TUM Data Innovation Lab

Munich Data Science Institute (MDSI)

Technical University of Munich

&

TUM Chair of Aerodynamics and Fluid Dynamics

Final report of project:

Equivariant GNN-Based Multi-Resolution Simulator for Particle-Based Fluid Mechanics

Authors	Kiran Bohaju, Marios Demestiades, Rundong Huang, Claudio Longo, Harish Ramachandran, Paul Schlack
Mentor	M.Sc Artur Toshev
Project Lead	Dr. Ricardo Acevedo Cabra (MDSI)
Supervisor	Prof. Dr. Massimo Fornasier (MDSI)

Jul 2023

Abstract

Particle-based fluid mechanics simulations play a crucial role in understanding and predicting complex fluid dynamics phenomena. However, the computational challenges posed by the multi-scale nature of these phenomena demand innovative approaches to enhance efficiency and accuracy. In this project, we aim to explore the potential of equivariant Graph Neural Networks (GNNs) in a multi-resolution framework for particle-based fluid mechanics simulations.

Our machine learning pipeline, developed from scratch to be independent of any specific deep learning framework, offers flexibility and compatibility with various backends. We aim to incorporate E(3) equivariant inputs, our model learns to exploit the rotational and translational symmetries inherent in fluid dynamics, leading to more meaningful representations and improved accuracy in predicting fluid behavior.

A key feature of our pipeline is the multi-resolution component, capable of handling varying numbers of particles in the temporal dimension while maintaining static shapes. This allows for the effective processing of fluid dynamics simulations with different levels of detail, enhancing adaptability and accuracy. Specialized data structures tailored specifically for fluid dynamics applications optimize performance and seamlessly integrate with the model.

Furthermore, we integrate a classical multi-resolution Smoothed Particle Hydrodynamics (SPH) solver into our framework. This provides a baseline for validation and benchmarking, enabling comparisons with machine learning-based predictions and further enhancing our understanding of fluid dynamics phenomena.

Looking ahead, our work opens avenues for further exploration and innovation in the field of fluid dynamics prediction with machine learning. The optimization of performance by computing the connectivity list directly on the GPU and exploring new neighborhood computation algorithms holds promise for improving computational efficiency. Additionally, computing coarse scores directly from latent information and investigating the use of steerable Multi Layer Perceptrons (MLPs) present exciting directions for future research and development.

Contents

1 Introduction

Problem Definition and Goals of the Project In the realm of particle-based fluid mechanics simulations, the utilization of equivariant Graph Neural Network techniques in a multi-resolution framework holds immense promise for advancing the efficiency and accuracy of complex physical systems' representations. From stellar collisions with sparsely populated regions to multi-phase flows encountered in additive manufacturing, particle-based Computational Fluid Dynamics (CFD) has emerged as the preferred approach. However, the multi-scale nature of these phenomena poses computational challenges, demanding substantial resources due to the vast number of particles required to resolve all relevant spatial and temporal scales.

To tackle these challenges, Coarse-Graining (CG) has been recognized as a potential solution. By employing larger particles, CG enables more extensive time steps while respecting the Courant-Friedrichs-Lewy (CFL) condition, leading to faster simulations without sacrificing accuracy. Remarkably, similarities in algorithmic principles between particle-based CFD and Molecular Dynamics (MD) simulations have been identified, paving the way for the adaptation of GNN-based CG techniques recently demonstrated in molecular simulations [1].

While [1] focused on CG in molecular simulations, the current work aims to extend their advancements to the field of fluid mechanics. The proposed equivariant GNN-based multi-resolution simulator seeks to harness the inherent algorithmic similarities between particle-based CFD and MD, providing a novel approach to accelerate fluid dynamics simulations. Moreover, the conventional uniform coarse-graining approach treats all regions equally, which may not fully exploit the potential efficiency gains. To address this limitation, this project explores the concept of multi-resolution coarse-graining, selectively refining regions with large variations while coarsening those with homogeneous properties. This strategy aims to optimize both efficiency and accuracy in fluid mechanics simulations, promising significant advancements in studying complex phenomena across varying scales.

This project presents a framework, leveraging equivariant GNN techniques within a custom multi-resolution framework for particle-based fluid mechanics simulations. By fusing these cutting-edge methodologies, we envision substantial improvements in computational efficiency and accuracy.

In the traditional multi-resolution SPH (MR-SPH) solver , several challenges may arise. Firstly, the determination of appropriate resolution levels for different regions of the fluid domain poses a significant hurdle. Achieving an optimal balance between accuracy and efficiency requires careful consideration, as overly coarse regions may lead to loss of critical details, while excessively fine regions may lead to unnecessary computational overhead. Secondly, maintaining accuracy in the transition between different resolution levels is another key challenge. Handling particle interactions across boundaries where the resolution changes can introduce errors or artifacts, if not appropriately managed. Finally, accounting for dynamic changes in fluid properties and evolving flow features throughout the simulation presents an ongoing challenge, as regions initially considered homogeneous may become heterogeneous over time. Addressing these challenges requires sophisticated algorithms to fully exploit the potential benefits of a multi-resolution approach while delivering accurate and reliable results in particle-based fluid mechanics simulations.

Our primary objective was to develop a machine learning pipeline from scratch that

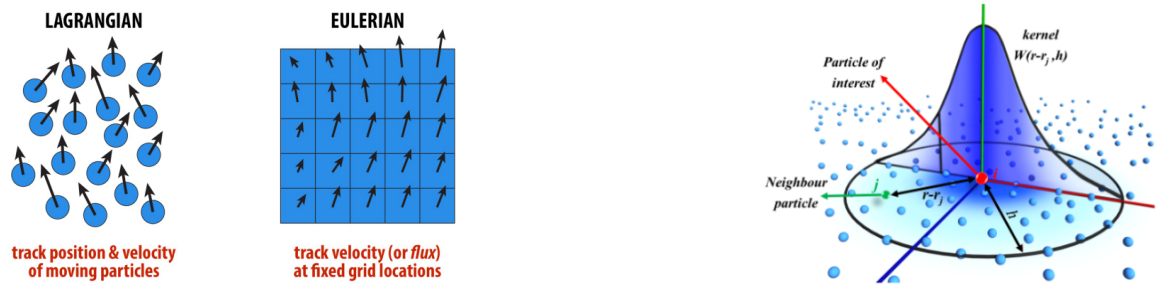
remains independent of any specific deep learning framework, ensuring flexibility and compatibility with various backends. We also aimed to leverage E(3) equivariant inputs to incorporate rotational and translational symmetries, allowing the model to learn and exploit these symmetries present in fluid dynamics.
Our comprehensive pipeline includes a multi-resolution component capable of handling varying numbers of particles over time while maintaining static shapes throughout. This multi-resolution feature enables the model to effectively process and analyze fluid dynamics simulations with different levels of detail, enhancing adaptability and accuracy.
Additionally, to support fluid dynamics applications specifically, we have implemented data structures tailored precisely for our needs. These specialized data structures are designed to efficiently handle multi-resolution information, optimizing performance and ensuring seamless integration with the model.
In addition to the machine learning pipeline, we also aimed to develop a classical multi-resolution SPH solver which can serve as a baseline for comparison with the machine learning-based predictions, enabling validation, benchmarking, and enhancing our understanding of fluid dynamics phenomena.

State-of-the-art Approaches In recent years, the MR-SPH solvers have gained a lot of traction. The majority of the methods differ in how they treat the coupling between the fine and coarse regions. In [2], the same smoothing kernel was used for both fine and coarse particles, this had an inherent downside of computational overhead in the fine particle region. When each region was assigned its own smoothing kernel, it brought down the computational cost significantly but it needed additional treatment at the interface as shown in [3] where an information exchange layer was implemented at the interface. In [4], correction matrices were incorporated which made the scheme consistent across the interface eliminating the need for an information exchange layer. This reference was used in this project for developing the traditional solver for dual resolution SPH consisting of particles with two discrete resolutions in the fluid domain. An extension of this work includes adaptive particle resolution [5] in the same domain. There has also been research interests in grid-based multi-resolution techniques for addressing fluid-structure interaction problems recently [6].
To the best of our knowledge, the exploration of multi-resolution particle-based methods leveraging GNNs for fluid dynamics prediction using machine learning remains an uncharted territory. However, neighboring techniques have exhibited promising outcomes in related fields. For instance, [7] presents an innovative approach that harnesses adaptive meshes, resulting in remarkable speed-ups of up to an order of magnitude when compared to conventional ground truth solvers. By dynamically adjusting the resolution of the mesh, this method optimizes computational efficiency while preserving the accuracy of the fluid dynamics simulation. Building on top of this, [8] employs reinforcement learning to learn the mesh adaption, being the first architecture that learns multi-resolution in an end-to-end fashion On the other hand, [9] adopts an alternative strategy by incorporating coarse-graining during an embedding step. However, a notable distinction is that they do not retain any fine particles during the simulation process. This approach still exhibits merits in certain scenarios, but it diverges from the notion of preserving multi-resolution particle-based information throughout the simulation.

2 Traditional Multi-Resolution Particle Solver

2.1 Smoothed particle hydrodynamics

In CFD, there are two fundamental approaches to representing fluid flow: the Eulerian approach and the Lagrangian approach. These approaches differ in how they discretize and track the fluid properties within a computational domain.
The Eulerian approach involves dividing the computational domain into a fixed mesh, where the fluid properties such as velocity, pressure etc. are defined at each grid point. The governing equations of fluid dynamics, such as the Navier-Stokes equations, are then solved numerically on this fixed grid to simulate the flow behavior.



