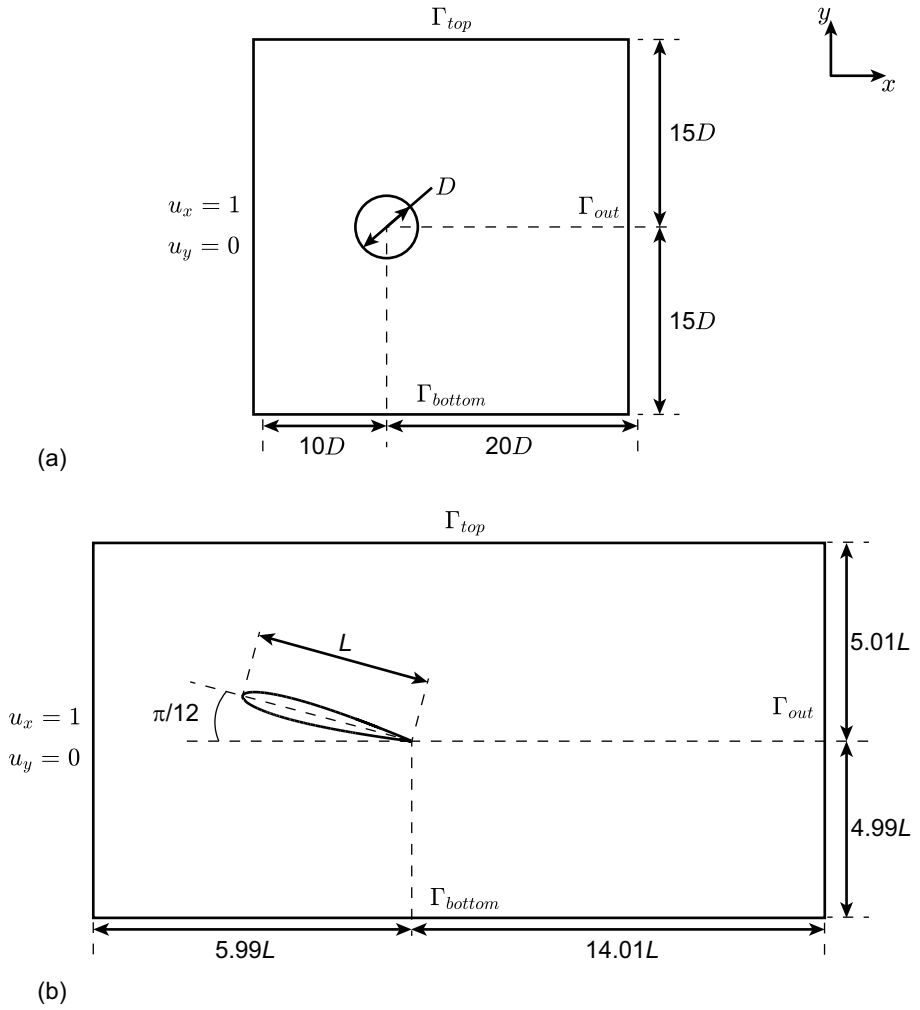


Figure 6: Schematic of the computational domain for the two data sets. (a) Flow around cylinder (b) Flow around airfoil.

Table 2: Train-test split of the data sets. The $a : b : c$ notation denotes a series of values starting from a and ends at c, with interval b.

Data set	Training Reynolds number(s)	Testing Reynolds number(s)	Time steps per training trajectory	Time steps per testing trajectory
Cylinder	200	200	2048	1200
Airfoil	2000:$\frac{100}{3}$:3000	1000:100:1900 2050:100:2950 3100:100:4000	512	800

A.2 Graph feature attachment

After being interpolated onto the coarser mesh, the flow data are then converted and re-attached to the graph/hypergraph that is converted from the coarse mesh (cf. Fig. 1). As the two data sets used in this work are both fluid flow data sets, they share the same feature attachment procedure. For features on the normal graph (the MeshGraphNet baseline), the node features are the velocity and pressure, as well as the boundary condition (encoded as a one-hot vector):

