A Hybrid Framework for Fluid Flow Simulations: Combining SPH with Machine Learning

Rene Winchenbach and Nils Thuerey
Technical University Munich
Munich, Germany
{rene.winchenbach,nils.thuerey}@tum.de

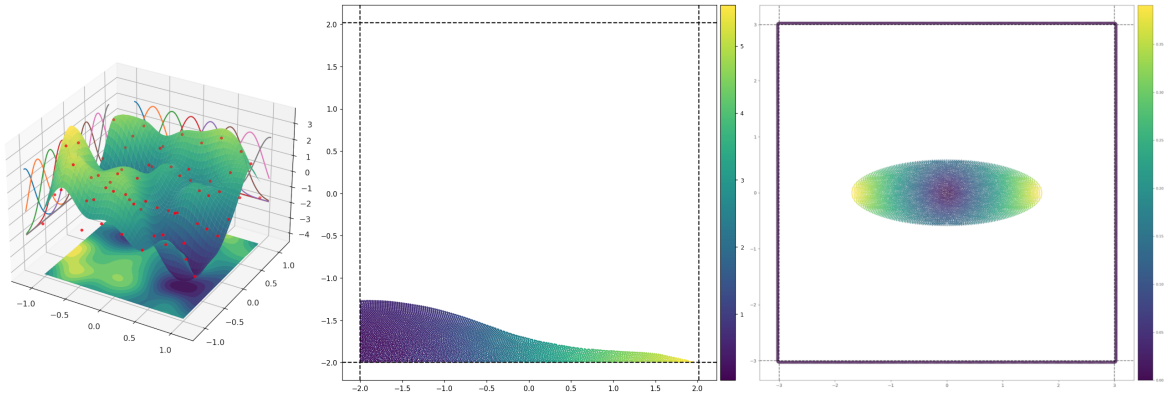


Fig. 1: Our simulation framework allows for a direct connection between the machine learning side, e.g., using spline convolutions (left), and SPH simulations (middle and right, velocity color coded).

***Abstract*—Machine Learning and Data Science have been a quickly growing and highly impactful field of research; however, thus far they find virtually no adoption within Smoothed Particle Hydrodynamics, due to a variety of fundamental issues. Recent machine learning approaches have introduced a variety of approaches, e.g., graph convolutional networks [1], continuous convolutions [2] and graph neural networks [3]; however, these approaches are often not validated as is typical within the CFD/SPHERIC community. The present paper aims to provide a foundation that would enable a more tight connection between the two fields. We achieve this by providing an Open Source SPH simulation built upon the PyTorch machine learning framework, using a variety of fluid and boundary treatments. By utilizing PyTorch as the underlying foundation we can readily utilize a variety of acceleration techniques, e.g., GPU accelation, whilst retaining a high level of abstraction in the source code. By utilizing PyTorch, we can directly link our simulation framework with machine learning approaches, which are also implemented within our overall framework. Basing the entire codebase on python in this regard also enables the framework being used in online platforms, e.g., Google Colab, enabling researchers and students to work with our framework without requiring personal hardware. Utilizing the this tight coupling, it is readily possible to evaluate the ability of a machine learning approach to replace components of the SPH simulation, e.g., an SPH density summation, or even the entire simulation step. We include a variety of traditional benchmark scenarios, e.g., oscillating drops, breaking dam scenarios and flows past obstacles, for validation, as well as scripts to generate randomized data for training. Our codebase is available online under an MIT license at https://github.com/wi-re/pytorchSPH.**

I. Introduction

Smoothed Particle Hydrodynamics (SPH) [4] is a well established numerical method for simulating fluid flows and other dynamic systems, offering a mesh-free, Lagrangian approach that is adept at handling complex geometries and large deformations. Meanwhile, the field of deep learning has experienced rapid advancements [5], resulting in a wide range of applications, including computer vision [6], natural language processing [7], and even for simulations [8]. The intersection of these two domains presents a broad area for novel research opportunities and applications in areas such as computer graphics [9], astrophysics [10], and environmental science [11]. However, applications to engineering, and especially Computational Fluid Dynamics (CFD) have only recently been emerging [12]. This paper aims to deliver an initial exploration of the integration of deep learning techniques with SPH simulations, providing an exposition of the underlying concepts, frameworks, and toy examples designed to build a foundation for deep learning in the SPH community.

In this work, we begin by examining the theoretical foundations of learning convolutions within the context of SPH, focusing on deep parametric convolutions [1] and continuous convolutions [9]. Additionally, we will discuss the PyTorch [13] and PyTorch Geometric [14] frameworks and their integration with SPH simulations. To promote a practical understanding of the core concepts associated with SPH and

deep learning, we provide a simple density function example, accompanied by a discussion of computational considerations and optimization using Torch JIT script.

Moreover, we will provide a example of setting up a network for replacing density estimations in SPH starting from generating training data to an analysis of hyperparameters and how they influence the achievable results. In addition, we will introduce the concept of Perlin noise [15] as a mechanism for generating synthetic data and offer heuristics for deep learning in the context of SPH. By presenting case studies that illustrate both successful and unsuccessful examples, we aim to provide valuable insights for practitioners and researchers alike. In conclusion, we will present a forward-looking perspective on the future of deep learning with SPH, highlighting emerging trends and potential applications.

