Beyond Grids: Neural Networks in Fluid Dynamics

Tooba Rahimnia
Department of Electrical & Computer Engineering
McGill University, Montreal

April 15, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

Fluid mechanics simulation is a crucial domain in real-world simulation, addressing a wide range of scenarios, including pouring liquids and vast ocean views, often becoming an indispensable tool for directors and artists. The Eulerian methodology stands as a significant approach in fluid simulation, segmenting the simulation space into grid cells to monitor fluid properties like velocity, pressure, and temperature. To ensure fluid incompressibility, the Eulerian technique engages in solving the Poisson equation, generating a sparse, symmetric, and positive definite linear system during projection, which can be solved iteratively; however, the iterative method, reliant on conventional numerical procedures, proves computationally intensive, particularly compared to other stages of the simulation.

In enhancing simulation speed, one avenue involves employing deep learning techniques to handle pressure projection, circumventing the need for analytical solutions via linear equations. In our present endeavor, we introduce a machine learning solution, distinct from traditional numerical approaches, training our model to comprehend fluid system behavior, thereby expediting the determination of fluid pressure values.

Abrégé

La simulation de la mécanique des fluides constitue un domaine vital au sein de la simulation du monde réel, répondant à des scénarios divers, de l'écoulement de liquides aux vastes vues océaniques, devenant souvent un outil indispensable pour les réalisateurs et les artistes. La méthodologie eulérienne se présente comme une approche significative dans la simulation de fluides, segmentant l'espace de simulation en cellules de grille pour surveiller les propriétés du fluide telles que la vitesse, la pression et la température. Pour garantir l'incompressibilité du fluide, la technique eulérienne résout l'équation de Poisson, générant un système linéaire, symétrique et défini positif lors de la projection, qui peut être résolu de manière itérative. Cependant, la méthode itérative, reposant sur des procédures numériques conventionnelles, se révèle intensément en termes de puissance de calcul, notamment par rapport aux autres étapes de la simulation.

Pour améliorer la vitesse de simulation, une voie consiste à utiliser des techniques d'apprentissage profond pour gérer la projection de la pression, contournant ainsi le besoin de solutions analytiques via des équations linéaires complexes. Dans notre effort actuel, nous introduisons une solution d'apprentissage automatique, distincte des approches numériques traditionnelles, formant notre modèle à comprendre le comportement du système de fluide, accélérant ainsi la détermination des valeurs de pression du fluide.

Acknowledgements

I would like to express my sincere gratitude to all those who have played a role in the completion of this thesis. Their support, guidance, and encouragement made a pivotal impact in my academic journey.

My sincere gratitude goes to my supervisor, Prof. Derek Nowrouzezahrai, for his exceptional guidance and support throughout the entire process of this thesis. His expertise, dedication, and unstoppable commitment to excellence have been remarkably influential in shaping this research and my academic growth as a graduate student.

I am grateful to Marc-Antoine Beaudoin, Olexa Bilaniuk, and Fabrice Normandin for their valuable insights, stimulating discussions, and shared experiences. Their intellectual contributions provided me a supportive environment to grow as a researcher. I also want to thank all my friends who stood by me, offering their encouragement and understanding, making this academic journey not just manageable but also enjoyable.

To my parents, Maria and Majid, thank you for your wisdom, guidance, and the invaluable life lessons you have imparted to me. Your work ethic, and determination have inspired me to always strive for my best. I am grateful for the sacrifices you have made to provide me with the best educational opportunities. Your unconditional supports have enabled me to pursue my passion.

Contribution of Authors

I am the sole author of this thesis, responsible for the entire composition. While working on this thesis, I received guidance from Prof. Derek Nowrouzezahrai to ensure its proper direction. In addition, I engaged in discussions with my colleagues to explore different topics and address any technical challenges.

In specific chapters, my role varied as follows: In the background (Chapter 2), and literature review (Chapter 3), I was the sole contributor and researcher. The execution of the neural network and machine learning components in Chapter 4 was exclusively my responsibility. And when designing and parameterizing the model, I primarily took the lead, with some assistance from my lab mates to familiarize myself with the software and neural network design process.

For the results and discussions in Chapter 5, I provided my own observations based on the data, although I sought input from my supervisor to identify areas where further elaboration or discussion was necessary.

Finally, in the conclusion and the notes for future work (Chapter 6), I shared my personal insights and observations regarding the entire project's progression and possible extensions.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Fluids are incredibly pervasive in our lives. They envelop our planet, with approximately 70% of its surface covered in water [1]. Even within our own bodies, fluids play a crucial role, particularly within our circulatory system, acting as biological fluid engines; however, it is easy to overlook the fact that we ourselves reside within a fluid environment: the atmosphere. This mixture of gases, essential for creating a habitable environment, is perhaps the fluid that subconsciously captures our attention the most on a daily basis. It influences the weather phenomena we witness, ranging from the formation of fluffy cumulus clouds to the powerful spectacle of thunderstorms. Undoubtedly, our existence is deeply intertwined with the dominance of fluids in our world.

Real-time simulation of fluid flow has posed a longstanding challenge in various fields of application. It finds relevance in computational fluid dynamics, where it serves industrial purposes, as well as in computer graphics and animation, particularly in the creation of realistic smoke and fluid effects. In the realm of computer graphics research, physically based fluid simulation has emerged as a highly significant area of exploration. The simulation process involves solving the Navier-Stokes equations, which are nonlinear partial differential equations. To discretize these equations, numerous numerical simulation methods have been employed, including Lagrangian methods [33] and Eulerian methods [5]. In the domain of high-resolution fluid simulation, Eulerian methods have gained wide adoption due to their enhanced accuracy in reconstructing and rendering fluid surfaces.

Figure 1.1: In 1998, the animated movie "Antz" by Dreamworks marked a significant milestone as the first film to feature fluid animations generated through simulation. Nevertheless, the process of animating the fluids in the movie was complex and time-consuming. The subsequent year, Jos Stam introduced a groundbreaking paper that streamlined the simulation procedure, making it more accessible and efficient for practical use [43].

The behavior of numerous physical phenomena is dictated by the incompressible Navier-Stokes equations, a set of partial differential equations that govern the fluid velocity field over time. To simulate these equations, two primary computational approaches are employed. The first is the use of Lagrangian methods, which approximate continuous quantities by employing discrete moving particles [21]. The second approach is the adoption of Eulerian methods, which approximate quantities on a fixed grid [17]. In the context of this work, we have chosen to utilize the latter approach.

In order to ensure the incompressibility of fluids, Eulerian methods tackle the Poisson equation, which gives rise to a widely recognized sparse, symmetric, and positive-definite linear system during the projection step. Due to the iterative nature of solving the Poisson equation using traditional numerical methods, the projection step often consumes more time compared to other stages of the simulation process. This can significantly tax computational resources [53] and pose challenges when minor adjustments are required in fluid simulation.

To enhance the efficiency of the projection step, numerous effective numerical methods have been proposed in the field of fluid simulation. Notably, traditional convex optimization techniques, such as the Preconditioned Conjugate Gradient (PCG) algorithm, or stationary iterative methods, like the Jacobi or Gauss-Seidel methods can yield exact solutions; however, PCG is limited by high time-constants and is generally not well-suited for GPU hardware. On the other hand, stationary iterative methods like the Jacobi or Gauss-Seidel methods are commonly employed in real-time applications, although the Jacobi method exhibits suboptimal asymptotic convergence. Furthermore, both of these algorithms have a computational complexity that strongly depends on the data (e.g., boundary conditions), and in real-time scenarios, iterative methods are often truncated to accommodate computational constraints.

In recent years, machine learning techniques have gained significant prominence in fluid simulation. Some researchers have approached the fluid simulation process as a supervised regression problem, training blackbox machine learning systems to predict outputs using random regression forests or neural networks for Lagrangian and Eulerian methods, respectively. Ladick´y et al. (2015) proposed an adaptation of regres-

Figure 1.2: To flood New York City in the film "The Day After Tomorrow", the team at Digital Domain had to create software capable of seamlessly combining foam and mists, conducting extensive simulations, and presenting the results in a user-friendly manner even for artists without prior experience in fluid animation. The outcome was the latest iteration of their FSIM software, earning prestigious SciTech Oscars for Doug Roble, Nafees Bin Zafar, and Ryo Sakaguchi [3].

sion forests for smoothed particle hydrodynamics, training a regressor to predict particle states based on handcrafted features [27]. Yang et al. (2016) trained a patch-based neural network to forecast ground truth pressure using local information from previous-frame pressure, voxel occupancy, and velocity divergence [61]. These machine learning-based approaches offer alternative strategies for efficient fluid simulation by harnessing the predictive power of blackbox models.

Some prior studies have utilized learning-based approaches in fluid simulation, which have shown their potential; however, these approaches have certain limitations. They typically rely on a dataset of linear system solutions obtained from an exact solver, which restricts their ability to compute targets during training. As a result, models trained in

Figure 1.3: "Sea of Thieves" released in 2018 for Xbox Series X and Series S, Xbox One, and Microsoft Windows, is a game that revolves around old-fashioned piracy and maritime adventures. Its standout feature is the remarkably realistic water graphics that dominate the gameplay. The water visuals in the game are so well-animated and simulated that they outshine the rest of the game's graphics. Notably, the water's appearance dynamically changes based on light sources, offering various shades from turquoise depths to deep cobalt [44].

this manner can predict ground-truth outputs effectively when provided with an initial frame generated by the exact solver. During testing, the initial frame is typically generated by the model itself, introducing a difference between the training and simulation phases. This difference can potentially lead to discrepancies and errors that accumulate throughout the generated sequence.

These challenges have some parallels with the difficulties faced when generating sequences using recurrent neural networks, as discussed in earlier research [4]. Moreover, while Yang et al. (2016) introduced a ConvNet architecture that demonstrated promising results and offered significant speed improvements compared to the PCG baseline [61], it is important to note that their approach is tailored to specific training conditions. This specialization limits its ability to accurately simulate long-range phenomena, such as gravity or buoyancy, which may be essential for more general use-cases.

1.1 Notation

From its earliest proposal stage, our approach aimed at a deep understanding of the underlying mathematical model in Eulerian fluid modeling while also addressing challenging software engineering aspects. We dedicated considerable effort to comprehending how different components of a fluid simulator function and how to construct and integrate them efficiently and elegantly.

Our primary source of guidance during development was Robert Bridson's "Fluid Simulation for Computer Graphics (Second Edition)" [5]. This textbook served as the foundation for our algorithms and the overall structure of the simulator. Building upon this knowledge, we propose a machine learning-based approach, inspired by the work done by Tompson et al. (2017), to accelerate the linear projection process [48]. Our approach is fast, exhibits data-independent complexity, and is suitable for general cases, harnessing the capabilities of deep learning to derive an approximate linear projection.

In this thesis, we provide background information in Chapter 2, review related works in Chapter 3, introduce the fluid simulation technique used and present our proposed model with implementation details in Chapter 4. Experimental results are detailed in Chapter 5, and we draw conclusions in Chapter 6.

Common symbols used in this thesis

$\mathbf{x}_G$	Eulerian grid position at G
$\mathbf{u}^n_{i,j,k}$	Velocity vector field. Located on the grid at time step n and position (i,j,k)
u	Velocity component in the x direction
v	Velocity component in the y direction
w	Velocity component in the z direction
$p_{i,j,k}$	Pressure in grid at position (i,j,k)
Δt	Time step between two frames
$\mathbf{f}$	External force

Chapter 2

Background

2.1 Fluid Equations

Fluids are all around us, from the air we breathe to the vast oceans covering two-thirds of Earth. They play a central role in some of the most captivating natural phenomena. Computer animation often focuses on Newtonian fluid dynamics, which are primarily described by the incompressible Navier-Stokes equations, a set of partial differential equations:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} + \frac{1}{\rho}\nabla p = \frac{\mu}{\rho}\nabla^2 \mathbf{u} + \mathbf{f}, \tag{2.1}$$

$$\nabla \cdot (\mathbf{u}) = 0. \tag{2.2}$$

In the given equations ρ represents the density, $\mathbf{u}$ is the velocity vector, p stands for pressure, μ is dynamic viscosity, and $\mathbf{f}$ denotes the external force acting on the fluid. It's important to notice that in specific cases, especially when dealing with fluids of low viscosity, we can eliminate the viscosity term without significantly compromising the accuracy of our simulations. Notably, this applies to fluids like water and air, often considered inviscid in computer animation, despite their real-world non-ideal behavior. Nevertheless, it's worth noting that even when we exclude the viscosity term, certain numerical methods used to approximate fluid equations can introduce a slight degree of artificial viscosity into the simulation.

In the context of the current project, our focus is on solving the Euler equations without the viscosity term. By disregarding viscosity, the computational complexity can be reduced while the essential behavior of the fluid flow is captured.

2.1.1 The Momentum Equation

Equation (2.1) is commonly referred to as the momentum equation, and within the realm of fluid dynamics, it represents Newton's Second Law of motion, expressed as $\mathbf{F} = m\mathbf{a}$. This equation elucidates the acceleration of the fluid resulting from the combined effects of internal and external forces acting upon it. While not immediately evident, it is worth noting that the terms present in the Euler equations closely parallel those encountered in Newton's Second Law, which will be further explained in the upcoming discussion.

To facilitate our exposition, let us consider a particle-based fluid simulation where individual particles symbolize discrete fluid entities characterized by mass m, volume V, and velocity $\mathbf{u}$. According to Newton's Second Law, the force $\mathbf{F}$ exerted on each particle is determined by its mass m and its acceleration $\mathbf{a}$:

$$\mathbf{F} = m\mathbf{a}$$

although the exact value of acceleration is unknown, we do have knowledge of the particle's velocity. Since acceleration is the derivative of velocity, we can express the equation in the following form:

$$\mathbf{F} = m\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} \tag{2.3}$$

the expression $\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t}$ represents the Material Derivative, which will be explained in detail later; however, for now, it can be understood as a standard derivative. Moving forward, the analysis involves identifying the forces acting on the particle. The primary force to consider is the force of gravity. It is essential to distinguish this force from simple acceleration due to gravity, as it takes into account the mass of the particle:

$$\mathbf{F}_{\text{gravity}} = m\mathbf{g}$$

it is evident that gravity uniformly influences all fluid particles as an external force. On the other hand, the internal forces within the fluid, which include viscosity and pressure, are more intricate. For the purposes of this project, we disregard viscosity, leaving pressure as the primary internal force of significance. In essence, fluid flows from regions of high pressure to regions of low pressure. To compute the pressure force at a specific particle position, we determine the gradient of the pressure field, denoted as ∇p. This gradient indicates the direction of the steepest pressure increase and is then inverted, resulting in $-\nabla p$, to signify movement away from high-pressure areas towards low-pressure regions. Note that ∇p essentially converts the scalar function (collection of partial derivatives) into a vector format:

$$-\nabla p = -\begin{bmatrix} \partial/\partial x \\ \partial/\partial y \\ \partial/\partial z \end{bmatrix} p = -\begin{bmatrix} \partial p/\partial x \\ \partial p/\partial y \\ \partial p/\partial z \end{bmatrix}.$$

In order to obtain the pressure force, the aforementioned vector should be integrated across the volume of the fluid particle; however, as a straightforward approximation, it suffices to multiply the vector by the volume, represented by V, associated with the particle blob:

$$\mathbf{F}_{\text{pressure}} = -V\nabla p.$$

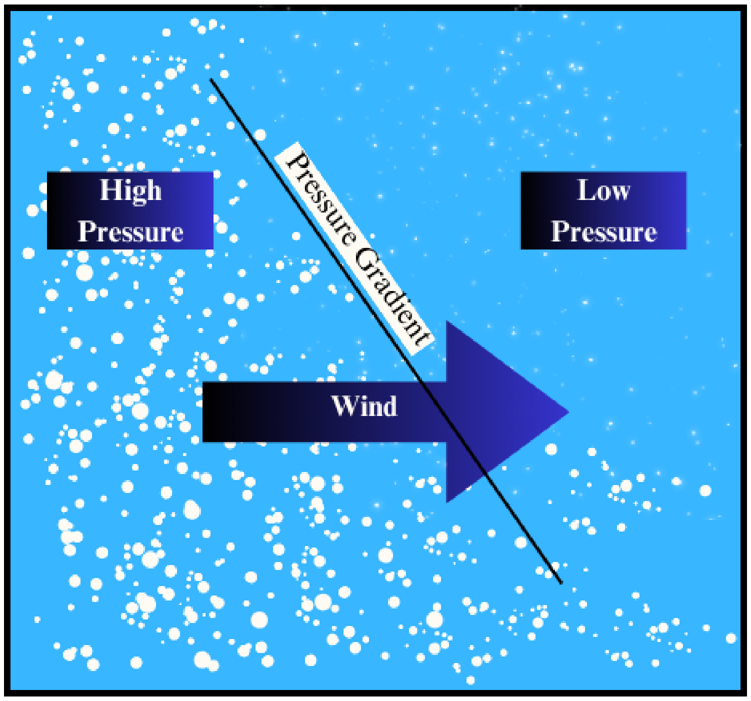


Figure 2.1: The force driving currents caused by the difference in pressure between the high-pressure region and low-pressure region is called the pressure gradient.