$$v_i = [u_{x,i}, u_{y,i}, p_i, \gamma_i], \tag{10}$$

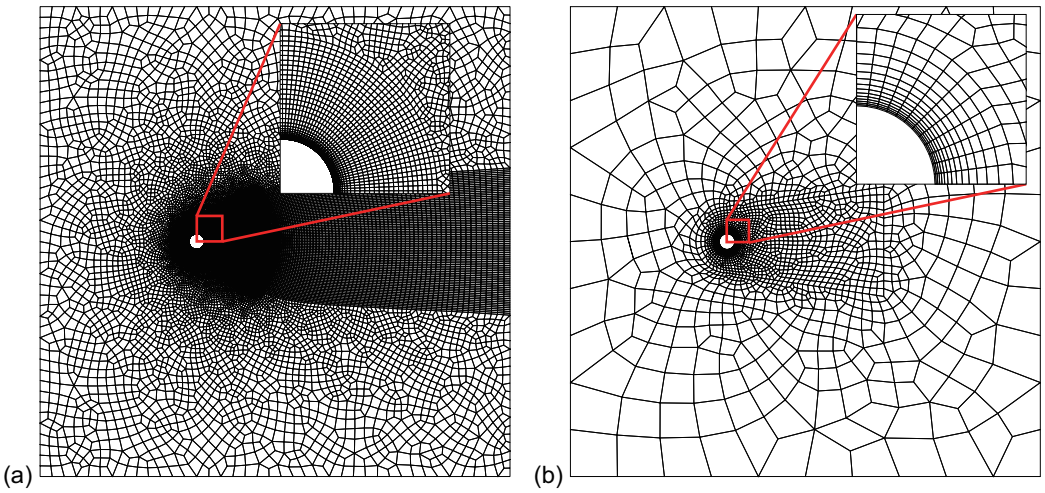


Figure 7: Schematic of the mesh used for the flow around cylinder case. (a) Computational mesh for the CFD solver. (b) Coarser mesh to be converted to graph/hypergraph for neural network.

The edge features encode the relative location information

$$e_{ij} = [x_i - x_j, y_i - y_j, \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}], \tag{11}$$

For the node-element hypergraph, the node features are the same as that of the normal graph. The features on the element-node edges are used to encode the relative location information. For an element connecting four nodes i, j, k, and l, the element-node features

$$e_{\square,p} = \left[x_p - \frac{1}{4} \sum_{r=i,j,k,l} x_r, y_p - \frac{1}{4} \sum_{r=i,j,k,l} y_r \right] \tag{12}$$

for $p = i, j, k, l$. The features on each element encode the area of the corresponding cell $S_\square$,

$$e_\square = [S_\square, -S_\square] \tag{13}$$

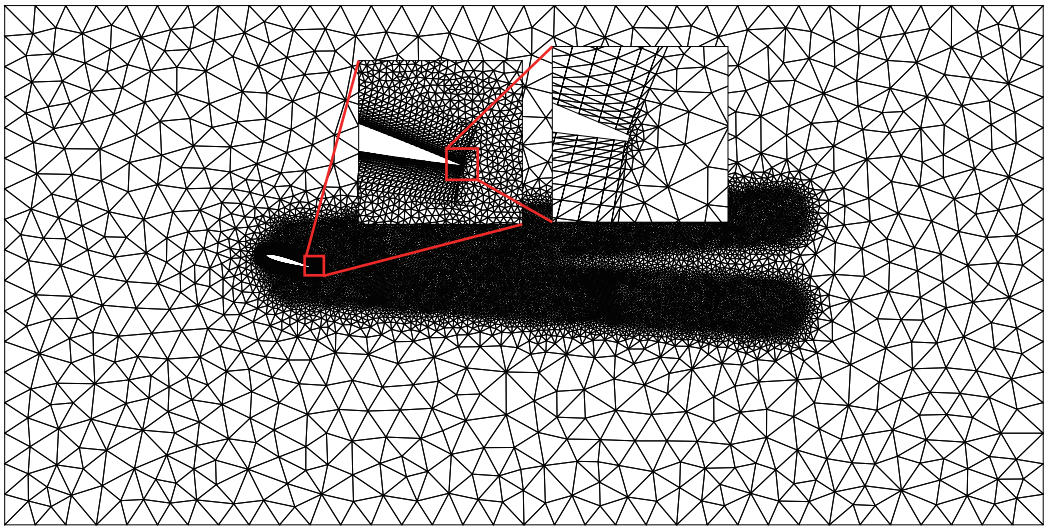
To avoid a feature vector of length 1, we augment the feature vector by concatenating it with its negative vector.