II. Convolutional Operators

Convolutional Neural Networks (CNN) [16] are a mathematical procedure for processing structured data and have found wide usage within deep learning, e.g., for image classification. CNN operations involve the application of *filters* (also often referred to as kernels in the literature, which should not be confused with SPH kernel functions), to a structured input data. Given an image of dimension $W \times H$, with spatial indices $x \in [0, W-1]$ and $y \in [0, H-1]$, and a filter $G(m,n)$ with filter indices m and n, the convolutional operator $\otimes$ can be expressed as:

$$\begin{aligned} C(x,y) &= F(x,y) \otimes G(m,n) \\ &= \sum_i \sum_j F(x+i, y+j) G(i,j), \end{aligned}$$

where the summations are performed over the filter dimensions. This operation is applied to every point in the input data and yields a feature map C. In the context of deep learning, the filter function G is evaluated using a matrix of parameters Θ, i.e., $G(i,j) = \Theta_{i,j}$, in which case the application of the filter can be efficiently implemented using matrix multiplications. However, this process is restricted to structured inputs, e.g., grid based simulations with integer coordinates, and is not directly applicable to unstructured simulations, such as SPH.

Instead of a discrete fixed filter G, a continuous convolution with a continuous filter function $g(x,y)$, for $x, y \in \mathbb{R}$, can be applied to an unstructured input $f(x,y)$ as

$$\begin{aligned} c(x,y) &= (f \otimes g)(x,y) \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y) g(x-x^\star, y-y^\star) dx^\star dy^\star. \end{aligned}$$

The Deep Parametric Continuous Convolutional Neural Network (DPC-Net) approach by Wang et al [1] realizes the filter function $f(x,y)$ using a multilayer perceptron (MLP). An MLP, in general, is a type of feedforward artificial neural network that consists of multiple layers of interconnected neurons [17]. It is composed of an input layer, in this case with 2 input neurons, one or more hidden layers, and an output layer, in this case with 1 output neuron. Each neuron in a layer receives input from the neurons in the previous layer, performs a weighted sum over its inputs followed by an application of an activation function, and passes the results to the neurons in the next layer. MLPs can learn complex, non-linear relationships between inputs and outputs by adjusting the weights of each input for each neuron and potential biases.

Similar to the motivation underpinning SPH, using such a filter function would not be practical as it has infinite support, analogous to a Gaussian kernel for SPH, and instead only compact support domains are utilized. Analogous to SPH, given an input of points (also referred to as vertices) local neighborhoods can be constructed based on either a support radius (analogous to SPH) or based on a fixed number of neighbors. These neighborhoods are then represented as edges connecting vertices, which yields a graph where the convolutional filter function f determines the weights on the edges according to the distance between points. These approaches are commonly referred to as Graph Neural Networks (GNNs) [18] and have found wide utilization for many tasks including segmentation of 3D point clouds [19] and image recognition [20].

Note that, in general, the filter function f determined by an MLP is neither radially symmetric nor does it need to have a spherical influence domain. This makes such GNNs very adaptable to many tasks, but by including more information, also called inductive biases, in the design of a neural network, the performance of the network can potential be improved significantly. One such inclusion of inductive biases can be found in the Continuous Convolution approach (CConv) by [9], where instead of utilizing an arbitrary MLP for the edge weights, an $n \times m$ grid of support points is utilized that spans the support domain where the contribution of points in between these support points is determined using a linear interpolation. Furthermore, an additional filter function, called window function, is utilized that is often based on SPH kernel functions such as the poly 6 kernel function [21], which ensures a spatially compact and spherical influence on vertices.

Within the context of GNNs, SPH can be viewed as a graph network by considering the particles and their interactions as nodes and edges, respectively, in the graph. In this representation, each particle in the simulation corresponds to a node, and the relationships between particles are represented by edges connecting the nodes. The edges in this graph can be either directed or undirected, depending on the nature of the SPH interactions.

The weights of the edges are determined by the kernel function, which is a smooth, radially symmetric function that depends on the distance between the particles. The kernel function effectively models the influence of neighboring particles on a given particle, thus capturing local interactions. As a result, the graph structure formed by the SPH particles resembles a weighted, undirected graph where edge weights represent the kernel function values between the particles.

In this graph representation, the SPH simulation can be seen as a process of information propagation across the network,

where each particle's properties (e.g., density, pressure, and velocity) are updated based on the weighted contributions from its neighbors. This view aligns with the core principles of GNNs, which aim to learn meaningful representations of graph-structured data by aggregating local information from neighboring nodes.

