Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Learning from Differentiable Physics to Simulate Liquids with Graph Networks

Jonathan Klimesch

Technical University of Munich
Department of Informatics

Bachelor's Thesis in Informatics

Learning from Differentiable Physics to Simulate Liquids with Graph Networks

Lernen von Differenzierbarer Physik um Flüssigkeiten mit Graph Netzwerken zu Simulieren

Author: Jonathan Klimesch
Supervisor: Prof. Dr. Nils Thuerey
Advisor: M.Sc. Philipp Holl

Submission Date: 14.04.2021

Chair of Computer Graphics and Visualization

Declaration of Originality

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Unterschleissheim, 14.04.2021

(Jonathan Klimesch)

Abstract

Simulating complex dynamics with traditional simulators is computationally challenging and often inaccurate. Deep learning models can be an efficient alternative for simulating such dynamics and can extend or replace parts of traditional simulators. Training such models within differentiable physics frameworks, allows for stronger interaction between the models and the underlying dynamics. We extend the Φ_{Flow} framework with a differentiable *Fluid-Implicit-Particle* (FLIP) simulator for incompressible, inviscid fluids. We test the physical behavior of our simulation in different scenarios and demonstrate its differentiability in simple optimization experiments. We then use data from this FLIP simulator to train and evaluate *Graph Network-based Simulators* (GNS), a recently proposed framework for learning simulations from data. We introduce a new multi-step loss which improves the capacity of the trained models to mitigate accumulating errors, and show that GNS models trained with this loss yield competitive results compared to models using artificial noise for error mitigation. We demonstrate that the generalization capabilities of GNS models can be improved by replacing domain-specific features with pure particle interactions. Our results indicate that the GNS models do not really learn the underlying physics, but problem-specific correlations between the input velocities and output accelerations.

Contents

Chapter 1

Introduction

Simulating complex dynamics is invaluable to many fields in computer graphics and the natural sciences. Numerical simulations of such dynamics are computationally challenging and are often inaccurate in approximating the underlying physical systems [59]. Recent work suggests, that deep learning models can be an efficient and accurate alternative to traditional, hand-crafted simulation approaches [64, 36, 74]. However, learning complex physics with such data-driven models is a challenging task. Recently developed differentiable physics frameworks [38, 68, 40] simplify the interaction between deep learning models and physics, and provide useful tools for running simulations and solving optimization problems.

The first goal of this thesis is to extend the differentiable physics toolbox by implementing a differentiable Fluid-Implicit-Particle (FLIP) [7, 84] simulator for incompressible, inviscid fluids within the Φ_{Flow} framework [38]. Recently, Sanchez-Gonzales *et al.* proposed the *Graph Network-based Simulators* (GNS) framework, where they approximate complex dynamics by learned message-passing on particle graphs [64]. Our second goal of this thesis is to gain better understanding of the GNS architecture by training it on data from our FLIP simulator, and to extend the GNS framework with new training variants.

In chapter 2, we describe the Euler equations, the physical basis of our FLIP simulation. We then explain the simulation approach of the FLIP method and provide a short introduction to deep learning techniques and the graph network architecture. In chapter 3 we summarize recent works in the field of learned simulators and graph networks. Chapter 4 describes the implementation of our FLIP simulator and demonstrates its physical behavior and differentiability. Chapter 5 includes an evaluation of GNS models, which were trained on data generated by our FLIP simulator. We first evaluate models trained with artificial noise and boundary distance features as proposed by Sanchez-Gonzales *et al.* [64]. We then propose a new multi-step loss and show that GNS models using this loss without any artificial noise, produce competitive results compared to the previous models. We further show that GNS models trained on domains with obstacle

boundaries, and without boundary distance features, have greater generalization capabilities. Our evaluations indicate that the GNS models do not really learn physical dynamics, but rather problem-specific correlations between input velocities and output accelerations. In chapter 6 we describe possible improvements of our FLIP simulator and propose possible future work with the GNS framework.

Chapter 2

Theory

2.1 Fluid Simulation

This section serves as an introduction to the basic equations of fluid mechanics which are used in this thesis, and gives an overview of possible numerical methods to discretize and simulate these equations.

2.1.1 Fluid Mechanics

Fluid mechanics is governed by the Navier-Stokes equations [3]. However, for the simulation purposes in this thesis, the assumption of an incompressible and inviscid fluid simplifies these equations to the so-called Euler equations. These can be derived from two basic principles: The conservation of mass, leading to the continuity equation, and Newton's second law, leading to the balance of momentum. The derivations shown here are mostly based on the text books by Chorin and Marsden [12] and Griebel *et al.* [30].

To derive the continuity equation, one starts with a fixed region Ω containing a fluid with a mass density $\rho(\mathrm{x}, \mathrm{t})$. The existence of the mass density is a simplification and is called the continuum assumption, as the molecular structure of the fluid is neglected. The mass of the fluid is then given by integrating this density over the region Ω:

$$\mathrm{m}(\Omega, \mathrm{t}) = \int_{\Omega} \rho(x, \mathrm{t}) \, \mathrm{dV}.$$

The principle of mass conservation can now be formalized as the equality between the rate of mass increase within region Ω and the rate with which mass is crossing the region boundary $\partial\Omega$:

$$\frac{\mathrm{d}}{\mathrm{dt}} \int_{\Omega} \rho \, \mathrm{dV} = - \int_{\partial\Omega} \rho \, \mathbf{u} \cdot \mathbf{n} \, \mathrm{dA}. \tag{2.1}$$

Here, $\mathbf{u}$ is the spatial velocity field of the fluid and $\mathbf{n}$ is the unit outward normal of the region boundary $\partial\Omega$. The divergence or Gauss theorem relates the flow through a boundary of a volume with the divergence (the sinks and sources) within that volume:

$$\int_\Omega \operatorname{div}(\mathbf{F})\,\mathrm{dV} = \int_{\partial\Omega} \mathbf{F}\cdot\mathbf{n}\,\mathrm{dA}$$

Applying this theorem to equation 2.1, the right hand-side can be rewritten and one derives

$$\int_\Omega \left[\frac{\partial\rho}{\partial t} + \operatorname{div}(\rho\mathbf{u})\right]\mathrm{dV} = 0$$

which holds for all fluid regions Ω and can therefore be simplified to the differential form of the law of conservation of mass, the *continuity equation*:

$$\frac{\partial\rho}{\partial t} + \operatorname{div}(\rho\mathbf{u}) = 0.$$

The simulations in this thesis only apply to liquids, which have a much lower compressibility than for example gases [23]. This justifies the assumption of an incompressible fluid which in turn ensures a constant mass density over time ($\rho(x,t) = \text{const}$). In this case the continuity equation takes the simple form

$$\operatorname{div}\mathbf{u} = 0 \tag{2.2}$$

To derive the equation of balance of momentum, one starts at Newton's second law ($\mathbf{F} = \mathrm{m}\mathbf{a}$). The acceleration $\mathbf{a}$ of a fluid particle which moves along the trajectory (x(t), y(t)) can be formalized as:

$$\mathbf{a} = \frac{\mathrm{d}^2}{\mathrm{dt}^2}(\mathrm{x(t)},\,\mathrm{y(t)}) = \frac{\mathrm{d}}{\mathrm{dt}}\mathbf{u}(\mathrm{x(t)},\mathrm{y(t)},\mathrm{t})$$

Applying the chain rule yields

$$\mathbf{a} = \frac{\partial\mathbf{u}}{\partial\mathrm{x}}\dot{\mathrm{x}} + \frac{\partial\mathbf{u}}{\partial\mathrm{y}}\dot{\mathrm{y}} + \frac{\partial\mathbf{u}}{\partial t} = \frac{\partial\mathbf{u}}{\partial\mathrm{x}}\mathrm{u} + \frac{\partial\mathbf{u}}{\partial\mathrm{y}}\mathrm{v} + \frac{\partial\mathbf{u}}{\partial t}$$

where u and v are the components of the particles velocity $\mathbf{u}$. Rewriting this equation using the gradient operator leads to

$$\mathbf{a} = \frac{\partial \mathbf{u}}{\partial \mathrm{t}} + \mathbf{u} \cdot \nabla \mathbf{u} \tag{2.3}$$

which can be rewritten again by defining the so-called material derivative:

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla$$

This definition can be interpreted as a connection of Lagrangian and Eulerian viewpoints of the fluid. In the *Lagrangian view*, the fluid is treated as a particle system carrying certain properties, whereas the *Eulerian view* considers fixed points in space (e.g. a regular grid) and measures the fluid properties as rates of change at these points [9]. The material derivative includes the term ∂_t to express the rate of change at a fixed point and the term $\mathbf{u} \cdot \nabla$ to account for the fact that the fluid is also flowing past that fixed point. Thus, it represents differentiation following the fluid, and answers the (Lagrangian) question of how fast a quantity is changing for a moving particle, by also considering fixed points from the Eulerian view [9].

For the application of Newton's law, one also needs to calculate the forces acting on the fluid. These can either be forces of stress (e.g. pressure and internal friction) or body forces which exert some force per volume (e.g. gravity or the magnetic force). The stress forces at the boundary of a region can be represented using the stress tensor $\boldsymbol{\sigma}$. With the second assumption of an inviscid fluid (the first being that the fluid is incompressible), internal friction of the fluid is neglected and the stress tensor can be solely defined by the pressure function p(x, t). In case of viscous fluids, the stress tensor would also contain a viscous parameter, leading to dynamic and kinematic viscosities in the momentum equation. However, for the simulation purposes in this thesis the assumption of an inviscid fluid is sufficient (reasons for this will be discussed in section 2.1.2) and the stress forces can be written as

$$S_{\partial\Omega} = -\int_{\partial\Omega} p\,\mathbf{n}\,\mathrm{dA} = -\int_{\Omega} \mathrm{grad}\,p\,\mathrm{dV} \tag{2.4}$$

where the the divergence theorem provided the last equality. The body forces can be calculated by integrating over the fluid region and taking into account $\mathbf{b}$(x, t), the body force per unit mass:

$$\mathbf{B} = \int_{\Omega} \rho\,\mathbf{b}\,\mathrm{dV} \tag{2.5}$$

Putting it all together (i.e. combining equations 2.3, 2.4 and 2.5), one finally derives the law of balance of momentum in its differential form:

$$\rho \frac{D\mathbf{u}}{Dt} = -\text{grad}\, p + \rho\, \mathbf{b} \tag{2.6}$$

Imposing the initial condition $\mathbf{u} = \mathbf{u_0}(\text{x, y})$ at $t = 0$, which has to satisfy the continuity equation 2.2, one can calculate the dynamics of the system (described by the quantities $\mathbf{u}$, p and ρ) using the momentum equation 2.6. The domain in which the fluid is defined has boundaries where certain conditions have to hold at all times. Thus, the problem becomes an initial-boundary value problem. For the simulations in this thesis, free-slip conditions are applied, meaning that the velocity components orthogonal to the domain boundaries are zero while the tangential components are not affected (no frictional losses at the boundaries). Thus, the Euler equations derived in this section are equations 2.2 and 2.6, together with a constant mass density $\rho(\text{x, t})$ and the boundary conditions:

$$\mathbf{u} \cdot \mathbf{n} = 0 \text{ on } \partial W \tag{2.7}$$

2.1.2 Simulation Approaches

As the simplifications and assumptions for the equations in section 2.1.1 suggest, the simulations in this thesis are not tailored for physical accuracy, but rather for realistic visual behaviour which is sufficient for the main goal of learning fluid motion. This section therefore summarizes simulation methods from the field of computer graphics. Accurate physical simulations from the realm of computational fluid mechanics are out of scope for this thesis.

In the computer graphics community there are two main simulation approaches based on the aforementioned viewpoints: Eulerian grids and Lagrangian particles [84]. In Eulerian methods, all variables of the fluid are stored at fixed grid points, together with a fluid mask indicating where the fluid is located. An example would be the method from Foster and Metaxas [25] which was the first grid-based 3D simulation implementing the full Navier-Stokes equations. Eulerian grids provide a simple discretization and simplify the calculations which ensure that the fluid satisfies the incompressibility constraint in form of the continuity equation 2.2. However, simulations using explicit Eulerian advection schemes can become unstable over time, forcing an upper boundary on the time step size and interactivity [71].

Lagrangian methods store variables on the different particles which represent different chunks of fluid. The variables are then carried along by moving the positions of the

individual particles, enabling simple, but accurate advection schemes using solvers for ordinary differential equations (ODE). One popular technique for such fluid simulations is smoothed particle hydrodynamics (SPH) which was originally developed for problems in astrophysics [51]. SPH is not based on a grid but uses analytical differentiation of interpolation formulae with which any function can be described by values at certain points, the particle positions. While Eulerian schemes struggle with advection, Lagrangian methods have difficulties with the incompressibility condition (Equation 2.2), often restricting the simulations to uniform particle spacings [84].