Currently, the exact role of pressure in the simulation may not be fully apparent; however, grasping the fundamental concept that high-pressure regions push the fluid away in the direction indicated by the negative pressure gradient is crucial (further elaboration on this in section 2.2.5). With this understanding, equation (2.3) can now be reformulated to incorporate the additional insights as follows:

$$m\mathbf{g} - V\nabla p = m\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t}$$

and by rearranging the equation slightly, we obtain the equation of motion for a fluid blob as follows:

$$m\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -V\nabla p + m\mathbf{g}. \tag{2.4}$$

When performing fluid calculations using a *finite* number of particles (eq. 2.4), there will be approximation errors since the values obtained from sampled particles cannot fully capture the values of unsampled ones. To overcome this limitation, a large number of particles, approaching infinity, is used to describe the fluid, forming what is known as a continuum model. According to Bridson [5], this continuum model has been experimentally proven to closely align with reality across a wide range of scenarios; however, a drawback of this approach arises when the number of particles tends to infinity, causing the mass and volume of each particle to approach zero and rendering the equations of motion meaningless. To address this, the momentum equation is divided by the volume prior to taking the limit as the number of particles approaches infinity,

$$\frac{m}{V}\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\frac{V}{V}\nabla p + \frac{m}{V}\mathbf{g}$$

and by recognizing that the ratio of mass to volume is equivalent to density, denoted as ρ (rho), the expression m/V can be substituted with ρ, resulting in the following formulation:

$$\rho\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\nabla p + \rho\mathbf{g}$$

and, ultimately, by dividing the equation by ρ, the material derivative can be isolated, leading to the final version of the momentum equation. This form of the equation is particularly useful for numerical solvers in order to facilitate the computational solution process:

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\frac{1}{\rho}\nabla p + \mathbf{g}.$$

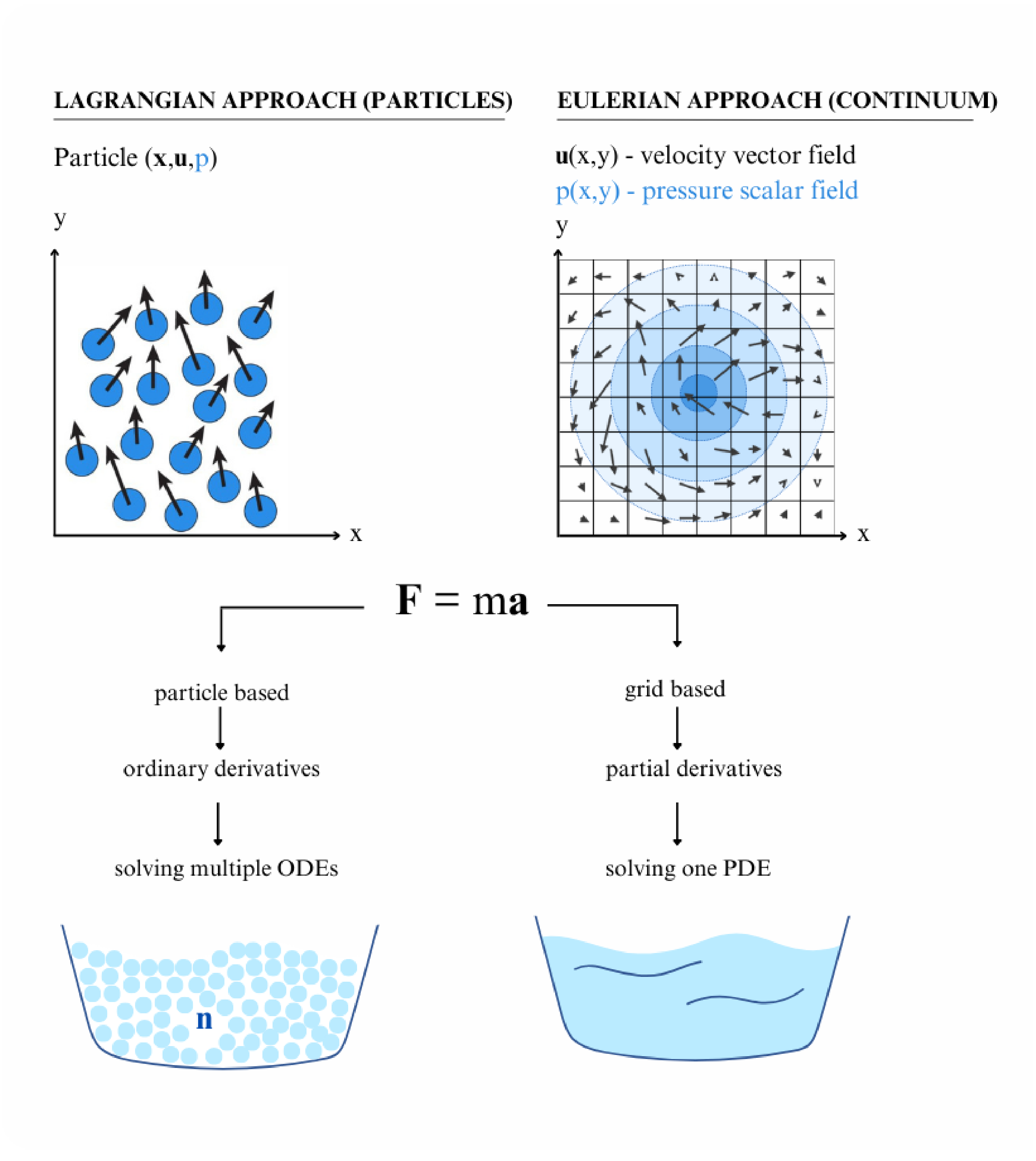


Figure 2.2: Comparing Lagrangian and Eulerian perspectives in fluid dynamics.

2.1.2 Material Derivative

Thus far, the acceleration of the particle has been treated as a regular derivative of velocity; however, in a continuum model such as fluids or deformable solids, there are different methods to track motion. Figure 2.2 illustrates two of such methods:

The first method is known as the Lagrangian method, named after Joseph-Louis Lagrange [58], which is commonly used in particle systems. In this method, points in space have a position $\mathbf{x}$ and velocity $\mathbf{u}$. By representing the fluid body with numerous particles and considering the active forces acting on them, the fluid's behavior can be simulated over time. Smoothed Particle Hydrodynamics (SPH) is an example of an approach that utilizes the Lagrangian method.

On the other hand, the Eulerian method, named after Leonhard Euler [57], takes a different approach. It focuses on fixed locations in space and measures the changes in various quantities (such as velocity, density, temperature) at those specific locations to determine how the fluid flows through the analyzed region. While this approach may not be as intuitive, it offers the advantage of facilitating easier approximation of spatial derivatives, such as pressure and temperature.

The link between these two approaches is established through the Material Derivative, which takes into account changes observed from both the Lagrangian and Eulerian perspectives. Figure 2.3 illustrates this connection by presenting two overlapping fields on the left: a velocity field $\mathbf{u}(\mathbf{x})$ and a scalar field $q(\mathbf{x})$. The trajectory of a particle is represented by the blue line. To provide an analogy, we can envision the particle moving through a river characterized by the velocity field $\mathbf{u}$, while traversing a smoke cloud (resulting from a nearby source) described by the scalar field q. The smoke cloud represents a scalar field, enabling the extraction of scalar measurements, such as smoke density, at

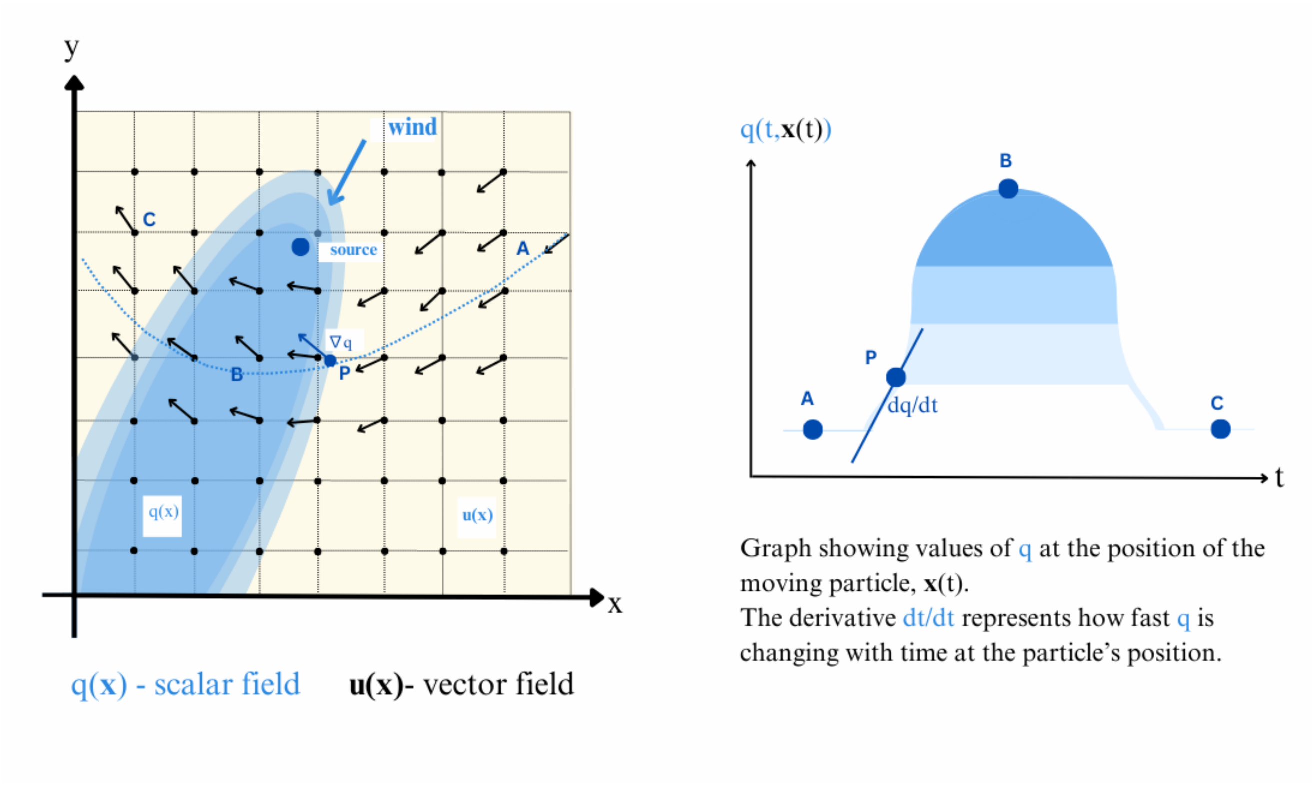

y
x
wind
source
C
A
B
P
∇q
q(x)
u(x)
q(x) - scalar field
u(x)- vector field
q(t,x(t))
B
P
A
C
dq/dt
t
Graph showing values of q at the position of the moving particle, x(t).
The derivative dt/dt represents how fast q is changing with time at the particle's position.

Figure 2.3: The material derivative.

various spatial locations.

The challenge lies in determining the rate of change of the scalar field q not at a fixed point in space, denoted by $\mathbf{x}$, but rather for a particle whose position is defined as a function of time, represented by $\mathbf{x}(t)$. Let's consider point P along the particle's trajectory at time t, denoted as $\mathbf{x}(t)$. At this point, the particle has a velocity $\mathbf{u}$(P). The maximum rate at which q can change from point P is determined by its gradient, represented as ∇q, which is a vector pointing from point P towards the region of q with the highest increase in value.

Therefore, the rate of change of q is primarily influenced by the alignment and magnitude of the velocity vector and the gradient vector. If the velocity vector aligns closely with the gradient vector, the change in q will be significant. To quantify the alignment, we can compute their dot product: $(\mathbf{u} \cdot \nabla q)$ (equivalently represented as $|\mathbf{u}||\nabla q| \cos(\theta)$). This addresses one aspect of the problem. Additionally, we need to consider Eulerian changes in q, which are not dependent on particles. These changes are captured by the term $\partial q/\partial t$. Combining both components, we obtain the following equation for the derivative of q at the position of a moving particle:

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

and the expanded form of the Material Derivative is:

$$\frac{\mathrm{D}q}{\mathrm{D}t} = \frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} + v\frac{\partial q}{\partial y} + w\frac{\partial q}{\partial z}$$

where u, v, and w are the there components of the velocity field, $\mathbf{u}$, in the Cartesian coordinate system.

In closing this section, we delve into applying the Material Derivative to vector functions, specifically the velocity field, which self-advects (as discussed in section 2.2.3). This involves merging the Eulerian and Lagrangian viewpoints to calculate a comprehensive derivative for each vector component:

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = \begin{bmatrix} \frac{Du}{Dt} \\ \frac{Dv}{Dt} \\ \frac{Dw}{Dt} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial t} \\ \frac{\partial v}{\partial t} \\ \frac{\partial w}{\partial t} \end{bmatrix} + \begin{bmatrix} \mathbf{u} \cdot \nabla u \\ \mathbf{u} \cdot \nabla v \\ \mathbf{u} \cdot \nabla w \end{bmatrix}.$$

2.1.3 Fluid Incompressibility

In the real world, fluids generally exhibit some degree of compressibility, although this is often imperceptible to the human eye; however, in the context of computer animation, fluid compressibility can be disregarded, and all fluids can be assumed to be incompressible. Incompressible fluids do not change their volume, and mathematically speaking, this implies that there is no net inflow or outflow across the fluid's surface. The normal component of velocity along the fluid surface ($\partial\Omega$) must be zero in order to maintain this condition:

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{volume}(\Omega) = \iint_{\partial\Omega} \mathbf{u} \cdot \hat{\mathrm{n}} \ = 0$$

and by applying the divergence theorem, the integral can be transformed into a volume integral:

$$\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{volume}(\Omega) = \iiint_{\Omega} \nabla \cdot \mathbf{u} \ = 0.$$

This condition must hold true for any Ω region within the fluid. And the only continuous function that integrates to zero regardless of the integration region is zero itself. Hence, the integrand must be zero everywhere (Bridson, 2015):

$$\nabla \cdot \mathbf{u} = 0$$

this is known as the **incompressibility condition**, which is the second part of the incompressible Navier-Stokes equations (2.2). To effectively satisfy this condition, the fluid's velocity field must be divergence-free. Enforcing this requirement involves utilizing the pressure term from the momentum equation, as will be demonstrated later.

2.1.4 Helmholtz-Hodge Decomposition

As Stam states in his paper [43], any vector field **w** can be expressed uniquely as a sum of a mass-conserving field and a gradient field, as shown bellow:

$$\mathbf{w} = \mathbf{u} + \nabla q \tag{2.5}$$

where **u** is the mass-conserving vector field: $\nabla \cdot \mathbf{u} = 0$ and q is the scalar field. In other words, **u** is the divergence-free component and ∇q is the gradient field component. Applying to both sides of the equation 2.5 the operator $(\nabla \cdot)$, we get:

$$\nabla \cdot \mathbf{w} = \nabla^2 q$$

since $\nabla \cdot \mathbf{u} = 0$. This is a **Poisson equation** that can be solved to get the scalar field q. There are different ways of solving a Poisson's equation, i.e., analytical or finite-difference solutions. Then q can be used to calculate **u** as follows:

$$\mathbf{u} = \mathbf{w} - \nabla q$$

and this result allows us to define an operator P which projects any vector **w** onto its divergence free part $\mathbf{u} = P\mathbf{w}$,

$$\mathbf{u} = P\mathbf{w} = \mathbf{w} - \nabla q.$$

Applying this projection operator on both sides of the (momentum) equation 2.1 we obtain an equation for the velocity:

$$\frac{\partial \mathbf{u}}{\partial t} = P(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \frac{\mu}{\rho}\nabla^2 \mathbf{u} + \mathbf{f}) \tag{2.6}$$

where we have used the fact that $\mathbf{u} = P\mathbf{w}$ and $P\nabla q = 0$. This is our fundamental equation from which we will develop a stable solver.

2.1.5 Boundary Conditions

The behavior of fluid at its boundaries and free surface is crucial for accurate simulations. Firstly, the fluid must be confined within the solid walls of its container, preventing it from flowing through. Secondly, a boundary between the fluid and its surrounding environment, known as the free surface, needs to be established. When dealing with a static solid boundary, the velocity component perpendicular to the solid surface should be set to zero, denoted by $\hat{n}$ as the normal to the solid boundary:

$$\mathbf{u} \cdot \hat{n} = 0.$$

On the other hand, tangential (denoted by $\hat{t}$) velocity along the solid surface can either be zero for viscous fluids (known as the no-slip condition) or left unchanged for inviscid fluids (the no-stick condition). In the current project, the latter condition is adopted.

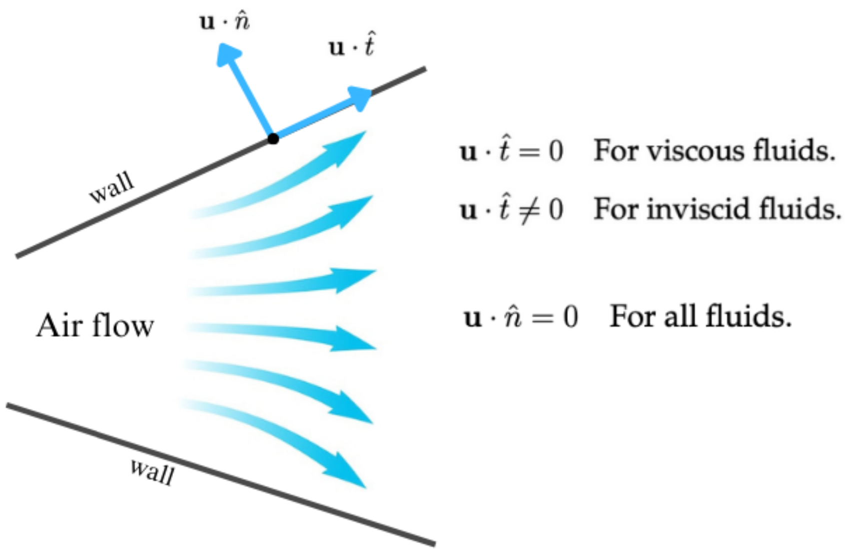

$\mathbf{u} \cdot \hat{n}$
$\mathbf{u} \cdot \hat{t}$
wall
Air flow
wall
$\mathbf{u} \cdot \hat{t} = 0$ For viscous fluids.
$\mathbf{u} \cdot \hat{t} \neq 0$ For inviscid fluids.
$\mathbf{u} \cdot \hat{n} = 0$ For all fluids.

Figure 2.4: Fluid's velocity components at the boundaries (wall).

Lastly, the free surface condition is handled by setting the pressure outside the fluid to zero, without imposing any control on the velocity. Through numerical implementations we can explore how velocities from the fluid are extrapolated in the free space, aiding in the accurate interpolation of velocities at the fluid boundary. This approach enables two essential processes: 1. interpolating missing data within data range and 2. extrapolating future data outside data range.

2.2 Solution Methodologies

Once the mathematical representation of fluid motion has been established, the subsequent phase in developing a fluid simulation involves solving the equations or, more precisely, approximating their solutions with a high level of accuracy. In this section, we provide a comprehensive explanation of the methods employed to achieve the aforementioned objective.

2.2.1 Fluid Mechanics Equations

Splitting is a methodology that involves solving individual components of a complex equation sequentially and then combining their effects to obtain the overall solution. This technique not only simplifies the solving process but also allows for the utilization of diverse numerical methods for different parts based on their suitability:

$$\text{advection} : \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t},$$

$$\text{body force} : \frac{\partial \mathbf{u}}{\partial \mathrm{t}},$$

$$\text{pressure projection} : \frac{\partial \mathbf{u}}{\partial \mathrm{t}} + \frac{1}{\rho}\nabla p = 0, \text{ while } \nabla \cdot \mathbf{u} = 0.$$

For instance, gravity, being a constant force, can be effectively handled with a forward Euler scheme, while advection often necessitates a more accurate method like Runge-

Kutta 2nd order or higher. As a result, instead of solving the Euler equations in a single step, they are divided into distinct parts to be solved independently (**Algorithm 1**).

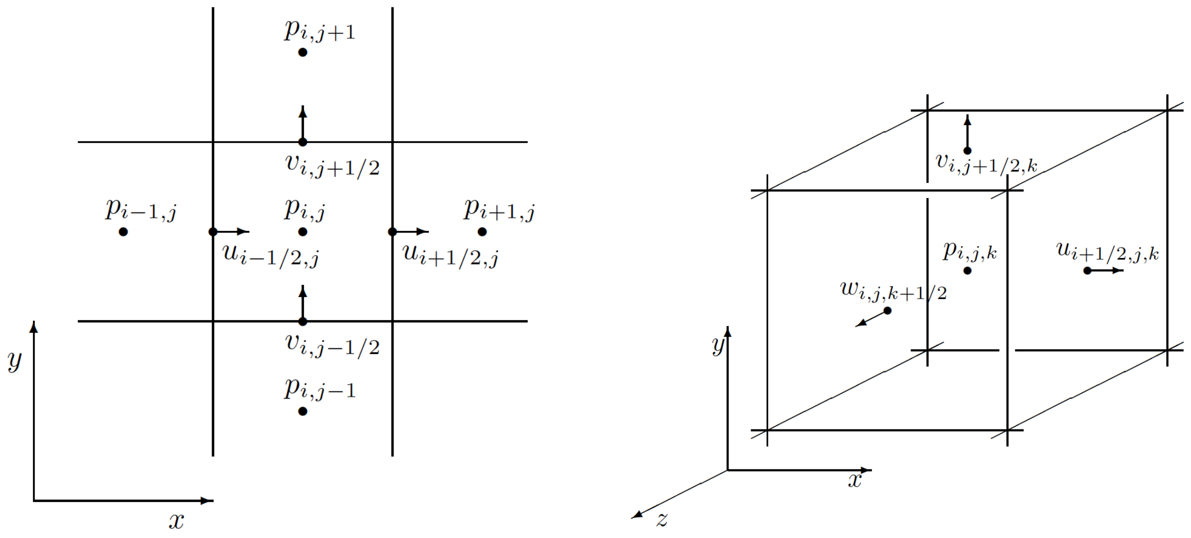
```
Algorithm 1: Basic Fluid Solver
  Step 1: Start with an initial divergence-free velocity field $\mathbf{u^0}$;
  Step 2: for $n \in \mathbb{N}$ do
      Determine an appropriate timestep $\Delta t$ to transition from time $t_n$ to time $t_{n+1}$;
      Advect the velocity field $\mathbf{u^0}$ to obtain $\mathbf{u^A}$;
      Apply external forces g to $\mathbf{u^A}$ to obtain $\mathbf{u^B}$;
      Make $\mathbf{u^B}$ divergence-free and enforce incompressibility;
```

2.2.2 MAC Grid

Before simulating fluid behavior by solving the equations, we must discretize fluid properties and quantities in space. In this project, our focus is on discretizing the velocity and pressure fields for mapping onto a two-dimensional grid.