III. PyTorch

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR) [13]. While PyTorch is primarily focused on deep learning, it also supports other machine learning algorithms and offers features useful to problems outside of machine learning. Some key features of PyTorch include its dynamic computation graph (eager execution), which allows for easy debugging and modification of models during runtime, strong GPU acceleration support, and a large ecosystem of tools and libraries. At its core, PyTorch is built on tensors, which are multi-dimensional arrays that serve as the fundamental data structure for building and manipulating data. They are very similar to NumPy [22] arrays, but with the added advantage of GPU support for faster computation. Tensors can hold data of various types, such as integers or floating-point numbers, and they can be used for various operations like element-wise addition, matrix multiplication, reshaping, and more. PyTorch tensors are highly optimized for efficient computation and support automatic differentiation, which is crucial for backpropagation in neural networks. The following is a simple example using PyTorch to create a random 2×3 tensor (i.e., a 2D Matrix), demonstrating first element-wise addition of tensors, reshaping of tensors into other shapes and finally matrix matrix multiplication:

```python
import torch
# Create a 2x3 tensor filled with zeros
tensor_zeros = torch.zeros(2, 3)
print(f"Zeros tensor:\n{tensor_zeros}")

# Create a 2x3 tensor filled with random values
tensor_random = torch.rand(2, 3)
print(f"\nRandom tensor:\n{tensor_random}")

# Element-wise addition of tensors
tensor_sum = tensor_zeros + tensor_random
print(f"\nSum of tensors:\n{tensor_sum}")

# Reshaping the tensor
tensor_reshaped = tensor_sum.view(3, 2)
print(f"\nReshaped tensor:\n{tensor_reshaped}")

# Matrix multiplication
tensor_mult = torch.matmul(tensor_reshaped,
    tensor_sum)
print(f"\nMatrix multiplication:\n{tensor_mult}")
```

An important aspect of PyTorch, and other machine learning libraries, is automatic differentiation. Automatic differentiation (autograd) [23] is a technique used in computational mathematics and machine learning to automatically and efficiently compute the derivatives (gradients) of functions with respect to their input variables. It is a key component in the optimization process of neural networks, as it enables efficient calculation of gradients needed for backpropagation and weight updates. Furthermore, automatic differentiation can also be used to realize differentiable solvers, which can be very useful for learning and optimization tasks.

In PyTorch, automatic differentiation is implemented using the *autograd* package. PyTorch tensors have an attribute called *requires_grad*, which, when set to *True*, keeps track of all operations performed on the tensor. This creates a computation graph, where each node represents a tensor and each edge represents an operation that generates an output tensor from one or more input tensors.

When the computation graph is complete (i.e., the forward pass of a neural network is finished), calling the *backward()* method on the final output tensor triggers the backpropagation process. During backpropagation, PyTorch computes the gradients of the output tensor with respect to each tensor in the graph that has *requires_grad=True*. These gradients are then stored in the *grad* attribute of each tensor, which can be accessed for updating the model weights during the optimization process. The following is a simple example of computing the derivative of $z = x^2 + y^2$ with respect to x and y, i.e., $\frac{\partial}{\partial x} z\Big|_{x=2,y=3}$ and $\frac{\partial}{\partial y} z\Big|_{x=2,y=3}$:

```python
import torch

# Create two tensors with requires_grad=True
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)

# Perform some operations on the tensors
z = x * x + y * y

# Call backward() to compute gradients
z.backward()

# Check the gradients
print(f"Gradient of z with respect to x: {x.grad}")
print(f"Gradient of z with respect to y: {y.grad}")
```

So far, we have only discussed using built-in operations, such as element-wise additions, however, an SPH simulation generally requires more complex computations. As we are using python as the backend for our simulation, writing for loops explicitly tends to be overly slow due to the interpreted nature of python and does not natively support parallelization across a GPU. TorchScript, through Just-In-Time (JIT) compilation, can optimize the computational performance of your PyTorch code, including operations like looping over vectors, by compiling the code to a lower-level intermediate representation (IR) before execution. This process can enable various optimizations, such as loop unrolling, operation fusion, and dead code elimination, which can lead to more efficient execution of the code. Note that this, effectively, is equivalent to compiling code before running it instead of interpreting the code on the go. The following is an example using torch jit script to perform an element-wise addition of two vectors with 1000 random elements each:

```python
import torch
from torch import jit

@jit.script
```

```
def loop_add(a: torch.Tensor, b: torch.Tensor) ->
    torch.Tensor:
    assert a.size() == b.size(), "Input tensors must
     have the same size"
    result = torch.zeros_like(a)
    for i in range(a.size(0)):
        result[i] = a[i] + b[i]
    return result

a = torch.rand(1000)
b = torch.rand(1000)

# Call the TorchScript function
c = loop_add(a, b)
```

In this example, we define a function *loop_add* that performs element-wise addition of two tensors using a loop. We use the *@jit.script* decorator to convert the function into a TorchScript function. It is important to note that while TorchScript may apply some optimizations to this loop, it's still recommended to use the built-in vectorized operations provided by PyTorch whenever possible for better performance.