Combining strengths from both Eulerian and Lagrangian schemes, the Particle-in-Cell (PIC) method was developed to handle advection with particles, while solving the incompressibility condition on a grid [33]. Grid point values are calculated as weighted averages from nearby particles at each time step and are used for calculating the pressure necessary for a divergence-free velocity field. Then, the grid point values are used to calculate new particle values by interpolating from the surrounding grid points. The particles get advected by simply moving their positions according to their velocities and their values are again mapped to the grid for the next time step. In order to simplify operations on the grid, the PIC method uses so-called *staggered grids*, which have been proposed in the Marker-and-Cell method (MAC) [34]. By storing grid values at the cell boundaries instead of the centers, central difference calculations (e.g. computing the divergence or gradient) get simpler and more accurate [9].

Due to the repeated interpolation of grid points for particle values, the PIC method shows strong numerical diffusion, resulting in undesirable smoothing of small-scale velocities. This disadvantage has been solved by introducing the Fluid-Implicit-Particle (FLIP) method [7]. The solution was to only map the change from the grid calculations back to the particles instead of the total velocity values. At each time step, the particle values are therefore only updated with the results from the grid calculations instead of getting replaced by them. Since its initial proposal, a multitude of FLIP variants have been developed. The Material Point Method (MPM) [73] extends FLIP to deformable, elastic materials like for example snow [72] or sponges [58]. Other methods combine FLIP with traditional grid-based solvers to reduce computation time and memory usage [22].

The simulations in this thesis are based on the FLIP method. To simulate inviscid, incompressible fluids, the Euler equations (Equations 2.2 and 2.6) have to be solved. One way to do this is to split these equations into multiple parts and solve each equation separately [9]. The law of balance of momentum (equation 2.6) can be split into two equations, one adding body forces and one adding stress forces while also satisfying the incompressibility constraint.

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{b}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho} \operatorname{grad} p \text{ satisfying } \nabla \cdot \mathbf{u} = 0 \tag{2.8}$$

In order to satisfy the continuity equation 2.2 the fluid must only be moved in a divergence-free velocity field. Thus, the sequence in which the equations get solved matters and the advection must use the output of the grid calculations. In order to ensure boundary conditions, the FLIP algorithm also corrects the positions of particles inside boundaries, or obstacles, by shifting them to valid positions inside the domain. Algorithm 1 shows the full FLIP procedure.

Algorithm 1 FLIP algorithm

1: Initialization of particle positions and velocities
2: **for** each time step **do**
3: Map particle values to staggered grid to get the velocity field $\mathbf{u}_\mathrm{i}$
4: Add forces (e.g. gravity) to $\mathbf{u}_\mathrm{i}$
5: Subtract pressure gradient from $\mathbf{u}_\mathrm{i}$ and receive divergence-free velocity field $\mathbf{u}_\mathrm{f}$
6: Map $\mathbf{u}_\mathrm{f} - \mathbf{u}_\mathrm{i}$ to each particle by interpolation
7: Advect particles and their values on the grid using an ODE solver
8: Push particles outside of boundaries or obstacles
9: **end for**

The above assumption of an inviscid fluid can be justified by the fact that our simulations are tailored to liquids where viscosity plays only a minor role, in contrast to highly viscous materials, such as honey. Furthermore, many numerical methods introduce dissipation effects which could be physically interpreted as viscosity [9]. Thus, adding viscosity would only have a negligible effect and make the equations more complex.

In order to solve equation 2.9 and calculate the pressure, it can be discretized using finite differences with a small time step dt [9]:

$$\mathbf{u}^{n+1} = \mathbf{u}^n - dt \cdot \frac{1}{\rho} \operatorname{grad} p \text{ satisfying } \nabla \cdot \mathbf{u}^{n+1} = 0$$

By applying the divergence on both sides this can be rewritten to:

$$\nabla \cdot \mathbf{u}^{n+1} = \nabla \cdot \left(\mathbf{u}^n - dt \cdot \frac{1}{\rho} \operatorname{grad} p \right) \stackrel{!}{=} 0$$

Thus, one has to solve the following system of linear equations:

$$\nabla \cdot \mathbf{u}^n = dt \cdot \frac{1}{\rho} \Delta p \tag{2.9}$$

The pressure field p is hereby processed by the laplace operator $\Delta = \nabla \cdot \nabla$. The implementation of the simulation in this thesis is discussed in detail in section 4.1.

2.2 Supervised Deep Learning

Due to a combination of advances in parallel computer hardware and efficient implementations of learning algorithms within well-designed software frameworks [55, 1], deep learning [67, 47, 28] has now a central role in machine learning and the broader context of artificial intelligence. After their initial success in computer vision [45], deep learning models have been applied in multiple fields of science, including climate science [60], biology [69], chemistry [27] and physics [17]. Geometric deep learning [10] summarizes the efforts to generalize deep learning models to non-Euclidean domains, where different models have successfully been applied to point clouds [31], meshes [32], graphs [65] and manifolds [13].

This chapter acts as a short introduction to deep learning with a focus on supervised learning. Section 2.2.1 states the general goal of supervised learning and introduces common optimization algorithms. Section 2.2.2 then describes common deep learning building blocks and discusses inductive biases and priors. Section 2.2.3 provides a more detailed introduction to graph networks. General introductions to deep learning can be found for example in the textbooks from Goodfellow *et al.* [28] or Nielsen [53].

2.2.1 Learning Goal and Algorithms

The formalism in this section is mostly based on the review from Carleo *et al.* [11]. In machine learning, many problems can be described by the form $\mathbf{y} = f(\mathbf{x})$ where $\mathbf{x}$ are inputs to some continuous function f which produces the outputs $\mathbf{y}$. An example would be an image classification problem where the inputs $\mathbf{x}$ are images of human faces and the outputs $\mathbf{y}$ are the names of the persons in the images. The function f would then map each image to the corresponding name of the person in it. In supervised learning, the goal is to approximate the function f by learning from pairs of inputs $\mathbf{x}$ and their corresponding outputs $\mathbf{y}$. A first step in approximating f is to express the function by a set of parameters Θ, leading to the parameterized function f_Θ. This is done by defining deep learning models which will be described in more detail in section 2.2.2. By designing a so-called *loss function* $\mathcal{L}(f_\Theta(\mathbf{x}), \mathbf{y})$ one can set the optimization goal for the

parameters Θ, so that $f_\Theta(\mathbf{x}) \approx f(\mathbf{x})$ when $\mathcal{L}$ is minimized. During the training process, the loss function is calculated for each input-output pair i and the parameters Θ get trained to minimize the mean loss

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(f_\Theta(x_i), y_i)$$

where n is the size of the training set. For parameter optimization, variants of the *gradient descent* algorithm are a popular choice. In this algorithm, the parameters get updated according to the formula

$$\Theta^{t+1} = \Theta^t - \eta \nabla_\Theta \mathcal{L}. \tag{2.10}$$

Thus, the weights get iteratively adjusted in the direction of the steepest descent of the loss where the step size η is called *learning rate*. The loss gradient is efficiently calculated using the so-called *backpropagation algorithm* [62] which is at the core of most deep learning applications. Details for this algorithm can be found in [53]. In order to speed up the training procedure, the loss can be calculated by only using subsets of the total training set, so-called *mini-batches*. When doing so, the algorithm is called *stochastic gradient descent* (SGD) which is a commonly used technique in deep learning [53]. Using small samples, the true loss can be estimated much faster than calculating it using the full training set.

The choice of the loss function $\mathcal{L}$ depends on the task at hand. For classification tasks, the so-called *log loss* (cross entropy loss) is most commonly used, whereas L_p functions like L_1 and L_2 are typically considered for regression problems. However, it has been shown that their use in classification problems is also reasonable [42]. However, the loss function can also be used to force certain priors upon the architecture of deep learning models. The work from Raissi *et al.* [56, 57, 81] shows examples for incorporating physical and biological laws into the loss functions of architectures in form of partial differential equations. Recently, deep learning architectures have been developed to respect physical symmetries and learn exact conservation laws from data by choosing loss functions which closely follow mathematical formulations of classical mechanics [29, 16]. Thus, the choice of the loss function has a significant effect on the outcome of the optimization procedure and is setting the learning goal of the model.

2.2.2 Deep Learning Architectures and Priors

A simple deep learning model for the parameterization of some continuous function f would be a *multi-layer perceptron* (MLP). A fully connected MLP [61] can have multiple

layers of so-called *neurons*, where each neuron is connected to all neurons of the previous and subsequent layer. An example of such an MLP is shown in figure 2.1. Layers between the input and output layers are called *hidden* and if there are multiple of such layers the neural network is called *deep*. The i-th layer of an MLP transforms its inputs $\mathbf{x}$ according to

$$g^{(i)}\left(\mathbf{w}^{(i)}\mathbf{x} + \mathbf{b}^{(i)}\right)$$

where $\mathbf{w}^{(i)}$ and $\mathbf{b}^{(i)}$ stand for the trainable parameter tensors of the i-th layer. The so-called *activation function* $g^{(i)}$ provides the layer with a non-linearity, common choices are variants of the rectified linear unit (ReLU), sigmoid or hyperbolic tangent (tanh) functions, exponential linear units (ELUs) [54] or sinusoids (i.e. $\sin(x)$) [56].

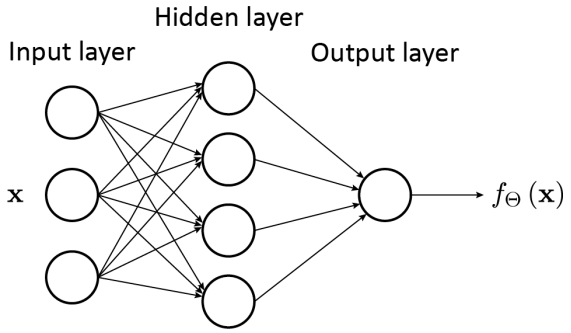

Figure 2.1: MLP with an input layer consisting of three neurons, one hidden layer with 4 neurons and one output layer with one neuron.

The work from Raissi *et al.* [56, 57, 81] shows examples for inserting certain priors (e.g. physical laws) into the loss function of the training procedure. Such priors can also be included into the architectures themselves and are then called *inductive biases* [5]. These inductive biases can introduce additional assumptions about a learning problem into an architecture and provide a model with a tendency towards certain solutions, independent of the given data [50]. For example, fully connected neural networks have a weak inductive bias as all neurons can interact with each other and determine the output in a similar manner [5]. Other deep learning building blocks, such as convolutional layers [26, 46] have an inductive bias towards translational invariance and locality, which is useful when analysing images. A general class of architectural inductive biases are so-called *relational biases* [5] which lead to an architecture called *graph network*, described in the next section.

2.2.3 Graph Networks

Battaglia *et al.* [5] name *combinatorial generalization* as one of the core building blocks of human intelligence. They argue, that humans understand the world in compositional terms, meaning that they understand complex systems as compositions of entities which interact with each other. For example, considering Lagrangian fluids, the fluid particles can be seen as entities and the forces between the particles as their relations. Battaglia *et al.* [5] formalize the principle of combinatorial generalization as a relational bias (see section 2.2.2) and propose an implementation of this bias in form of a generalized version of *graph neural networks* [65] which they call *graph network* (GN). More information about graph neural networks can be found in the reviews [10, 83, 80].

The architecture of a GN operates on graphs, which consist of nodes connected by edges. As mentioned, the nodes could for example be fluid particles and the edges could represent the forces the particles exert onto each other. A GN takes such a graph and processes it with different *update* and *aggregation* functions. Update functions alter node and edge features, while aggregation functions summarize features from multiple nodes or edges. Battaglia *et al.* [5] formalize such uses of update and aggregation functions into the GN framework. A given graph structure is formalized as a 3-tuple $G = (\mathbf{u}, V, E)$. $\mathbf{u}$ encodes global attributes of the graph, for example the total energy of a system. V is the set of nodes where the attributes of the i-th node are encoded as $\mathbf{v}_i$. E is the set of graph edges where the attributes of the k-th edge are encoded as $\mathbf{e}_k$. A graph is said to be directed, if it contains only one-way edges which connect a *sender* node ($\mathrm{v_s}$) with a *receiver* node ($\mathrm{v_r}$). Figure 2.2 shows an abstract example of a directed graph with 5 nodes.

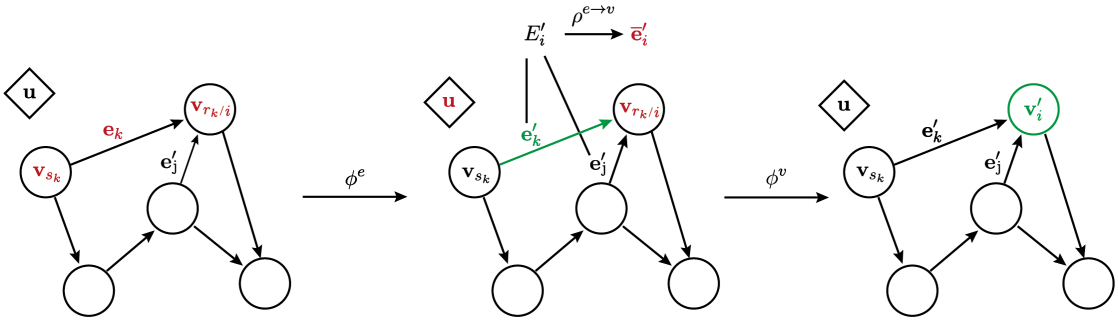

Figure 2.2: Update mechanism of an Interaction Network [4] applied to a simple, directed graph with five nodes. Red colors indicate the variables which get used in the next update step, green colors indicate which variables have been updated. For visualization purposes the update and aggregation functions are only applied to one edge and one vertex.