The Marker-and-Cell (MAC) method, introduced by Harlow and Welch (1965), is employed for solving incompressible fluid flow [23]. This method involves creating a spatial grid where variables are stored at different locations in a staggered arrangement. The rationale behind this arrangement may not be immediately apparent, but as we will see, it greatly simplifies the calculation of pressure to enforce incompressibility. Fig. 2.5a depicts a two-dimensional MAC grid, illustrating the arrangement of variables.

In the MAC grid, the pressure values $p_{i,j}$ are stored at the centers of each cell (i, j). On the other hand, the velocity components are not stored at the cell centers, but rather split into horizontal and vertical components, which are then stored at the centers of the cell faces. Each face is shared by two neighboring cells: the horizontal velocity component is stored at the centers of vertical faces, and the vertical velocity component is stored at the



(a) 2D cell. **(b)** 3D cell.

Figure 2.5: Staggered MAC grids. Images by Bridson (2015).

centers of horizontal faces. A similar arrangement holds in three dimensions (Fig. 2.5b).

The rationale behind the staggered arrangement of pressure and velocity, as extensively discussed by Bridson (2015), can be attributed to its ability to facilitate precise central differences during the computation of spatial derivatives, such as calculating derivatives of the u-component at a specific point i:

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1/2} - u_{i-1/2}}{\Delta x}. \tag{2.7}$$

The use of half indices in the staggered arrangement simply indicates that the velocity positions are located halfway between the cell centers. While in actual code these velocities are referred to using integer indices, using half indices when describing algorithms and formulas provides a more intuitive understanding.

One downside of the staggered arrangement is the increased complexity when interpolating velocity. Since no velocity vectors are actually stored, interpolations for all velocity components are needed whenever the actual velocity vector is required at a known or arbitrary point within the grid. For arbitrary points, a bilinear interpolation (in 2D) or trilinear interpolation (in 3D) is always necessary; however, at grid points (where pressure is stored) and cell face centers (where the velocity components themselves are stored), simple averaging is sufficient:

$$\mathbf{u}_{i,j,k} = \left(\frac{u_{i-1/2,j,k} + u_{i+1/2,j,k}}{2}, \frac{v_{i,j-1/2,k} + v_{i,j+1/2,k}}{2}, \frac{w_{i,j,k-1/2} + w_{i,j,k+1/2}}{2} \right),$$

$$\mathbf{u}_{i+1/2,j,k} = \left(u_{i+1/2,j,k}, \frac{\begin{array}{c} v_{i,j-1/2,k} + v_{i,j+1/2,k} \\ +v_{i+1,j-1/2,k} + v_{i+1,j+1/2,k} \end{array}}{4}, \frac{\begin{array}{c} w_{i,j,k-1/2} + w_{i,j,k+1/2} \\ +w_{i+1,j,k-1/2} + w_{i+1,j,k+1/2} \end{array}}{4} \right).$$

2.2.3 Advection

Advection refers to the movement of fluid particles or blobs with the velocity field $\mathbf{u}$. In the context of fluid dynamics, the advection equation states that the quantities being advected remain constant in the Lagrangian viewpoint and only change their position as they are transported by the flow:

$$\frac{Dq}{Dt} = \frac{\partial q}{\partial t} + \mathbf{u} \cdot \nabla q = 0.$$

In the context of fluid simulation, consider the scenario where each fluid particle has a corresponding temperature value. According to the advection equation, as these particles are transported by the velocity field, their temperature values remain unchanged. This concept is further elaborated upon by Bridson (2015) in Section 1.3.2 of "Fluid Simulation for Computer Graphics".

To solve the advection equation, the Semi-Lagrangian method introduced by Stam [1999] is used. This method is chosen for its simplicity, ease of implementation, and unconditional stability. The idea behind semi-Lagrangian advection is to trace the particle's path backward in time from the point of interest, rather than using forward integration for the time derivative $\partial q/\partial t$ and an accurate central difference for the spatial derivative $\mathbf{u} \cdot \nabla q$.

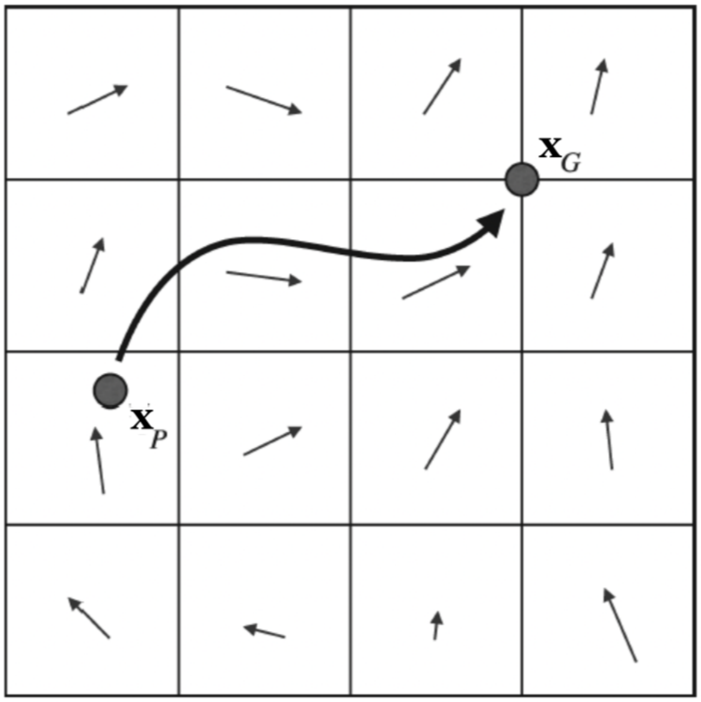


Figure 2.6: The Semi-Lagrangian Method.

In a practical illustration, Fig. 2.6 showcases to determine the value of velocity components of a particular point at position x_G in the new timestep, a hypothetical particle (hence the term "semi" in "semi-Lagrangian") is traced back one timestep using the reversed velocity field to its previous position x_P. At that location, an interpolation between the two nearest u-components and v-components are conducted to retrieve the old $\mathbf{u}$ value, which is then directly assigned to $\mathbf{x}_\mathrm{G}$. The advection algorithm for velocity in a 3D scenario is outlined in **Algorithm 2** below:

Algorithm 2: The Advection Algorithm

```
for every cell $(i, j, k)$ do
    for each velocity component $(u, v, w)$ do
        Process for the $u$-component is outlined as follows:
        Perform interpolation at $\mathbf{x_G}$ where $\mathbf{u_G}$ is stored to find the full velocity
          vector;
        Reverse the velocity vector;
        Integrate one timestep: $\mathbf{x_P} = \mathbf{x_G} + \mathbf{u_G}\Delta t$;
        At $\mathbf{x_P}$, interpolate the velocity component $\mathbf{u_P}$;
        Use the old value at the old position as the new value at the new position:
          $\mathbf{u_G} = \mathbf{u_P}$;
```

2.2.4 Boundary Conditions and Extrapolation

When performing advection, it is possible for the imaginary particle to trace back to a position outside the boundaries of the fluid, such as within a solid wall or in open air. To ensure correct velocity interpolation in these areas, a technique called velocity extrapolation is employed.

In the case where the particle ends up inside a solid, based on the rationale explained in section 2.1.5, tangential velocities within solids must be set to values that do not impede fluid movement along the solid walls. The natural choice is to mirror the velocity values inside the fluid, as illustrated in Fig. 2.7. This ensures that there is no change in tangential velocity during interpolation at the boundary, as interpolation between two identical values results in the same value.

Extrapolating velocity from the fluid to the surrounding air is more complex and won't be discussed in this work (numerous extrapolation algorithms can be found through online sources). Essentially, it involves averaging velocities starting from the fluid surface and extending outward into the surrounding air.

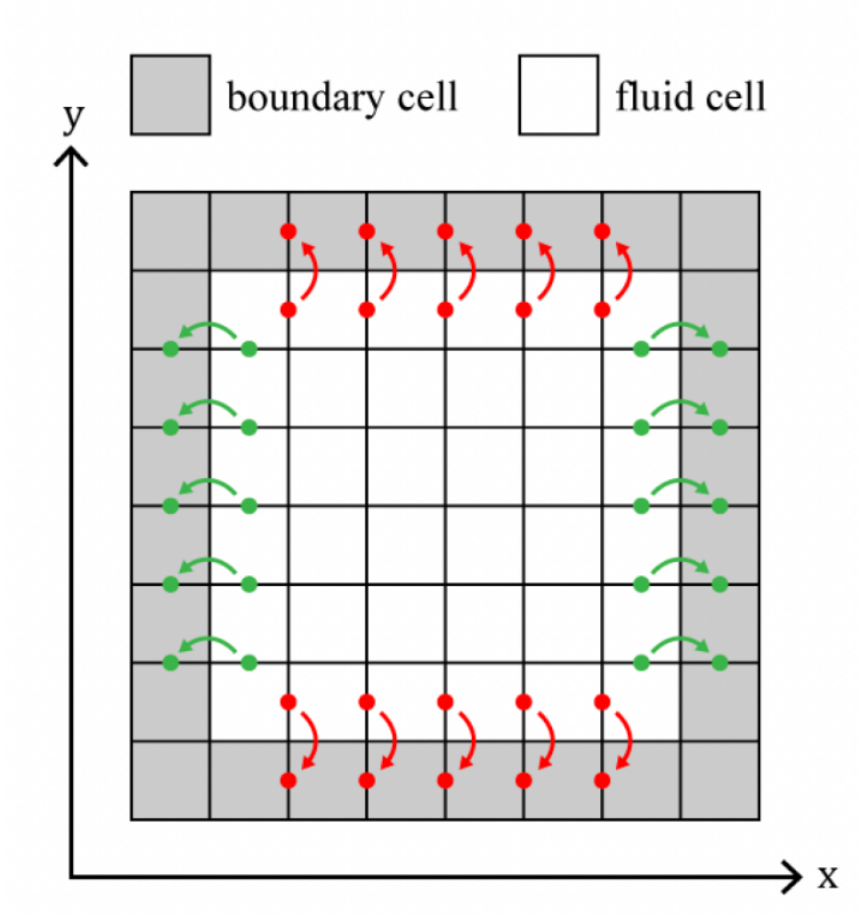


Figure 2.7: Extrapolating fluid velocities to solid walls (the no-stick condition).

2.2.5 Pressure Equation

To ensure that the velocity field is divergence-free and preserves fluid volume, an additional force must be introduced. This force acts to eliminate any divergence, enforcing incompressibility and maintaining boundary conditions.

As explained in section 2.1.1, regions of high pressure exert a force that pushes the fluid away, in the direction opposite to the negative pressure gradient. Therefore, during the velocity update at time *n+1*, according to the momentum equation, the gradient of pressure should be subtracted (or equivalently, the negative gradient of pressure can be added) from the intermediate velocity field $\mathbf{u}$ obtained from the advection step:

$$\mathbf{u}^{n+1} = \mathbf{u} - \Delta t \frac{\nabla p}{\rho} \tag{2.8}$$

the resulting velocity field fulfills the requirement of incompressibility:

$$\nabla \cdot \mathbf{u}^{n+1} = 0 \tag{2.9}$$

in addition, solid wall boundary conditions are imposed as $\mathbf{u}^{n+1} \cdot \hat{n} = 0$ and a free surface condition is enforced where $p = 0$.

The staggered arrangement of variables becomes evident in the context of subtracting a component, $\partial p/\partial x$, $\partial p/\partial y$, or $\partial p/\partial z$ of the pressure gradient ∇p from the corresponding component of velocity $\mathbf{u}$. This arrangement ensures that there are two neighboring pressure values on either side of the velocity component, allowing us to approximate equation 2.8 using central differences for ∇p as follows:

$$\begin{aligned}
u^{n+1}_{i+\frac{1}{2},j,k} &= u_{i+\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\left(\frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}\right) \\
v^{n+1}_{i,j+\frac{1}{2},k} &= v_{i,j+\frac{1}{2},k} - \frac{\Delta t}{\rho}\left(\frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x}\right) \\
w^{n+1}_{i,j,k+\frac{1}{2}} &= w_{i,j,k+\frac{1}{2}} - \frac{\Delta t}{\rho}\left(\frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x}\right)
\end{aligned} \tag{2.10}$$

however, it is not necessary to update all velocities in the grid using the pressure gradient. Air regions that do not border any fluid, for instance, do not require updating since preserving the volume of air is not a concern. Similarly, velocities at solid walls do not need to be updated with the pressure gradient as they are set directly according to the specified boundary conditions (zero for perpendicular flow and unchanged for tangential flow). The velocities that need to be updated with the pressure gradient are only those within the fluid and those located at the free surface of the fluid, which are adjacent to air cells as shown in Fig. 2.8.

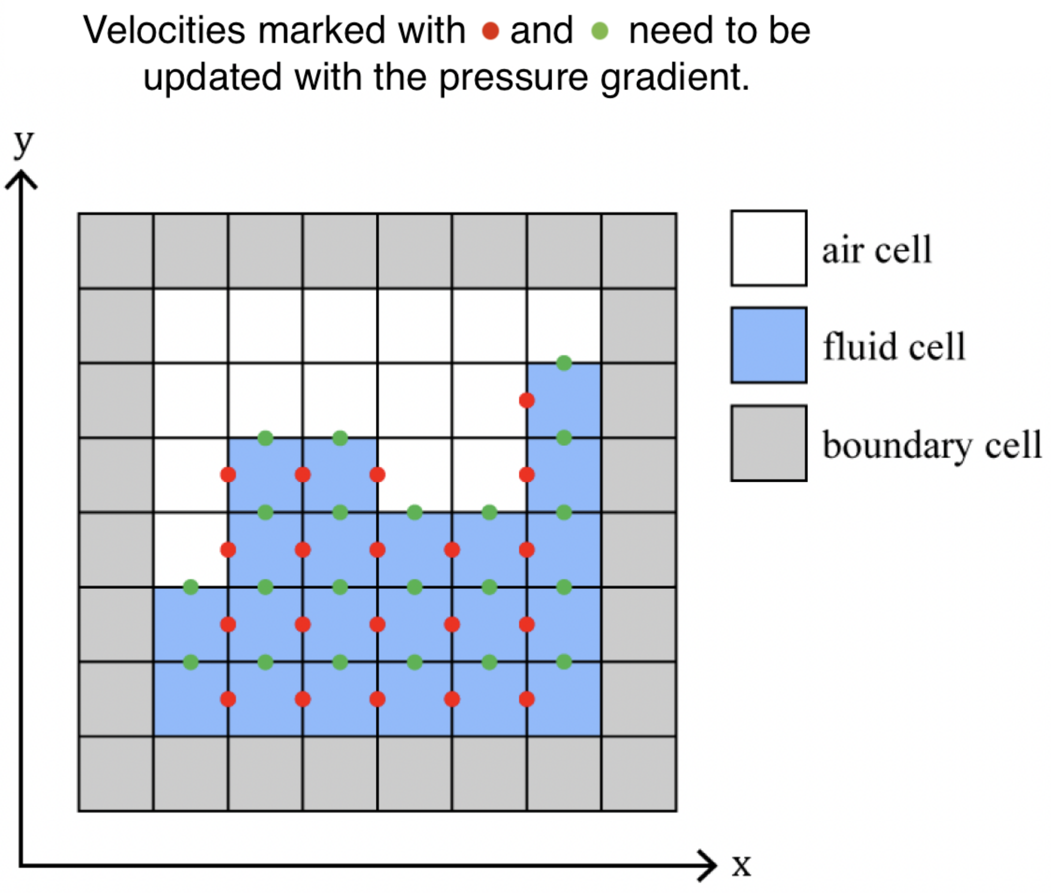

Velocities marked with ● and ● need to be updated with the pressure gradient.
y
x
air cell
fluid cell
boundary cell

Figure 2.8: Velocities affected by the pressure gradient update in voxelized domain.

Up until now, we have discussed the pressure update (equation 2.8), but we still need to ensure the satisfaction of the incompressibility condition (equation 2.9). Fortunately, because velocity is conveniently stored on the (MAC) grid, calculating the divergence becomes a straightforward procedure. The three-dimensional expression for divergence can be stated as follows:

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$

and approximating it for cell (i, j, k) using central finite differences results in:

$$(\nabla \cdot \mathbf{u})_{i,j,k} \approx \frac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{\Delta x} + \frac{v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k}}{\Delta x} + \frac{w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}}}{\Delta x}. \tag{2.11}$$

Similar to the pressure update, the calculation of the divergence is only necessary for cells identified as fluid, while it is not crucial for air or solid objects to undergo volume changes.

Although we have established how to update velocity using the pressure gradient, there is an important missing piece: the pressure itself. The pressure update only addresses velocity, without providing any immediate information about the pressure values. We already know that pressures in the air are set to a constant value of zero (**Dirichlet** boundary condition).

Additionally, within solid objects, the solid boundary condition specifies the normal derivative of pressure instead of storing an explicit pressure value (**Neumann** boundary condition); however, these boundary conditions are not a concern since velocities at fluid-solid boundaries are manually set at each timestep. Thus, the remaining challenge is to determine the pressure values inside the fluid to achieve incompressibility when updating velocity.