Finally, PyTorch Geometric [14] is an extension library for PyTorch that focuses on geometric deep learning, providing tools to work with graph-structured data and perform graph neural network operations. It simplifies the implementation of graph-based machine learning models by offering efficient GPU-accelerated operations, easy-to-use data handling, and various pre-implemented graph neural network layers. Furthermore, it includes support for various tasks that are common in SPH, such as neighbor searches and passing information along edges of a graph. In the following example,we use the *radius* function to find the neighbors of a random set of 10 particles with a fixed search radius of 0.3 and then utilize the *scatter-add* function to sum up a random value assigned to each particle, i.e., we compute $\langle A_i \rangle = \sum_j A_j$ for $|\mathbf{x}_i - \mathbf{x}_j| \leq 0.3$:

```
import torch
from torch_geometric.nn import radius
from torch_scatter import scatter_add

# Generate random 2D points and random values
points = torch.rand(10, 2)
values = torch.rand(10, 1)

# Specify the radius for the neighborhood search
search_radius = 0.3

# Perform radius-based neighbor search
adj_matrix = radius(points, points, r=search_radius,
     batch=None)

# Gather source values from the adjacency matrix
source_values = values[adj_matrix[1]]

# Perform scatter_add to sum up the values
sum_values = scatter_add(source=source_values, index
    =adj_matrix[0], dim_size=points.size(0))

print("Values per point:")
print(values)
print("\nSum of values per point:")
print(sum_values)
```

In this example the neighborhoods are represented as an adjacency matrix, which is a $2xn$ dimensional structure that contains all particle to particle pairings $i \times j$ based on a global search radius.

IV. TorchSPH

After covering the fundamentals of PyTorch, we can now move to describing our simulation framework. The goal of our framework was two-fold as our goal was to provide an easily extensible and versatile codebase, via the usage of Python, and an easy integration with deep learning frameworks, via the usage of PyTorch. Within our framework we generally distinguish three types of code as (i) general and utility functions that are universally used, e.g., kernel functions, (ii) modules that implement specific SPH simulation techniques, e.g., kinematic viscosity and (iii) simulators that collect modules and handle the processing of configurations, time integration and file output. Before discussing the second and third kind of code in general, discussing some aspects of our implementation is warranted.

There are many potential designs for SPH frameworks [24]–[26] and how processing an individual SPH interpolation can be performed. As we aim to be similar in concept to GNNs, our choice of code design is intended to use the same building blocks as a GNN. As an example consider the following example of a density summation:

```
@torch.jit.script
def density(radialDistances, areas, restDensities,
    neighbors, support):
    with record_function("sph - density summation"):
        i,j = neighbors
        message = kernel(radialDistances, support) *
     areas[j] * restDensities[j]
        rho =  scatter_sum(message, i, dim=0,
    dim_size=areas.shape[0])
        return rho
```

This function is given the edge weights of our simulation graph, i.e., the euclidean distances between particles, in the *radialDistances* parameter, the area and rest density of each particle, the adjacency matrix containing the neighbors and a global constant support radius. This information is used to construct a message $m_{i,j} = a_j \rho_{j,0} W_{ij}$ for each particle pairing and the messages are aggregated along the edges of the graph using a summation operation. Accordingly, this realizes an SPH density summation $\rho_i = \sum_j m_j W_{ij}$ using a graph based conceptual model and implements this using *@torch.jit.script* to allow for efficient performance. The use of *record_function* enables the use of performance profiling tools commonly used in deep learning.

Each module of our simulation framework now consists of a collection of these message passing operations that work on either the entire simulation graph or only parts of it. To handle boundary handling this process is slightly inverted as a boundary handling approach changes how messages between boundaries and fluid particles are processed, i.e., the implementation is different for different boundary handling approaches. To realize this, we utilize a *BoundaryHandling* class that handles the generation and export of boundaries in the SPH simulation and that is required to implement

the various processes required, e.g., the *BoundaryHandling* class contains an *evalBoundaryDensity* abstract function that computes the density contribution of boundaries on fluid particles and computes the density of the boundary itself. It is important to consider that modules are not very uniform in their complexity as some modules only require a few lines of code, e.g., the density summation described before, whereas other modules, e.g., the implicit shifting module, require hundreds of lines of code.

In general, the modular design is intended to be flexible, extensible and maintainable by providing a common and simple interface with a compartmentalization of configuration processing. For our simulation framework we chose to utilize the TOML language to configure the simulation as it is a human-readable and easily parseable configuration format. By using a human-readable format and restricting the configuration to a single TOML file, configurations might be limited in their ultimate complexity, but as we aim to provide an easy to use and understand framework, we chose to make this trade-off. A given TOML configuration file is not processed at once by a central function but is delegated to each module in a simulation which can process the configuration as required. To provide a general basis for these configurations, we utilize a *Parameter* class that ensures that parameters exist in the configuration, if they are required, and that they are assigned plausible default values.

Based on the modules, the third type of code (simulators), works by using a common base class (*SPHSimulation*) to handle common tasks, such as time integration and parameter processing, and combining this base with simulator specific functionality. In our current framework we provide a δ-SPH [27] based simulator, using a continuum formulation for the density formulation and an explicit pressure term based on the Tait equation, and a *DFSPH* [28] based simulator using a summation density formulation and an implicit pressure solver to ensure incompressibility. Simulators may use very different underlying modules but are only required to implement two functions. These two functions are the constructor, which handles the gathering of modules and loading of the configuration file and a *timestep* function that give the current *simulationState* computes a rate of change for the fluid velocity, position and density. Integrating these rates of change is performed in the base class and supports various integration schemes such as an explicit Euler integration or RK4. By combining all of this, the following example shows how a simulation can be loaded, ran for 3200 timesteps and the results stored (excluding imports for readability):

```
config = 'configs/dambreak_deltasph.toml'
parsedConfig, simulationModel = loadConfig(config)
sphSimulation = simulationModel(parsedConfig)
sphSimulation.initializeSimulation()
for i in tqdm(range(3200)):
    sphSimulation.integrate()
sphSimulation.outFile.close()
```