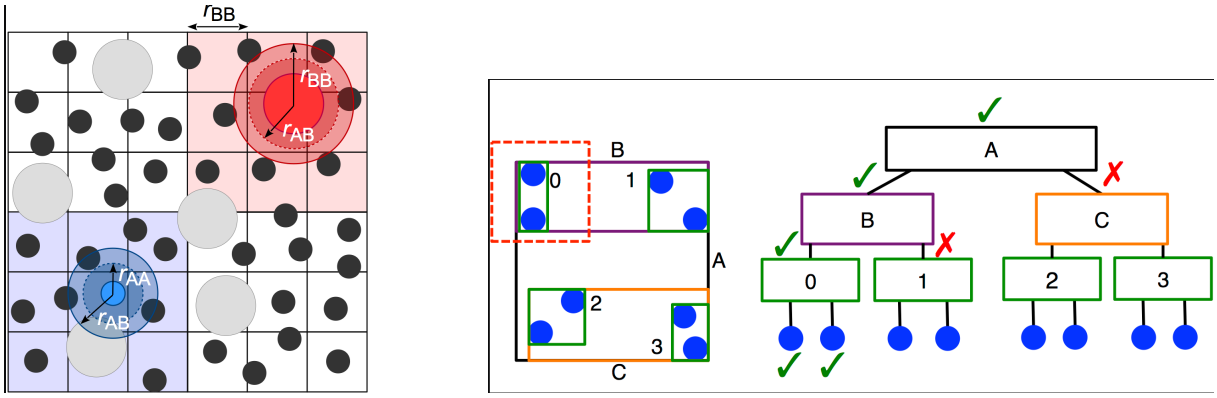
(a) Eulerian vs Lagrangian Approach (b) Gaussian Kernel

Figure 1: Basic blocks of SPH formulation [10]

On the other hand, the Lagrangian approach offers an alternative method to represent fluid flow by following the motion of individual fluid particles or 'parcels' throughout the computational domain. Instead of dividing the domain into a fixed grid, Lagrangian methods track the position, velocity, and other properties of discrete fluid particles over time. In contrast to the Eulerian methods, the Lagrangian methods naturally handle moving boundaries, free-surfaces and can simulate fluid-solid interactions with relative ease. Figure 1 depicts the key difference between the Lagrangian and the Eulerian scheme. The SPH is a Lagrangian method to simulate and analyze fluid flows. This method discretizes the fluid domain into a set of particles that move with the fluid flow. Each particle carries macroscopic properties such as position, velocity, and density, allowing for a natural representation of fluid motion and deformation. This method was initially developed by Monaghan and Gingold [11] in 1977 and independently by Lucy [12] for astrophysical simulations.
As the simulation progresses, for a given particle, a kernel function is used which weights the macroscopic properties of neighbouring particles based on distance and assigns it to the particle in concern. The simplest kernel function is a Gaussian function as shown in Figure 1 with a fixed cut-off radius. Other kernel functions popularly used in SPH are the Wendland Quintic Spline and the Cubic Spline. The details on the basic SPH theory is mentioned in Appendix F for curious readers.

2.2 Neighbors Search

Neighbor search algorithms play a crucial role in SPH. We discuss their utility for both single-resolution and multi-resolution problems, highlighting the benefits of different cutoff radii. Below mentioned are some standard methods of utilizing the neighbor search.



(a) Cell list grid division (b) k-d tree method

Figure 2: (a) Grid division with cell list and (b) kdtree method [13]

2.2.1 Cell Lists, Stencils and Tree Methods

Cell lists divide the simulation space into cells and maintain a list of particles within each cell. By checking neighboring cells, particles within the cutoff radius can be efficiently identified. Cell lists provide advantages such as reduced search space, improved cache utilization, and decreased time complexity, especially for simulations with low particle density [14]. Stencil uses a similar approach to the cell lists but the search area for each particle is fixed width cell, in contrast to the cell list which uses a cutoff radius.
A k-d tree, short for a k-dimensional tree, is a tree data structure. It organizes points or particles in a hierarchical structure, dividing the space into hyperplanes along each dimension. This allows for efficient search operations by recursively partitioning the space and narrowing down the search region. At each node, a splitting plane is chosen perpendicular to one of the dimensions, dividing the points into two child nodes. This process continues until each leaf node contains several points, forming a tree-like structure. The k-d tree enables efficient nearest neighbor search by pruning branches that are unlikely to contain the nearest neighbor. By traversing the tree based on spatial relationships, the algorithm efficiently identifies the closest points or particles [15]. Figures 2a and 2b show the difference between the two search methods.

2.2.2 Comparison of Neighbor Search Methods for Use in Multi-Resolution SPH

In terms of effectiveness, cell lists typically outperform k-d trees in neighbor search performance, especially in simulations with a lot of particles. By segmenting the simulation domain into cells, cell lists provide a localized search strategy that can speed up the detection of nearby particles. For large-scale simulations, k-d trees can be slower since they

need a hierarchical traversal of the tree structure. Additionally, cell lists are able to manage particles with varied radii more readily than k-d trees in multi-resolution scenarios, i.e., variable radii. Based on the particle radii, alternative cell sizes or cutoff distances can be assigned using cell lists. This makes it possible to conduct effective neighbor searches within the range required for each particle size. k-d trees, on the other hand, frequently have a set partitioning structure based on the spatial distribution of the particles, which might not be able to adjust adequately to changing particle radii. Finally, cell lists typically use less memory than k-d trees in terms of memory use. While k-d trees require memory to store the tree structure, this requirement might grow large for lengthy simulations, whereas the memory requirements for cell lists are proportionate to the number of cells.

2.2.3 Current Neighbor Search Open-Source Implementations

Current neighbor search open-source implementations, (refer Table 1) include Matsicpy, JAX-MD, scipy.kdtree, MDAnalysis, and HOOMD Blue.
The two candidates with the best chances are JAX-MD and scipy.kdtree. On the positive side, it is possible to jit the JAX-MD neighbor list update function, improving speed performance, significantly as the number of particles increase, but the downside is that the static type architecture and absence of a variable radii option. The second candidate is scipy.kdtree, which can build a uni-directional neighbor list in contrast to other candidates who employ cell lists, and can satisfy all the requirements of the current project, including variable radii. For instance, if a coarse particle X having a large cutoff radius has a fine particle Y in its vicinity with a small cutoff radius, then Y is a neighbor of X but not necessarily the other way around. A new check using Y's cutoff radius should be performed to determine whether X is in the neighborhood of Y. This uni-directional relationship between particles is essential for a multiresolution domain.

Table 1: Neighbor search package comparison where N is the number of particles and k is the number of particles per cell

Package	Neighborhood Search Method	PBC	Library	Var Radii	Avg Complexity
Matsicpy	cell list	Yes	c backend	yes	$\mathcal{O}(kN)$
JAX-MD	cell list	Yes	JAX	no	$\mathcal{O}(kN)$
scipy.kdtree	tree	Yes	scipy	yes	$\mathcal{O}(N \log N)$
MDAnalysis	tree	Yes	scipy	yes	$\mathcal{O}(N \log N)$
HOOMDBlue	cell list/stencil/tree	Yes	c backend	yes	all[1]

2.3 Multi-Resolution Approaches

In SPH, to accurately capture high gradients of field variables, we need sufficiently many fine particles in these regions of high gradients. But discretizing a domain with a single fine resolution requires a huge amount of memory overhead and it is a significant waste of

[1]The HOOMD Blue search methods involve both complexities: $\mathcal{O}(kN)$ and $\mathcal{O}(N \log N)$. The actual complexity depends on the specific method.

resources in regions of low gradients. This is the motivation to proceed to multi-resolution schemes i.e. to discretize the regions of interest with fine resolution and to have coarse particles elsewhere in the domain.
In our proposed approach following [4], the domain has particles of only two discrete resolutions: fine and coarse. The fine particles have a smaller cut-off radius h_H and the coarse particles have a bigger cut-off radius h_L. As shown in Figure 3, all particles are placed initially in a regular lattice with fine particles having a spacing of Δx_H and the coarse particles having a spacing of Δx_L. The resolution ratio $\gamma_{LH} = \frac{\Delta x_L}{\Delta x_H}$ is set to 2 in our solver.

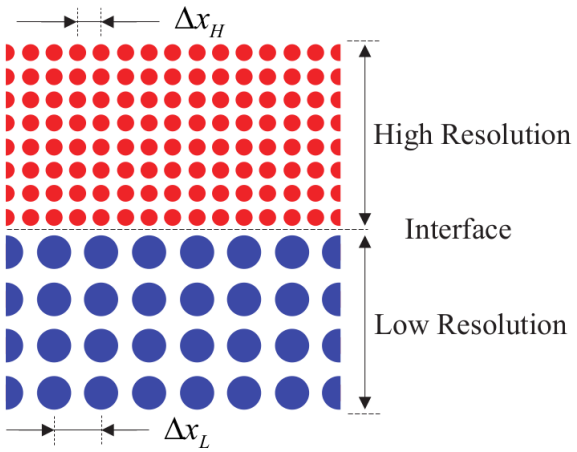


Figure 3: Initial particle configuration of the MR-SPH setup [4]

.

We implement dynamic splitting and merging of particles. As the particles advect with the flow, some fine particles may enter a coarse region which have to be merged or some coarse particles may enter the fine region where they need to be split.

2.3.1 Splitting

When a coarse particle moves into the fine particle region, it is split into 4 fine particles following the procedure described in Algorithm 4.

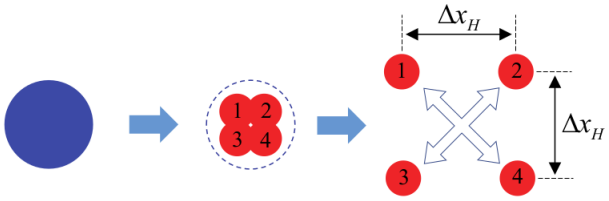


Figure 4: Refinement in 2D: Splitting a coarse particle into four fine particles [4].

Algorithm 1: Splitting Algorithm

1: At the position of the coarse particle, replace it with 4 fine particles each having the velocity and density of the big particle.
2: Move the fine particles such that they have an equal spacing of Δx_H as shown in Figure 4.
3: Correct the density of the fine particles using Equation 11.
4: Update the pressure of the fine particles using Equation 13.

2.3.2 Merging

The merging algorithm is a bit more involved than the splitting algorithm.