To solve this problem, we elaborate on two important pieces of information that we know: how the pressure updates velocity and the condition that the resulting velocity must satisfy. This combination is achieved by formulating a linear system of equations, with one equation for each fluid cell. Specifically, equation (2.10) is substituted into equation (2.11) as follows:

$$(\nabla \cdot \mathbf{u})_{i,j,k} \approx \frac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{\Delta x} + \frac{v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k}}{\Delta x} + \frac{w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}}}{\Delta x} = 0,$$

$$\begin{aligned}\frac{1}{\Delta x}\Bigg[&\left(u_{i+\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}\right) \\ &- \left(u_{i-\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\frac{p_{i,j,k} - p_{i-1,j,k}}{\Delta x}\right) \\ &+ \left(v_{i,j+\frac{1}{2},k} - \frac{\Delta t}{\rho}\frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x}\right) \\ &- \left(v_{i,j-\frac{1}{2},k} - \frac{\Delta t}{\rho}\frac{p_{i,j,k} - p_{i,j-1,k}}{\Delta x}\right) \\ &+ \left(w_{i,j,k+\frac{1}{2}} - \frac{\Delta t}{\rho}\frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x}\right) \\ &- \left(w_{i,j,k-\frac{1}{2}} - \frac{\Delta t}{\rho}\frac{p_{i,j,k} - p_{i,j,k-1}}{\Delta x}\right)\Bigg] = 0\end{aligned} \tag{2.12}$$

and by performing algebraic simplifications, we can obtain a numerical approximation to the Poisson problem $-\frac{\Delta t}{\rho}\nabla \cdot \nabla p = -\nabla \cdot \mathbf{u}$ for a fluid cell (i, j, k):

$$-\frac{\Delta t}{\rho}\begin{pmatrix} \dfrac{-6p_{i,j,k} + p_{i+1,j,k} + p_{i,j+1,k} + p_{i,j,k+1}}{\Delta x^2} \\ +\dfrac{p_{i-1,j,k} + p_{i,j-1,k} + p_{i,j,k-1}}{\Delta x^2} \end{pmatrix} = -\begin{pmatrix} \dfrac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{\Delta x} \\ +\dfrac{v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k}}{\Delta x} \\ +\dfrac{w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}}}{\Delta x} \end{pmatrix}. \tag{2.13}$$

The equation presented above (equation 2.12) represents the general formula for calculating pressure in each fluid cell of the grid; however, when a cell is located at the boundary of the fluid, there are predefined pressure values based on the boundary conditions. These known pressure values can be substituted into equation (2.12) to derive a slightly modified version of equation (2.13).

Although the resulting differences are minimal, it is important to note the adjustments without rewriting the entire equation. Specifically:

- when neighboring cells are filled with air, their corresponding pressure terms are simply set to zero, and
- when neighboring cells are solid, their corresponding pressure terms are removed, and the coefficient preceding the $p_{i,j,k}$ term is decreased by one.

2.2.6 Finding and Applying Pressure

Once equation (2.13) is derived for each fluid cell in the grid, it gives rise to a substantial system of linear equations that must be solved to determine the unknown variables, which are the pressures p. This system can be represented as a coefficient matrix, A, multiplied by a vector containing all the unknown pressures, denoted as x, which is equal to a vector consisting of the negative divergence of the velocity field for each fluid cell, represented as b:

$$Ax = b. \tag{2.14}$$

There exist numerous methods for solving linear systems, and in the following section we briefly analyze some of these methods that are dominantly used in the animation field. Once the linear system solver returns the computed pressures for each cell, the fluid velocities can be updated using equation (2.10). The resulting velocity field, denoted as $\mathbf{u}^{n+1}$, is divergence-free and satisfies the specified boundary conditions. This updated velocity field can then be utilized to advect the marker particles, serving as the final step in the entire fluid calculation process.

Before concluding this section, let's provide an outline of the steps involved in the pressure update routine $(\Delta t, \mathbf{u})$:

- calculate the negative divergence for each fluid cell, taking into account modifications at solid boundaries,
- construct the coefficient matrix A by iterating over each fluid cell and determining the types of neighboring cells (fluid, air, or solid),
- solve the linear system $Ax = b$ using a linear system solver, which provides the vector of solved pressures x, and
- compute the new velocity field $\mathbf{u}^{n+1}$ by applying the pressure gradient update formula, equation (2.10), to the existing (non-divergent) velocity field $\mathbf{u}$ and the solved pressure vector x.

By following this routine, the pressure update process ensures that the velocity field is adjusted in accordance with the pressure changes, leading to a more accurate and consistent simulation of the fluid behavior.

Chapter 3

Literature Review

Eulerian-based fluid simulation is a widely used approach for modeling and simulating fluid dynamics. It relies on dividing the simulation domain into a fixed grid or mesh, where fluid properties, such as velocity, pressure, and density, are computed at each grid cell. This approach has been foundational in the field of computer graphics and computational fluid dynamics (CFD). In this literature review, we highlight some key developments and contributions.

3.1 Comparison of Numerical Methods

3.1.1 Navier-Stokes Equations

The Navier-Stokes equation is a fundamental partial differential equation in fluid mechanics [34], employed to describe the behavior of incompressible fluids [54]. This equation represents an extension of Leonhard Euler's 18th-century equation [20], which originally described the motion of frictionless and incompressible fluids. In 1821, French engineer Claude-Louis Navier [25] introduced the crucial concept of viscosity, accounting for fluid friction and making it applicable to a broader range of real-world scenarios, particularly for viscous fluids. Subsequently, during the mid-19th century, British physicist and mathematician Sir George Gabriel Stokes further refined this framework [30], though the exact solutions were only achievable for simple two-dimensional flows. The complex dynamics involving vortices, turbulence, and chaos in three-dimensional fluid systems, including gases, as velocities increase, have remained challenging to fully analyze, with practical solutions mainly relying on numerical approximations. Euler's original equation is:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{\nabla p}{\rho} \tag{3.1}$$

where $\mathbf{u}$ is the fluid velocity vector, p is the fluid pressure, ρ is the fluid density, and ∇ indicates the gradient differential operator. The Navier-Stokes equation, on the other hand, has the form:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{\nabla p}{\rho} + \nu \nabla^2 \mathbf{u} \tag{3.2}$$

where all the terms are identical to the Euler's equation with the addition of ν (kinematic viscosity) and ∇^2 (Laplacian operator).

3.1.2 Finite Differences

Finite difference methods are one of the earliest numerical techniques used to solve partial differential equations, including the Navier-Stokes equations. In 1922, Lewis Fry Richardson published a paper titled "Weather Prediction by Numerical Process" in which he applied finite difference methods to approximate solutions to the primitive equations of atmospheric motion, a simplified form of the Navier-Stokes equations for atmospheric flow [38]. Richardson's work laid the foundation for numerical weather prediction and can be considered an early example of using finite differences for fluid dynamics simulations.

When it comes to solving large-scale fluid simulations, especially in three dimensions, directly applying finite difference methods can be computationally expensive and may not be efficient. This is where iterative solvers come into play. Iterative solvers are used to solve linear systems of equations efficiently, which is a common step in solving fluid flow problems.

Below, we present a concise overview of the key iterative methods employed for solving the linear system of equation (also called the Poisson Equation) in fluid simulation. For more comprehensive information on these methods, we highly recommend consulting Saad's book [41].

Jacobi

In the context of an $n \times n$ real matrix A and a real n-vector b, the problem at hand involves seeking a solution x within $\mathbb{R}^n$, satisfying the following:

$$Ax = b.$$

Let $Ax = b$ be a square system of n linear equations, where:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}, \quad x \;=\; \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, and \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}. \tag{3.3}$$

The formula, based on elements and applicable to each row indexed by i, can be expressed as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, ..., n$$

and as shown in the equation 3.3, in this context, we use a_{ij} to represent the element of matrix A at row i and column j, and $x_j^{(k)}$ refers to the j^{th} component of vector of unknowns p at time step k.

It's worth noting that the positive definiteness of A ensures its invertibility, with all diagonal elements a_{ii} being positive, thus validating our update rule. Additionally, every element of $x^{(k+1)}$ is reliant on known values $x_j^{(k)}$, which inherently lends Jacobi to parallelization. This characteristic makes it straightforward to implement and efficient to execute on a GPU.

However, this method tends to exhibit a slow convergence rate, often necessitating a substantial number of iterations before achieving a satisfactory solution. This side effect would bring us to the next method.

Gauss-Seidel (GS)

Let's contemplate adjusting the Jacobi update rule to incorporate the newly available values $\mathbf{x_j^{(k+1)}}$. By doing so, we arrive at the definition of the Gauss-Seidel method as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, ..., n$$

and to be more specific, when calculating the i^{th} component of $x^{(k+1)}$, we update the matrix-vector product by substituting $x_j^{(k)}$ with $x_j^{(k+1)}$ for $j < i$, while leaving everything else unchanged for $j > i$. The procedure is generally continued until the changes made during an iteration fall below a specified tolerance, such as a sufficiently small residual.

Unlike the Jacobi method, when implementing GS method, only one storage vector is required as elements can be overwritten as they are computed, which can be advantageous for very large problems. The Gauss-Seidel method often demonstrates a swifter convergence rate, albeit at the expense of relinquishing Jacobi's parallel nature, transforming into a sequential algorithm.

Conjugate Gradient (CG)

The Conjugate Gradient algorithm stands out as a highly effective iterative approach for solving linear systems with sparse symmetric positive definite matrices. In essence, this method embodies the concept of orthogonal projection onto the Krylov subspace $\mathcal{K}_m(r_0, A)$, where r_0 denotes the initial residual. Notably, when dealing with a symmetric matrix A, leveraging certain simplifications derived from the three-term Lanczos recurrence results in more elegant and efficient algorithms.

We can define two non-zero vectors $\mathbf{u}$ and $\mathbf{v}$ as conjugate with respect to A if:

$$\mathbf{u}^{\mathsf{T}} A\mathbf{v} = 0.$$

As A exhibits the properties of symmetry and positive definiteness, the expression on the left-hand side defines an inner product:

$$\mathbf{u}^{\mathsf{T}} A\mathbf{v} = \langle \mathbf{u}, \mathbf{v} \rangle_A := \langle A\mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{u}, A^{\mathsf{T}}\mathbf{v} \rangle = \langle \mathbf{u}, A\mathbf{v} \rangle.$$

Two vectors are considered conjugate if and only if they are orthogonal with respect to this inner product. The concept of conjugacy is symmetric, meaning that if $\mathbf{u}$ is conjugate to $\mathbf{v}$, then $\mathbf{v}$ is also conjugate to $\mathbf{u}$. Let's assume:

$$P = \{p_1, \ldots, p_n\}$$

is a collection of n mutually conjugate vectors concerning A, which means that $p_i^{\mathsf{T}} A p_j = 0$ for all $i \neq j$. In this scenario, P serves as a basis for $\mathbb{R}^n$, allowing us to represent the solution $\mathbf{x}_*$ of $Ax = b$ using this basis:

$$x_* = \sum_{i=1}^{n} \alpha_i p_i \Rightarrow Ax_* = \sum_{i=1}^{n} \alpha_i A p_i \tag{3.4}$$

and when we left-multiply the equation $Ax = b$ by the vector p_k^{T}, we obtain:

$$p_k^{\mathsf{T}} b = p_k^{\mathsf{T}} A x_* = \sum_{i=1}^{n} \alpha_i p_k^{\mathsf{T}} A p_i = \sum_{i=1}^{n} \alpha_i \langle p_k, p_i \rangle_A = \alpha_k \langle p_k, p_k \rangle_A \tag{3.5}$$

and so

$$\alpha_k = \frac{\langle p_k, b \rangle}{\langle p_k, p_k \rangle_A}. \tag{3.6}$$

According to the method described in [36], the approach for solving the equation $Ax = b$ involves discovering a series of n conjugate directions and subsequently determining the coefficients α_k.

Concluding Thoughts

Based on the data provided by [51] an in-depth analysis of three different methods was undertaken. The findings revealed that the Conjugate Gradient Method exhibits faster convergence and requires fewer iterations to attain its final solutions compared to the other two methods (Table 3.1). Additionally, it was observed that the Conjugate Gradient Method yields lower computational errors in the process. Consequently, the Conjugate Gradient Method is deemed to be a more reliable and accurate approach, making it the preferred choice among these methods.

Method	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$	$\mathbf{x_5}$	**Iteration**
Jacobi	7.8957	0.5406	0.4229	0.0736	0.0106	91
Gauss-Seidel	7.8945	0.5316	0.4239	0.0745	0.0107	31
Conjugate Gradient	7.8975	0.54326	0.4249	0.0737	0.07105	5

Table 3.1: Comparison of Numerical Methods.

3.1.3 Real-Time and Interactive Fluids

Over the years, the field of fluid simulation has witnessed significant advancements. In 1996, Foster and Metaxas made substantial progress in enhancing the realism of liquid simulations by addressing the limitations of prior computer graphics fluid models and introducing more complex behaviors. This marked the foundation for more realistic liquid animations [17]. In 1999, Jos Stam's "Stable Fluid" paper was a pivotal moment in animation, introducing an unconditionally stable model for fluid flow, enabling real-time interaction and faster simulations [43]. This model allowed animators to effortlessly generate intricate fluid-like behaviors. Shortly thereafter, Fedkiw et al. (2001) proposed an innovative approach to numerical smoke simulation in 2001 [16]. Their method demonstrated remarkable speed and efficiency, particularly for inviscid Euler equations.

Grid-based Eulerian methods are powerful but can face challenges in handling complex fluid behaviors. Grid-free techniques, like Smoothed Particle Hydrodynamics (SPH), offer an alternative. Enright and colleagues introduced a "particle level set method" in 2002, presenting a new perspective on achieving visually realistic water surface behaviors [15]. Their method also introduced velocity extrapolation techniques. Additionally, Bargteil et al. (2005) presented a semi-Lagrangian surface tracking method for fluid simulations in 2005, maintaining an explicit polygonal mesh and octree data structure for tracking surface characteristics [2].

In 2006, Min and Gibou introduced a second-order accurate projection method for incompressible Navier-Stokes equations, achieving stability on non-uniform adaptive Cartesian grids [31]. A year later, "Real-time Breaking Waves for Shallow Water Simulations" brought a new approach to enhancing shallow water simulations, including the effects of overturning waves [46].

Three years later, Thürey and Rüde made a substantial contribution in fluid simulation by developing an algorithm based on the lattice Boltzmann method (LBM) for free surface flow simulations [47]. Their work reduced computational time by over threefold for simulations with large volumes of fluid.

Figure 3.1: Disney's 'Moana' was a watershed moment for the studio, featuring over 900 scenes with complex ocean interactions. To meet the film's demands efficiently, Disney introduced a versatile authoring system based on a lightweight implicit ocean representation [35].

Advancements in graphics hardware and algorithms have led to real-time Eulerian fluid simulations for use in video games and interactive applications where they play a crucial role in crafting lifelike fluid animations and effects. These simulations have become integral components of gaming engines, enabling game developers to seamlessly integrate fluid dynamics into gameplay. In 2009, Nvidia introduced an efficient method for tracking fluid surfaces, employing triangle meshes [32]. This approach emphasizes computational efficiency and memory conservation, offering direct control over volume and feature preservation. It mitigates memory usage by preventing grid stretching, distinguishing itself from traditional implicit methods. Another noteworthy contribution was in 2010, when Chentanez and Müller introduced a hybrid water simulation method that integrates grid-based and particle-based approaches [8]. This innovation enables real-time simulations of large-scale water bodies while preserving intricate small-scale details. Implemented within the CUDA framework, the method leverages modern GPUs for real-time performance.

In the same year, another paper addressed PDE solvers for fluid simulations on large grids. This approach introduced an efficient numerical solver tailored for the Poisson equation, accommodating Neumann and Dirichlet boundary conditions on irregular voxelized domains [29].

Nvidia's more recent work in real-time Eulerian simulation introduced an innovative grid-based method capable of simulating fully three-dimensional large-scale scenes in real time [9]. This method facilitates real-time flood simulations, incorporating physics and rendering elements while maintaining performance.

In the realm of fluid simulation, these contributions collectively form a rich tapestry of techniques and methods that have advanced the field over the years.

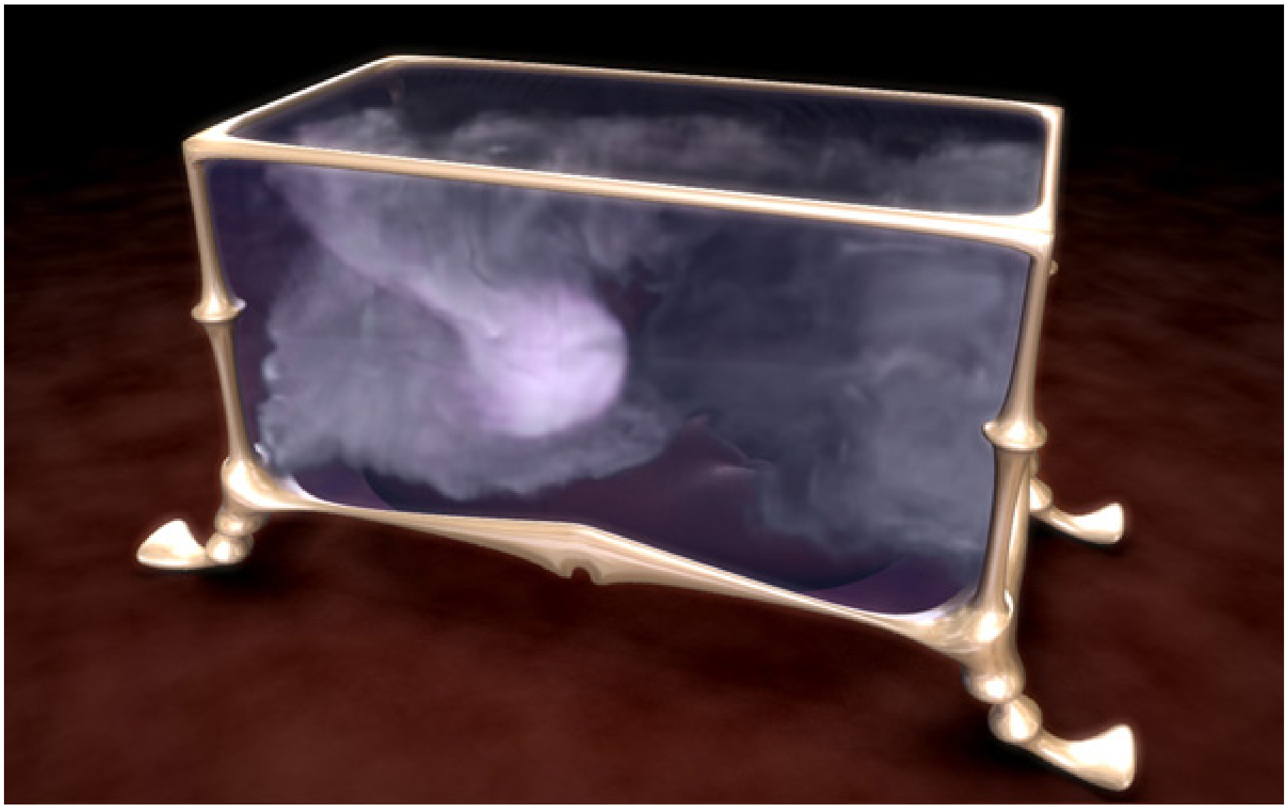


Figure 3.2: Released on December 14, 2006, this demo was originally designed for NVIDIA GeForce 8-Series Graphics Cards and is aptly named 'A Box Full of Smoke.

3.2 Machine Learning Integration

The pursuit of realism in fluid dynamics simulations has led to numerous challenges, particularly in addressing complex physical phenomena efficiently. The emergence of machine learning and deep learning techniques has revolutionized fluid simulation, offering innovative solutions to these challenges [55], [10]. In this section we investigate the integration of machine learning in fluid simulation, exploring the core concepts and applications. The review comprises four key subsections that discuss the potential of neural networks for solving partial differential equations (PDEs), the popular neural network architectures tailored for fluid simulation, the manifold advantages of employing neural networks, and finally, the existing challenges in the field.

In the interest of transparency, we will offer a high-level description of the methods and techniques utilized in the subsequent papers; the operational intricacies of these methods fall outside the scope of this thesis.

Figure 3.3: AI generated water simulation. Image by Pixabay.