Regarding computational considerations, we were able to implement virtually all parts of the simulation directly using PyTorch JIT script, however, there is one exception. A general problem within Python are for loops due the interpretive nature of Python, and even within PyTorch JIT script these can be difficult to implement if loops are contained within other loops and have variable lengths. This does not, generally, occur within SPH, except two exceptions: (i) the actual summation, which we were able to efficiently implement using PyTorch Geometric's scatter operation, and (ii) the construction of neighborlists. While PyTorch Geometric does offer a radius search that we utilized before, the algorithm used does a simple each-against-each search for neighbors, i.e., of complexity $\mathcal{O}(n^2)$, which is not useful for larger simulations. Accordingly, we utilized the neighbor search of [29] and implemented this process directly in C++ for the CPU implementation and CUDA for the GPU based implementation. Note that including such code is straight forward in PyTorch and can be compiled just in time, even when running the code in an online environment.

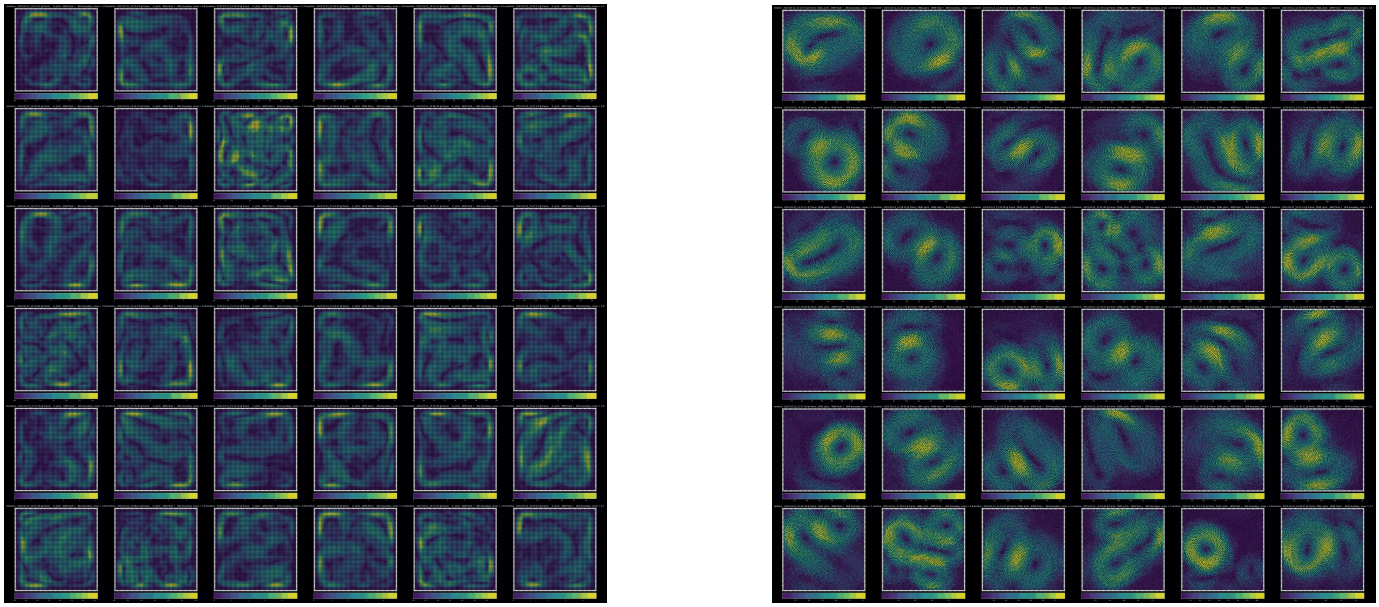
V. Data Generation

Data generation and collection play a crucial role in deep learning, particularly when the goal is to learn the underlying principles governing physical systems. A well-curated dataset can facilitate the training process, enabling a model to learn essential features and relationships within the data. In the context of learning physical systems, such as fluid dynamics simulations, generating a diverse and realistic dataset is paramount for the model's ability to generalize and accurately predict new scenarios.

In our work, we synthesize a dataset by simulating an enclosed box filled with liquid, with no gravity applied to it. This setup allows us to focus on the fluid's inherent behavior and the interactions between particles, without the influence of external forces. Furthermore, this limits the complexity of the problem, i.e., there are no free surfaces, which makes the learning task more narrow and more realistic to achieve. Creating variance in the dataset is essential for ensuring that our model can learn the underlying physics of the system, generalize well, and handle complex scenarios.

To introduce variance and interesting physics into our dataset, we employ an initial velocity field generated using isotropic Perlin noise [15] on the potential function. This technique results in a so-called *curl noise*, which is divergence-free and provides a physically plausible initial condition for the fluid particles. The use of Perlin noise enables us to have a strong control over the complexity of the initial velocity, i.e., we can control the frequency and amplitude of the velocity field, to change the complexity of the simulations. Furthermore, by utilizing Octave noise, which is a summation of noise fields of doubling frequencies, we can efficiently generate complex flow phenomena for this seemingly simple flow scenario.