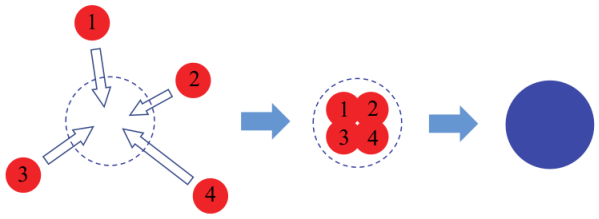


Figure 5: Coarsening in 2D: Merging four fine particles into a coarse particle [4].

Algorithm 2: Merging Algorithm

1: Determine the fine particles that have entered the coarse region and enter their IDs into a list
2: Sort this list of fine particle IDs based on how much they have penetrated the coarse region, the further they are inside the coarse region, the more priority is given to merge them.
3: Now determine the three nearest neighbors of these fine particles which are in this sorted list, they may lie in either the fine region or the coarse region.
4: Determine the Center of Mass (COM) of these particle groups and merge these particles only if the COM lies inside the coarse region.
5: Average the velocities of the four fine particles and assign it to the coarse particle.
6: Correct the density of the coarse particle using Equation 11.
7: Update the pressure of the coarse particle using Equation 13.

2.4 Results From the Traditional Solver

A 2D Reverse-Poiseuille Flow (RPF) test was setup for testing the implementation for the Multi-resolution solver. This test case was specifically chosen because it has Periodic Boundary Conditions (PBC) in both directions reducing the additional complexity of dealing with wall particles.
The domain was initialized with a total of 8000 particles out of which 6400 particles had a fine resolution and the rest 1600 had a coarse resolution. A time step, $dt = 10^{-7}$, was

chosen respecting the CFL conditions. Figure 6 shows the velocity profile of the above chosen configuration. This parabolic profile is an expected outcome for this laminar case. Even though there is no dynamic merging and splitting occurring for this low Reynolds number flow scenario, it is a first proof that multi resolution SPH actually works.

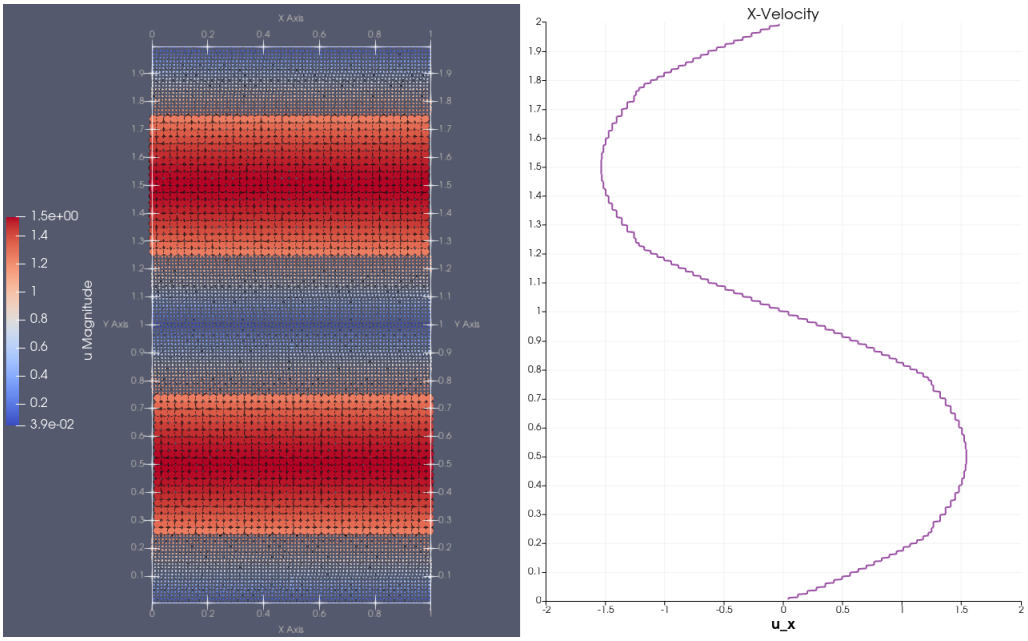


Figure 6: RPF test setup for Re = 0.015

3 Machine Learning Code-Base

3.1 Literature Review

Graph Neural Networks Graph neural networks have become prevalent in molecular dynamics and adjacent fields. They serve as the foundation for Graph Network-based Simulators (GNS) introduced in [16], which forms the basis of our machine learning pipeline. The GNS is composed of three key components: an *Encoder*, a *Processor*, and a *Decoder*. Each part plays a specific role in the simulation process:

- ***Encoder***: This component constructs a meaningful latent graph from the input state. It utilizes Multi-Layer Perceptrons (MLPs) to process the nodes independently, without considering neighboring nodes or edge information. The Encoder's primary function is to capture essential features and create a latent representation of the input graph.

- ***Processor***: The Processor is a message passing layer, as described in [17]. It consists of two MLPs: an edge MLP and a node MLP. The edge MLP processes edge latent vectors, taking as input the latent vectors of the two nodes connected by the edge, as well as the edge latent vector. The output is the new latent vector for the edge. On the other hand, the node MLP takes the aggregated edge latent vectors and its own latent vector as input. The aggregation function has to be a permutation invariant function, such as mean or max or summation, which we use. The node MLP outputs the new latent vector for the node.

- ***Decoder***: This component extracts relevant information from the final latent graph representation. Similar to the Encoder, the Decoder is also implemented as an MLP that operates independently on the nodes, without considering edge or neighboring node information.

By combining the power of MLP-based Encoders and Decoders with message passing in the Processor, the GNS can effectively simulate complex molecular dynamics systems [9, 7, 16].

Group-Equivariant Deep Learning E(3) equivariance, i.e. rotational and translational equivariance, is a critical concept that our model should learn, given that the laws of physics governing fluid dynamics exhibit E(3) equivariant properties. By incorporating E(3) equivariant methods into our model architecture, we can take advantage of the fact that the same patterns and properties can be recognized in different orientations or translations, reducing the number of unique configurations the model needs to learn. This results in enhanced sample efficiency and faster convergence during training.
Additionally, symmetry-aware representation plays a crucial role in understanding molecular systems. By incorporating E(3) equivariance into the model architecture, the model becomes more aware of the symmetries present in molecular structures and interactions. This can lead to more meaningful and concise representations of molecular systems, ultimately enhancing the model's predictive capabilities.
To achieve these goals, E(3) equivariant neural networks have been developed. One example is EGNN [18], a message passing GNN with restrictions on the initial latent vectors.

The initial latent vectors have to be E(3) invariant, and distances and relative differences are chosen to ensure this invariance. This approach has been adapted throughout our code-base to maintain E(3) equivariance. The message passing step itself is still invariant though.
Furthermore, SEGNN [19] builds upon this work by incorporating steerable MLPs. Steerable MLPs interleave linear mappings with non-linear activation functions, just like regular MLPs, but use linear transformations conditioned on geometric information. Additional care has to be taken on selecting a non-linear activation function that works with steerable vectors. This technique allows for full exploitation of E(3) equivariant properties.

3.2 Framework Comparison

Firstly, we go through the main reference paper [1] at the beginning of the project and try to understand the underlying fundamental concepts, theories and various methodologies related to simulation of time-integrated coarse-grained molecular dynamics(CGMD). The codebase for this paper [1] implements multi-scale GNN simulators without the need of force or energy. Python 3.8, PyTorch 1.11 and CUDA 11.3 were all used to test this implementation.
PyTorch itself provides a lot of flexibility however it can sometimes lead to boilerplate code and make it challenging to maintain structured and scalable projects. Pytorch Lightning framework is used for organizing and simplifying PyTorch code. It aims to alleviate above issues by providing a lightweight abstraction over PyTorch, making it easier to organize code, separate concerns, and improve readability. It introduces a standard structure for organizing models, training loops, and data handling, allowing researchers and engineers to focus more on the high-level logic of their models.
The overall overview of codebase with various components related to data preprocessing, GNNembeddings, clustering, DynamicsGNN is shown in figure 7. During the preprocessing steps, an MD system is embedded and coarse-grained to a coarse-level graph.. The CG MD simulator featurizes historical data using the featurizer before the Dynamics GNN forecasts the positions for the following step. A Score GNN is an option for further fine-tuning the projected placements.
We planned to implement the codebase in PyTorch 2.0 [20](without using PyTorch Lightening framework) and in JAX [21] to compare the inference speed and performance between those frameworks.

3.2.1 Pytorch 2.0

Performance comes in second place to flexibility and hackability in the PyTorch concept. Since PyTorch's debut in 2017, hardware accelerators like GPUs have dramatically increased in speed. Parts of PyTorch's internals were transferred into C++ in order to retain high-performance eager execution, which decreased hackability and raised the bar for code contributions.
Performance comes in second place to flexibility and hackability in the PyTorch concept. Since PyTorch's debut in 2017, hardware accelerators like GPUs have dramatically increased in speed. Parts of PyTorch's internals were transferred into C++ in order to retain high-performance eager execution, which decreased hackability and raised the bar

for code contributions. PyTorch compiler is develop to improve the performance without losing in PyTorch experience. A new feature torch.compile is added in PyTorch 2.0 [20]. It provide fundamental support for TorchDynamo, AOTAutograd, PrimTorch and TorchInductor. In addition, [22] discuss about deep learning compiler technologies that powers PyTorch 2.0 and the different phases of the compilation process.

3.2.2 JAX

JAX [21] is an open-source Python library developed by Google Research that provides an ecosystem for high-performance numerical computing and machine learning. JAX was designed to address some of the limitations of traditional automatic differentiation libraries and to leverage the full potential of hardware accelerators, such as GPUs and TPUs. It is built on top of NumPy and incorporates elements from various research projects like Autograd and XLA (Accelerated Linear Algebra).
Some of the key features of JAX are Numpy compatibility, automatic differentiation, functional programming and XLA integration.