3.2.1 Partial Differential Equations Solvers

Deep learning has been a prominent player in commercial applications for a considerable period, with recent notable advancements. Its growing potential is evident in scientific computing, where it demonstrates the ability to forecast solutions to complex partial differential equations (PDEs). Traditionally, these equations posed significant computational challenges due to their expensive numerical solutions. Data-driven deep learning techniques hold the promise of transforming scientific and engineering applications, encompassing fields like aerodynamics, oceanography, climatology, and reservoir modeling.

In the realm of PDE solving, Holl et al. (2020) introduce a deep learning approach that controls complex physical systems over extended periods through an end-to-end trainable system [24]. This method combines a predictor network for trajectory planning and a control network for parameter inference, leading to rapid and real-time interactions with intricate physical systems, including the incompressible Navier-Stokes equations. Similarly, Um et al. (2021) present the concept of differentiable physics networks, enhancing PDE solutions by accounting for factors unrepresented by discretized PDEs [49]. This approach is effective for a range of PDEs, including non-linear advection-diffusion systems and three-dimensional Navier-Stokes flows, ultimately improving solution accuracy and accommodating diverse physical behaviors.

In a different approach, Cai et al. (2020) introduce an unsupervised deep learning method for solving PDEs using deep neural networks and least-squares functionals as loss functions, focusing on the first-order system least-squares functional for partial differential equations in one dimension [6].

Furthermore, Cheng et al. (2021) delve into the utilization of deep neural networks to solve the 2D Poisson equation for electric field computation in plasma fluid simulations. Their findings offer insights into optimal neural network configurations and boundary conditions, paving the way for novel computational strategies to address unsteady problems involving the Poisson equation [7].

Recent developments from AWS AI Labs explore the challenges of incorporating physical constraints into machine learning models when solving PDEs: The ICML paper by Hansen et al. (2023) emphasizes the enforcement of conservation laws, resulting in improved accuracy [22]. The ICLR paper by Saad et al. (2023) highlights the enforcement of physics through boundary conditions, demonstrating significant performance enhancements in solving PDEs [40]. These advancements signify the growing synergy between deep learning and scientific computing, promising to revolutionize various industries.

3.2.2 Neural Networks' Architectures

By replacing the computationally intensive components of simulations with machine learning algorithms, data-driven models can achieve real-time simulation speeds [14,19, 26,27,50,59]. In this section, we will explore state-of-the-art neural architectures that capture fine-grained details and behaviors of fluid simulations, highlighting their potential to enhance the realism and efficiency of fluid animation [11,48].

UNet

The U-Net architecture is a deep learning model originally designed for biomedical image segmentation but has found applications in various fields, including fluid simulation. It consists of a U-shaped neural network with a contracting path to capture context and an expansive path for precise localization. The key innovation is skip connections that concatenate feature maps from the contracting path to the expanding path [39]. In fluid simulation, U-Net can be employed to predict fluid behavior, such as velocity or pressure fields, from input data like boundary conditions and obstacles. By training on simulation data, it learns to capture complex fluid dynamics, making it useful for real-time or data-driven fluid simulations, where it can replace traditional numerical solvers for certain tasks, offering speed and accuracy benefits.

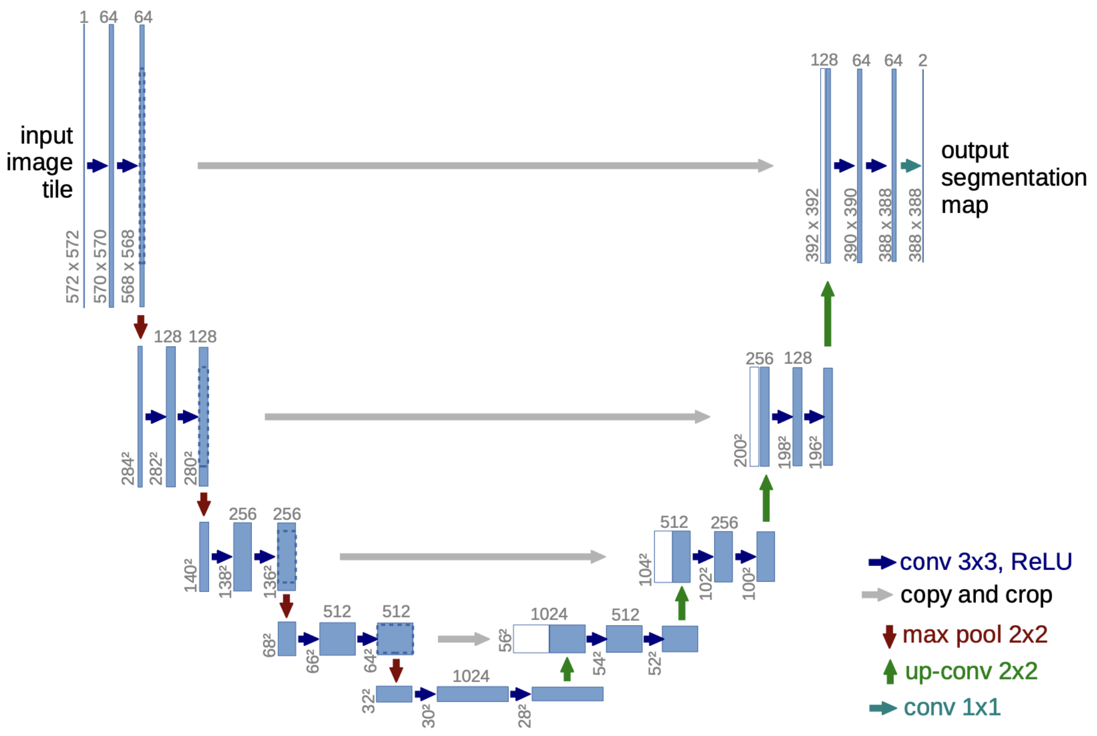


Figure 3.4: U-net architecture. Blue boxes correspond to a multi-channel feature map. The number of channels associated to each box is written above it. The arrows denote the different operations. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. Image by [39].

FluidNet

FluidNet is a convolutional neural network (CNN) architecture specifically designed and tailored to simulate the behavior of fluid flow. It captures intricate fluid dynamics at various scales. FluidNet takes as input the current state of a fluid simulation and predicts the velocity and pressure fields for the next time step. By training on large datasets of fluid simulations, the model learns to generalize fluid behavior, making it capable of producing accurate predictions [48].

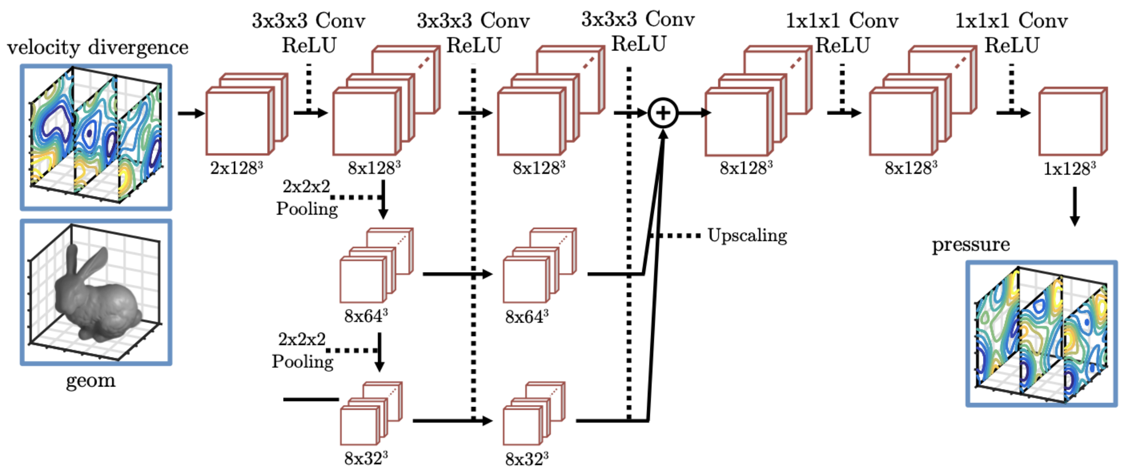


Figure 3.5: FluidNet: Convolutional network for pressure solve by Tompson et al. (2017). This model employs five stages of convolution and ReLU layers, capturing local interactions that reflect the sparsity structure of the linear system. To handle long-range physical phenomena, it incorporates multi-resolution features by downsampling the initial layer, processing each resolution in parallel, and then upsampling the results. Image by [48].

In practice, FluidNet can be used to speed up fluid simulations by replacing or augmenting traditional numerical solvers. It accomplishes this by providing quick estimates of velocity and pressure fields, making it suitable for real-time applications such as video games, simulations, and scientific visualizations, where computational efficiency is crucial while maintaining realism and detail in fluid behavior.

MSNet

MSNet (Multi-Scale Neural Network) is an extension of UNet that addresses the limitation of UNet in capturing long-range dependencies. It is a neural network architecture designed for video prediction. This model leverages a multi-scale approach to capture hierarchical information across multiple levels of abstraction in video sequences. MSNet consists of an encoder-decoder structure with skip connections, allowing it to predict fu-

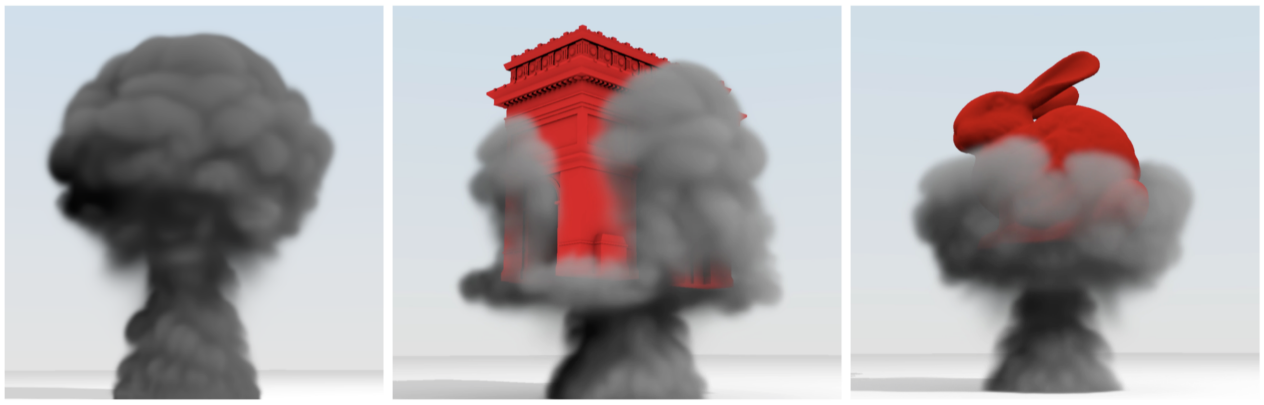


Figure 3.6: Plume simulation with neural network, by Tompson et al. [48].

ture frames at different resolutions simultaneously [28].

By training on sequences of large-scale fluid simulations, the model learns to understand the intricate dynamics of fluids, enabling it to generate accurate predictions of fluid motion, velocity fields, or pressure distributions. This approach can significantly accelerate fluid simulations, making them suitable for real-time applications like gaming, scientific visualization, and engineering simulations, while maintaining high-quality results.

GANs

The application of Generative Adversarial Networks (GANs) in fluid simulation has opened up new avenues for generating highly realistic fluid behaviors and improving visual fidelity in simulations. [18], [12].

For instance, TempoGAN is a GAN-based method that upscales fluid velocity fields in real-time, providing high-resolution fluid simulations without the need for computationally expensive grid refinements. This approach is particularly useful for interactive applications, such as computer games and virtual reality, where real-time fluid animation is essential [60].

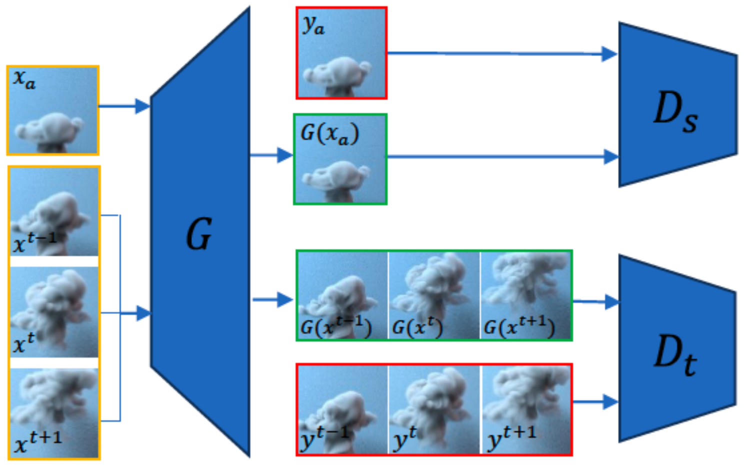
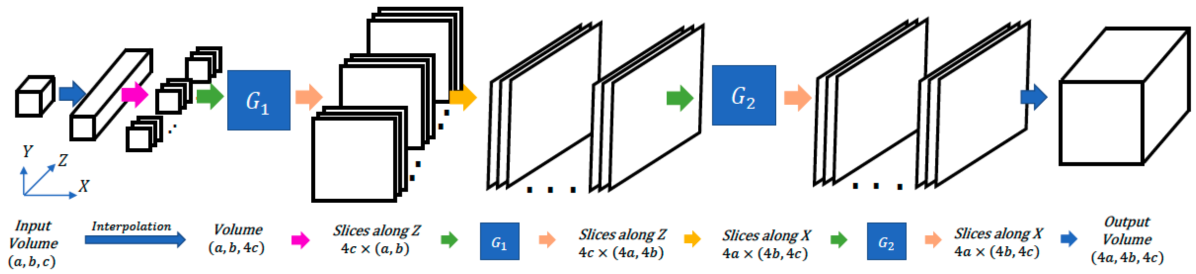


Figure 3.7: This illustration provides a broad perspective of tempoGAN's methodology: during training, a generator on the left is instructed by two discriminator networks on the right. One of these networks concentrates on spatial elements (D_s), while the other emphasizes temporal aspects (D_t); however, at runtime, both discriminator networks are disregarded, and only the generator network is put to use for evaluation. Image by [60].

In a related context, another study by Werhahn in 2019 [52] presents the concept of a "multi-pass GAN" volumetric training pipeline. This innovative approach simplifies the training process, making it possible to employ more complex networks. The paper demonstrates the effectiveness of the multi-pass GAN in enhancing the resolution of fluid flow, particularly buoyant smoke, utilizing tempoGAN and progressive GAN architectures. Additionally, the paper introduces a unique multi-pass method for training networks with time sequences of 3D volume data, combining progressive training and temporal discriminators. This novel approach enables an 8x upscaling of 3D fluid simulations through deep neural networks. Fig. 3.8 provides an illustration of the proposed multi-pass GAN pipeline.



Input Volume (a,b,c)
Interpolation
Volume $(a,b,4c)$
Slices along Z $4c \times (a,b)$
G_1
Slices along Z $4c \times (4a,4b)$
Slices along X $4a \times (4b,4c)$
G_2
Slices along X $4a \times (4b,4c)$
Output Volume $(4a,4b,4c)$
Y
Z
X

Figure 3.8: Multi-Pass GAN: Following an upsampling operation in the z-direction, the data is subjected to two distinct directions using two generator networks, G1 and G2, which have been trained adversarially. The initial upsampling step ensures that all the unknowns within the data are uniformly handled by these networks. Image by [52].

3.2.3 Advantages of Using Neural Networks

Machine learning integration in fluid simulation brings a multitude of advantages to the table, as discussed in this section. The application of neural networks offers real-time, high-fidelity simulations, overcoming limitations associated with traditional methods. These advantages include the ability to handle complex geometries, achieve visual realism, and reduce computational costs. The utilization of neural networks has redefined the landscape of fluid simulation, making it accessible for applications ranging from video games to scientific research.

Recently, several works have showcased the capabilities of neural network architectures in fluid simulation. For instance Wiewel et al. (202) proposed a deep learning approach that infers physical functions based on data, enabling neural networks to provide predictive capabilities for complex inverse problems, potentially leading to faster forward simulations [56]. Thüerey et al. (2020) demonstrated deep learning models for inferring Reynolds-Averaged Navier-Stokes solutions for airfoil flows, introducing a modernized Unet architecture and publicly available source code to facilitate research in deep learning methods for physics problems [45].

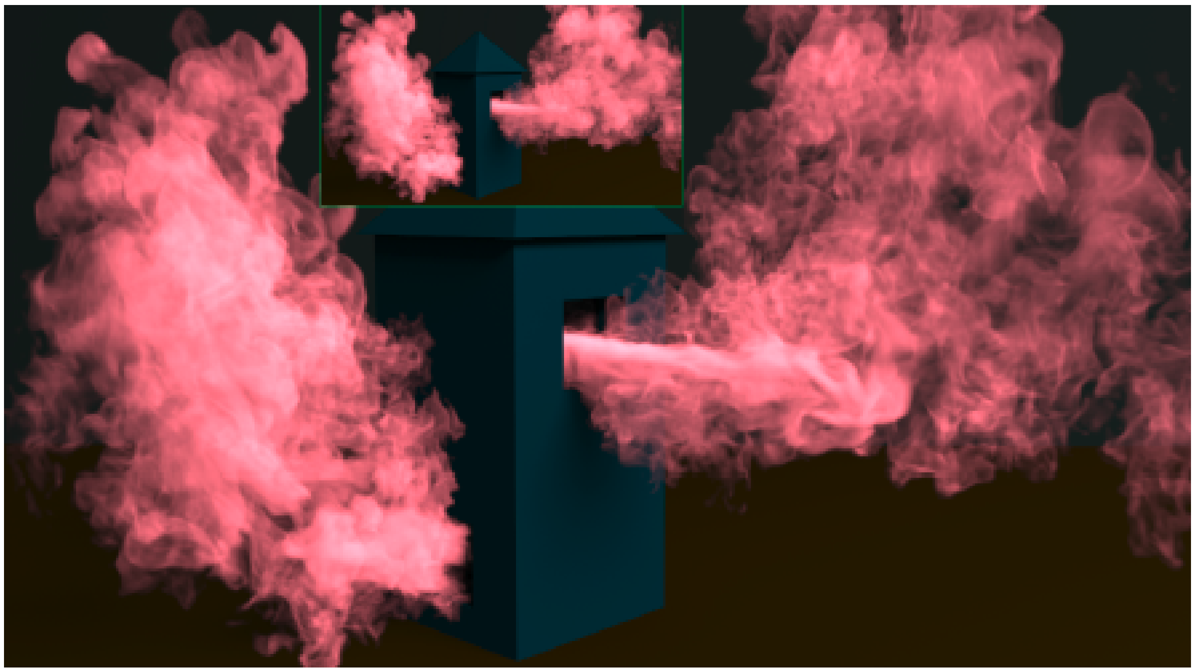


Figure 3.9: A temporally coherent generative model addressing the superresolution problem for fluid flows. Image by [60].

Another noteworthy example is Smart-fluidnet, as proposed by Dong et al. (2019), which automates model generation and application, providing significant speedup and improved simulation quality compared to state-of-the-art models [13]. Rao et al. (2020) leveraged physics-informed deep learning to reconstruct dynamic fluid phenomena from sparse multiview RGB videos, handling both synthetic and real scenes flexibly [37].

Other works, such as [48], showcased how data-driven approaches utilizing convolutional networks can efficiently solve the incompressible Euler equations and obtain highly realistic fluid simulations. [63] presented a hierarchical region proposal network for natural disaster damage assessment in aerial videos, while [39] proposed a network and training strategy for biomedical image segmentation using convolutional networks.