There are two ways to compute this velocity field as we can either compute the potential field on a high resolution underlying mesh, using mesh finite difference operators to



(a) Initial simulation states

(b) Final simulation states

Fig. 2: Our dataset consists of 36 different randomly initialized simulations. Each simulation contains 4096 particles that are initially placed on a rectilinear sampling and then simulated over time using δ-SPH.

find the velocity and sampling the velocity field at the initial particle positions and sampling the potential field on the initial particle positions and using an SPH operator to compute the initial velocity field. Only the latter approach results in a divergence free velocity field, as evaluated using SPH divergence operators, whereas the former results in an initially non divergence free velocity field that results in undesired shocks. Overall, our data setup generates 4096 fluid particles (using a regular 64x64 sampling) and generates 36 different simulations for training using different initial seeds for the velocity field, see Fig.2

VI. Learning Density

While trying to learn the behavior of the underlying PDE in our simulation, i.e., the incompressible Navier-Stokes equation, is a very interesting approach, practically achieving this goal to within an acceptable precision in CFD is not practically possible using current state of the art techniques. However, by understanding how current state of the art methods perform on simpler problems, we aim to foster a strong foundation for future avenues of research. To this extent, we focus on learning the SPH density summation and observing how different hyperparemters influence the accuracy of the result. Accordingly, the task of the network is that given a set of particle positions p_i, $i \in [0, 4096)$, and a constant feature vector of $f_i = 1$ for every particle, to compute the density $\rho_i = \sum_j m_j W_{ij}$. As all particles in our dataset have identical masses, i.e., $m_i = \text{const}$, the learning task is to learn a scaled kernel function $\hat{W}_{ij} = mW_{ij}$ based on the edge weights, i.e., the euclidean distance between particles $\mathbf{x}_{ij}$. Within our

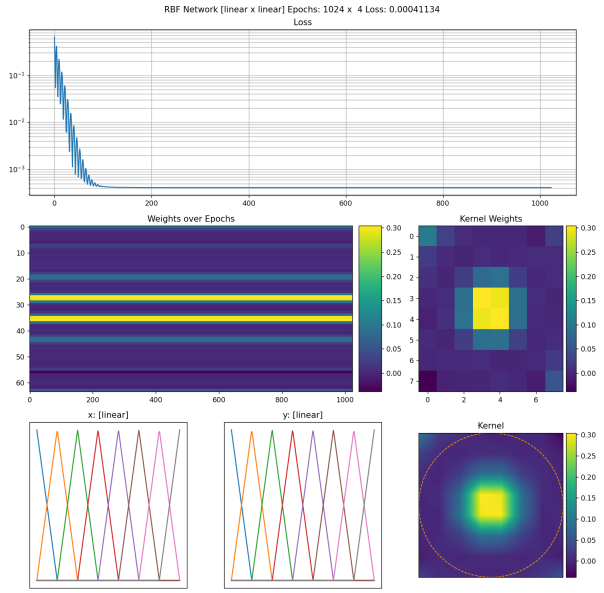


Fig. 3: Training Results for learning an SPH density summation on the Perlin dataset using Cartesian distances

simulation we utilize the Wendland2 kernel function given as

$$W(\mathbf{x}, h) = \frac{7}{\pi}\frac{1}{h^2}\left[1 - \frac{|\mathbf{x}|}{h}\right]_+^4 \left(1 + 4\frac{|\mathbf{x}|}{h}\right), \qquad (1)$$

As we use the CConv approach as our underlying network architecture, the goal is to now learn a function $f(\mathbf{x}, \Theta)$, where Θ describes the set of parameters for a bilinear interpolation in

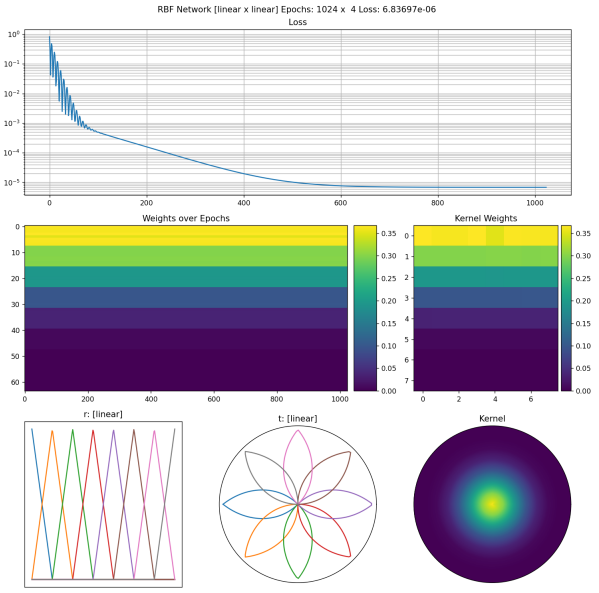


Fig. 4: Training Results for learning an SPH density summation on the Perlin dataset using Polar distances

2D. As discussed before, using a window function can significantly improve the stability of the CConv approach, however, using a window function based on the Wendland2 kernel in this case trivializes the learning problem and, accordingly, we do not utilize a window function for our experiments. Regardless, using a window function that is identical to the underlying kernel function can serve as a sanity check of the training setup as in this case the learned function $f(\mathbf{x}, \Theta)$ should equal the global mass m for all inputs, which is achieved using $\Theta = m$. By starting from different initial guesses for Θ we can observe the convergence of a network to this trivial solution to help in adjusting some hyperparameters. However, for brevity we skip this optimization here and move on to learning the aforementioned density operation.