3.2.3 Comparison between PyTorch 2.0 and JAX

Both PyTorch 2.0 and JAX are just-in-time compilable, support fully functional programming, and have been employed in major production systems, such as ChatGPT and Bard respectively, making them seemingly similar at a glance. However, there are significant differences in priorities and design decisions between these two frameworks.
In PyTorch, JIT compiling using Torch.compile is more straightforward compared to JAX's jax.jit. The ease of JIT compilation in PyTorch is due to its support for both data-dependent control flow and dynamic shapes. On the other hand, JAX requires programs to be written in a purely functional manner, with static shapes and limited data-dependent control flow.
PyTorch's TorchDynamo plays a crucial role in JIT compiling arbitrary Python code into FX graphs. By dynamically modifying Python bytecode just before execution through hooks into the frame evaluation API in CPython, TorchDynamo handles data-dependent control flow by employing graph breaks and extracting different FX graphs. This feature ensures almost complete backward compatibility with PyTorch 1.x, albeit with a trade-off of convenience for performance.
One notable advantage of PyTorch is its experimental support for dynamic shapes, which generally perform better than static shapes when compiled with PyTorch [20].
In contrast, JAX prioritizes different aspects. It is described on the JAX website as bringing together Autograd and XLA. JAX is a purely functional framework heavily based on the Numpy API. The powerful intermediate language, XLA, employed by JAX has the disadvantage of requiring static shapes. As a result, JAX is more restrictive compared to PyTorch, necessitating more effort to handle padding.
However, this restrictiveness allows JAX to generate much faster compiled code. Although JAX does not have graph breaks, it provides a somewhat similar functionality in the form of external callbacks. Nevertheless, the downside is that external callbacks in JAX will always be executed on the host side.

3.3 Methodology

In the domain of fluid dynamics, existing tools in JAX are predominantly designed with a focus on molecular dynamics applications, making them sub-optimal for our needs. Furthermore, our research project aimed to create a code base that seamlessly integrates with both JAX and PyTorch, facilitating a meaningful and robust comparison between these two frameworks in the context of fluid dynamics prediction. To accomplish this, we adopted a deliberate approach of avoiding the use of Jraph and JAX-MD completely. Instead, we undertook the development of custom data structures and connectivity computation methods tailored precisely to the specific demands of fluid dynamics simulations. While coming at the cost of having to build a lot of code from scratch, this decision allowed us to fine-tune our solutions to the task at hand, yielding superior performance and ensuring our code remains well-organized and maintainable.

3.3.1 Data Handling

Dataset Our dataset comprises a comprehensive long-term reverse poison flow simulation, spanning a total of $20,001$ timesteps. Each timestep includes particle positions, alongside particle types that, for our specific case, hold no significance and are uniformly set to zero.
To extract meaningful information from the dataset, we adopt a time window approach. Specifically, we load a stack of $n+1$ consecutive timesteps and calculate the velocity of particle positions through finite differences:

$$\mathbf{V}_t^{t+n} = \frac{\mathbf{X}_{t+1}^{t+n} - \mathbf{X}_t^{t+n-1}}{\Delta t} \tag{1}$$

where, $\mathbf{V}_t^{t+n}$ represents the velocity computed between timestep t and $t+n$, while $\mathbf{X}_t^{t+n}$ denotes the positions of particles from timestep t to $t+n$. The parameter dt corresponds to the time step size.
Furthermore, our dataset preprocessing includes an option to compute the initial graph connectivity during the data loading.

Dataloader For our dataloader implementation, we designed a custom padding function, distinct from the Jraph library, tailored to leverage our custom *GraphsTuple* data structure, as described in the subsequent section. The padding function efficiently accommodates inputs of varying lengths, ensuring seamless processing during data preparation, even though this is not actually needed for our dataset at hand. Subsequently, we collate the data and perform the conversion from Numpy data structures to the required backend, which could be either JAX, PyTorch, or Numpy, depending on the specific use case.
To optimize compatibility and ease of use, we opted to employ the standard PyTorch dataloader. PyTorch's dataloader has native support for numpy arrays, facilitating a seamless integration of our custom data structures into the data pipeline.

3.3.2 Basics

Data Structures Our *GraphsTuple* is tailored to seamlessly integrate with JAX as a PyTree while also maintaining native compatibility with PyTorch. It adapts many

of Jraph's *GraphTuple*'s attributes, adapting them to our needs. It encompasses the following attributes:

- **nodes:** A named tuple holding the node attributes. Some of these attributes may not have values attached to them during all processing steps, allowing for flexibility.
- **edges:** A named tuple containing the edge attributes.
- **senders:** An array containing the sender node indices for the edges.
- **receivers:** An array containing the receiver node indices for the edges.

The attributes of our nodes are as follows:

- **latent:** The latent vectors of the nodes.
- **position:** The positional history $\mathbf{X}_t^{t+n}$ of the nodes, with the time dimension leading.
- **velocity:** The velocity history of the nodes $\mathbf{V}_t^{t+n}$, with the time dimension leading.
- **acceleration_mean:** The predicted mean of the multivariate normal acceleration distributions.
- **acceleration_covariance:** The predicted covariance of the multivariate normal acceleration distributions.
- **mass:** The masses of the nodes, where a value of 1 denotes fine-resolution nodes, and 4 indicates coarse nodes.
- **is_coarse:** The coarse mask of the nodes.
- **is_padding:** In contrast to *jraph*, we employ padding masks to keep track of padded nodes. This allows us to freely permute the nodes while retaining information about which nodes are padding nodes. This is essential during the multi-resolution call.
- **target_position:** The target position $\mathbf{x}_{t+n+1}$.
- **coarse_score:** The probability that the particle is a coarse node or a fine node. This score is used to generate the merge and split assignments.

Layers We decided on building all layers from scratch, with the exception of the basic multi-layer perceptron, which we leveraged from existing implementations such as Haiku or Torchvision. By doing so, we ensured that our code base remains highly adaptable, allowing for easy substitution of the standard MLP with more specialized versions, such as steerable MLPs, as needed.
An essential aspect of our implementation strategy was to make our code base compatible with both PyTorch and JAX. To accommodate any differences in function names or keyword arguments between the two frameworks, we introduced wrapper functions wherever needed, that bridge the gap, enabling smooth integration of equivalent functions in their respective backends.
We incorporated the encoder-processor-decoder architecture introduced in [16].
The message passing layers in the processor are designed to operate in three distinct modes:

- **Weight Sharing:** In this mode, the layers share the same weight, reducing the amount of parameters needed at the cost of possibly less expression power.
- **Independent Weights:** Alternatively, layers can operate with independent weights, providing more expressive power, albeit increasing the model's parameter count.
- **Mixed Mode:** We also implemented a mixed mode, where specific layers have new weights, and the processor performs multiple message passing steps. This approach combines the advantages of weight sharing and independent weights, resulting in a total of $n \cdot k$ message passing steps, where n is the number of layers with shared weights, and k is the number of message passing steps performed by each layer with new weights.

YAML configuration loader The YAML configuration loader plays an important role in neural networks by providing a flexible and user-friendly way to specify and manage various aspects of the network's configuration. By using a configuration loader, we can load the YAML file and adjust the network settings based on specific requirements or experiments. This flexibility is particularly useful when conducting hyperparameter tuning or comparing different network configurations. Configuration loader are implemented for following modules: Dataloader, Wandb logger, EmbeddingGNN, DynammicsGNN.

3.3.3 Multi-Resolution Kit

The multi-resolution kit manages the merging and splitting of particles in the graph, along with the computation of neighborhoods. Given the necessity for data-dependent control flow in this context, the kit incorporates an external callback that operates independently of JAX. Instead, we have predominantly implemented the callback using Numpy and Scipy, offering a seamless transition to Numba in the future. By adopting Numba, we could effortlessly vectorize the function, presenting a promising avenue for performance optimization.
During the multi-resolution process, we leverage the positions that were initially used to compute the coarse scores. These positions precisely correspond to the predicted positions of the next time step before applying the merge and split function. This approach ensures that subsequent operations are executed based on accurate, predicted positions, bolstering the reliability and precision of our merging and splitting decisions.

External Callback The external callback generates the merge assignments, split assignments, and new edge connectivity while preserving the static output of the graph. Ensuring static output necessitates maintaining a constant number of merges and splits. Specifically, after splitting, the number of nodes becomes:

$$|V| \leftarrow |V| - 3 \cdot |num\_merges| + 3 \cdot |num\_splits| \tag{2}$$

when splitting and merging four nodes at once.
To accommodate a dynamic number of merges and splits, we employ padding. If there aren't enough merges or splits, we merge or split padding nodes instead. However, we must be cautious about potential padding underflows, as the number of padding nodes is limited. We track underflows and overflows in the state to ensure proper handling. To

work in the JAX framework, we modify the merge and split assignments to maintain a consistent output regardless of whether under or overflows occur.
Initially, we compute the split and merge assignments based on the coarse scores. Subsequently, we calculate the merged positions, and finally, we determine the connectivity using kd-trees with different cutoff radii, based on the mass, for the new positions.
The external callback enables dynamic adjustments without compromising the model's overall performance, enhancing the model's capacity to handle diverse fluid dynamics scenarios with precision and accuracy.

On-Device To preserve gradients and ensure seamless integration with the deep learning framework, we execute the actual merging and splitting operations directly on the device within the model. Initially, we compute the split pattern, which has the potential to be predicted from the latent features of the coarse-grained nodes. However, for now, it is represented as a simple axis-aligned square.
With the the external callback outputs, we proceed to perform the splitting and merging of particles on the device. This on-device execution ensures that gradients remain intact throughout the model's operations, maintaining consistency during both forward and backward passes.
Additionally, the on-device function can be just-in-time (JIT) compiled, given that it is not data dependent and exhibits predictable shape outputs.

3.4 Software Engineering

Unit Tests Ensuring the robustness and correctness of our implementation, we have crafted unit tests for all layers. These tests rigorously assess the returned shapes, as well as validate static shapes, ensuring that the code can be successfully JIT-compiled.
Moreover, we have constructed a comprehensive test case for the multi-resolution kit. These tests not only validate shapes but also assess the correctness of the merge and split actions, guaranteeing that the kit functions as intended. We also ensure that it can be successfully JIT-compiled, in order to be able to be incorparated in our models.
To streamline the testing process, all unit tests are automatically triggered with each push to the git repository. This automated testing system allows us to promptly identify and address any issues, promoting continuous integration and reliable code development.