These works collectively demonstrate the remarkable potential of neural network architectures in fluid animation, as they continue to revolutionize the field with their ability to learn from data, optimize complex problems, and offer more efficient and accurate solutions.

3.2.4 Challenges and Limitations

Neural networks hold tremendous promise, yet they are accompanied by inherent limitations. These challenges encompass the quality of training data, the ability to generalize across diverse scenarios, and the risk of overfitting. In certain cases, neural network models lack interpretability, raising concerns about the physical consistency of their predictions.

For instance, a fundamental issue arises as deep learning models trained on physical data can often produce predictions that ignore essential physical principles. These models may violate conservation laws, resulting in solutions to heat transfer and fluid flow problems that fail to maintain energy or mass conservation. They may also disregard boundary conditions, allowing heat transfer through insulating boundaries, even in the absence of such violations in the training data. These inaccuracies can occur because the models illegitimately extrapolate from patterns during inference.

Future research directions are poised to address these limitations. One avenue involves the development of hybrid methods that blend conventional numerical techniques with neural networks, leveraging the strengths of both approaches. This integration holds the potential to yield more accurate and adaptable predictions. Another important area for exploration is to enhance the interpretability and ensure the physical consistency of neural network predictions. This means advancing our capacity to comprehend and trust the decisions made by these models, particularly in fields where the precision of predictions has substantial real-world implications. Further elaboration and exploration of these research directions fall outside the scope of this thesis.

Chapter 4

Implementation

In this chapter, we will delve into two key aspects of our implementation. The first part of the chapter will focus on Software Architecture and Coding Analysis, where we will elaborate on the underlying structure and codebase of our software system. This examination is crucial for understanding how our solution is engineered and functions. The second part of the chapter will center around data creation and model design. We will discuss the processes involved in generating the datasets used for training and evaluation, as well as the design and architecture of the deep learning model employed in our research.

4.1 Software Architecture and Coding Analysis

Here, we aim to provide a comprehensive examination of the core structure of our software system. This involves a detailed exploration of the software's architecture, which encompasses the arrangement of components, modules, and their interactions. Moreover, we will delve into the coding aspects of our solution. This involves a thorough code review, where we scrutinize the implementation details, coding standards, and practices followed during the software development process.

4.1.1 Method of Solution

In this implementation, we consider a fluid in a 2 dimensional setting that has zero viscosity and is incompressible. Given that the velocity and pressure are known for some initial time $t = 0$, such fluid can be modeled by the Euler equations as previously discussed in section 2.1:

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla \mathbf{u} - \frac{1}{\rho} \nabla p + \mathbf{f}, \tag{4.1}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{4.2}$$

In the equation 4.1 above, $\mathbf{f}$ is the summation of external forces applied to the fluid body such as gravity and buoyancy. The boundary condition type we consider here is the fixed boundary condition when the fluid lies in some bounded domain *D*.

We numerically solve all the spatial partial derivatives using finite difference (FD) methods on MAC grid [23]. As discussed in section 2.2.2, MAC grid representation samples velocity components on the face of the voxel cells, and the scalar quantities (e.g. pressure or density) at the voxel center. For simplicity, we set density (ρ) to a constant value of one. This representation resolves the non-trivial nullspace of central differencing on the standard uniformly sampled grid and it simplifies boundary condition handling, since solid-cell boundaries are collocated with the velocity samples [5].

To solve equation 4.1 we use the splitting method that involves adding external force, advection update, and pressure projection step. The process proceeds through three units of time, each with a duration of Δt. We initiate this sequence by utilizing the solution $\mathbf{u_0}(x)$, which corresponds to $\mathbf{u}(x, t)$ from the previous time step. We then sequentially address each term present on the right side of equation 4.1, followed by a projection onto fields that are divergence-free. The overall procedure is illustrated in the figure below:

$$\begin{tikzcd} \mathbf{u}_0(\mathbf{x}) \arrow[r, "\text{add force}"] & \mathbf{u}_1(\mathbf{x}) \arrow[r, "\text{advect}"] & \mathbf{u}_2(\mathbf{x}) \arrow[r, "\text{project}"] & \mathbf{u}_3(\mathbf{x}) \end{tikzcd}$$

Figure 4.1: Sequence of fluid animating operations.

The solution at time $t + \Delta t$ is subsequently determined based on the most recent velocity field: $\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u_3}(\mathbf{x})$. Our simulation evolves through a series of iterations involving these steps. An outline of the algorithm for updating velocity at each time step is presented in **Algorithm 3**, inspired by the work on accelerating Eulerian fluid simulation by Tompson and colleagues [48].

Algorithm 3: Euler Equation Velocity Update Algorithm

```
Step 1: Advection and force update to calculate $\mathbf{u_t^*}$;
    Step 1.1: Add external forces $\mathbf{f_{body}}$;
    Step 1.2: Self-advect velocity field $\mathbf{u_{t-1}}$;
    Step 1.3: Set normal component of solid-cell velocities;
Step 2: Pressure projection to calculate $\mathbf{u_t}$;
    Step 2.1: Solve the Poisson equation: $\nabla^2 p_t = \frac{1}{\Delta t}\nabla \cdot \mathbf{u_t^*}$ to find $p_t$;
    Step 2.2: Apply velocity update: $\mathbf{u_t} = \mathbf{u_{t-1}} - \frac{1}{\rho}\nabla p_t$;
```

At a macroscopic level, the algorithm depicted in **Algorithm 3** deliberately excludes the pressure term ($-\nabla p$) from equation 4.1. This omission leads to the generation of an advected velocity field, denoted as u_t^*, which contains undesired divergence quantity. At step 2, the algorithm resolves for the pressure, p, ensuring compliance with the incompressibility constraint outlined in equation 4.2. This results in the emergence of a velocity field, u_t, which is indeed divergence-free.

To solve the advection component in step 1.2, we employ a semi-Lagrangian advection procedure inspired by the Maccormack method [42]. In this approach, we trace each point of the field back in time. Consequently, the new velocity at a given position, $\mathbf{x}$, is determined as the velocity the particle possessed at its former location a time interval of Δt ago.

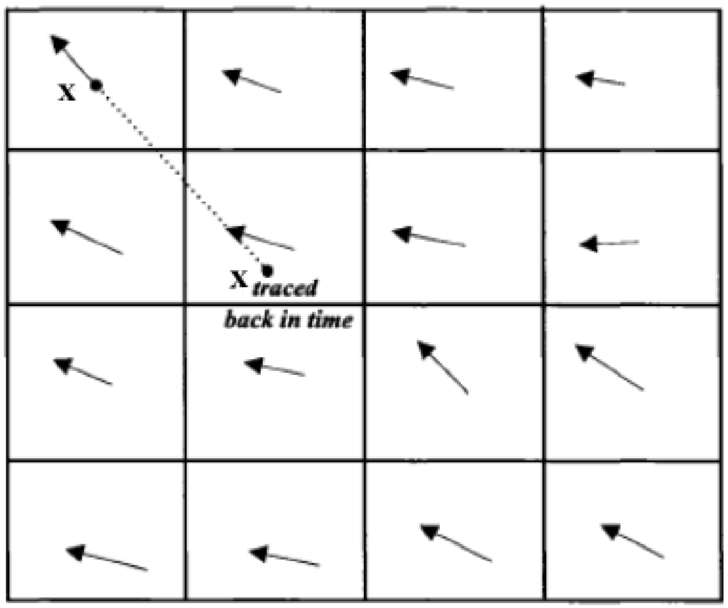


Figure 4.2: Semi-Lagrangian advection scheme.

In **Algorithm 3**, step 2.1 is by far the most computationally demanding component. It involves solving the following Poisson equation:

$$\nabla^2 p_t = \frac{1}{\Delta t}\nabla \cdot \mathbf{u}_t^*, \tag{4.3}$$

the equation described above gives rise to a large sparse system of linear equations, denoted as $Ap_t = b$, where matrix A encompasses the coefficients of pressure fields, b represents the right-hand side of equation 4.3, and x signifies the pressure field, the solution to this equation.

While matrix A exhibits the properties of symmetry and positive semi-definiteness, the associated linear system often has a significant quantity of free parameters. Consequently, employing conventional iterative solvers requires a substantial number of iterations to achieve a sufficiently low residual. Since the required iteration count is strongly data-dependent, our implementation opts to replace the precise analytical solver in step 2.1 with a learned alternative.

At this stage, we aim to harness the capabilities of deep neural networks for the purpose of predicting the pressure values derived from the Poisson equation. Specifically, we design a multilayer perceptron (MLP) model with an activation function to estimate p_t. The intricacies of this model will be comprehensively discussed in the following sections.

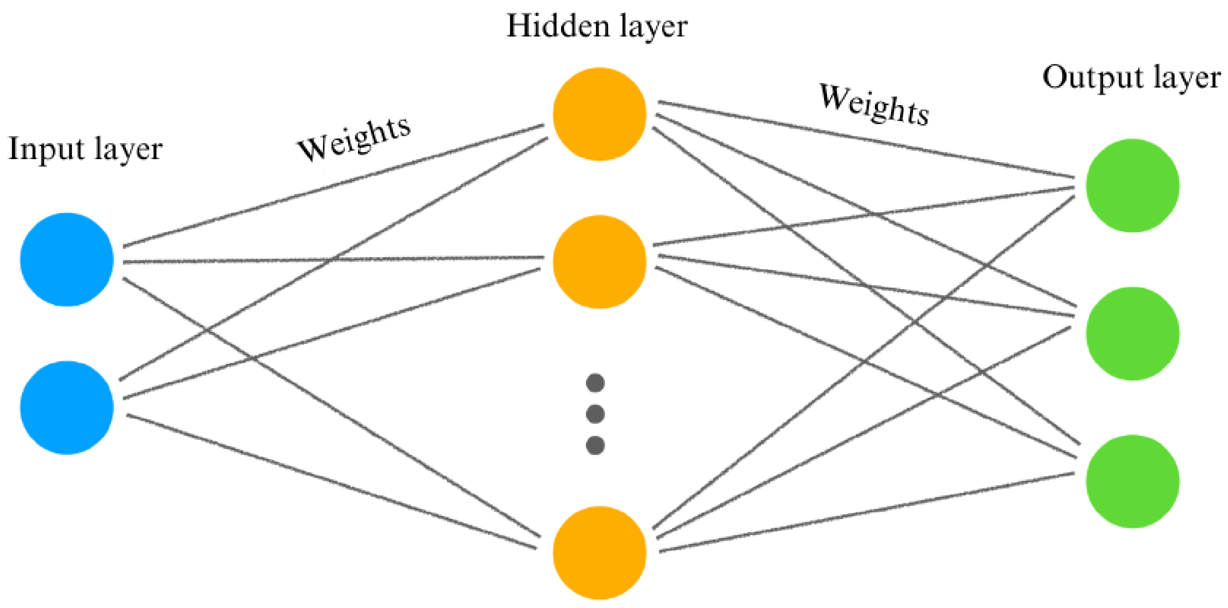


Figure 4.3: A visual representation of a multilayer perceptron model.

Following the pressure solution, we employ step 2.2 to determine the divergence-free velocity field, denoted as u_t. In order to fulfill the slip-condition boundaries (Fig. 4.4) at the interfaces between the fluid and solid cells, we establish the velocity of MAC (Marker-and-Cell) cells in a manner that ensures the component aligned with the normal of the boundary face is identical to the normal component of the object's velocity (i.e., $\hat{n} \cdot u_{fluid} = \hat{n} \cdot u_{solid}$). The representation provided by the MAC grid simplifies this process, given that each solid cell boundary aligns with the sampling location of the velocity grid.

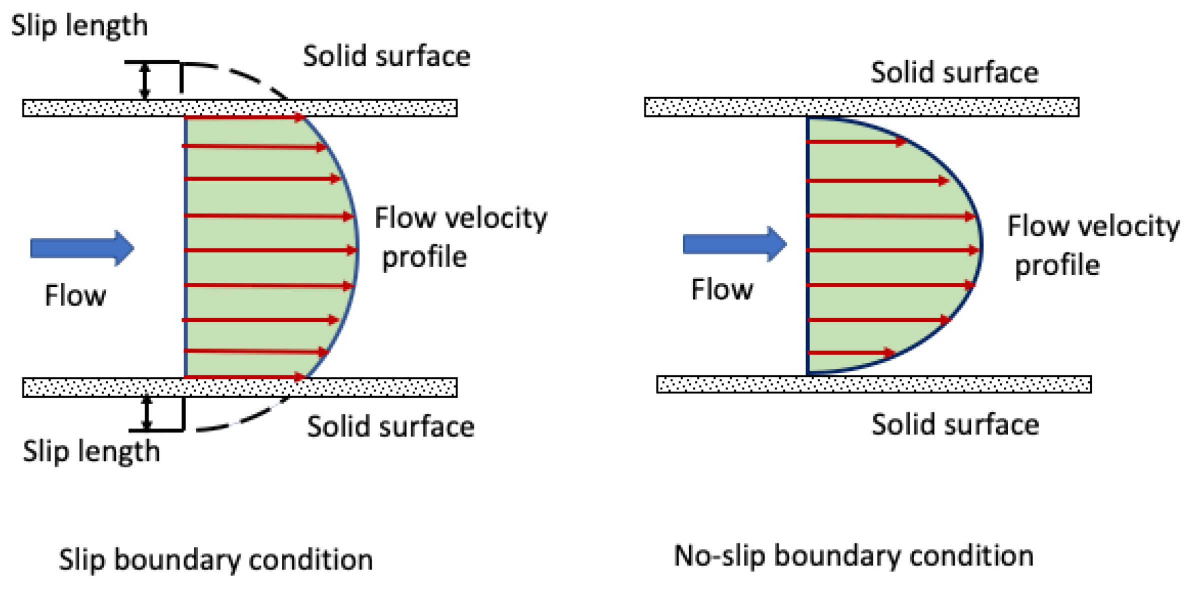


Figure 4.4: Slip and no-slip boundary conditions: in no-slip boundary conditions, the speed of the fluid at the wall is zero, whereas in slip boundary conditions there is relative movement between the wall and the fluid. Image by [62].

4.1.2 Classes and Functions

The code for this Master's thesis was written in the Pytorch framework on macOS 13.4.1 Ventura operating system. The project was written and tested on a laptop with an Apple M2 Pro chip and 16 GB RAM.

Below you can find the classes and functions used in the code, along with their descriptions and correspondence to the fluid simulation theory presented earlier in this document.

Classes

- `FluidNet`: This class defines the architecture of the neural network called FluidNet. It inherits from the nn.Module class, which is a base class for all PyTorch neural network modules. The `FluidNet` class contains the neural network layers and defines the forward pass through the network.
- `FluidDataset`: This is a custom dataset class. It inherits from torch.utils.data.Dataset and is responsible for loading the input and target data for training or validation. It overrides the `'__init__'`, `'__len__'`, and `'__getitem__'` methods.
- `FluidTestDataset`: This class is a custom dataset class, similar to the `FluidDataset` class mentioned above. It inherits from torch.utils.data.Dataset and is responsible for loading the test input and target data for evaluation. It overrides the `'__init__'`, `'__len__'`, and `'__getitem__'` methods.

Functions

- `test_model`: This function is defined to evaluate the model on the test dataset. It calculates the test loss, mean squared error (MSE), and predictions.
- `calculate_mse`: This function is defined to calculate the mean squared error.

- `forcing_function(time, point)`: This function defines the forcing function that applies forces to specific points in the domain. The function calculates the forced value based on time and point location.

- `partial_derivative_x(field)`: This function calculates the partial derivative of a field with respect to the x-coordinate using central differences.

- `partial_derivative_y(field)`: This function calculates the partial derivative of a field with respect to the y-coordinate using central differences.

- `laplace(field)`: This function calculates the discrete Laplacian of a field using central differences.

- `divergence(vector_field)`: This function calculates the divergence of a vector field by taking the sum of its partial derivatives with respect to x and y coordinates.

- `gradient(field)`: This function calculates the gradient of a scalar field by computing its partial derivatives with respect to x and y coordinates.

- `advect(field, vector_field)`: This function performs the advection of a field using backward tracing and interpolation to calculate the new field values.

- `poisson_operator(field_flattened)`: This function applies the discrete Poisson operator to a flattened scalar field.

- Utility Functions: There are some utility functions like `count_parameters` (which counts the number of trainable parameters in a model) and `main()` (which is the entry point of the script and where the main training loop is implemented).

- Visualization: The code also includes plotting and visualization of training and validation loss over epochs using Matplotlib.

4.2 Dataset Creation and Model Design

In this section we discuss the various steps taken to ensure data integrity, including data collection, preprocessing, and data formatting to suit the requirements of the model.

4.2.1 Generating Dataset

In the process of constructing our dataset, we have employed the conjugate gradient method to solve the Poisson equation, as outlined in Section 4.1.1 of this study. In this endeavor, we have utilized the independent variable of the equation, which encapsulates the divergence of the velocity field. Subsequently, we have utilized the final solution of the equation, represented as the corresponding pressure, as our output data. Both the input and output data components manifest as scalar fields, and they assume a structured format, specifically a shape of `(num_samples, num_points, num_points)`. In this context, `num_points` signifies the spatial coordinates of individual points within the grid space, while `num_samples` alludes to the overall count of such points.

Our dataset creation process has led to the formulation of three distinct datasets, each thoroughly curated for the purpose of training, validation, and testing. Each dataset is inherently accompanied by its own pair of input and output data.

Providing a more detailed perspective on each dataset:

1. Training Set: This dataset exhibits a shape of (40080, 51, 51), effectively comprising 668 distinct simulations, each encompassing 600 sequential time steps. Notably, every simulation is distinguished by its unique initial force, characterized by both magnitude and direction.

2. Validation Set: With dimensions of (11005, 51, 51), this dataset encompasses a diverse array of simulations. It includes 25 simulations, each spanning 120 time steps, along with 30 simulations covering 100 time steps, 26 simulations spanning 115 time

steps, and finally, 27 simulations involving 110 time steps. It's worth emphasizing that each simulation within this set has a distinct initial force, varying in terms of both location and direction.

3. Testing Set: Concluding our dataset ensemble, the testing set exhibits dimensions of (18120, 51, 51). It consists of 302 simulations, each characterized by 60 time steps. Notably, the variations across these simulations depends on the initial force's magnitude and its associated location.

In summary, our dataset creation process is underpinned by the utilization of the conjugate gradient method for solving the Poisson equation, and we have meticulously compiled separate datasets for training, validation, and testing. These datasets showcase distinctive characteristics in terms of simulation count, time steps, initial force configurations, and associated spatial coordinates.

4.2.2 Design and Training of Model

Given the nature of our dataset and problem set, we decided to create a multi-layer perceptron (MLP) architecture, as our initial attempt. This feedforward neural network model, named FluidNet, has multiple linear layers and activation functions, and is constructed using the PyTorch library.