Formally, a GN block consists of update functions, denoted by ϕ, and aggregation functions, denoted by ρ. The order and use of update and aggregation functions can vary

and are often tailored to specific problems. For our deep learning experiments in chapter 5, we make use of one specific variant, the *Interaction Network*, which was first proposed by Battaglia *et al.* [4]. The mechanism of this GN variant can be formalized as the three equations 2.11 and is shown in figure 2.2.

$$\begin{aligned} \mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}\right) \\ \overline{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) \\ \mathbf{v}'_i &= \phi^v\left(\overline{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) \end{aligned} \tag{2.11}$$

At first, the GN updates single edges by applying its edge update function ϕ^e to the edge features $\mathbf{e}_k$, sender node features $\mathbf{v}_{s_k}$ and receiver node features $\mathbf{v}_{r_k}$. For each node, the GN then summarizes the features of all incoming edges (E'_i) with its aggregation function $\rho^{e\to v}$. Finally, it updates each node by applying its node update function ϕ^v to the edge feature summary $\overline{\mathbf{e}}'_i$, the node features $\mathbf{v}_i$ and the global graph attributes $\mathbf{u}$. Descriptions of more general GN blocks can be found in the original description by Battaglia *et al.* [5].

In general, the actual implementation of the update and aggregation functions can vary and does not necessarily include any deep learning building blocks. However, in case of the interaction networks, ϕ^e and ϕ^v are implemented as separate MLPs which are reused for all edges and nodes respectively. The aggregation function $\rho^{e\to v}$ is implemented as an elementwise sum. By reusing the same MLPs for multiple edge and node updates, GNs have a bias towards the aforementioned combinatorial generalization. Graphs often consist of similar entities, like for example fluid particles which exert similar forces onto each other. Training GNs on such graphs provides them with strong generalization capacities to larger graphs with similar entities, which is the key effect of combinatorial generalization [63, 4].

Furthermore, multiple GN blocks can be composed to form complex architectures. For example, one could construct an architecture divided into an encoder, processor and decoder, where the processor consists of a sequence of GN blocks (e.g. interaction networks). One such configuration has proven effective in learning complex dynamics such as fluid motion [64] and is used for our deep learning experiments in chapter 5.

2.3 Differentiable Physics

In recent years, the intersection of machine learning, automatic differentiation and physical sciences became a very active field of research [14]. High quality automatic differentiation libraries like Autograd [49], PyTorch [55], Tensorflow [1] or JAX [8] allowed the integration of classical simulators into deep learning systems [76, 66] and initiated the

development of differentiable physics solvers. These solvers are often used for solving inverse problems in various fields, like for example fluid or molecular dynamics [66, 79]. When calculating gradients of simulation parameters, regular solvers can often only provide approximations by finite differencing. Solvers using automatic differentiation resolve this issue by efficiently computing analytic derivatives with respect to arbitrary inputs [38]. Examples for such solvers are Φ_{Flow} [38], a framework-independent solver providing multiple physics scenarios, DiffTaichi [40], which extends the Taichi programming language [39] with automatic differentiation capabilities and JAX MD [68], which uses JAX to provide multiple physics simulation environments. While most research into the combination of automatic differentiation and simulations resulted in highly specialized solutions [6, 75, 41, 21], these solvers aim to provide general purpose simulation environments which can be easily integrated into other systems to either guide training [38], solve optimization problems [66] or complement parts of classical solvers with learned components [76].

We integrate our FLIP simulation into the Φ_{Flow} solver [38]. Φ_{Flow} was designed in a framework-independent way, providing support for multiple automatic differentiation backends. Its operations make use of the aforementioned staggered grids [34] while implementing various differential operators (gradient, divergence, curl, laplace). Furthermore, the framework provides methods for solving Poisson problems or advection steps by using algorithms like the conjugate gradient [37] or semi-Lagrangian advection [71].

Chapter 3

Related Work

Our deep learning architectures are based on recent work from Sanchez-Gonzales et al. [64], who proposed a framework called *Graph Network-based Simulators* (GNS) which can learn to simulate the dynamics of different materials like water and sand. The authors apply the GNS to datasets generated by classical simulators using the aforementioned SPH [51] and MPM [73] methods. They also use another technique, called *position-based dynamics* (PBD) [52], which directly predicts and works with position changes instead of focusing on forces and accelerations like the former methods. Their work is summarized in chapter 5 and is based on the long-term effort of using graph networks to build data-driven simulators [4, 27, 5, 63, 48].

More broadly, graph networks have become popular in natural sciences such as collider physics [70], astrophysics [17] or chemistry [27]. Another important area of study is making graph networks more understandable, e.g. by reformalizing them using symbolic regression [15, 17] and by providing tools for analyzing these systems [82], which are currently considered as black boxes.

Another, non-graph-based technique for learning fluid dynamics is the work from Ummenhofer *et al.* [77] where the authors represent fluids as point clouds and apply network architectures with continuous convolution layers to learn the mechanics. Continuous convolutions were also used by Schenck *et al.* [66], who implement differentiable fluid dynamics layers which can be integrated into deep learning systems. Continuous convolutions on point clouds can be seen as an independent research branch in the field of geometric deep learning. An overview of different methods is given in the review by Guo *et al.* [31].

Chapter 4

Differentiable FLIP Simulation for $\Phi_{\mathbf{Flow}}$

This chapter describes the implementation of our differentiable FLIP simulation for the second version of Φ_{Flow}. Section 4.1 explains the implementation details. Section 4.2 shows simulation examples and verifies the expected behaviour of the simulation in different scenarios. Section 4.3 demonstrates the differentiability of the simulation in simple optimization experiments.

4.1 Implementation

As introduced in section 2.3, the Φ_{Flow} solver is a differentiable physics solver which can be used to implement physics scenarios in different backends (SciPy [43] and Numpy [35], PyTorch [55], TensorFlow [1] or JAX [8]). Our first contribution is to extend this framework with a differentiable FLIP simulator. We update an existing FLIP simulation from the first version of Φ_{Flow} and make it compatible to the second version which is currently under development. Furthermore, we eliminate known bugs and extend the code to support obstacles and inflows.

Using the *Domain* object from Φ_{Flow}, the physical properties of the simulation space can be initialized with free-slip boundary conditions as defined by equation 2.7. The initial particle distribution can be specified using the Φ_{Flow} *Geometry* object to generate a binary mask marking the grid cells which should contain particles. As discussed by Zhu and Bridson [84], assigning 8 particles to each marked grid cell prevents voids in the fluid while avoiding too much noise. Thus, 8 particles are generated in each marked cell by default. Their initial x and y positions can be chosen from a uniform distribution within each cell or can be set to the cell centers which is useful in later symmetry tests (section 4.2). Their initial velocity can be specified by a single velocity vector which is then broadcasted to all particles. Obstacles can be initialized with the Φ_{Flow} *Obstacle* class, specifying their extends and a rotation angle.

The simulation itself is implemented as described in algorithm 1. Fluid particles are represented using the *PointCloud* class of Φ_{Flow} which provides functions to map particle

values to the staggered grid of the domain and also to sample particle values from the grid. After interpolating the velocity grid from the particle velocities, possible forces are added to the velocity grid by Euler integration. One standard force tensor T_{force} is for example gravitational acceleration in negative y-direction, where dt is the time-step of the simulation:

$$\mathbf{v}_{\text{Force}} = \mathbf{v} + dt \cdot \mathbf{T}_{\text{force}}$$

Next, to satisfy the continuity equation 2.2, that is to ensure a divergence-free velocity field for the advection step, the pressure is calculated by solving the system of linear equations derived in equation 2.9. Therefore, the divergence of the velocity field (which was already updated with the force tensor) gets calculated using the Φ_{Flow} divergence operator for staggered grids. Using the divergence of the velocity field, the pressure is computed by the high-level linear equation solver of Φ_{Flow} which automatically generates a sparse matrix from the given equation and solves the system by making use of the conjugate gradient method [37]. Calculating the pressure gradient with the corresponding Φ_{Flow} operator and subtracting it from the velocity field results in the divergence-free velocity. A FLIP simulation updates the particle velocities with the change between the initial and the divergence-free velocity field (as explained in section 2.1.2). Thus, only the difference between the divergence-free velocity field and the initial velocity field are interpolated and added to the particle velocities by again making use of the Φ_{Flow} sampling mechanism between staggered grids and the *PointCloud* class. When mapping grid velocities to particles, the particle values are interpolated from neighboring grid cell values. As the grid velocities are only nonzero for cells which hold particles, this can lead to deteriorating liquid shapes as the particles at the borders of the shape will get slowed down due to the interpolation from zero velocities outside of the liquid shape. Figure 4.1 shows an example of this effect on the left, where border particles (e.g. the red particle) interpolate from zero velocities and are therefore slower than inner particles (e.g. the green particle). This leads to a deterioration of the liquid shape. To avoid this, we implement an extrapolation method which extrapolates the grid velocities, so that border particles don't interpolate from zero velocities, but instead have the same velocities as inner particles, shown schematically in figure 4.1 on the right.

The particles are advected using a fourth-order Runge-Kutta scheme, shifting the particles to their new positions. Using this fourth-order Runge-Kutta scheme at high velocities can also lead to shape deterioration when sampling higher-order component velocities from grid values outside of the liquid shape. To avoid this, we extend the Runge-Kutta scheme with velocity-dependent extrapolation for each component.

Particles which were falsely advected into the domain boundary, or into obstacles, are moved to valid positions within the domain after each simulation step. For this function-

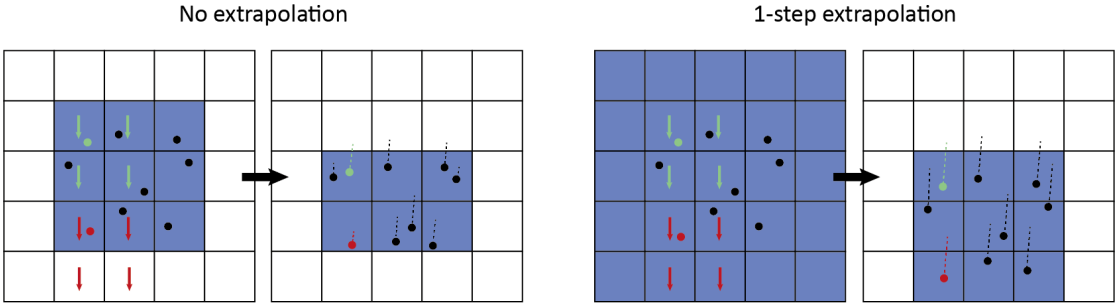

Figure 4.1: Effect of extrapolation on falling liquid shapes for one staggered grid component. Dots represent particles, blue grid cells have nonzero velocities, white grid cells have zero velocities. Particles interpolate their new velocity values from the neighboring grid cells, indicated by the red and green arrows for 2 particles. Dashed lines indicate the position change after advection.

ality, we extended the Φ_{Flow} *Box* class by shift functions which detect particles inside the geometry domain, and push them towards the nearest border. Finally, possible new particles are added at their corresponding positions by concatenating *PointCloud* objects. Then, the next iteration of the simulation starts using the updated position and velocity values.

Figure 4.2 shows three examples of liquid simulations with different scenarios. Links for videos of these examples can be found in table A.1 in the appendix. We generate the simulations with a time step of 0.1 seconds using a 64×64 domain. The first example, showing a block falling into a pool, has a total of $11,520$ particles. The obstacles shown in the second and third example of figure 4.2 are represented by points only for visualization purposes and are actually implemented on grid level.

To enable the execution of our simulation in all of the Φ_{Flow} backends, we add additional scatter functions in PyTorch and TensorFlow. All our additional simulation methods are implemented for a variable number of dimensions, allowing the FLIP simulator to also work in 3 dimensions. Our code, demos and an additional test suite for the FLIP simulations are available in the Φ_{Flow} GitHub repository.

4.2 Physical Verification of the FLIP Simulation

Having demonstrated our FLIP simulation in different scenarios we now verify it in two more aspects. First, we inspect its behaviour when artificial viscosity is added. Second, we measure its ability to retain a symmetric state throughout a simulation.