Documentation Ensuring clarity and accessibility for our codebase, we have created extensive documentation. Our documentation is built using Sphinx, a powerful and popular documentation generation tool. It provides a structured, easily navigable format, and can be viewed as an HTML file in any standard web browser.
Through our documentation, we aim to provide in-depth insights into our code architecture, design choices, function descriptions, and usage guidelines.

3.4.1 Model

Pre-processing The pre-processing stage is responsible for transforming the fine-resolution graph into a mixed-resolution graph, setting the foundation for subsequent model operations.

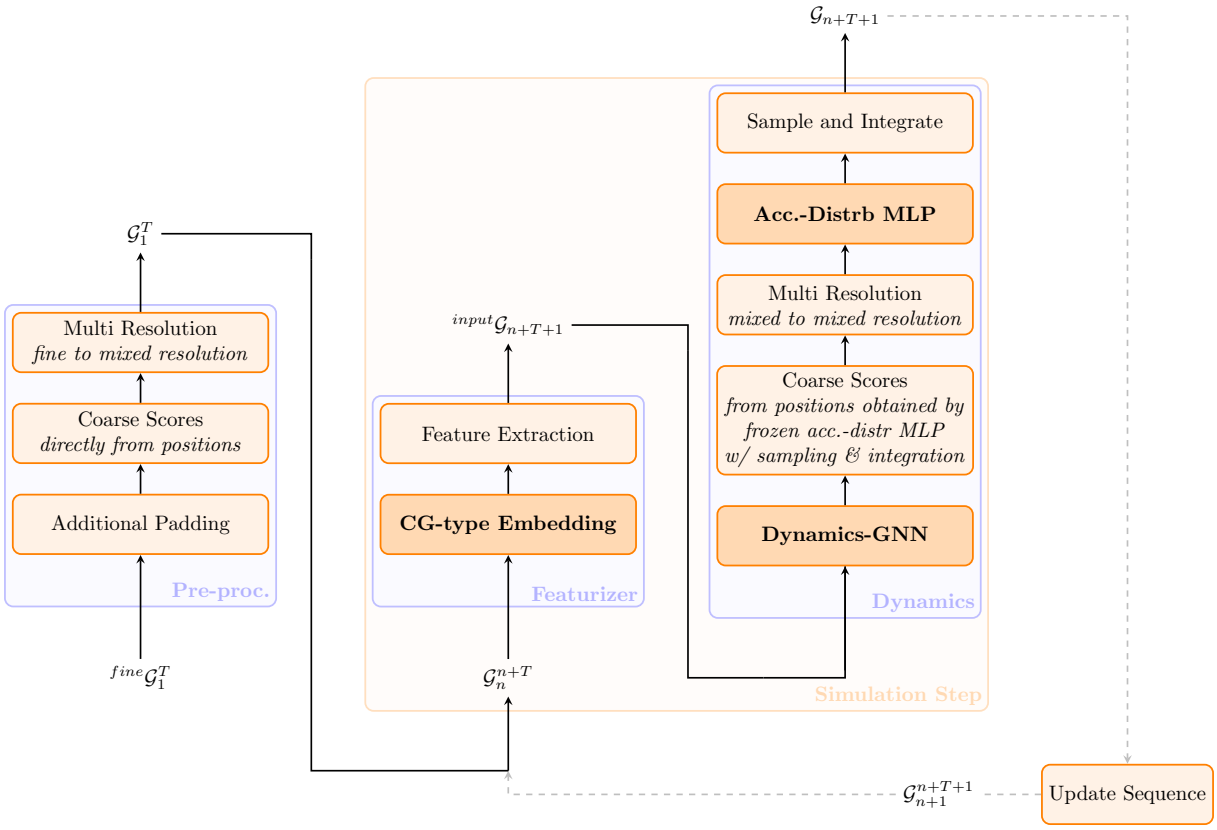


Figure 7: Overview of the model architecture. The model pipeline consists of two main steps: a pre-processing step and a simulation step. The blue boxes represent separate files in the code base. Dark orange layers contain learnable parameters, while the lightly shaded ones are pure functions, without any learnable parameters.

Firstly, additional padding is applied to the graph. While the graph is already padded, the mixed-resolution process might require further padding to accommodate the merge and splits accurately, preventing any potential underflows.
Subsequently, the pre-processing stage computes the coarse scores, which play a pivotal role in the subsequent multi-resolution computations. These coarse scores are derived based on the distance to the nearest interface. Each particle is assigned a binary classification in the form of a positive or negative coarse score, depending on whether it lies within the coarse region or not. The coarse score is positive if the particle is within the coarse region and negative otherwise, representing a binary classification for each particle. Finally, to facilitate further processing and ensure that the coarse scores lie within the $(0, 1)$ interval, a sigmoid function is applied.

$$f(x_i) = \begin{cases} 1, & \text{if } \mathbf{x}_i \in \text{Coarse Region} \\ -1, & \text{otherwise} \end{cases}$$
$$cs_i = \sigma\left(f(\mathbf{x}_i) \cdot \min\{||\mathbf{x}_i - \mathbf{f}_j|| \quad | \quad \mathbf{f}_j \in \text{Interfaces}\}\right) \tag{3}$$

In future versions, we aim to learn the coarse scores directly. This approach is feasible due to the multi-resolution kit's ability to handle proper gradient flow through the coarse scores, even though the splitting assignments are computed in the external callback.

Featurizer The featurizer computes the initial latent vectors for both nodes and edges, forming the foundation for subsequent computations within the model. For nodes, the initial latent vector comprises two key components. Firstly, the flattened velocity history of the particles is incorporated, capturing the dynamic movement patterns of each particle over time. Secondly, a one-hot encoding of the particle type is included, indicating whether the particle is coarse-grained or fine-grained. By leveraging this information, the model can distinguish between different particle types, enabling more accurate and context-aware predictions.
On the other hand, for edges, the initial latent vector is determined based on two factors: the displacement and distance between the two connected nodes. This representation captures the spatial relationship between nodes, providing crucial geometric information that contributes to accurate edge computations.

Dynamics The dynamics module is the heart of our model, where the majority of the actual computations occur. It processes the latent vectors from the featurizer and performs all steps to predict the fluid dynamics behaviors.
The dynamics-GNN plays a crucial role by mapping the latent vectors from the featurizer to a new latent representation. This mapping extracts essential features that effectively capture the underlying dynamics of the fluid system, enabling the model to gain a deeper understanding of the fluid behavior.
Next, the acceleration is computed using the acceleration-distribution MLP. The weights of the MLP are frozen at this stage. The accelerations are then sampled from the resulting distribution, and this information is doubly integrated to obtain the positions of the current time step.
These positions are then used to compute the coarse scores for the subsequent multi-resolution computation. Additionally, the same positions are used to compute the connectivity in the multi-resolution function, ensuring accurate merging and splitting operations based on the positional information before the merge and split operation.
Following the merging and splitting operations, the acceleration-distribution is recomputed using the acceleration-distribution MLP. This step is essential as we cannot merely merge the acceleration distributions obtained earlier due to the non-linear nature of the acceleration distribution MLP. Subsequently, we sample from the new distribution and integrate twice to predict the velocities and positions, respectively.
Finally, we update the sequence, using the position and velocity history of the T most recent time steps as input for the next simulation step.

4 Refinements with Diffusion

This section deals with score-based refinement methods for long MD simulations and is based on the reference paper [23], which we adapted to our particle framework.

Motivating Refinement Procedures In MD simulations, one often faces stability issues along the simulated MD trajectories [23]. In particular, the predicted particle positions may not be optimal or at worst even infeasible from a physics point of view. One approach to dealing with this problem would be to 'correct' the predicted particle positions using refinement methods, which is also explored by the reference paper.

4.1 Literature Review

Score refinements The reference paper [23] addresses the stability problems in long simulations by following a predict-then-refine procedure. A two-step scheme is applied, where the Dynamics GNN is used to predict the forward step particle positions. Next, the learned scores are applied to refine ('denoise') the predicted particle positions. The training of the score model follows the noise conditional score network (NSCN) framework with denoising score matching [24]. Starting from the ground truth state, noise at different scales is added to the particles. The score model is then trained to predict the added noise.

[25] derives a concept for the pre-training of molecular structure data for down-stream representation tasks. By learning an associated force field, the model allows for refinement steps resulting in close-to-equilibrium or lower-energy structures, hence 'relaxing' the structures. The resulting learning objective is equivalent to the objective applied in [23].

Diffusion models In [26], the authors aim to predict ligand-to-protein docking poses. For this purpose they apply a diffusion generative model that generates a sample of candidate ligand poses. The proposed ligand-to-protein poses are then ranked by a confidence model based on their RMSD[2] score, reflecting the likelihood of a pose being feasible.

[27] introduces a learning framework based on diffusion probabilistic models that allows both for (image) generation and denoising refinement steps. In stark contrast to score refinements, the diffusion-based approach does not rely on learning the scores. Instead a 'reverse process' is trained to reverse a diffusion that gradually adds noise to the data. The learned denoising transition steps can then be applied to refine the current states.

Our ideas to potentially improve the framework in [23] are:

- define an energy function over the number of bead-collisions or the MSE of particle distances and train towards low-energy particle positions (inspired by [25])
- pre-train a Score GNN on 'relaxed' (equilibrium) particle positions (following [25])
- generate multiple candidate refinements and add a confidence model to assess the likelihood that the refined particle positions are feasible (adapted from [26])
- replace the Score GNN by a diffusion model and combine diffusion-based refinement steps with a confidence model to assess the predicted positions (see [26] and [27])

[2]The root-mean-square deviation (RMSD) is a measure of the average distance between atoms.

4.2 Methodology

Please note that we did not implement any of the ideas for potential improvements over the reference paper due to time limitations. In this section, we therefore cover the framework of the reference paper. Note that we provide an educative minimal working example[3].

Before introducing the methodology of the NSCN framework, we want to showcase how the learned scores are applied to refine and denoise particle positions.