B Details of implementation and training

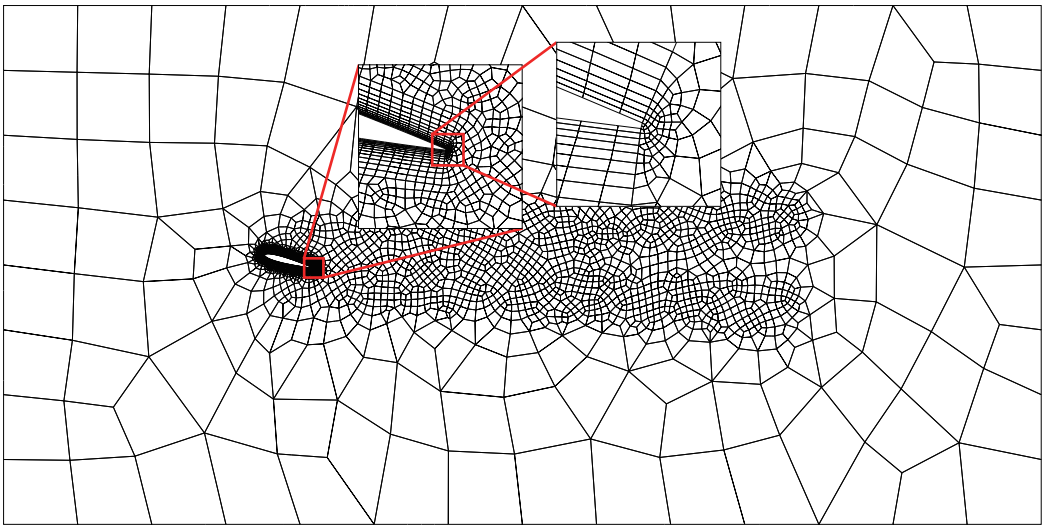
We use PyTorch [25] to implement all the models. In all reported message-passing models, all MLPs (encoders, decoders, processors) have two hidden layers with hidden layer width 128. A total of 15 message-passing layers are used for all reported models. Residual links are added for all message-passing layers. In order to improve convergence, layer normalization [31] is added at the end of all encoders and processors for models with ReLU-like activation functions. All trainings are performed with all random seeds fixed at 1 and batch size 4, using an Adam [26] optimizer with PyTorch default setup, and lasts for 207 epochs. In the warmup stage (first 7 epochs), the momentum of the Adam optimizer is reset at the start of each epoch. The detailed training scheme is reported in Table 3. All trainings and evaluations are completed on a single Nvidia RTX 3090 GPU with CPU being AMD Ryzen 9 5900 @ 3 GHz $\times$ 12 cores.

For the MeshGraphNet baseline, we translate from the TensorFlow implementation provided by Pfaff et al. [3]. It should be noted that our implementation is different from the original version in that:

1. We no longer gather normalization statistics (mean and standard deviaton of inputs and outputs) on-the-fly, but rather use the whole training data set to directly calculate them in pre-processing stage.
2. We adopt the modification suggested by Lino et al. [7] to also include pressure in the temporal roll-out, in addition to the velocity. Due to this modification, the outputs of the neural network at the domain boundaries are now included in the calculation of loss.



(a)



(b)

Figure 8: Schematic of the mesh used for the flow around airfoil case. (a) Computational mesh for the CFD solver. (b) Coarser mesh to be converted to graph/hypergraph for the graph neural network.

3. The results reported, except specifically mentioned, are generated from models trained without the use of training noise. This is because we notice that the optimal amount of training noise varies with the data sets and networks used, leading to another hyper-parameters that is too computationally expensive to optimize for fair comparison.
4. The training scheme is slightly different from the original implementation. We add a learning rate warm up stage of 7 epochs, and the learning rate do not decay immediately from the start. The max learning rate (10^{-4}) and min learning rate (10^{-6}) are kept the same as original training scheme in MeshGraphNet. We apply the same training scheme for all reported models.

For the node-element message-passing with ReLU-like activation functions, we use Gaussian Error Linear Units (GELU) [32] activation function rather than ReLU in the MeshGraphNet baseline. This is because MeshGraphNet gives better results with ReLU compared to GELU, while the reverse is true for node-element hypergraph message-passing networks.

For the message-passing networks with sinusoidal activation function, we use the sine activation [20] along with the specific initialization schemes described in the work.

Table 3: Neural network training scheme

Stage	Batch size	Epochs	Starting learning rate	Learning rate decay per epoch	Reset momentum at every epoch
1	4	7	10^{-6}	2.152	True
2		50	10^{-4}	1	False
3		100	10^{-4}	0.955	False
4		50	10^{-6}	1	False

Table 4: Inference time per prediction roll-out step

Data set	MeshGraphNet	Node-edge-sin	Node-elem	Node-elem-sin
Cylinder	$\approx$ 13.6 ms	$\approx$ 15.9 ms	$\approx$ 14.3 ms	$\approx$ 17.2 ms
Airfoil	$\approx$ 14.3 ms	$\approx$ 16.7 ms	$\approx$ 14.9 ms	$\approx$ 15.4 ms

The message-passing layers are implemented in a gather-scatter scheme, which is similar to that in PyTorch Geometric [19] but not exactly the same. Due to the use of the GPU atomic operation in the scatter functions, the training and evaluation processes are not deterministic even when all random numbers are fixed, and some differences are expected between the network parameters obtained from running the provided code and the trained network parameters provided.

C Additional results

Inference speed Theoretically, the computational complexity of the models in concern should be similar, as a generalized graph message-passing layer and a node-element hypergraph message-passing layer require almost the same number of gather functions, scatter functions, and MLP evaluations. The actual inference time cost per step for both data sets during evaluation, reported in Table 4, confirms this hypothesis. The node-element hypergraph message-passing networks run slightly slower than the graph message-passing layers, probably due to the fact that there is a data copying step in the implementation of node-element hypergraph message-passing layer that we found unnecessary in theory, but difficult to circumvent in practice. When using sinusoidal activation functions, the networks generally run slightly slower (except Node-elem-sin for airfoil data sets), probably due to the fact that the evaluation of sine function is slower than ReLU or ReLU-like activation functions.