As described in section 2.1.2, the PIC method produces simulations with strong numerical diffusion which leads to smoothing of small-scale velocities. The FLIP method uses

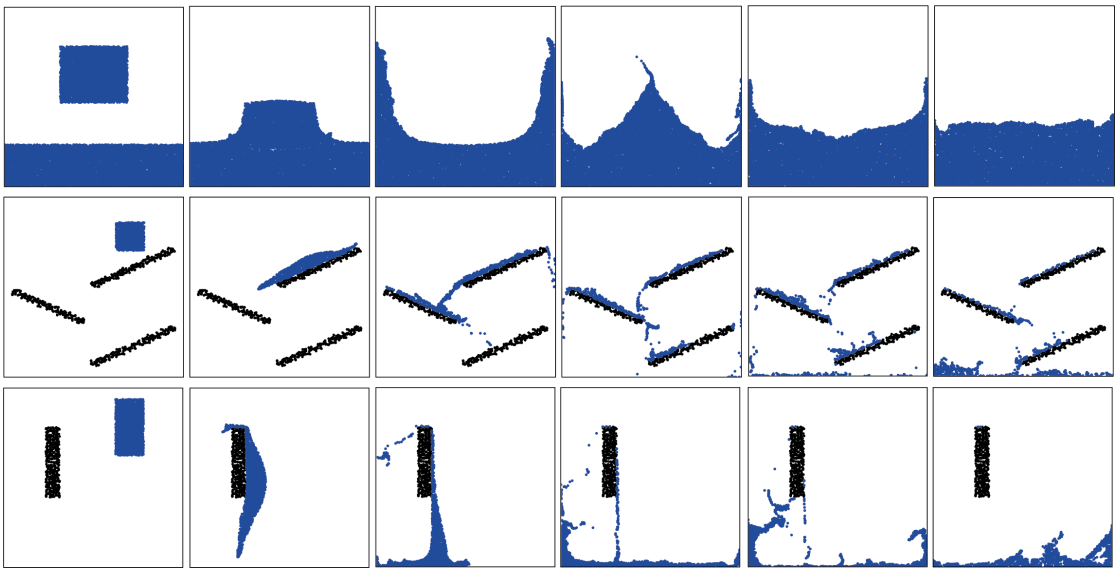

Figure 4.2: Three example scenarios generated by our FLIP simulator. Obstacles are implemented on grid-level but are visualized as point clouds. Time increases from left to right.

the velocity difference instead of the total velocities as updates to the particle velocities. It therefore shows less diffusion and higher level of detail. We have implemented our fluid simulation in such a way that the user can switch between the FLIP and PIC methods. Figure 4.3 shows a qualitative comparison of both simulation approaches. Links to videos and further examples can again be found in table A.1 in the appendix. As expected, and in agreement with the findings of Zhu and Bridson [84], the smoothing effect of the PIC method is clearly visible in contrast to the result of the FLIP method which is rich of small-scale velocities. The smoothing effect of the PIC method can also be interpreted as numerical viscosity. Often, FLIP and PIC are combined by a weighted average to adjust the amount of this numerical viscosity [84].

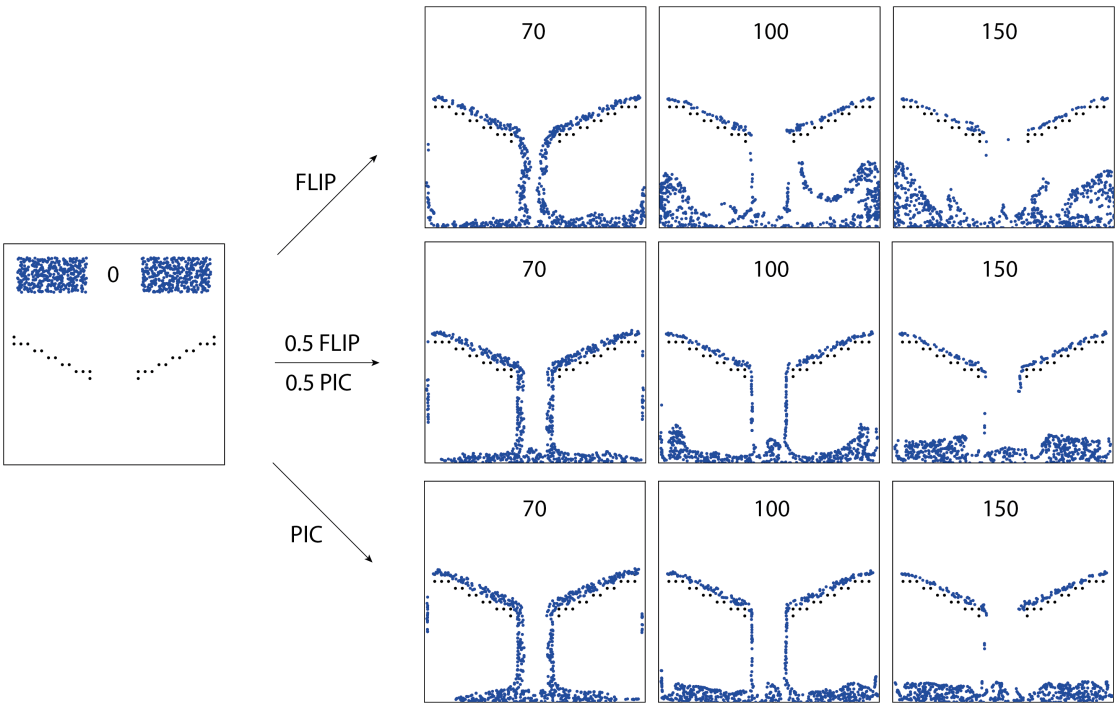

Figure 4.3: Comparison between FLIP and PIC simulations and a mixture of the two methods. Numbers indicate time steps of the simulations.

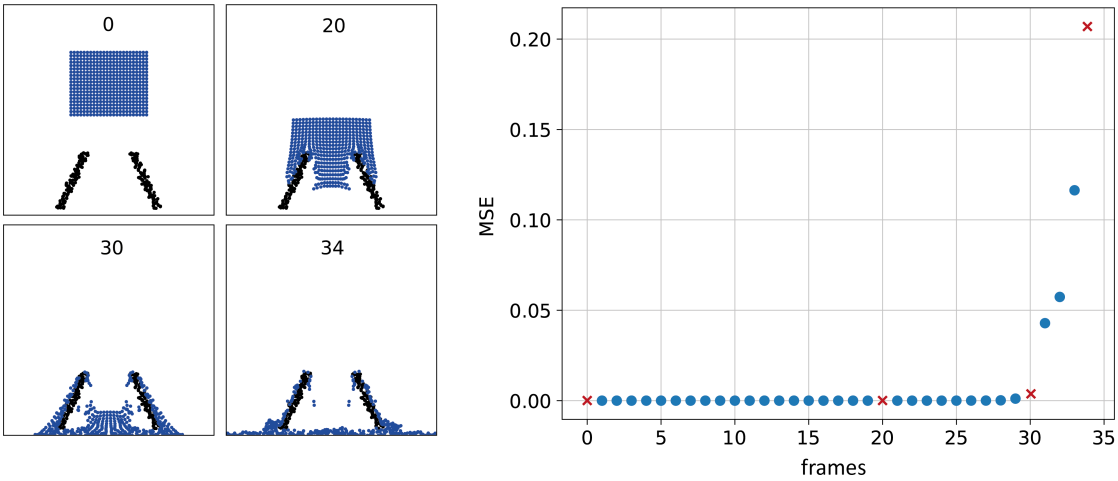

Figure 4.4: Symmetry experiment. The left side shows the simulation state at different frames, indicated by the number at the top. The diagram on the right shows the time evolution of the MSE error between particle positions of both sides of the domain. Red crosses indicate frames where the simulation state is shown on the left.

When choosing a symmetric scenario, one would expect the simulation result to be symmetric as well. In order to test the implemented simulation in this regard, we conduct a symmetry test of which the results are presented in figure 4.4. To ensure initial symmetry, we sample the particles at the centers of the underlying grid cells, forming a block in the middle of the domain. The right side of figure 4.4 shows the time evolution of the mean-squared-error (MSE) between the particle positions on both sides, until the number of particles on both sides differs. From the time evolution of the MSE and the corresponding images at the respective frames, it becomes clear, that the simulation is indeed nearly symmetric until frame 30, showing only negligible differences between the particle positions on both sides. However, due to numerical errors which get accumulated over time, the MSE increases exponentially and the position differences become noticeable from frame 30 on, until the first particle changes sides at frame 35. Aside from numerical errors, fluid motion is chaotic in itself. The symmetry violation after a certain amount of time is therefore expected. Table A.1 in the appendix contains links to videos of the example in figure 4.4 and others.

4.3 Demonstration of the Differentiability of the FLIP Simulation

To demonstrate the differentiability of our FLIP simulation, we conduct example optimization experiments using PyTorch. The upper left of figure 4.5 shows the initial state. The experiment starts with a liquid block, which gets transformed by 20 steps of the FLIP simulation. The initial particle positions are then optimized to approximate the target positions shown in the lower right of figure 4.5. The optimization is done using gradient descent (learning rate of 1) and the L_2 loss between the transformed positions of the initial and target states. As shown by the intermediate steps, the optimization converges within 15 cycles, transforming the initial liquid block into the two spheres of the target. The initial increase of the L_2 loss from 14,129 to 18,141 can be explained by the chaotic nature of fluid dynamics which might cause the particles to get advected in counterproductive directions during the 20 steps of FLIP simulation. Figure 4.6 shows another example of this problem. The optimization shown in the left column stops improving at a relatively high loss and is unable to approximate the target sphere. Again, this can be explained by the fact that shifting the positions by the gradients retained by backpropagation would only improve the current state of the advected fluid, but not the fluid state after applying the FLIP simulation to the new particle positions.

The optimization experiment in the right column of figure 4.6 shows another complication when solving inverse problems. The optimization reaches a relatively small loss, but achieves this without approximating the actual sphere shape of the target. In this case,

the optimization problem has multiple solutions and the optimization is unable to escape one of the local minimums.

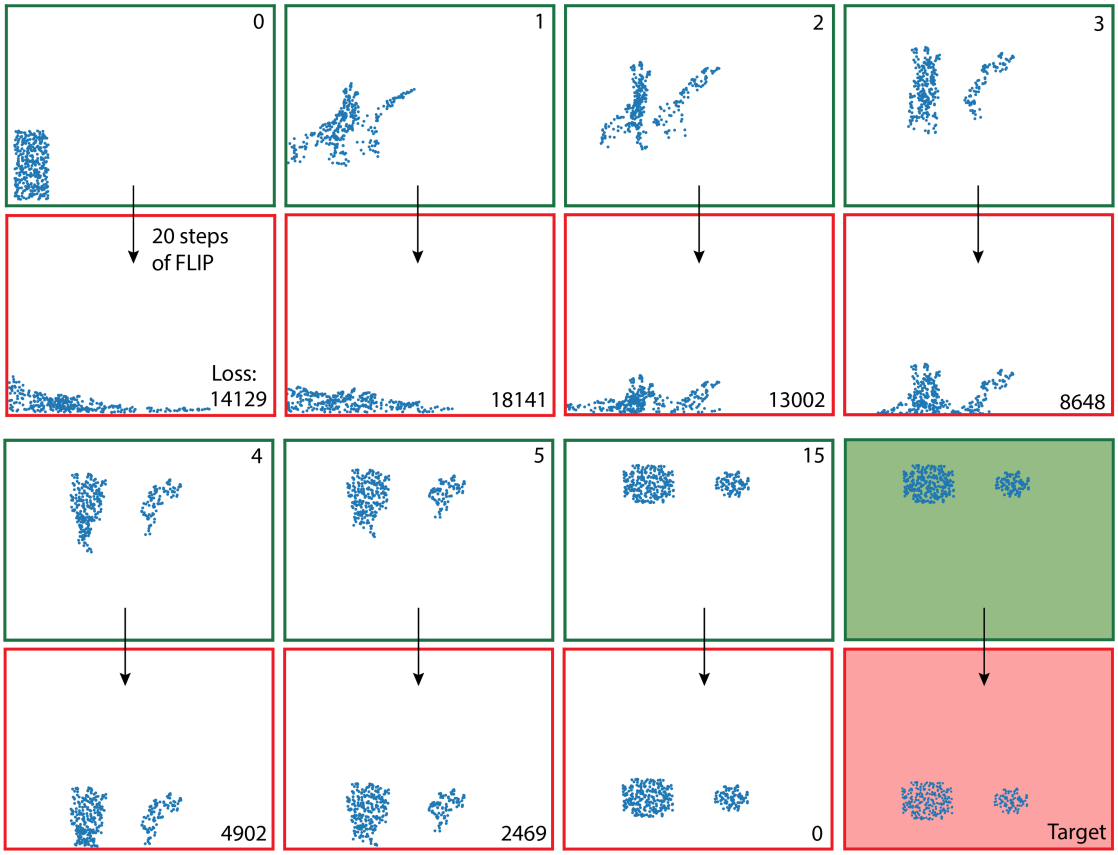

Figure 4.5: Optimization of particle positions with gradient descent through the differentiable FLIP simulation. The green boxes show the initial particle positions, the red boxes show the particle positions after 20 steps of the FLIP simulation. The target of the optimization and its initial positions are shown in the shaded boxes. The optimization steps are indicated by the numbers in the green boxes, the red boxes show the L_2 loss of each step.