4.2.1 Score refinements: annealed Langevin dynamics

Suppose that $p(x) \in \mathbb{R}^n$ is the distribution of particle positions in equilibrium[4]. Then the corresponding score-function is defined as

$$\nabla_x \log p(x) = \left(\frac{\partial \log p(x)}{\partial x_1}, \ \ldots, \ \frac{\partial \log p(x)}{\partial x_n} \right)^T,$$

where $p(x)$ denotes the density[5] and $\log p(x)$ is the log-density function.

Given the score function, one can apply a concept called 'Langevin dynamics' to sample from the distribution $p(x)$. This is an iterative MCMC procedure that allows us to draw a sample from $p(x)$ by using only its score function $\nabla_x \log p(x)$.

Algorithm 3: Langevin dynamics

Input:

- learned scores: $\nabla_x \log p(x)$
- parameters: # of iteration steps K, step size $\varepsilon > 0$
- starting point: $x_0 \sim \pi$ (arbitrary prior distribution)

for $i = 0, 1, \ldots, K$ **do**

$z_i \sim \mathcal{N}(0, I)$

$x_{i+1} \leftarrow x_i + \varepsilon \, \nabla_x \log p(x) \, |_{x=x_i} + \sqrt{2}\varepsilon \, z_i$

end

Output:

- sample x_K with approximate distribution $p(x)$

Under enough regularity, when $\varepsilon \downarrow 0$ and $K \to \infty$, the distribution $\hat{p}(x_k)$ of the iterated sample x_K converges to $p(x)$. This yields a theoretical basis and motivation for the score-based refinement steps to be applied to the potentially erroneous particle positions [23].

[3] See git@github.com:CLongoTUM/pre-training-via-denoising.git.

[4] We refer to Appendix Pre-training via Denoising for further reading. In [25], the authors introduce a physical distribution to describe the statistical behavior of physical systems in equilibrium.

[5] The data distribution and density are often both denoted interchangeably by $p(x)$ (e.g. see [24]).

4.3 Score-matching with multiple noise perturbations

We refer to the appendix for a gentle introduction to score models and score matching. Further, we make the remark that score matching has limitations when it comes to estimating scores in low data density regions, due to the potential lack of sample data [24].

Noise perturbation. To overcome difficulties with low data density regions, so-called noise conditional score networks apply noise perturbations at different scales, to 'fill' low sample data regions. We can define a set of different noise-scales by

$$\mathcal{S} = \Big\{\sigma_1, \dots, \sigma_L \;\Big|\; \frac{\sigma_1}{\sigma_2} = \dots = \frac{\sigma_{L-1}}{\sigma_L} > 1 \text{ for } i = 1, \dots, L-1\Big\}.$$

For each noise-level $\sigma_i \in \mathcal{S}$, we perturb the data $x \in \mathbb{R}^D$ by adding Gaussian noise

$$\tilde{x} = x + \sigma_i \varepsilon_i, \;\; \text{where } \; \varepsilon_i \sim \mathcal{N}(0, 1).$$

The distribution of the perturbed data $\tilde{x}$ is given as

$$q_{\sigma_i}(\tilde{x}) = \int p(t)\, \mathcal{N}(\tilde{x} \mid t, \sigma^2 I)\, dt.$$

Training. Following [24], the NCSN $s_\theta(x, \sigma)$ is trained by jointly estimating the scores of all perturbed data distributions,

$$s_\theta(x, \sigma_i) \approx \nabla_x \log q_{\sigma_i}(x) \quad \forall\, i = 1, \dots, L.$$

The NCSN can be trained via denoising score matching.

Learning objective. The process of perturbing the input data with noise at scale σ_i can be described as a diffusion process[6] with transition probabilities

$$q_{\sigma_i}(\tilde{x} \mid x) = \mathcal{N}(\tilde{x} \mid x, \sigma^2 I).$$

The scores are given by

$$\nabla_{\tilde{x}} \log q_{\sigma_i}(\tilde{x} \mid x) = \nabla_{\tilde{x}} \Big(\frac{1}{(2\sigma_i \pi)^{n/2}} - \sum_{j=1}^{n} \frac{(\tilde{x}_j - x_j)^2}{2\sigma_i^2} \Big) = \frac{\tilde{x} - x}{\sigma_i^2} = \frac{1}{\sigma_i} \varepsilon_i$$

and hence, the model is trained to predict the added noise over the scales $\sigma_1, \dots, \sigma_L$ by minimizing a function[7] of residues of the form

$$||s_\theta(x, \sigma_i) - \frac{\tilde{x} - x}{\sigma_i^2}|| = ||s_\theta(x, \sigma_i) - \frac{1}{\sigma_i} \varepsilon_i||.$$

Yet again we end up with a score-model that is trained to predict noise for denoising. In the next section, we focus on the implementation of the diffusion framework in JAX.

[6] This diffusion-based view falls in line with [27]. We refer to the diffusion section.

[7] We refer to the reference paper [23] for the exact definition of the objective function.

4.4 Implementation

The score-based refinement method[28] is constructed using the JAX and Haiku libraries for efficient neural network operations. It involves iteratively refining the positions of particles using annealed Langevin dynamics. During each iteration, the model preprocesses a score graph based on the current positions and history latent representation augmented as uncertainty embeddings. The obtained scores are then used to adjust the positions, and Langevin noise is added to explore the solution space effectively. This iterative process is performed multiple times for different sigma values, allowing the model to progressively improve the accuracy of its predictions.

4.4.1 Data Augmentation

To capture the temporal relationships among embeddings generated by a dynamic model, we propose augmenting the learned embeddings in the representation space with an uncertainty estimation technique[29]. We then apply masked mix-up to the augmented embeddings to further enhance the diversity of the data. To ensure more robust predictions, we utilize contrastive learning to constrain both the original and augmented embeddings.

After obtaining the interaction embeddings from dynamic encoder, we characterize the embeddings as uncertainty distributions, which are used to generate augmented embeddings. Specifically, denote a batch of interaction embeddings encoded by the dynamic encoder as $Z \in R^{B\times D} = (z_1, ..., z_B)$, where B is the batch size and D is the embedding size. $z_i \in R^{1\times D}$ represents the embeddings of $i-th$ interaction. We first calculate the embeddings mean and standard deviation of each event, which can be formulated as:

$$\mu(z_i) = \frac{1}{D}\sum_{d=1}^{D} z_i^d, \quad \sigma^2(z_i) = \frac{1}{D}\sum_{d=1}^{D}\left(z_i^d - \mu(z_i^d)\right)^2. \tag{4}$$

Denote the statistics $\mu(Z) \in R^B = (z_1, ..., z_B)$ and $\sigma(Z) \in R^B = (z_1, ..., z_B)$ as the embedding mean and standard deviation of all events in the batch. The uncertainty estimation of the embedding mean μ and standard deviation σ are calculated as:

$$\Sigma_\mu^2(Z) = \frac{1}{B}\sum_{b=1}^{B}\left(\mu(z_b) - \mathbb{E}_b[\mu(z_b)]\right)^2, \quad \Sigma_\sigma^2(Z) = \frac{1}{B}\sum_{b=1}^{B}\left(\sigma(z_b) - \mathbb{E}_b[\sigma(z_b)]\right)^2. \tag{5}$$

Finally, the uncertainty embeddings $Z_{un} \in R^{B\times D}$ are generated as follows:

$$Z_{un} = (\sigma(Z) + f\epsilon_\sigma\Sigma_\sigma(Z))\left(\frac{Z-\mu(Z)}{\sigma(Z)}\right) + (\mu(Z) + f\epsilon_\mu\Sigma_\mu(Z)) \tag{6}$$

where ϵ_μ and ϵ_σ both follow the standard Gaussian distribution, f is a hyper-parameter to control the scope of uncertainty estimation. The uncertainty estimation based on the training data observations can provide an appropriate and meaningful variation range for each interaction's embedding, which does not harm model training but can simulate diverse and reasonable potential shifts.

4.4.2 Score Refinement

The model structure consists of a score-based prediction with Dynamics GNN and refinement approach with Score GNN. It begins with an initialization step, where parameters and attributes are set up. A range of sigma values is created to be used for annealed Langevin dynamics during the refinement process. A score GNN is instantiated to refine the predicted positions.

Prediction. The prediction step takes input historic positions, particle types, bonds, and weights. Preprocessing is performed to obtain a history graph representation. The history graph is fed into a Dynamics GNN to predict the next time stamp positions. The predicted positions are split into mean and standard deviation components. The standard deviation is adjusted using a non-linear activation function. If deterministic mode is enabled, the mean is used as the prediction; otherwise, a sample is drawn from a normal distribution defined by the mean and standard deviation. The predicted positions are then decoded to obtain the current positions.

Refinement. During the refinement process, reverse diffusion using annealed Langevin dynamics is applied to improve the predicted positions. The current positions are initialized with the decoded positions and undergo iterative updates. The score graph is constructed using the current positions, history latent representation, and other input parameters. Scores for position refinement are obtained from a score graph neural network. Langevin noise is added to the current positions to simulate diffusion.

Training. The training step involves processing input sequences, next positions, particle types, bonds, and other optional arguments. Preprocessing is performed to prepare the data for training. The positions are augmented with noise to introduce diversity into the training data. Target values for the score-based loss are calculated based on the difference between the noisy next positions and the original next positions, normalized by the sampled sigmas. The history graph and predicted mean and standard deviation values are obtained using the Dynamics GNN. The standard deviation is adjusted using a non-linear activation function. Target values for the accuracy loss are calculated based on the inverse of the decoder post-processor. The score graph is constructed using the noisy next positions, history latent representation, and other input parameters. Scores for position refinement are obtained from a score graph neural network and rescaled by the inverse of the sampled sigmas. The score loss is calculated as the mean squared error between the scores and the target values. If a property prediction network is included, property predictions are obtained and the property loss is calculated. The overall loss is a combination of the accuracy loss, score loss, and property loss.