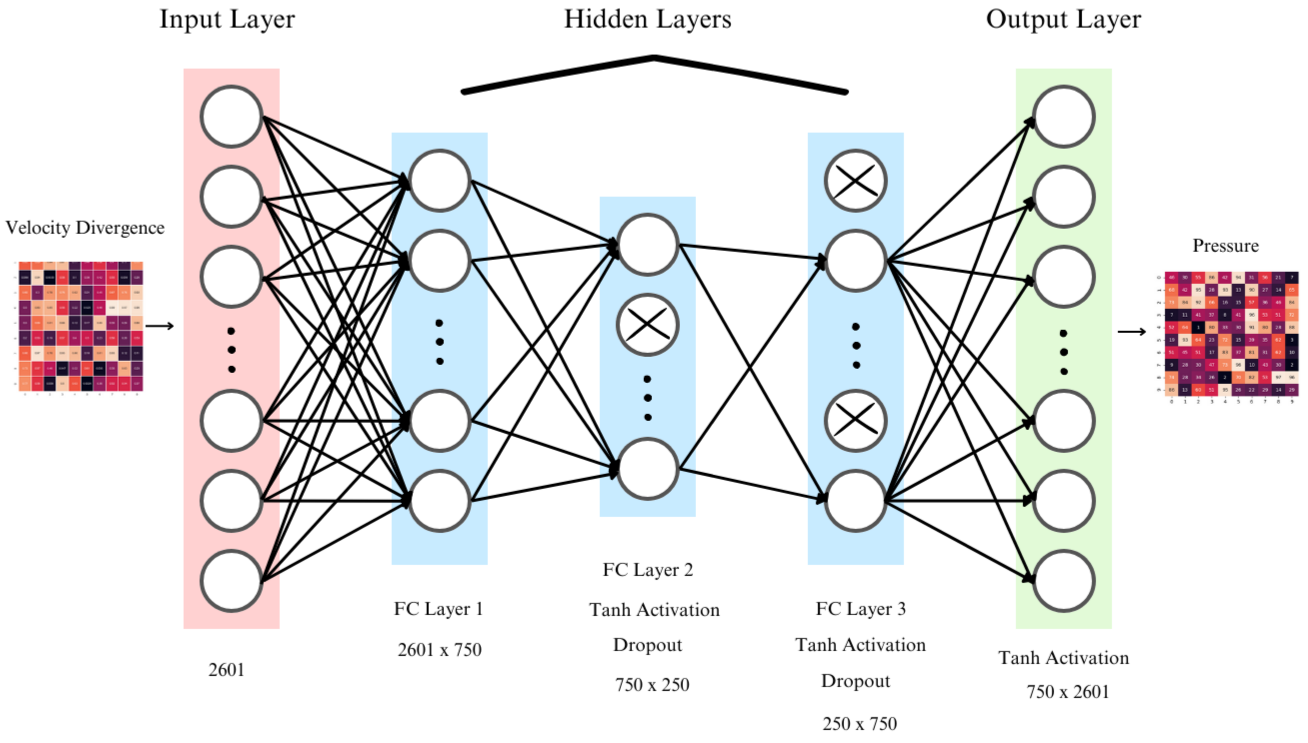


Figure 4.5: Our convolutional network for pressure solve.

The architecture consists of several key components. The input layer receives flattened input data representing features of the fluid simulation. This data is then passed through three hidden layers, two of which comprising a linear transformation followed by a hyperbolic tangent (Tanh) activation function. These layers introduce non-linearity and help the network learn complex patterns within the input data. Additionally, a dropout layer is applied to the outputs of the two last hidden layers, preventing overfitting by randomly deactivating certain neurons during training. A comprehensive depiction of the neural network, including detailed information, is illustrated in Figure 4.5.

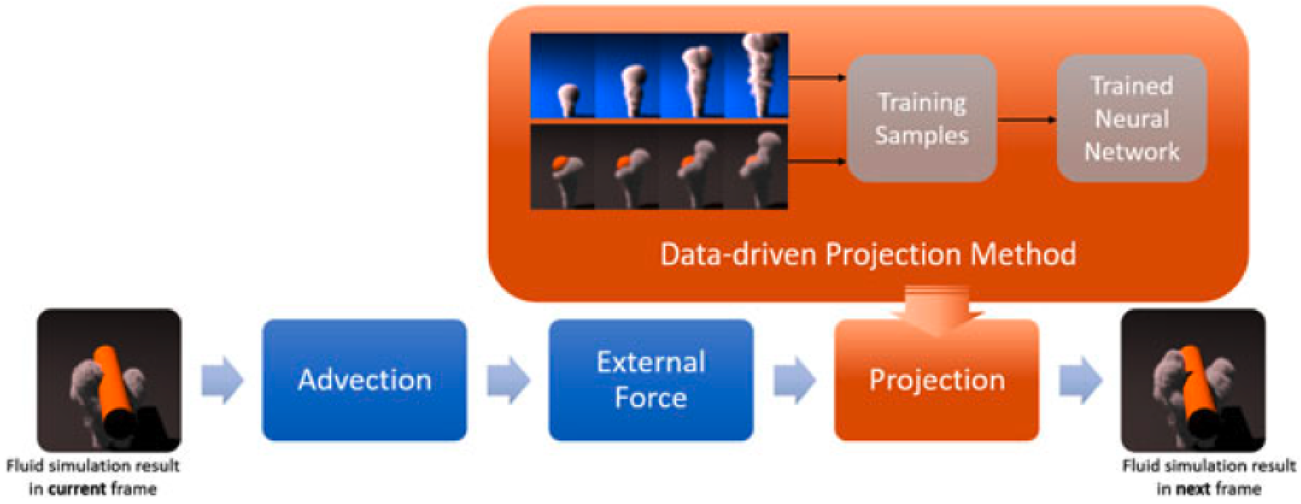

Training Samples
Trained Neural Network
Data-driven Projection Method
Advection
External Force
Projection
Fluid simulation result in **current** frame
Fluid simulation result in **next** frame

Figure 4.6: Here, our approach closely aligns with the methodology employed by Yang et al. (2016) Specifically, our data-driven projection method seamlessly integrates into the established framework of traditional grid-based fluid simulations. By using a basic dataset for training, our data-driven projection method significantly speeds up solving the Poisson equation compared to traditional methods. Image by [61].

The architecture's output layer produces predictions that correspond to the solution of the fluid simulation in our supervised learning approach. The entire network is designed to capture the relationship between the given input data and the corresponding fluid pressure output. Weight initialization for the linear layers is performed using a normal distribution, ensuring a proper starting point for training. The neural network's design allows it to learn and represent the dynamics of fluid behavior. Through a sequence of linear transformations and activation functions, our FluidNet architecture effectively translates input data into accurate predictions of fluid pressure, making it well-suited for our specific fluid simulation tasks. We will elaborate more on the performance of our neural network in chapter 5.

Moving on to the training process of our neural network model, we divide the code into several sections, each serving a specific purpose.

Training Setup

1. Data Preparation: The code starts by setting up various parameters such as device choice (GPU or CPU), the number of training epochs, batch size, and dimensions of the input grid (N_POINTS). The input data, represented as input_train and target_train, are loaded from .npy files. These files contain simulation data in the form of independent variable inputs and corresponding output data, which are both scalar fields with dimensions (num_samples, num_points, num_points).

2. Preprocessing: The loaded data is reshaped to a (num_samples, num_features) format to prepare it for model input. The input and target data undergo min-max scaling normalization to ensure consistency in the range of values. This helps stabilize the training process and improve convergence.

3. Dataset Creation and Loading: A custom `FluidDataset` class is defined, which inherits from PyTorch's Dataset class. This class is responsible for encapsulating the input and target data, facilitating easier loading during training. Training and validation datasets are created using instances of this class and loaded into PyTorch DataLoader objects. This helps manage data loading efficiently, applying batch processing and shuffling during training.

4. Model Definition and Training: The FluidNet model architecture is created. The architecture facilitates learning the complex relationships between input data and the corresponding fluid pressure output. The neural network is then trained using a mean squared error loss function (MSELoss) and stochastic gradient descent (SGD) optimizer. The training loop iterates through epochs, backpropagating errors and updating the model's parameters.

5. Training Progress Visualization: During training, the code logs and prints the training and validation losses for each epoch, providing insight into the model's performance. These loss values are stored and plotted against the number of epochs, allowing for easy visualization of the training process. The log scale on the y-axis is used to better visualize changes in loss values over time.

6. Saving the Model and Loss Plot: After training, the trained model's state dictionary is saved to a .pth file. Additionally, a loss plot is generated and saved as an image file, aiding in analyzing the model's learning progress.

Testing Setup

The final code segment involves the testing and evaluation of our trained neural network model. It starts by preparing the testing input data, reshaping it into a suitable format, and performing min-max scaling normalization. The same preprocessing steps are applied to the testing output data. Subsequently, a custom dataset class named `FluidTestDataset` is defined for organizing the testing data, and a DataLoader is created for batching and loading the data efficiently.

The code then defines utility functions to calculate the Mean Squared Error (MSE) and test the model's performance on the testing dataset. The trained FluidNet model is loaded, and the loss function (MSE) is defined using PyTorch's nn.MSELoss. The model is tested on the testing data using the test_model function, which calculates the test loss, computes the MSE, and obtains predictions for pressure values.

At the end, we visualize the predicted and calculated pressure values over time for a specific point (x, y) in the fluid simulation grid. We plot the pressure values using time as the x-axis and pressure as the y-axis, displaying both predicted and calculated pressure curves. This visualization aids in assessing the model's performance in predicting fluid pressure dynamics. The resulting plot, as well as a 2D simulation comparison, is saved as an image file for further analysis.

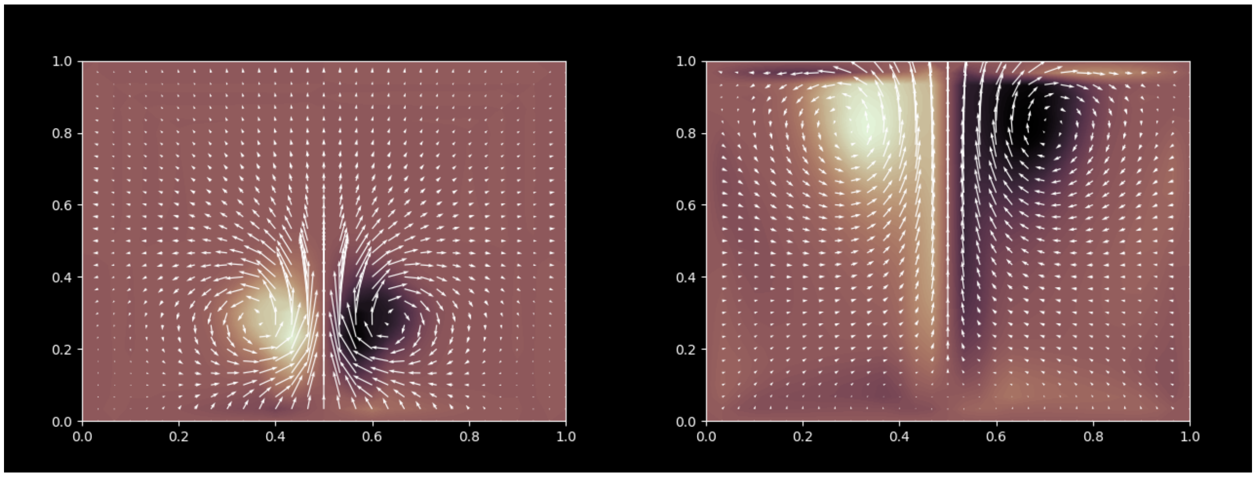
Chapter 5

Results and Analysis

We employ GPU-accelerated solvers, including conjugate gradient and neural network solvers, in Python using NVIDIA's CUDA (version 11.7). These computations utilize the Scipy and Torch libraries. All experimentation took place on an Apple M2 Pro CPU equipped with 16 GB of RAM, while the external (Mila) cluster features an NVIDIA UNIX x86_64 Kernel Module 535.86.10.

5.1 Test Scene and Parameters

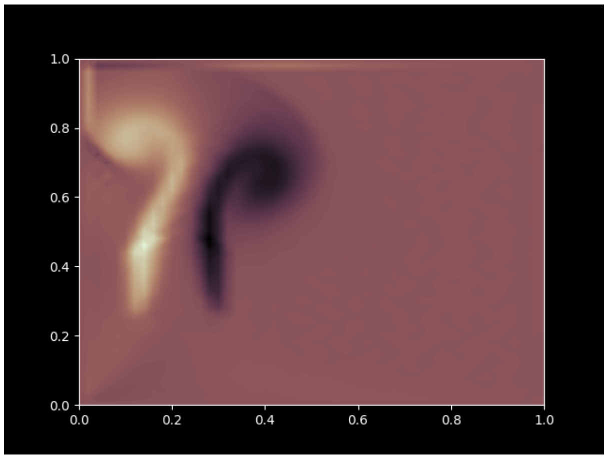
In this study, we designate the outcome obtained from the conjugate gradient (CG) solver as the ground truth. For the visual representation of the outputs, we employ a color-coded depiction of the motion of plumes, accompanied by their velocity field path. To enhance the assessment of the Neural solver's performance relative to the ground truth, we distinguish the color-coded representation from the vector field path. We then examine the results at three distinct stages, spanning from the initiation to the conclusion of the simulation.



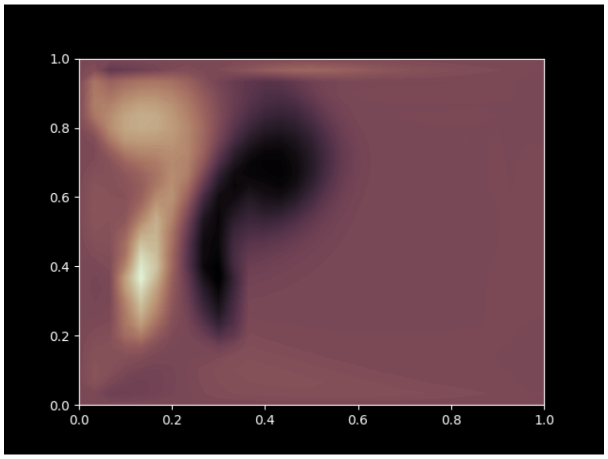
(a) Flow motion at time step 10. **(b)** Flow motion at time step 50.

Figure 5.1: Ground truth: sample fluid simulation with velocity field.

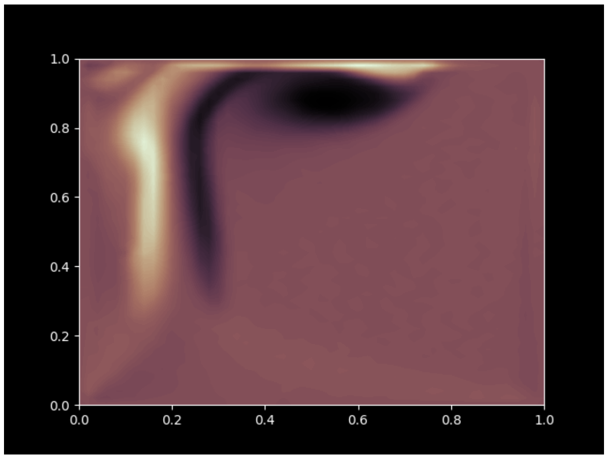
Fig. 5.2 illustrates the testing scenario we devised to assess the performance and convergence of our neural network solver in comparison to this ground truth. The visually striking scene portrays the emission of two neighboring plumes, one in gold and the other in purple, while the grayscale rendition of the scenes in Fig. 5.3 depicts the trajectory of the plumes' velocity field within the frame. Our experiments maintain fixed boundary conditions and an initial buoyancy force, without the inclusion of vorticity confinement or gravity. To ensure the consistency and persuasiveness of our findings, we maintained identical test environment parameters across all experiments, aligning them with those of the ground truth.



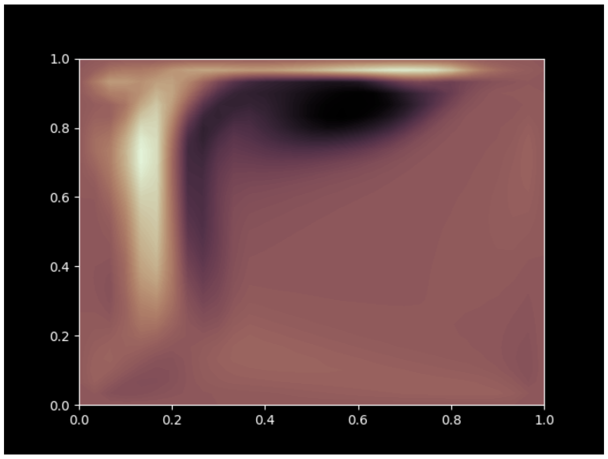
(a) Neural solver at time step 20.



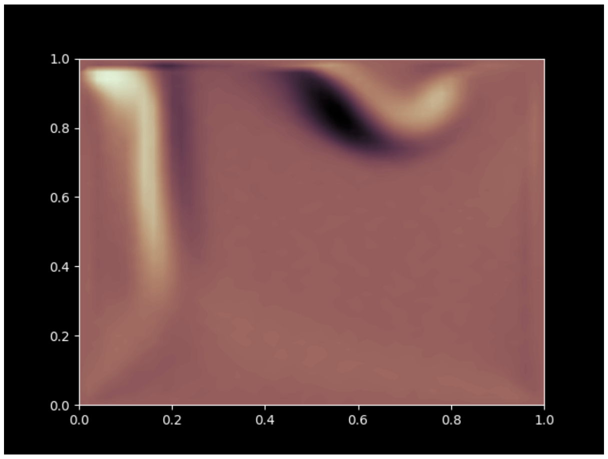
(b) Ground truth at time step 20.



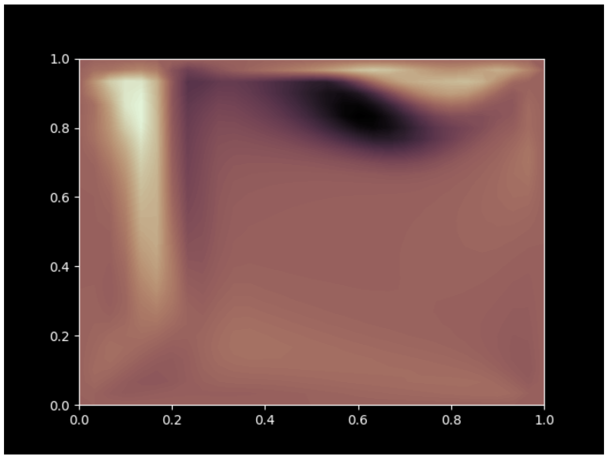
(c) Neural solver at time step 40.



(d) Ground truth at time step 40.

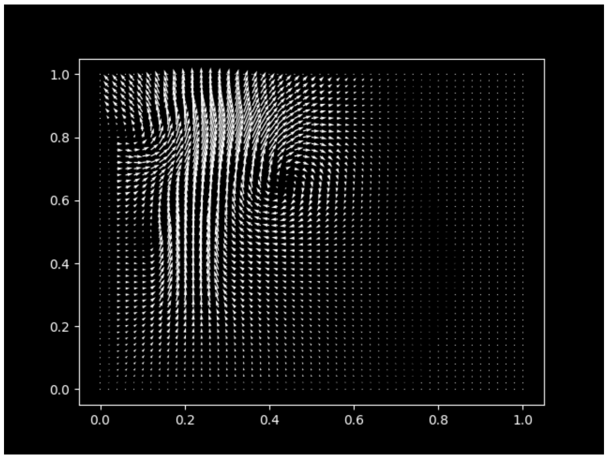


(e) Neural solver at time step 60.