There are many hyperparameters in a CConv network, e.g., the number of layers, number of weights per filter, number of features per layer, choice of activation function, choice of window function, training parameters, mini batching choices, data augmentation amongst many more. While discussing all of these parameters would be interesting, such an in-depth discussion is far beyond the scope of our paper and we will, accordingly, only focus on a two aspects here. The two aspects we chose are the coordinate mapping for the CConv layer and the influence of the training setup for various numbers of weight per filter.

When we discussed the CConv operation before, we only described the learned function $g(\mathbf{x}, \Theta)$ depending on the euclidean distance, i.e., the edge weights, of particle pairs. However, this distance does not necessarily need to be in cartesian space and, due to the spherical nature of the neighborhoods, other choice can be more useful. Whilst the original CConv paper utilizes a volume preserving mapping [9], we chose a polar coordinate transformation and compare it against the cartesian variant, see Figs3 and 4. What can be seen from this experiment is that the polar variant results in a much better approximation of the density summation, i.e., a loss of 10^{-5} instead of 10^{-3}, and the resulting function is a much better approximation of what an SPH kernel function should look like, at an equivalent number of parameters. However, whilst the polar coordinate does yield a lower overall result, the training process also takes more iterations and can, in more complex examples, yield problems as there is an over parametrization for close particle pairs.

Whilst the previous experiment was performed on a fairly diverse and large training dataset, a more naïve dataset for this density interpolation would consist of regularly spaced particles with some jitter added to the positions of particles to create some minor variance in the data. Furthermore, a naïve approach may be to increase the number of weights per filter to provide a close approximation of an SPH kernel. Note that this experiment was performed with the volume preserving mapping of the original CConv approach and, accordingly, the visualized filters are distorted w.r.t. the original cartesian simulation space. Performing this experiment, see Fig5, we can, however, observe several problems. While on the one hand, increasing the number of weights per filter can result in a filter that has a smoother visual appearance, training a large filter with a very low amount of jitter results in a filter that is zero almost everywhere as no particle pairings during training ever were at this specific distance relative to each other. Increasing the amount of jitter in this naïve setup does improve the results, in regards to filling out the larger filter everywhere, the results still appear noisy as the convergence rate is relatively low and ensuring an isotropic distribution of particle pairings across the dataset is not practically possible. A very important take away from this result is that using an *ideally* incompressible SPH simulation for training may not be useful for training neural networks as particle shifting techniques and incompressibility ensure a minimum (and uniform) spacing between particles during training. This, during inference, then leads to problems as any deviation of the NN from the solver it was trained on will lead to particles moving into relative distances that were never, or only sparringly, observed during training.

VII. Limitations and Outlook

In this paper we have given a brief overview of our PyTorch based SPH simulation framework and provide some brief insights into what happens when a NN is trained to replace the density summation of an SPH simulation. Based on these insights, we hope to provide some insights into the process of developing neural networks for SPH simulations in a way that can foster future research. While the results we presented here are limited in their scope, partially due to constraints of space and partially due to the lack of neural networks that can achieve acceptable numerical accuracy for a CFD application, our framework can be utilized to develop neural networks that could perform entire simulation steps.

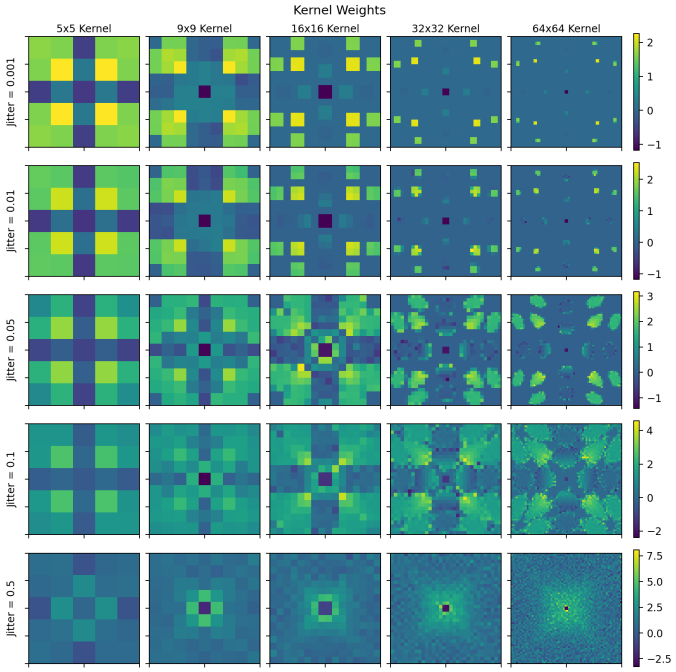


Fig. 5: Training results for learning an SPH density summation on a regular particle distribution for a volume preserving coordinate mapping using different amounts of jitter during training and different filter sizes

References

[1] S. Wang, S. Suo, W.-C. Ma, A. Pokrovsky, and R. Urtasun, "Deep parametric continuous convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2589–2597.