In summary, the model structure involves predicting initial positions using a Dynamics GNN and then refining the positions using reverse diffusion with annealed Langevin dynamics. The model is trained with a combination of accuracy loss, score loss, and property loss.

5 Conclusions

This project aimed to explore the potential of equivariant graph neural networks in a multi-resolution framework for particle-based fluid mechanics simulations.
The development of a machine learning pipeline from scratch, independent of any specific deep learning framework, allows for flexibility and compatibility with various backends. Incorporating E(3) equivariant inputs enables the model to learn and exploit rotational and translational symmetries present in fluid dynamics, leading to more meaningful representations. If full E(3) equivariance is needed, our architecture facilitates changing the GNNs to SEGNNs by simply exchanging the MLPs with steerable MLPs. This is achieved by changing a single config parameter.
Our multi-resolution component, capable of handling varying numbers of particles in the temporal dimension while maintaining static shapes, empowers the model to process fluid dynamics simulations with different levels of detail effectively. The specialized data structures tailored for fluid dynamics applications further optimize performance and integration with the model. The development of the machine learning pipeline for particle-based fluid mechanics simulations turned out to be a more intricate task than anticipated. The complexity of the project grew, resulting in a substantial amount of code. In fact, we ended up writing approximately 6,000 lines of code dedicated solely to the machine learning pipeline, even after removing redundant or outdated code.
The extensive documentation of our code base, coupled with the Continuous Integration and Continuous Deployment (CI/CD) pipeline, guaranteed maintainable and robust code. Additionally, the inclusion of a classical multi-resolution SPH solver complements our machine learning-based approach, providing a baseline for validation and benchmarking.

Looking Ahead our work opens up exciting opportunities for further exploration and innovation in the field of fluid dynamics prediction with machine learning. Notably, we have identified specific areas that hold great potential for enhancing the efficiency and performance of our current implementation.
One significant area for improvement lies in optimizing the computation of the connectivity list transfer from the CPU to the GPU. Currently, this process represents one of the most computationally expensive tasks in our code-base. Exploring and implementing new neighborhood computation algorithms, such as the promising approach introduced in [30], could yield substantial benefits. These algorithms should be adaptable to work within the bounds of static shapes and without data-driven control flow, potentially being JAX jit compilable, while mitigating the memory footprint issues associated with JAX-MD. By computing the connectivity directly on the GPU, we could significantly improve the overall efficiency and speed of our simulations.
Another avenue for enhancement lies in directly computing the coarse scores from the latent information. Our code-base was designed with this in mind, making it well-suited for this approach. By directly deriving the coarse scores from the latent vectors, we can potentially adapt the multi-resolution approach to more difficult problems than the Reverse-Poiseuille Flow.
Additionally, exploring the addition of steerable MLPs is a promising. This would practically change all of our GNNs to SEGNNs.

References

[1] Xiang Fu, Tian Xie, Nathan J. Rebello, Bradley D. Olsen, and Tommi Jaakkola. Simulate time-integrated coarse-grained molecular dynamics with geometric machine learning, 2022.

[2] Pourya Omidvar, Peter K. Stansby, and Benedict D. Rogers. Wave body interaction in 2d using smoothed particle hydrodynamics (sph) with variable particle mass. *International Journal for Numerical Methods in Fluids*, 68(6):686–705.

[3] Xin Bian, Zhen Li, and George Karniadakis. Multi-resolution flow simulations by smoothed particle hydrodynamics via domain decomposition. *Journal of Computational Physics*, 297:132–155, 09 2015.

[4] Wei Hu, Wenxiao Pan, Milad Rakhsha, Qiang Tian, Haiyan Hu, and Dan Negrut. A consistent multi-resolution smoothed particle hydrodynamics method. *Computer Methods in Applied Mechanics and Engineering*, 324:278–299, 2017.

[5] Wei Hu, Guannan Guo, Xiaozhe Hu, Dan Negrut, Zhijie Xu, and Wenxiao Pan. A consistent spatially adaptive smoothed particle hydrodynamics method for fluid–structure interactions. *Computer Methods in Applied Mechanics and Engineering*, 347:402–424, apr 2019.

[6] Tianrun Gao, Huihe Qiu, and Lin Fu. A block-based adaptive particle refinement sph method for fluidâ€“structure interaction problems. *Computer Methods in Applied Mechanics and Engineering*, 399:115356, 2022.

[7] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning Mesh-Based Simulation with Graph Networks.

[8] Tailin Wu, Takashi Maruyama, Qingqing Zhao, Gordon Wetzstein, and Jure Leskovec. Learning Controllable Adaptive Simulation for Multi-resolution Physics.

[9] Xiang Fu, Tian Xie, Nathan J. Rebello, Bradley D. Olsen, and Tommi Jaakkola. Simulate Time-integrated Coarse-grained Molecular Dynamics with Geometric Machine Learning.

[10] Giuseppe Giorgi. *GIORGI PhD Thesis 2018 FINAL*. PhD thesis, 03 2019.

[11] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 12 1977.

[12] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. , 82:1013–1024, December 1977.

[13] J. A. Anderson, J. Glaser, and S. C. Glotzer. Hoomd-blue: A python package for high-performance molecular dynamics and hard particle monte carlo simulations. *Computational Materials Science*, 173:109363, 2020.

[14] D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, 2002.

[15] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[16] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to Simulate Complex Physics with Graph Networks.

[17] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks.

[18] Victor Garcia Satorras, Emiel Hoogeboom, and Max Welling. E(n) Equivariant Graph Neural Networks.

[19] Johannes Brandstetter and Rob Hesselink. GEOMETRIC AND PHYSICAL QUANTITIES IMPROVE E(3) EQUIVARIANT MESSAGE PASSING.

[20] PyTorch 2.0 — PyTorch.

[21] Jax: High-performance array computing.

[22] Shashank Prasanna. How pytorch 2.0 accelerates deep learning with operator fusion and cpu/gpu code-generation, 2023.

[23] Xiang Fu, Tian Xie, Nathan J. Rebello, Bradley D. Olsen, and Tommi Jaakkola. Simulate time-integrated coarse-grained molecular dynamics with geometric machine learning, 2022.

[24] Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution, 2020.

[25] Sheheryar Zaidi, Michael Schaarschmidt, James Martens, Hyunjik Kim, Yee Whye Teh, Alvaro Sanchez-Gonzalez, Peter Battaglia, Razvan Pascanu, and Jonathan Godwin. Pre-training via denoising for molecular property prediction, 2022.

[26] Gabriele Corso, Hannes StÃ¤rk, Bowen Jing, Regina Barzilay, and Tommi Jaakkola. Diffdock: Diffusion steps, twists, and turns for molecular docking, 2023.

[27] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *CoRR*, abs/2006.11239, 2020.

[28] Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution, 2020.

[29] Xiaotong Li, Yongxing Dai, Yixiao Ge, Jun Liu, Ying Shan, and Ling-Yu Duan. Uncertainty modeling for out-of-distribution generalization, 2022.

[30] Xinye Chen and Stefan GÃ¼ttel. Fast exact fixed-radius nearest neighbor search based on sorting.

[31] Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural Computation*, 23(7):1661–1674, 2011.

Appendix

A Introducing score models

We follow [31] for the sake of introducing score models.

Given the sample data $x_1, \dots, x_N \overset{i.i.d.}{\sim} p(x)$, the task is to fit a model to the data distribution $p(x)$ and to compute the scores.

An intuitive way of modeling the data distribution is via energy-based models.

A.1 Energy-based Models

For a parametric family of energy functions $\{E_\theta : D \to \mathbb{R},\, \theta \in \Theta\}$, we define a density by

$$p_\theta(x) = \frac{e^{-E_\theta(x)}}{Z_\theta} \quad \text{with total energy} \quad Z_\theta = \int_D e^{-E_\theta(x)}\, dx.$$

In this setup $\theta \in \Theta$ is a learnable parameter which may be obtained by maximizing the log-likelihood function $\ell(x) : \Theta \mapsto \mathbb{R}, \theta \mapsto \sum_{k=1}^N \log\, p_\theta(x_k)$ of the sample $x_1, \dots, x_N$.

However in order to compute the pdf $p_\theta(x)$, one is required to evaluate the corresponding normalizing constant Z_θ, which is intractable for general $f_\theta(x)$. Thus, to make maximum likelihood training feasible, one needs to either restrict the model to make Z_θ tractable or approximate the normalizing constant, which may be computationally exhaustive.

By introducing score-based models, we can sidestep intractable normalizing constants.

A.2 Score models and score matching

For $\theta \in \Theta$, we define a score model s_θ by

$$s_\theta(x) = \nabla_x \log\, p_\theta(x) = -\nabla_x E_\theta(x) - \underbrace{\nabla_x \log\, Z_\theta}_{=0} = -\nabla_x E_\theta(x).$$

Hence, the score-based model is independent of Z_θ.

In this setup $\theta \in \Theta$ is a learnable parameter which may be obtained by minimizing the Fisher divergence between the model and the data distribution, defined by

$$\mathbb{E}_p[\, ||\nabla_x \log\, p(x) - s_\theta(x)||_2^2\,] = \int p(x)\, ||\nabla_x \log\, p(x) - s_\theta(x)||_2^2\, dx.$$

The Fisher divergence measures the ℓ_2 distance between the score-based model and the ground-truth data score $\nabla_x \log\, p(x)$.

B Pre-Training via Denoising for molecular property prediction

[25] derives a concept for the pre-training of molecular structure data to generate useful representations of 3D molecular structures for downstream prediction tasks. By learning an associated force field, the model allows for refinement steps resulting in close-to-equilibrium or lower-energy structures.

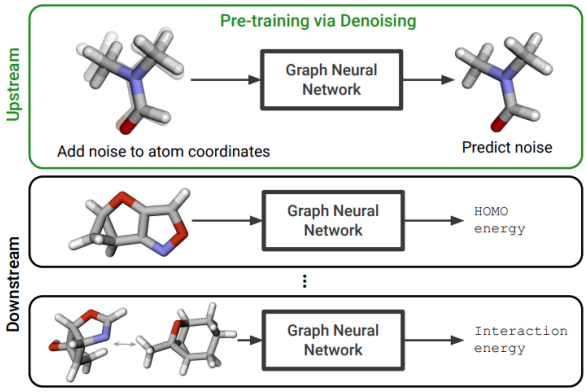


Figure 8: Illustration. This figure was taken from [25] Fig. 1

Learning Objective Surprisingly, the corresponding learning objective is equivalent to the denoising score matching objective in [24] that is also applied in the reference paper.