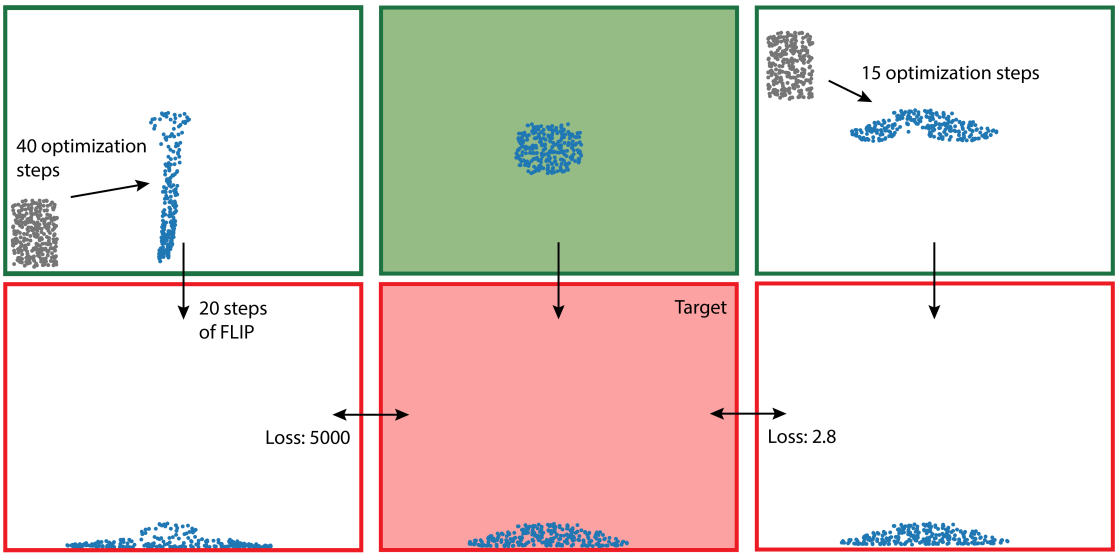

Figure 4.6: Two optimization experiments with problematic outcomes. Grey blocks indicate the initial states of the liquid shapes. Green boxes contain the particle positions before 20 steps of the FLIP simulation, the red boxes show particles after simulating. The shaded boxes contain the optimization target and its initial particle positions.

Chapter 5

Learning to Simulate

This chapter describes our deep learning experiments. We adopt the Graph Network-based Simulators architecture from Sanchez-Gonzalez *et al.* [64], which is described in section 5.1, and train it on the dynamics of the FLIP simulation from section 4.1. Section 5.2 describes the dataset of FLIP simulations which was used for the experiments and section 5.3 explains different learning scenarios and model variants in which the GNS was applied to this dataset. Section 5.4 then compares these scenarios and models with different metrics, and inspects some of their generalization capabilities in more detail.

5.1 Learning FLIP Simulations with Graph Network-based Simulators

Sanchez-Gonzalez *et al.* [64] compose several GN blocks (described in section 2.2.3) into an encode-process-decode configuration which they name *Graph Network-based Simulators* (GNS). They demonstrate that the GNS is capable of learning complex dynamics of different materials such as fluids and sand. Figure 5.1 displays the GNS architecture schematically. On a high level, the GNS is a parameterization of dynamics which map the current state of the world to a future state. In case of the FLIP simulation, the dynamics are described by fluid dynamics, summarized in section 2.1.1. In the Lagrangian view, the state of a fluid can be described in terms of single particles which interact with each other. The GNS takes the positions of these particles from the FLIP simulation as input and uses its encoder to transform these positions into a graph, where each node corresponds to a particle, and edges indicate particles interacting with each other. For each particle, edges are added to all neighboring particles within a certain *connectivity radius*, which is a hyperparameter of the GNS architecture. The node features are calculated by an MLP which takes the five previous particle velocities, the particle type and its distance to the domain boundaries as input, and outputs a latent vector of size 128. The edge features are calculated by another MLP, which takes the relative dis-

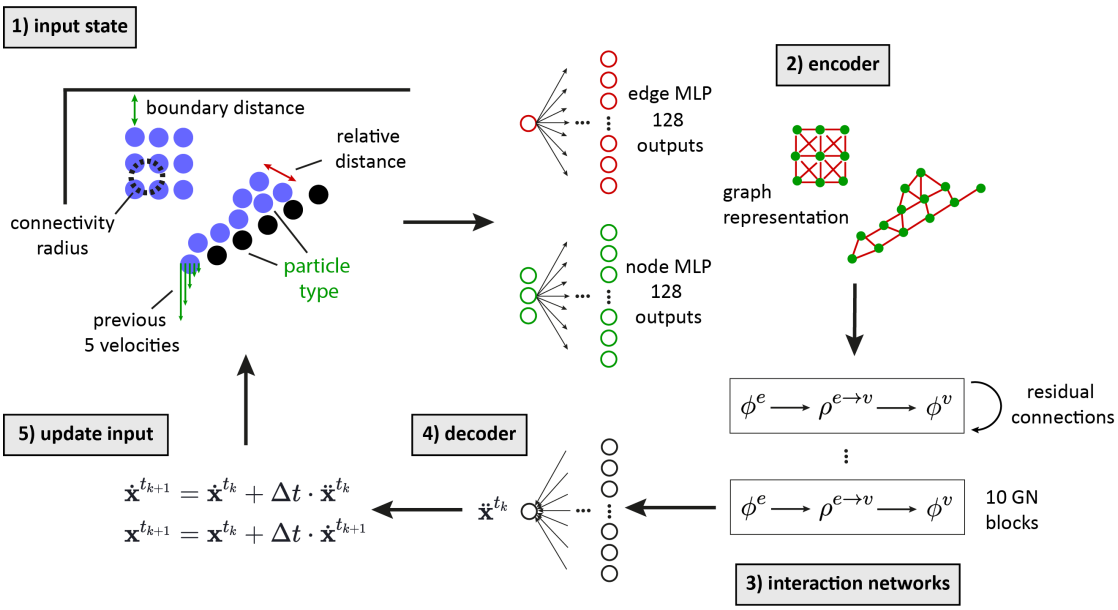

Figure 5.1: Schematic overview of the Graph Network Simulator (GNS) architecture.

tance between the connected nodes as input and produces another latent vector of size 128.

The particle interactions are then computed by performing message passing on the resulting graph. This is done by a sequence of 10 different interaction networks, where each network uses the mechanism defined in equation 2.11 (without the global graph attribute) [5]. All interaction networks have residual connections between the node and edge attributes at input and output. The update functions of the interaction networks are implemented as MLPs and the aggregation function as an elementwise sum (as described in section 2.2.3). The parameterization of the fluid dynamics thus takes the form of message-passing on a graph, performed by MLPs.

The resulting graph is then used to map the input state, the particle positions, to the next state. After the message passing, the GNS first uses its decoder, another MLP, to transform the node features into particle-wise accelerations. These accelerations are then used to calculate future velocities and positions for each particle using Euler integration (see figure 5.1).

All MLPs consist of two hidden layers with ReLU activations and size 128. The MLPs of the encoder use LayerNorm [2] after their output layer. A detailed analysis of the hyperparameters of the GNS architecture (e.g. number of velocities to calculate node features, number of interaction networks) can be found in the work of Sanchez-Gonzales

et al. [64]. They implemented the GNS architecture in TensorFlow 1 [1], using Sonnet 1 [19] and the Graph Nets library [20]. The implementation was published on GitHub and is also used to apply different variants of the GNS to the FLIP simulation of this thesis, as described in section 5.3.

5.2 FLIP Dataset

To apply the GNS architecture to the FLIP simulation from section 4.1, we first use our simulator to generate training, validation and test sets. Table 5.1 summarizes the parameters which we use to generate the training and validation sets. The training set consists of 2000 simulations with 400 frames each, using a time step of 0.05 s. The scenes are randomly generated and can contain liquid blocks, elongated obstacles and a liquid pool at the bottom. Figure 5.2 shows initial simulation states as examples of the training set. The validation set consists of 20 simulations with length 400 and is generated with the same random distribution as the training set, using a different random seed. We generate our test set manually to evaluate the behaviour of the models in 10 challenging scenarios. Examples from the test set are shown in figure 5.3.

Parameter description	**Value**
Trajectory duration (steps)	400
Time step (s)	0.05
Maximum number of particles	1300
Domain size	32×32
Probability for pool	0.3
Pool height	[3, 8]
Block size	[2, 20]
Probability for multiple blocks	0.3
Number of blocks if there are multiple	[2, 3]
Probability for obstacles	0.8
Number of obstacles	[1, 5]
Obstacle length	[2, 20]
Obstacle rotations	[0, 90]
Probability for initial velocity	0.3
Initial velocity range	[-5, 5]

Table 5.1: Parameter specifications for dataset generation. Tuples indicate ranges from which values were drawn randomly. The use of some of the parameters is dependent on the corresponding probabilities (e.g. number of obstacles is only considered if probability for obstacles is met).

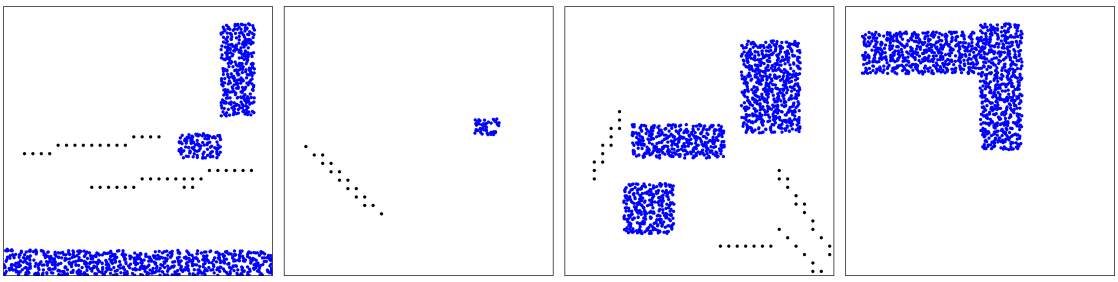

Figure 5.2: Simulation examples from our training set.

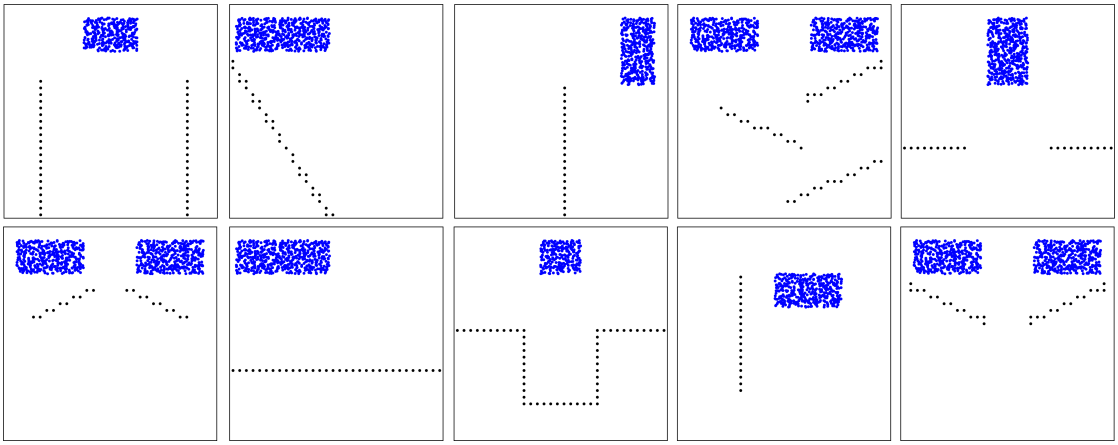

Figure 5.3: Simulation examples from our custom-designed test set.

Sanchez-Gonzales *et al.* [64] normalize all input and target vectors elementwise to zero mean and unit variance. Thus, we calculate the dataset statistics online during data generation and use them later for normalization during training and inference. Due to memory constraints, we limit the number of particles in each FLIP simulation to less than 1300. All trajectories are generated on domains of size 32×32 and scaled to size 0.8×0.8 with positions laying between 0.1 and 0.9.

The training set used by Sanchez-Gonzales *et al.* [64] is mostly generated by simulators based on the Material Point Method (MPM, see section 2.1.2). Sanchez-Gonzales *et al.* show that the connectivity radius is one of the most important hyperparameters of the GNS architecture. Their experiments indicate that greater connectivity radii yield lower errors. As explained in their work, larger particle neighborhoods enable long-range communication among the particles, supporting the message passing which applies the dynamics. However, this communication benefit stands in trade-off with the amount of computation and memory, which increases with larger radii due to the size of the graphs.