[2] L. Prantl, B. Ummenhofer, V. Koltun, and N. Thuerey, "Guaranteed conservation of momentum for learning particle-based fluid dynamics," 2022.

[3] A. A. Ramabathiran and P. Ramachandran, "Spinn: sparse, physics-based, and partially interpretable neural networks for pdes," *Journal of Computational Physics*, vol. 445, p. 110600, 2021.

[4] J. J. Monaghan, "Smoothed particle hydrodynamics," *Annual review of astronomy and astrophysics*, vol. 30, no. 1, pp. 543–574, 1992.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[6] A. Voulodimos, N. Doulamis, A. Doulamis, E. Protopapadakis *et al.*, "Deep learning for computer vision: A brief review," *Computational intelligence and neuroscience*, vol. 2018, 2018.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[8] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia, "Learning to simulate complex physics with graph networks," in *International conference on machine learning*. PMLR, 2020, pp. 8459–8468.

[9] B. Ummenhofer, L. Prantl, N. Thuerey, and V. Koltun, "Lagrangian fluid simulation with continuous convolutions," in *International Conference on Learning Representations*, 2019.

[10] P. Lemos, M. Cranmer, M. Abidi, C. Hahn, M. Eickenberg, E. Massara, D. Yallup, and S. Ho, "Robust simulation-based inference in cosmology with bayesian neural networks," *Machine Learning: Science and Technology*, vol. 4, no. 1, p. 01LT01, 2023.

[11] J. A. Weyn, D. R. Durran, R. Caruana, and N. Cresswell-Clay, "Sub-seasonal forecasting with a large ensemble of deep-learning weather prediction models," *Journal of Advances in Modeling Earth Systems*, vol. 13, no. 7, p. e2021MS002502, 2021.

[12] L.-W. Chen, B. A. Cakal, X. Hu, and N. Thuerey, "Numerical investigation of minimum drag profiles in laminar flow using deep learning surrogates," *Journal of Fluid Mechanics*, vol. 919, p. A34, 2021.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[14] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[15] R. Bridson, J. Houriham, and M. Nordenstam, "Curl-noise for procedural fluid flow," *ACM Trans. Graph.*, vol. 26, no. 3, p. 46, 2007.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*. NIPS, 2012, pp. 1097–1105.

[17] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1998.

[18] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. A. Riedmiller, R. Hadsell, and P. W. Battaglia, "Graph networks as learnable physics engines for inference and control," in *International Conference on Machine Learning*, 2018.

[19] M. Fey, J. E. Lenssen, F. Weichert, and H. Müller, "Splinecnn: Fast geometric deep learning with continuous b-spline kernels," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 869–877.

[20] Z.-M. Chen, X.-S. Wei, P. Wang, and Y. Guo, "Multi-label image recognition with graph convolutional networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 5177–5186.

[21] M. Müller, D. Charypar, and M. Gross, "Particle-Based Fluid Simulation for Interactive Applications," *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animat.*, no. 5, pp. 154–159, 2003.

[22] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[23] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[24] R. Winchenbach. (2019) openmaelstrom. Accessed: 2020-08-04. [Online]. Available: http://www.cg.informatik.uni-siegen.de/openMaelstrom

[25] P. Ramachandran, A. Bhosale, K. Puri, P. Negi, A. Muta, A. Dinesh, D. Menon, R. Govind, S. Sanka, A. S. Sebastian, A. Sen, R. Kaushik, A. Kumar, V. Kurapati, M. Patil, D. Tavker, P. Pandey, C. Kaushik, A. Dutt, and A. Agarwal, "PySPH: A Python-based Framework for Smoothed Particle Hydrodynamics," *ACM Transactions on Mathematical Software*, vol. 47, no. 4, pp. 1–38, Dec. 2021.

[26] J. M. Domínguez, G. Fourtakas, C. Altomare, R. B. Canelas, A. Tafuni, O. García-Feal, I. Martínez-Estévez, A. Mokos, R. Vacondio, A. J. Crespo *et al.*, "Dualsphysics: from fluid dynamics to multiphysics problems," *Computational Particle Mechanics*, vol. 9, no. 5, pp. 867–895, 2022.

[27] S. Marrone, M. Antuono, A. Colagrossi, G. Colicchio, D. Le Touzé, and G. Graziani, "δ-sph model for simulating violent impact flows," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13-16, pp. 1526–1542, 2011.

[28] J. Bender and D. Koschier, "Divergence-free smoothed particle hydrodynamics," in *Proceedings of the 14th ACM SIGGRAPH/Eurographics symposium on computer animation*. ACM, 2015, pp. 147–155.

[29] R. Winchenbach and A. Kolb, "Multi-level-memory structures for adaptive sph simulations," 2019.