Relevance This allows for a different perspective view on the concept of 'denoising' as relaxing the particle system into an 'equilibrium state'. Following [25], one could argue to either add a pre-training of the score model to our framework or to replace it at all by a pre-trained version. However, a potential limitation for our use-case is that the pre-training in [25] would require a dataset of 'equilibrium structures'. In particular, it is not clear how an 'equilibrium' would look like for particles, neither there is a dataset available. A potential idea would be to define an energy function over the number of bead-collisions or the MSE of particle distances and looking at low-energy 'particle positions'.

C Diffusion steps, twists, and turns for molecular dockin

In [26], the authors aim to predict ligand-to-protein docking poses. For this purpose they apply a diffusion generative model that generates a sample of candidate ligand poses. The proposed ligand-to-protein poses are then ranked by a confidence model based on their RMSD[8] score, reflecting the likelihood of a pose being feasible.

Relevance An idea that could be applied to our framework is the use of a confidence model to assess the likelihood that the refined particle positions are feasible. This would add a confidence step to our model that consists of generating n independent refined structures and ranking them based on an appropriate metric like particle collisions or distances, which could lead to an improvement over the reference paper.

[8]The root-mean-square deviation (RMSD) of atomic positions is a measure of the average distance between the atoms of a molecular structure.

D Diffusion models

In [27] diffusion probabilistic models are introduced. They are parameterized Markov chains that model how data points transition through the latent space and can be written as a mixture of the latent variables $x_1, \ldots, x_T$,

$$p_\theta(x_0) = \int p_\theta(x_0, x_1, \ldots, x_T)\, dx_1 \ldots dx_T.$$

The task is to learn p_θ by learning the joint distribution $p_\theta(x_0, x_1, \ldots, x_T)$.

Diffusion process A diffusion process is a Markov chain that gradually adds noise to a starting point x_0 [27]. In the context of diffusion models, one assumes that by moving through the latent space, more and more Gaussian noise is added to x_0.
Thus, by construction, a diffusion process q has Gaussian transition probabilities

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\sigma_t}\, x_{t-1}, \sigma_t\, I) \;\text{ for }\; t = 1, \ldots, T.$$

Learning the reversed process Assume that the transition of the data through the latent space can be described by a diffusion process ('forward process') q with Gaussian transition probabilities. By the Markov property, learning the so-called 'reverse process' $p_\theta(x_0, x_1, \ldots, x_T)$ can be simplified to learning the transition probabilities $p_\theta(x_{t-1} \mid x_t)$,

$$p_\theta(x_0, x_1, \ldots, x_T) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t), \text{ with}$$
$$p(x_T) = \mathcal{N}(x_T; 0, I) \;\text{ and }\; p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t); \Sigma_\theta(x_t, t)).$$

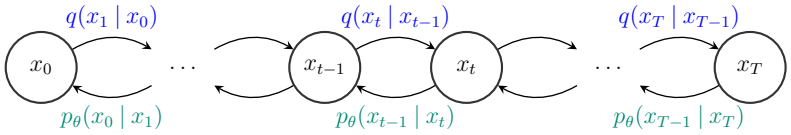


Figure 9: Illustration of the diffusion process and reverse process

It can be shown that the training objective is equivalent to the NCSN objective [24].

Relevance There is a fundamental difference between the previous score-based frameworks and a diffusion model. While the previous methods learn the scores and apply score-based refinements, the diffusion framework yields a generative model (the learned 'reverse process' p_θ). In particular, using the reverse process and $x_T \sim p(x_T)$, one can directly generate a sample $x_0 \sim p_\theta(x_0)$. It is also possible to include small refinement steps by iteratively applying the transition probabilities to a current state.

An improvement of the reference paper, could be replacing the score-based refinement steps by applying transition steps to sample a relaxed particle position. However, this could potentially be error-prone and one should consider connecting this with a confidence model as discussed in the molecular docking framework [26].

E Refinement Algorithm

Algorithm 4: Refinement Step

```
for i ← 1 to L do
    for j ← 1 to K do
        Score Graph Preprocessing:
            Construct score graph
        Score GNN:
            Obtain scores for position refinement: scores = ScoreGNN(score_graph)
        Add Langevin Noise
    end
end
```

F Introduction to basics of SPH

In SPH, a kernel function weights the contributions of particles in position x_j according to their distance to the particle in concern at position x_i as follows:

$$W_{ij} = W\left(\frac{||x_i - x_j||}{h}\right) = W(R). \tag{7}$$

For our SPH solver, we have implemented the Quintic Spline kernel function defined as follows:

$$W_{ij} = \alpha_d \begin{cases} (3-R)^5 - 6(2-R)^5 + 15(1-R)^5 & 0 \leq R < 1 \\ (3-R)^5 - 6(2-R)^5 & 1 \leq R < 2 \\ (3-R)^5 & 2 \leq R < 3 \\ 0 & R \geq 3. \end{cases} \tag{8}$$

Here $R = \frac{||x_i - x_j||}{h}$, h is the cut-off radius of the kernel and α_d assumes the value $7/478\pi h^2$ for 2D simulation scenarios.
Any field quantity A_i associated with a particle positioned at x_i, is approximated as:

$$A_i = \sum_j V_j A_j W_{ij} = \sum_j \frac{m_j}{\rho_j} A_j W_{ij}. \tag{9}$$

where V_j,m_j,ρ_j are the volume, mass and density of the j^{th} neighbor of particle i respectively.
The incompressible Navier-Stokes essentially consists of the continuity and momentum equation stemming from conservation of mass and momentum respectively. For the SPH algorithm, we discretize these equations along with an equation of state to close the system of equations.
The density of particle i is determined from the continuity equation as follows:

$$\rho_i = \sum_j m_j W_{ij}. \tag{10}$$

The key difference in terms of equations for MR-SPH is the density update. The update for density for our solver is given as follows:

$$\rho_i = \frac{\sum_j m_j W_{ij}}{\sum_j V_j W_{ij}}. \tag{11}$$

The acceleration of the particle i is determined from the momentum equation as follows:

$$\begin{aligned} \mathbf{a_i} = \frac{d\mathbf{v_i}}{dt} = \frac{1}{m_i}\sum_j (V_i^2 + V_j^2)\left[-\tilde{p_{ij}}\frac{\partial W}{\partial x_{ij}}\mathbf{e_{ij}} + \tilde{\eta_{ij}}\frac{\mathbf{v_{ij}}}{x_{ij}}\frac{\partial W}{\partial x_{ij}}\right] + \mathbf{g_i}. \\ \text{with } \tilde{p_{ij}} = \frac{\rho_j p_i + \rho_i p_j}{\rho_i + \rho_j}, \tilde{\eta_{ij}} = \frac{2\eta_i \eta_j}{\eta_i + \eta_j}. \end{aligned} \tag{12}$$

Here $\mathbf{v_{ij}} = \mathbf{v_i} - \mathbf{v_j}$ is the relative velocity of particles i and j and $x_{ij} = x_i - x_j$ is the relative distance between interacting particles.
The Equation of State which relates the pressure and density is given as follows:

$$p = p_0\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right] + \chi. \tag{13}$$

where p_0, ρ_0, χ are the reference pressure, reference density and background pressure. The exponent γ is usually chosen as 7 in the literature to limit the density variation to 1 %.
Time integration using a second order explicit predictor-corrector scheme is shown in Equations 14, 15 and 16 to determine the velocity v_i and position x_i at time $t + \Delta t$.
In the prediction step, the intermediate velocity $\overline{\mathbf{v_i}}$ and $\overline{\mathbf{x_i}}$ at the intermediate step $t + \frac{\Delta t}{2}$ are predicted as follows:

$$\begin{cases} \overline{\mathbf{v_i}}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right) = \mathbf{v_i}(\mathbf{t}) + \frac{\Delta t}{2}\mathbf{a_i}(\mathbf{t}). \\ \overline{\mathbf{x_i}}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right) = \mathbf{x_i}(\mathbf{t}) + \frac{\Delta t}{2}\mathbf{v_i}(\mathbf{t}). \end{cases} \tag{14}$$

The corresponding density and pressure at this intermediate time step is obtained using Equations 10 and 13 respectively. Using this intermediate information, the acceleration is evaluated at this intermediate time-step using Equation 12 which is used for correction as follows:

$$\begin{cases} \mathbf{v_i}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right) = \mathbf{v_i}(\mathbf{t}) + \frac{\Delta t}{2}\mathbf{a_i}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right). \\ \mathbf{x_i}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right) = \mathbf{x_i}(\mathbf{t}) + \frac{\Delta t}{2}\mathbf{v_i}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right). \end{cases} \tag{15}$$

The final particle position and velocity at $t + \Delta t$ is obtained as:

$$\begin{cases} \mathbf{v_i}\left(\mathbf{t} + \mathbf{\Delta t}\right) = 2\mathbf{v_i}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right) - \mathbf{v_i}(\mathbf{t}). \\ \mathbf{x_i}\left(\mathbf{t} + \mathbf{\Delta t}\right) = 2\mathbf{x_i}\left(\mathbf{t} + \frac{\mathbf{\Delta t}}{\mathbf{2}}\right) - \mathbf{x_i}(\mathbf{t}). \end{cases} \tag{16}$$

As far as the step size Δt is concerned, it is constrained by the Courant $-$ Friedrichs $-$ Lewy (CFL) condition given by:

$$\Delta t \leq \min\left(0.25\,\frac{h}{c}, 0.25\,\min\left(\frac{h}{\mathbf{a_i}}\right), 0.125\,\frac{h^2}{\nu}\right). \tag{17}$$

here c is the speed of sound and ν is the kinematic viscosity.