Sanchez-Gonzales *et al.* use a connectivity radius of 0.015 on a domain of size 0.8×0.8. Using this connectivity radius to train the GNS on FLIP trajectories yielded unstable results where the fluid blocks were torn apart within the first few time steps. Instead, a higher connectivity radius of 0.03 was necessary in order to produce physical trajectories. Figure 5.4 shows the time evolution of the distribution of neighboring particles (averaged over 50 simulations), using a connectivity radius of 0.03 for the FLIP and MPM WaterRamps dataset from [64]. The left side of figure 5.4 shows that the neighbor number in our FLIP simulations drops at around frame number 50 indicating that the liquid blocks splash at the bottom. After that initial drop, the mean number of neighbor particles (thick red dots) stays nearly constant. This is expected due to the incompressibility constraints of the simulation. For the MPM trajectories (figure 5.4 on the right), the mean number of neighbor particles slightly increases, indicating that the fluid gets compressed over time. This enables long-range communications with a smaller radius. The maximum number of neighbor particles differs strongly between FLIP and MPM simulations, reaching higher values in the MPM case. This further indicates that even a low connectivity radius enables long-range communication between particles in the MPM case while a higher radius is required in case of our FLIP dataset.

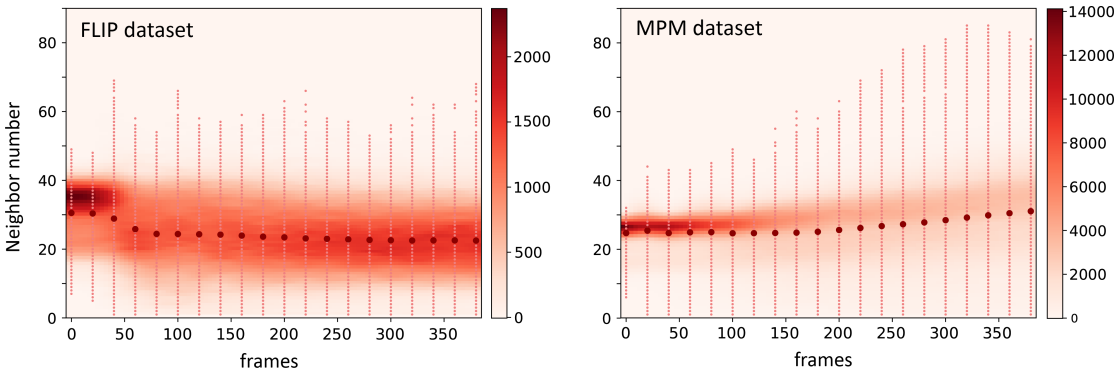

Figure 5.4: Distribution of neighbor numbers for the connectivity radius 0.03 at different time steps, averaged over 50 dataset samples. The left side shows results for the FLIP dataset, the right side shows results for the MPM WaterRamps dataset used in [64]. The thick red dots mark the mean value of neighbor numbers. The distribution was interpolated between values at the positions of the thin red dots. The color indicates the number of particles with the respective number of neighbors.

5.3 Learning Variants

Sanchez-Gonzales *et al.* [64] optimize the GNS parameters Θ using the L_2 loss

$$\mathcal{L} = \left\| \ddot{\mathbf{p}}^t_{\mathrm{GNS}} - \ddot{\mathbf{p}}^t_{\mathrm{GT}} \right\|^2 \tag{5.1}$$

which compares the predicted accelerations (the outputs of the GNS) with the corresponding ground truth (GT) accelerations at time step t. A trained GNS can then be used to generate longer simulation rollouts by using the GNS outputs as its new inputs for the next time step. However, due to the complex mechanics and imperfect optimization, the GNS would then accumulate its own error over time. With the loss defined in equation 5.1 the GNS model is only trained on one-step data and is therefore not trained for handling this accumulated error. One countermeasure used by Sanchez-Gonzales *et al.* [64] is to corrupt the model input during training with artificial noise, which ideally matches the noise produced by the model during rollout generation.

First, we verify the capabilities of the GNS framework by training it on our FLIP dataset. Except for the connectivity radius described in the last section, we train the first GNS (named 1-step-noise, 1sn) with the same procedure as used by Sanchez-Gonzales *et al.* [64]. Additionally, we train another GNS (named 1-step, 1s) without the additional artificial noise.

As a next step, we implement an alternative to the artificial noise. This alternative uses a loss function which spans over more than one step and is described as

$$\mathcal{L} = \frac{1}{n} \left(\left\| \ddot{\mathbf{p}}^t_{\mathrm{GNS}}(\mathbf{p}^{t-1}_{\mathrm{GT}}) - \ddot{\mathbf{p}}^t_{\mathrm{GT}} \right\|^2 + \sum_{i=1}^{n} \left\| \ddot{\mathbf{p}}^{t+i}_{\mathrm{GNS}}(\mathbf{p}^{t+i-1}_{\mathrm{GNS}}) - \ddot{\mathbf{p}}^{t+i}_{\mathrm{GT}} \right\|^2 \right) \tag{5.2}$$

where n is the number of additional steps using the GNS positions as input for the next prediction. The GNS first takes the five previous ground truth velocities as input ($\mathbf{p}^{t-1}_{\mathrm{GT}}$), outputs the particle-wise accelerations and calculates the new velocities and particle positions. In the next step, it takes the four previous ground truth velocities and the newly predicted velocities as input ($\mathbf{p}^{t+i-1}_{\mathrm{GNS}}$) and calculates the next velocities which replace another ground truth velocity in the next step. This way, the model is constantly confronted with its own error which should improve its error mitigation capabilities during rollout generation. Due to memory constraints, we test this alternative only with $n = 1$. We optimize one GNS using this loss from scratch (named 2-step-scratch, 2ss) and another one by initializing it using pretrained weights from the 1-step model described above (named 2-step-initialized, 2si).

As described in section 5.1, the node features in the graph encoding are calculated by an MLP using the five previous velocities, the particle types and the distances to the

domain boundaries as input. However, giving the GNS information about the domain boundaries restricts its generalization capabilities to the domain it has seen during training. An alternative which we explore in this thesis is to model the domain boundaries as obstacles and to remove the additional domain boundary distances from the node feature calculation. This alternative has two advantages. First, the model is not restricted to any specific domain, but operates solely on the interaction between fluid and obstacle particles. Second, the model sees more interactions between fluid and obstacle particles during training, which might improve its capabilities even further. We train a fifth model on a new version of our FLIP training set where we represent the domain boundaries as obstacle particles. We train this model with artificial noise, using the loss from equation 5.1 and name it 1-step-noise-bounded (1snb).

We optimize the parameters of our GNS models with the Adam optimizer [44] and a batch size of 2. The learning rate is decreased exponentially from 10^{-4} to 10^{-6}. We adopted this optimization procedure from Sanchez-Gonzales *et al.* and use it for all experiments in this thesis.

5.4 Quantitative Comparison

To compare the model variants from section 5.3 quantitatively, we use 4 different metrics. The first metric is the particle-wise mean-squared error of one-step acceleration predictions (MSE-acc 1) as defined in equation 5.1. The second metric (MSE 20) is the MSE averaged across time, particles and spatial dimensions of 20 frames, taken at each 20 steps of the full 400-step rollouts. This metric indicates the model performance at different stages of the FLIP trajectories (e.g. the falling liquid block in the beginning or the sloshing liquid at the bottom of the domain at later time steps). The third metric (MSE 400), is the MSE averaged across time, particle and spatial dimensions, but applied to the full rollouts of length 400.

Evaluating rollouts solely with particle-wise MSE can be misleading. Considering two particles A and B, the MSE could be high if the model predicts particle A at the position of particle B and vice versa, even so the particle distributions of prediction and ground truth match. Due to the chaotic nature of fluid motion, the MSE can therefore be misleading and one might favour a distributional metric, which is invariant under particle permutations. Therefore, we evaluate the model variants from section 5.3 with a fourth metric (EMD) using optimal transport (OT) and the earth mover's distance or Wasserstein metric [78, 18].

The optimal transport problem describes an optimization problem of the form

$$d_M(\mathbf{r}, \mathbf{c}) = \min_{P \in U(\mathbf{r},\mathbf{c})} \sum_{i,j} P_{ij} M_{ij}. \tag{5.3}$$

Here, $\mathbf{r}$ and $\mathbf{c}$ describe the probabilistic weights (sum to 1) of all particles in the source and target distribution and $U(\mathbf{r}, \mathbf{c})$ is the set of positive $n \times m$ matrices defined by

$$U(\mathbf{r}, \mathbf{c}) = \left\{ P \in \mathbb{R}_{>0}^{n \times m} \mid P\mathbf{1}_m = \mathbf{r}, P^\top \mathbf{1}_n = \mathbf{c} \right\}.$$

M is the cost matrix, which in case of a fluid with n particles has the size $n \times n$ and contains the particle-wise distances. The optimization problem, described by equation 5.3, is to find the Wasserstein distance $d_M(\mathbf{r}, \mathbf{c})$ which is calculated by using the element of $U(\mathbf{r}, \mathbf{c})$ which yields the lowest distance between the particle distribution of ground truth and prediction, measured by multiplying it with the cost matrix M. This distance is also called the *earth mover distance* (EMD). This metric therefore relates each particle of the predicted particle distribution to its nearest neighbors in the target distribution and is invariant under particle permutations. For our evaluations, we calculate the Wasserstein distance using the Python Optimal Transport (POT) library [24].

We evaluate the performance of each model on the validation set during training. Figure 5.5 shows exemplary screening results using all four metrics for the 1-step-noise model variant described in section 5.3. As the MSE-acc 1 is calculated in the same way as the loss with which this model was trained (equation 5.1) this metric shows a stable downwards trend with increasing number of training steps. The MSE 20 shows a similar behavior as it only includes 20-step rollouts and is still quite similar to the training loss. The MSE 400 and EMD metrics produce relatively noisy screenings which still show a strong downwards trend. The increase in noise is explained by the fact that the model is not directly optimized for the tasks measured by these metrics. Due to the chaotic weight updates within the network during training, small updates for the optimization with the one-step loss can have large impacts on the performance in the full trajectory rollout task.

To favour models which have a high performance on generating full-scale rollouts, we use the MSE 400 to choose model checkpoints for further analysis and apply the best checkpoint of each model variant to the test set. Figure 5.6 summarizes the performances of all models evaluated by the four metrics described before. The exact scores underlying this figure can be found in table B.1 in the appendix. Figure 5.7 shows the trajectories of the MSE 400 and the EMD over time for all model variants, averaged over the entire test set. Figure 5.8 shows exemplary rollout predictions from all model variants for one

of the test set scenarios. The link to videos of the rollout predictions can be found in table A.1 in the appendix.

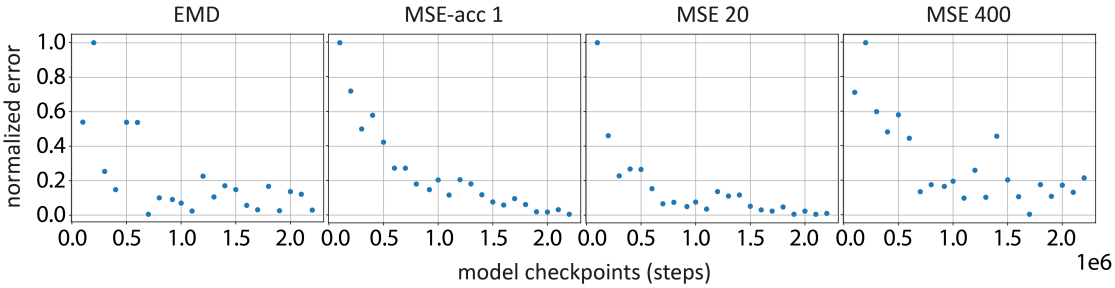

Figure 5.5: Screening results of the 1-step-noise model on the validation set during training.

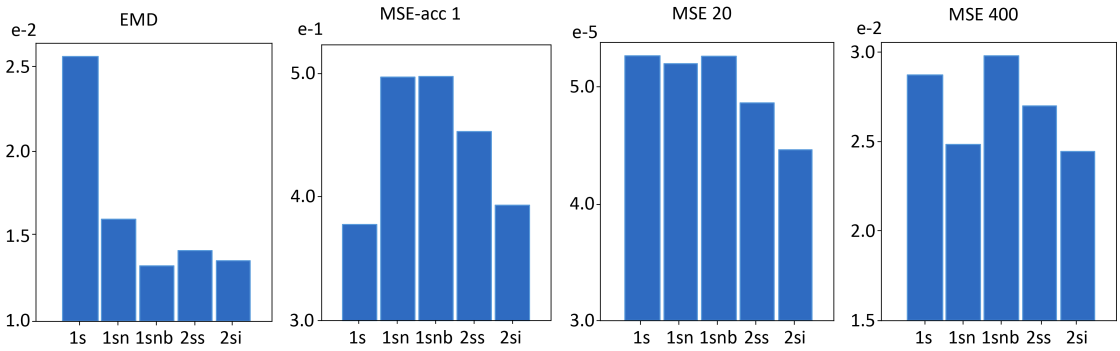

Figure 5.6: Test set performances of the 1-step (1s), 1-step-noise (1sn), 1-step-noise-bounded (1snb), 2-step-scratch (2ss) and 2-step-initialized (2si) models.

The 1-step model variant shows the best performance with respect to the MSE-acc 1 metric. This model is solely trained for this task, without any artificial noise or multi-step error, leading to its superior performance on the specific task of one-step predictions. While its MSE 20 and MSE 400 evaluations are rather similar to other model variants, its EMD evaluation shows a far worse performance. Therefore, the model's rollout prediction follows the ground truth rollout closely, but rather tries to hold the particles at similar positions, than to really apply dynamics which produce a similar visual behaviour compared to the FLIP trajectory. The rollout example in figure 5.6 shows that the model sometimes produces strange, nonphysical motions (last column).

The 1-step-noise model shows a much better EMD performance, but performs worse than the 1-step model when evaluated with MSE-acc 1. This is likely due to the artificial noise which challenges this model with an additional difficulty during training. Figure 5.8 shows that this model variant leads to fluid motion which is more compressed than

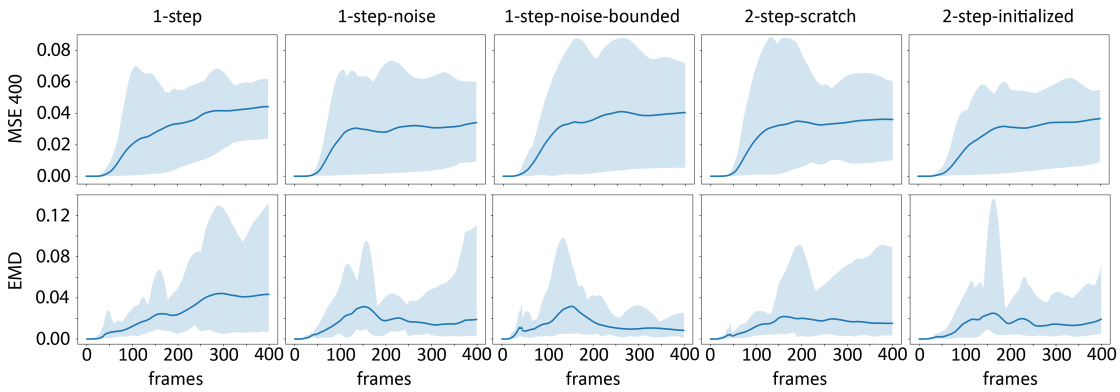

Figure 5.7: Error trajectories of the MSE 400 (MSE averaged over full rollouts) and the EMD metrics for all five model variants. The blue line represents the mean (over the entire dataset) and the shaded area indicates the range of possible values.

in the ground truth rollout. Similar compression can be seen in the rollout of the 1-step model.

The 1-step-noise-bounded model has the best EMD performance, resembles the 1-step-noise model's performance for MSE-acc 1, but has lower performance in case of MSE 20 and MSE 400. With removal of the boundary distance features, the model has to extract this information solely from the interaction between fluid particles and obstacles. Figure 5.8 shows that this model variant produces rollouts with a similar density than the ground truth. However, it produces dynamics which cause the fluid to slide faster along the boundaries, possibly caused by the constant interactions with boundary particles which normally lets a fluid block splash at an obstacle.

For the 2-step model variants, the 2-step-initialized model has better performances in all metrics compared to the 2-step-scratch model. It surpasses all other variants on the MSE 20 and MSE 400 metrics and reaches similar performances to the best models for the MSE-acc 1 and EMD metrics. Thus, using a 1-step model as initialization for 2-step models seems to have a positive effect. However, figure 5.8 shows that it has inherited the compression tendency of the 1-step model, leading to a lower density of the fluid during rollout. In contrast, the 2-step-scratch model preserves the density throughout the rollout (similar to the 1-step-noise-bounded model).

As described in section 5.3 the 1-step-noise-bounded model should have the capability to generalize to larger domains. Figure 5.9 shows two examples from such a generalization experiment (link to videos can be found in table A.1 in the appendix). We extend the domain size of 32×32 to 32×64 and adapt the scaling to put the position values into the range of 0.1 to 0.9 in x and 0.1 to 1.8 in y direction. Then, we use both models, 1-step-noise and 1-step-noise-bounded (trained on 32×32 domains), to generate rollouts

for these new scenarios. Figure 5.9 shows that the 1-step-noise model produces dynamics which let the fluid splash when crossing half of the new domain. This height equals the bottom of the domain seen during training where fluid particles normally splash and are accelerated in x-direction. Therefore, the model seems to have learned to correlate accelerations in x-direction with the distance to the boundaries.

The 1-step-noise-bounded model shows better behaviour than the 1-step-noise model but still deforms the fluid block after crossing half of the domain. This indicates that the model normally not only correlates accelerations and boundary distances but also accelerations and velocities. In the training examples, most fluids had similar velocities right before splashing at the bottom. When the fluid crosses half of the new domain at height 32, the particles cross this velocity threshold, causing the model to decelerate the particles in y- and accelerate the particles in x-direction.

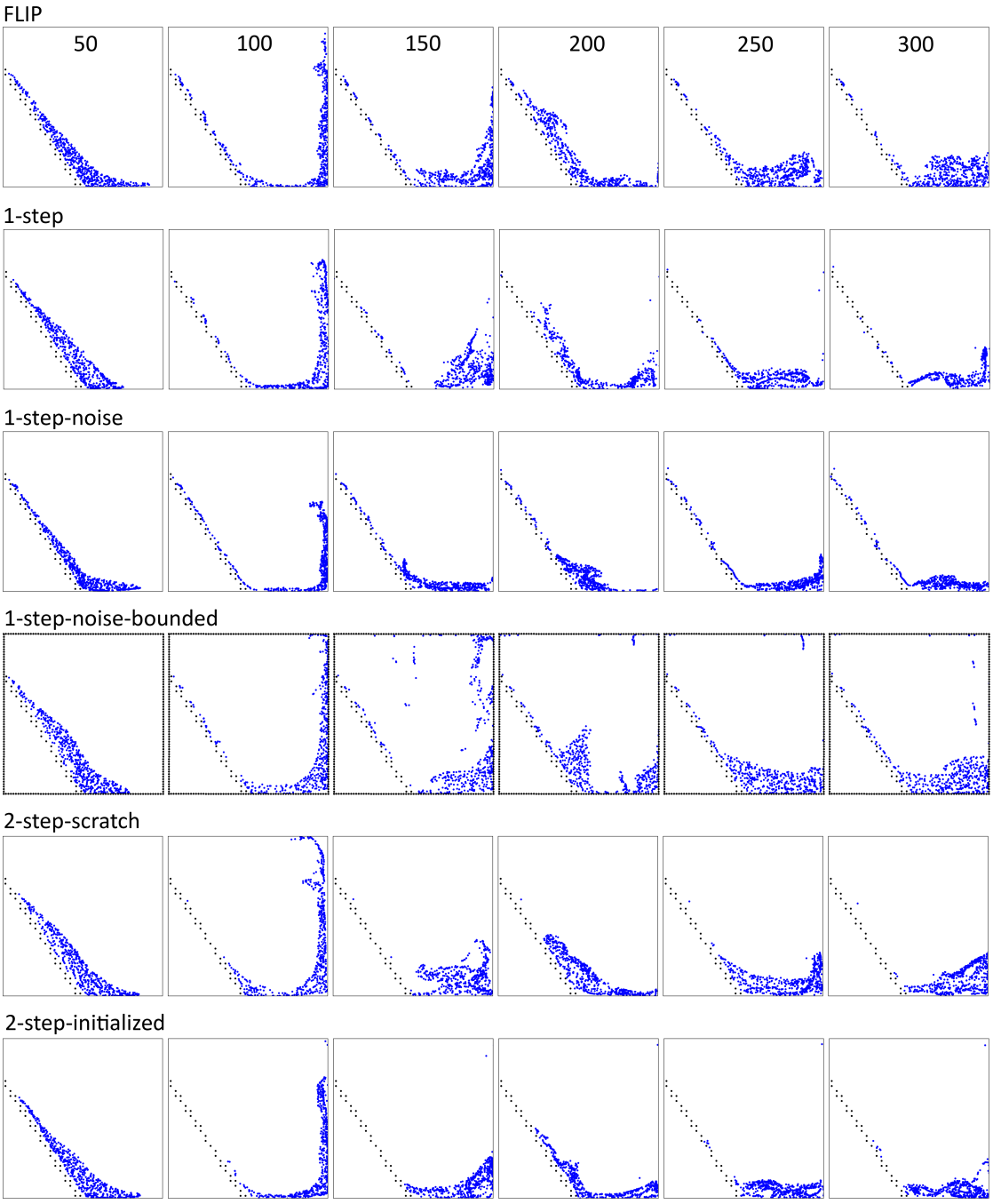

Figure 5.8: Example predictions for all five model variants. Different time steps are indicated at the top. The top row shows the ground truth trajectory. The initial state of the trajectory is shown at the top of the second column of figure 5.3.

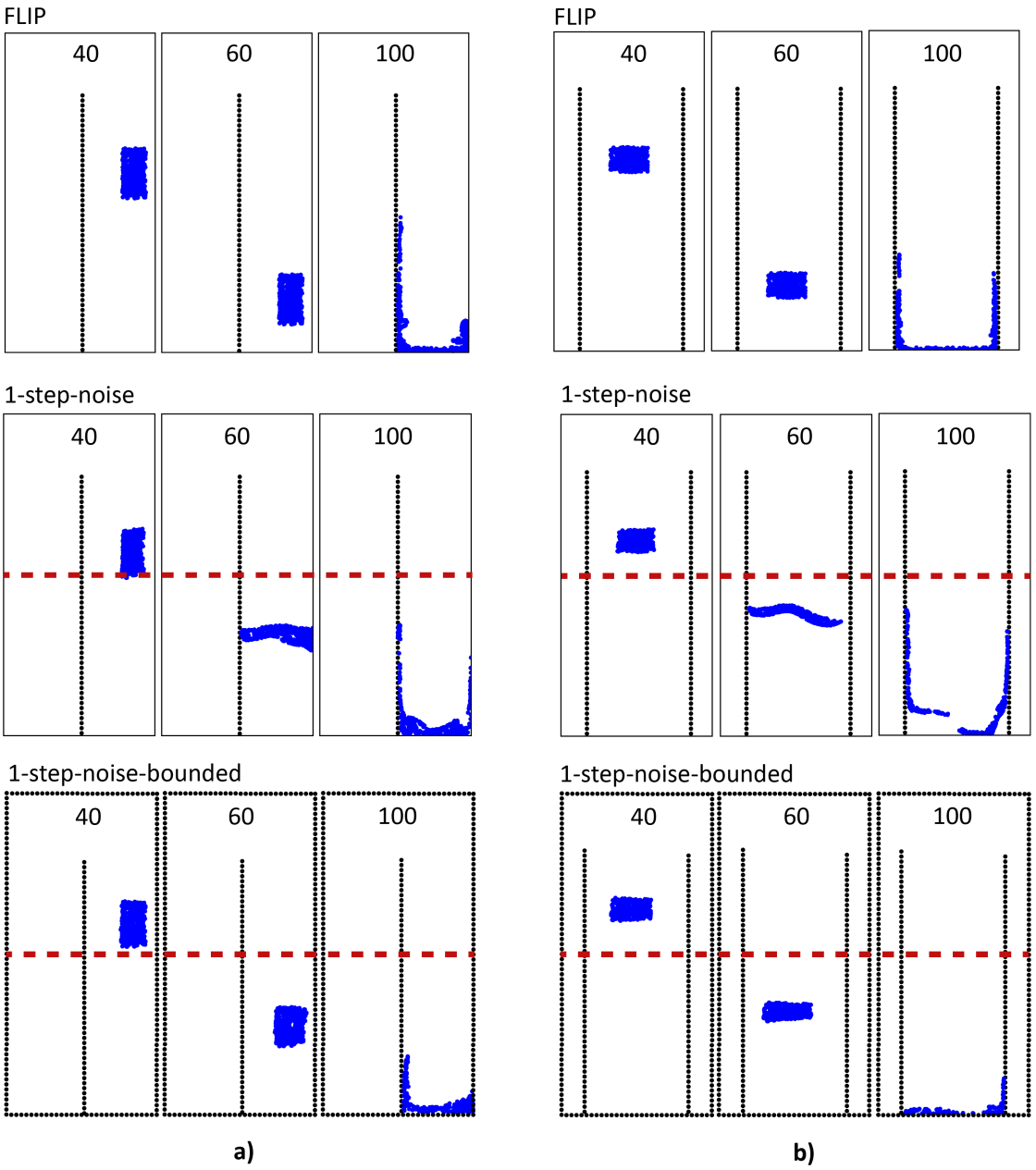

Figure 5.9: Two example predictions from the domain generalization experiment. Both left and right examples show ground truth (FLIP), predictions from 1-step-noise and predictions from 1-step-noise-bounded at the frames indicated by the numbers. The red dashed lines indicate the original domain size.

Chapter 6

Conclusions and Outlook

Our differentiable FLIP simulation for incompressible, inviscid fluids, extends the differentiable physics framework Φ_{Flow} and can be used to solve inverse optimization problems. Our simulator enables the integration of graph- and point-based deep learning models into fluid simulations and supports the development of new, physics-based losses and architectures.

We have shown, that GNS models are capable of learning fluid dynamics from a dataset produced by our FLIP simulator. Training the GNS models with our multi-step loss enabled the models to mitigate accumulating errors in simulation rollouts and yielded competitive results compared to models trained with the artificial noise proposed by Sanches-Gonzales *et al.* [64]. Removing the domain-specific boundary distance features and training GNS models on domains with obstacle boundaries, increased the models capacity to generalize to larger domains. However, we find that these models still have the tendency to deform the fluids as soon as they are passing the original domain size seen during training. This indicates that the GNS does not really learn the underlying physics but rather problem-specific correlations between input velocities and output accelerations. This is supported by the fact that the architecture takes the previous five velocities as inputs and does not only rely on just the previous velocity as one would expect from physical dynamics. Furthermore, most models are unable to retain the original density of the fluid and compress the fluid particles. This is another unphysical behaviour supporting our conclusion.

Extending the GNS architecture with strong inductive biases towards physical laws and symmetries could improve its physical understanding and force it to learn actual dynamics instead of problem-specific correlations. Future work should also concentrate on examining the physical reasoning of learned simulators in more detail. Transforming parts of learned simulators into symbolic models [17, 15] and extending tools like the recently proposed GNNExplainer from Ying *et al.* [82] could provide further insights into the reasoning of Graph Networks and could yield new ideas on how to improve their physical understanding.

Having a differentiable fluid simulation and learned, differentiable simulators, future work could also compare their respective behaviour in optimization problems and explore the use of learned simulators for solving inverse problems.

Another possibility for future work could be to solve current limitations of our FLIP simulator. Specifically, the obstacle interaction could still be improved by implementing boundary conditions with different behaviour at obstacle bottom and top. This could improve the physical behaviour of the simulation by preventing fluid particles from sticking to the lower side of obstacles, which happened in some of the tested scenarios. Our optimization experiments in section 4.3 have shown, that the chaotic nature of fluid dynamics can prevent gradient descent from converging to satisfying solutions. This could be improved by replacing the L_2 loss with a distributional loss which is invariant to particle permutations. One possibility would be to use a frequency loss on the density distribution of the fluid on the underlaying grid. This requires implementing custom-designed gradients which is currently a work-in-progress.

Appendix A

Video Links

Simulation examples (section 4.1)	https://git.io/JOUjI
FLIP vs. PIC (section 4.2)	https://git.io/JOUh1
Symmetry experiments (section 4.2)	https://git.io/JOUhS
Example predictions from GNS models (section 5.4)	https://git.io/JOOQG
Example generalization experiments (section 5.4)	https://git.io/JOOQn

Table A.1: Links to videos of different simulations and predicted rollouts.

Appendix B

Performance Table of GNS Models

Model variant	EMD (10^{-2})	MSE-acc 1 (10^{-1})	MSE 20 (10^{-5})	MSE 400 (10^{-2})
1-step	2.598	**3.775**	5.653	2.853
1-step-noise	1.617	4.980	5.574	2.469
1-step-noise-bounded	**1.336**	4.985	5.648	2.959
2-step-scratch	1.428	4.535	5.181	2.682
2-step-initialized	1.367	3.933	**4.711**	**2.430**

Table B.1: Testset performance of all model variants from section 5.3, evaluated with multiple metrics. The MSE is reported for one-step (MSE-acc 1) acceleration predictions or averaged over either 20-step rollouts (MSE 20) or full rollouts (MSE 400). Bold values indicate the best score of the respective metric.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. USENIX Association, 2016.

[2] J. Ba, J. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. Cambridge University Press, 2000.

[4] P. W. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, pages 4502–4510, 2016.

[5] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[6] F. A. Belbute-Peres, K. Smith, Kelsey R. Allen, J. Tenenbaum, and J. Z. Kolter. End-to-end differentiable physics for learning and control. In *NeurIPS*, 2018.

[7] J.U. Brackbill and H. M. Ruppel. Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics*, 65(2):314–343, 1986.

[8] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

[9] R. Bridson. *Fluid Simulation*. A. K. Peters, Ltd., USA, 2nd edition, 2015.

[10] M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34: 18–42, 2017.

[11] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, and L. Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91:045002, 2019.

[12] A.J. Chorin and J.E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Texts in Applied Mathematics 4. Springer-Verlag, 2nd edition, 1990.

[13] T. Cohen, M. Weiler, B. Kicanaoglu, and M. Welling. Gauge equivariant convolutional networks and the icosahedral cnn. In *ICML*, 2019.

[14] K. Cranmer, J. Brehmer, and G. Louppe. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences*, 117:30055–30062, 2020.

[15] M. Cranmer, R. Xu, P. Battaglia, and S. Ho. Learning symbolic physics with graph networks. *arXiv preprint arXiv:1909.05862*, 2019.

[16] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho. Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*, 2020.

[17] M. Cranmer, A. Sanchez-Gonzalez, P. W. Battaglia, R. Xu, K. Cranmer, D. Spergel, and S. Ho. Discovering symbolic models from deep learning with inductive biases. *arXiv preprint arXiv:2006.11287*, 2020.

[18] M. Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *NIPS*, 2013.

[19] DeepMind. Sonnet, 2017. `https://github.com/deepmind/sonnet`.

[20] DeepMind. Graph nets library, 2018. `https://github.com/deepmind/graph_nets`.

[21] J. Degrave, M. Hermans, J. Dambre, and F. Wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in Neurorobotics*, 13, 2017.

[22] F. Ferstl, R. Ando, C. Wojtan, R. Westermann, and N. Thuerey. Narrow band flip for liquid simulations. *Computer Graphics Forum*, 35(2):225–232, 2016.

[23] R. Fine and F. Millero. Compressibility of water as a function of temperature and pressure. *The Journal of Chemical Physics*, 59:5529–5536, 11 1973.

[24] R. Flamary and N. Courty. Pot python optimal transport library, 2017. URL `https://pythonot.github.io/`.

[25] N. Foster and D. Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 181–188, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[26] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[27] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *Proceedings of the 34th International Conference on Machine Learning*, 70:1263–1272, 2017.

[28] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[29] S. Greydanus, M. Dzamba, and J. Yosinski. Hamiltonian neural networks. *Neural Information Processing Systems*, pages 15353–15363, 2019.

[30] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics, a Practical Introduction*. SIAM, Philadelphia, 1998.

[31] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun. Deep learning for 3d point clouds: A survey. *arXiv preprint arXiv:1912.12033*, 2019.

[32] R. Hanocka, A. Hertz, N. Fish, R. Giryes, S. Fleishman, and D. Cohen-Or. Meshcnn: a network with an edge. *ACM Transactions on Graphics (TOG)*, 38:1–12, 2019.

[33] F. H. Harlow. The particle-in-cell method for numerical solution of problems in fluid dynamics. *Experimental arithmetic, high-speed computations and mathematics*, 1963.

[34] F. H. Harlow and J. E. Welch. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[35] 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. Fern'andez del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[36] S. He, Y. Li, Y. Feng, S. Ho, S. Ravanbakhsh, W. Chen, and B. Póczos. Learning to predict the cosmological structure formation. *Proceedings of the National Academy of Sciences of the United States of America*, 116:13825–13832, 2019.

[37] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–435, 1952.

[38] P. Holl, V. Koltun, and N. Thuerey. Learning to control pdes with differentiable physics. *arXiv preprint arXiv:2001.07457*, 2020.

[39] Y. Hu, T. Li, L. Anderson, J. Ragan-Kelley, and F. Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):201, 2019.

[40] Y. Hu, L. Anderson, T. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand. Difftaichi: Differentiable programming for physical simulation. *ICLR*, 2020.

[41] J. Ingraham, A. J. Riesselman, C. Sander, and D. S. Marks. Learning protein structure with a differentiable simulator. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[42] K. Janocha and W. Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[43] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001. URL `http://www.scipy.org/`.

[44] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, 2012.

[46] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[47] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[48] Y. Li, J. Wu, R. Tedrake, J. B. Tenenbaum, and A. Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. In *ICLR*, 2019.

[49] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy. *ICML 2015 AutoML Workshop*, 238, 2015.

[50] T. M. Mitchell. The need for biases in learning generalizations. *Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ.*, 1980.

[51] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992.

[52] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2006.

[53] M. A. Nielsen. *Neural networks and deep learning*. Determination Press, 2018. `http://neuralnetworksanddeeplearning.com/`.

[54] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[55] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. De-Vito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. *NIPS-W*, 2017.

[56] M. Raissi, A. Yazdani, and G. E. Karniadakis. Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data. *arXiv preprint arXiv:1808.04327*, 2018.

[57] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[58] D. Ram, T. F. Gast, C. Jiang, C. Schroeder, A. Stomakhin, J. Teran, and P. Kavehpour. A material point method for viscoelastic fluids, foams and sponges. *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2015.

[59] S. Rasp, M. Pritchard, and P. Gentine. Deep learning to represent subgrid processes in climate models. *Proceedings of the National Academy of Sciences of the United States of America*, 115:9684–9689, 2018.

[60] D. Rolnick, P. Donti, Lynn H. Kaack, K. Kochanski, A. Lacoste, K. Sankaran, A. Ross, N. Milojevic-Dupont, N. Jaques, A. Waldman-Brown, A. Luccioni, T. Maharaj, E. Sherwin, S. K. Mukkavilli, K. P. Körding, C. Gomes, A. Ng, D. Hassabis, J. C. Platt, F. Creutzig, J. Chayes, and Y. Bengio. Tackling climate change with machine learning. *ArXiv*, 2019.

[61] F. Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. *American Journal of Psychology*, 76:705, 1963.

[62] D. E. Rumelhart, Hinton G. E., and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[63] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, Merel J., M. Riedmiller, R. Hadsell, and P. W. Battaglia. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

[64] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and Battaglia P. W. Learning to simulate complex physics with graph networks. *arXiv preprint arXiv:2002.09405*, 2020.

[65] F. Scarselli, M. Gori, A. C. Tsoi, Hagenbuchner M., and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[66] C. Schenck and D. Fox. Spnets: Differentiable fluid dynamics for deep neural networks. In *Proceedings of the Second Conference on Robot Learning (CoRL)*, Zurich, Switzerland, 2018.

[67] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks: the official journal of the International Neural Network Society*, 61:85–117, 2015.

[68] S. Schoenholz and E. D. Cubuk. Jax, m.d.: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. *arXiv preprint arXiv:1912.04232*, 2019.

[69] A. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. ídek, A. W. R. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P.t Kohli, D. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577:706–710, 2020.

[70] J. Shlomi, P. W. Battaglia, and J. Vlimant. Graph neural networks in particle physics. *arXiv preprint arXiv:2007.13681*, 2020.

[71] J. Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[72] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle. A material point method for snow simulation. *ACM Trans. Graph.*, 32(4), July 2013.

[73] D. Sulsky, S.-J. Zhou, and H. L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer physics communications*, 87(1-2):236–252, 1995.

[74] N. Thuerey, K. Weissenow, H. Mehrotra, and N. Mainali. Deep learning methods for reynolds-averaged navierstokes simulations of airfoil flows. *AIAA Journal*, 58: 15–26, 2018.

[75] M. Toussaint, K. R. Allen, K. Smith, and J. Tenenbaum. Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems*, 2018.

[76] K. Um, Y. Fei, R. Brand, P. Holl, and N. Thuerey. Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. *arXiv preprint arXiv:2007.00016*, 2020.

[77] B. Ummenhofer, L. Prantl, N. Thuerey, and V. Koltun. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*, 2020.

[78] C. Villani. *Topics in optimal transportation*. American Mathematical Soc., 2003.

[79] W. Wang, S. Axelrod, and R. Gómez-Bombarelli. Differentiable molecular simulations for control and learning. *arXiv preprint arXiv:2003.00868*, 2020.

[80] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

[81] A. Yazdani, L. Lu, M. Raissi, and G. E. Karniadakis. Systems biology informed deep learning for inferring parameters and hidden dynamics. *PLOS Computational Biology*, 16(11):1–19, 11 2020.

[82] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems*, 32:9240–9251, 2019.

[83] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[84] Y. Zhu and R. Bridson. Animating sand as a fluid. *ACM Trans. Graph*, 24(3): 965–972, 2005.

Acknowledgement

I would like to thank Nils Thuerey for his supervision and helpful suggestions.

Special thanks goes to my supervisor Philipp Holl for the many weekly meetings throughout the last months, for his continuous advice and guidance and for proof-reading this thesis.

I wish to thank my family for their continuous support and love and for the countless interesting discussions.

Finally, as always, I'm glad for the crazy ones, the misfits, the rebels and the troublemakers without whom this world would be a far less interesting and inspiring place.