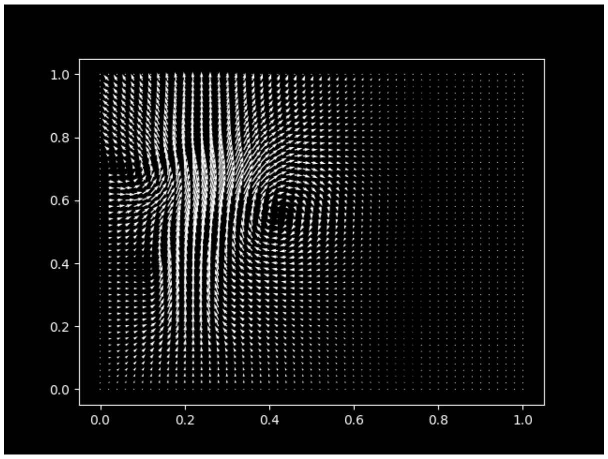


(f) Ground truth at time step 60.

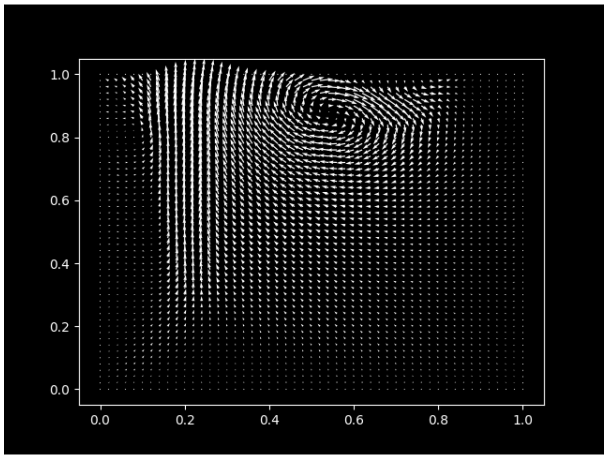
Figure 5.2: Contour representation of neural solver (left) vs ground truth (right): Fluid motion for a total of 60 time steps with a step size of 0.1 seconds. There are a total of 51x51 cell points containing the properties of the fluid.



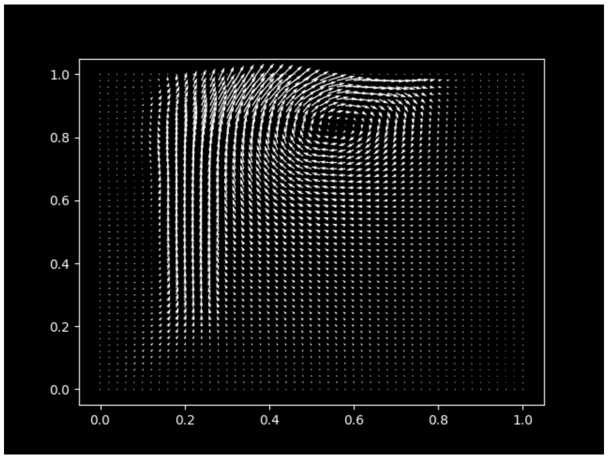
(a) Neural solver at time step 20.



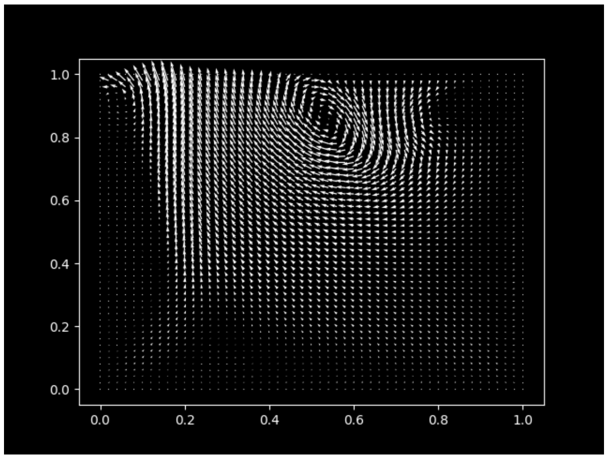
(b) Ground truth at time step 20.



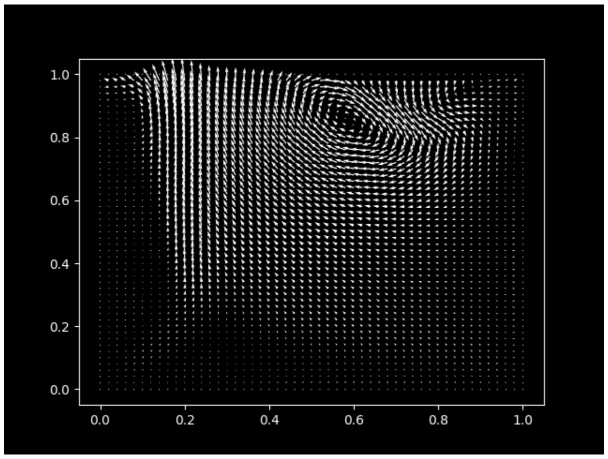
(c) Neural solver at time step 40.



(d) Ground truth at time step 40.



(e) Neural solver at time step 60.



(f) Ground truth at time step 60.

Figure 5.3: Quiver representation of neural solver (left) vs ground truth (right): Vector field trajectory for a total of 60 time steps with a step size of 0.1 seconds. There are a total of 51x51 cell points containing the properties of the fluid.

In the context of our analysis, when examining both Fig. 5.2 and Fig. 5.3, it becomes evident that our neural solver exhibits a notably high degree of accuracy in predicting fluid motion.

Note that in the neural solver's results, the initial position of fluid motion appears slightly elevated (starting point at $y \approx 2.8$) when compared to the ground truth (starting point at $y \approx 2.0$); however, it is crucial to emphasize that the direction of the fluid flow aligns closely with the ground truth result, signifying the neural solver's proficiency in replicating the actual behavior of fluid dynamics. This observation underscores the neural solver's effectiveness in accurately modeling fluid motion, despite the initial discrepancy in starting positions.

Furthermore, an additional noteworthy observation arises when comparing paired Figures (5.2c, 5.2d) and (5.2e, 5.2f). These comparisons reveal that the flow pattern generated by the neural solver exhibits a tendency to fold up, whereas the ground truth illustrates a more expansive and diffused fluid behavior. This discrepancy further underscores the distinct characteristics of the neural solver's predictions in capturing specific intricacies of fluid dynamics compared to the ground truth.

5.2 Model Performance

5.2.1 Parameter Tuning and Loss Functions

After extensive experimentation and a series of trial-and-error iterations, we determined that the model achieves its highest level of stability when employing the Tanh activation function with Mean Squared Error loss function and Stochastic Gradient Descent (SGD) optimizer. Furthermore, we initialized the weights of all the linear layers with random values drawn from a normal distribution with mean = 0 and std = 0.1 (standard deviation). We found that setting appropriate initial values for weights can help improve convergence and avoid issues such as vanishing or exploding gradients during training; however, this approach was coupled with the right choice of activation function in order to guarantee the convergence of our model.

We initially opted for the ReLU activation function, as it is widely acknowledged to exhibit better performance across a broad spectrum of problems, but encountered issues with system divergence and instability. For our second trial, we made the more suitable choice of activation function, which significantly improved our model's performance and resolved the issues. Here are the formulas for the activation functions we used in our experiments:

$$\text{First trial:} \quad ReLU(x) = \begin{cases} x, & x < 0 \\ 0, & otherwise \end{cases},$$

$$\text{Second trial:} \quad tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad \text{if } x < 0.$$

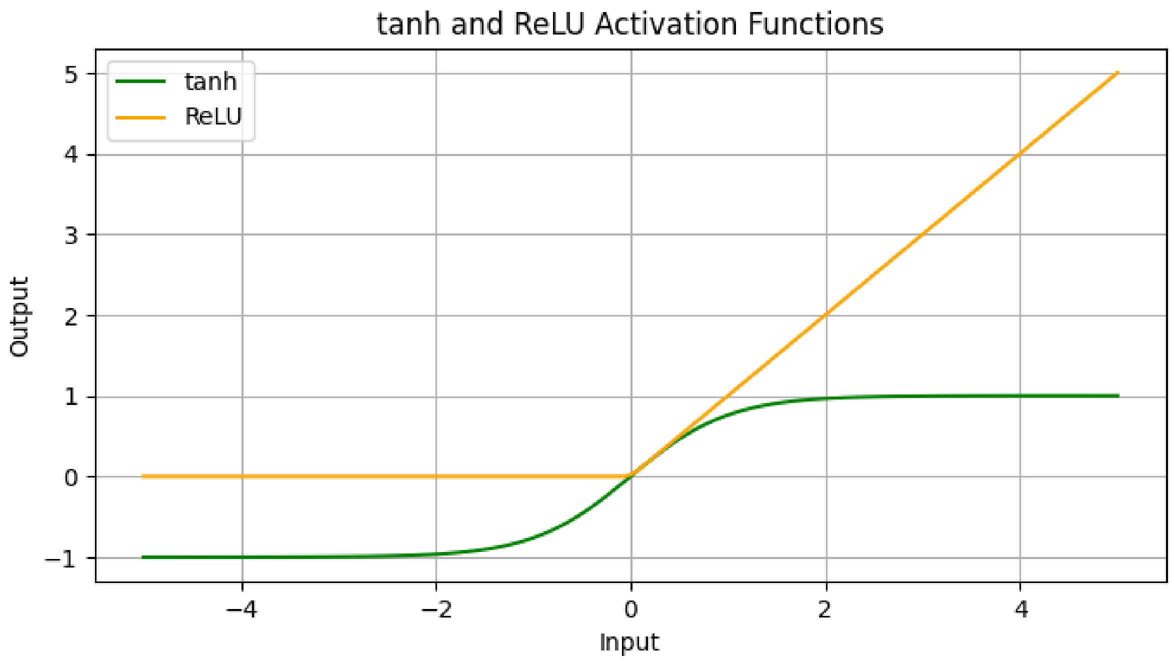


Figure 5.4: Evaluated activation functions.

Fine-tuning the learning rate proved to be a tricky task. Through multiple test cases, we observed three discernible patterns in our model's performance. For our experimental purposes, we established that in the majority of instances, the model achieved proficiency in learning from the data within 100 epochs. In the following analysis, we clarify the behavior of each loss graph and present our best pick. To enhance the visibility of the relatively small loss values in our scenario, we opted to employ a logarithmic scale for visualizing the loss functions:

- Scenario One:

  In this scenario, we are examining a loss function graph that indicates underfitting. Despite the model being trained for an extended period, the training loss consistently decreases throughout the training process. This observation is indicative of a learning rate that may be too small (in this case, set at 0.0001), causing the model to learn very slowly and potentially struggle to capture the underlying patterns in the data. As a result, even with prolonged training, the model fails to fit the training data adequately, leading to underfitting where it cannot generalize well to unseen data despite the low training loss. This situation highlights the importance of finding an appropriate learning rate that balances the trade-off between fast convergence and effective learning.

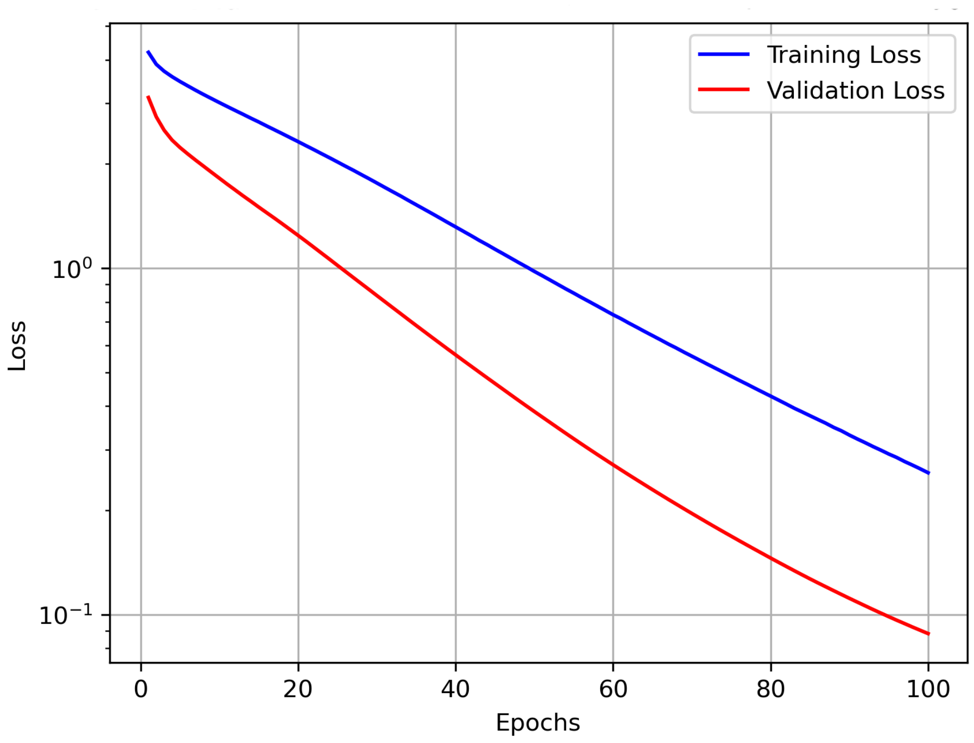


Figure 5.5: Training and validation loss over epochs (learning rate = 0.0001).

- Scenario Two:

  In this scenario, the model exhibits clear signs of overfitting. Shortly after the training begins, the training loss drops to a very low value, indicating that the model is fitting the training data almost perfectly; however, the validation loss remains relatively constant and larger than the training loss. This discrepancy between the training and validation losses suggests that the model has memorized the training data but struggles to generalize to unseen data, which is typical of overfitting.

The choice of a very high learning rate (0.99 in this case) likely contributes to this overfitting behavior. A high learning rate can cause the model to quickly adjust its parameters to minimize the training loss, potentially leading to fitting noise or outliers in the training data. Consequently, the model's performance on the validation data suffers, as it fails to generalize beyond the specifics of the training set. We have noted that this situation brings up the importance of selecting an appropriate learning rate and implementing techniques like regularization to mitigate overfitting.

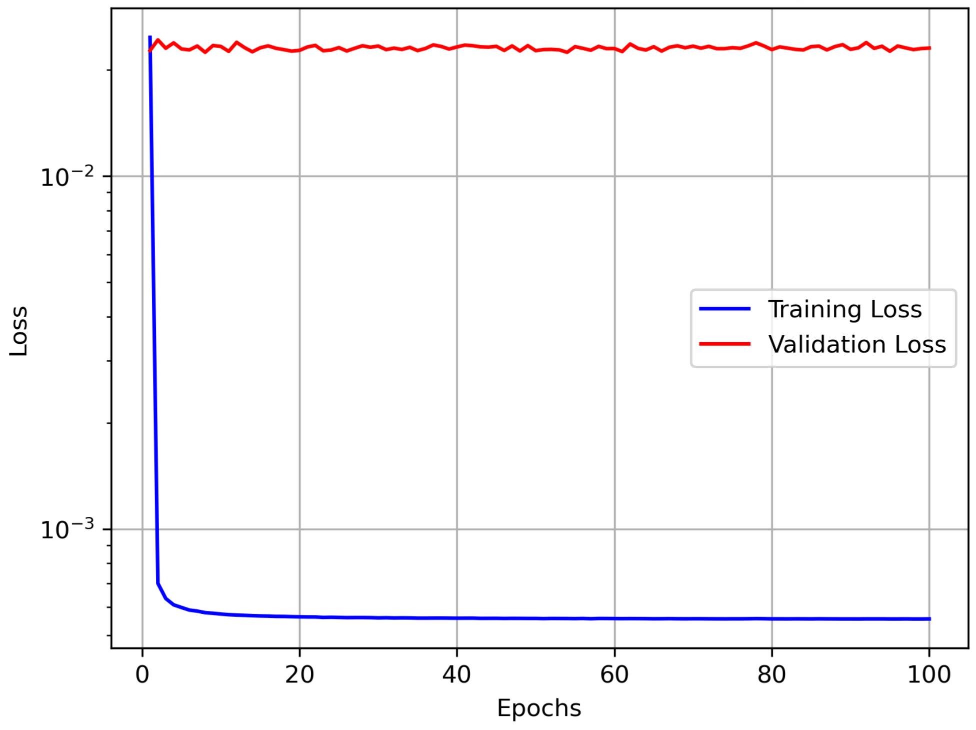


Figure 5.6: Training and validation loss over epochs (learning rate = 0.99).

- Scenario Three:

  In this scenario, we observe a good fit learning curve, characterized by both the training and validation loss decreasing to points of stability. The learning rate of 0.01 appears to be appropriate for this model, as it allows for a gradual convergence to an optimal solution without the issues of rapid overfitting or slow convergence.

The training loss decreases steadily and eventually stabilizes, indicating that the model effectively learns from the training data without overfitting. Similarly, the validation loss also decreases steadily and converges to a stable value. The small gap between the training and validation loss suggests that the model generalizes well to unseen data, as the validation loss closely tracks the training loss. Overall, this scenario demonstrates a well-balanced learning curve where the model achieves good performance on both the training and validation datasets, highlighting again the importance of an appropriate learning rate choice. Therefore, we opted for the final scenario with a learning rate of 0.01.

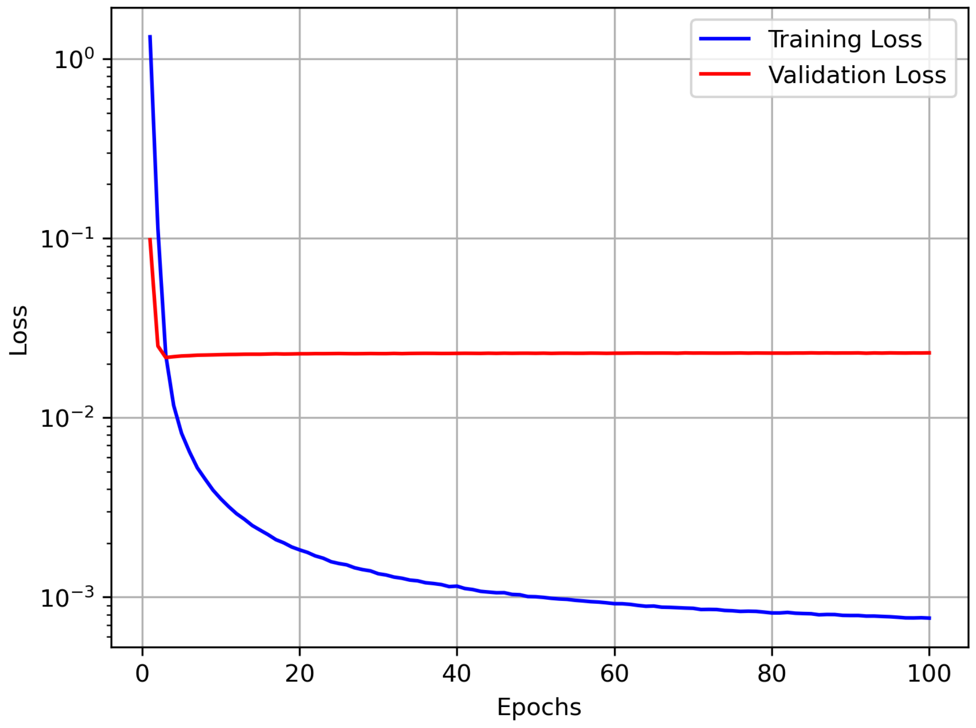


Figure 5.7: Training and validation loss over epochs (learning rate = 0.01).

5.2.2 Resolution

- Without Regularization:

  In our initial approach, we designed a model without any regularization methods, compounding with a high number of parameters exceeding 400 million. We utilized CUDA version 11.7, and the simulation runtime remained consistent at approximately 5.3 seconds for both traditional and neural solver methods; however, when it comes to computation time, the neural solver took about 6.19 seconds, while the traditional numerical solver (ground truth) required approximately 11.89 seconds.

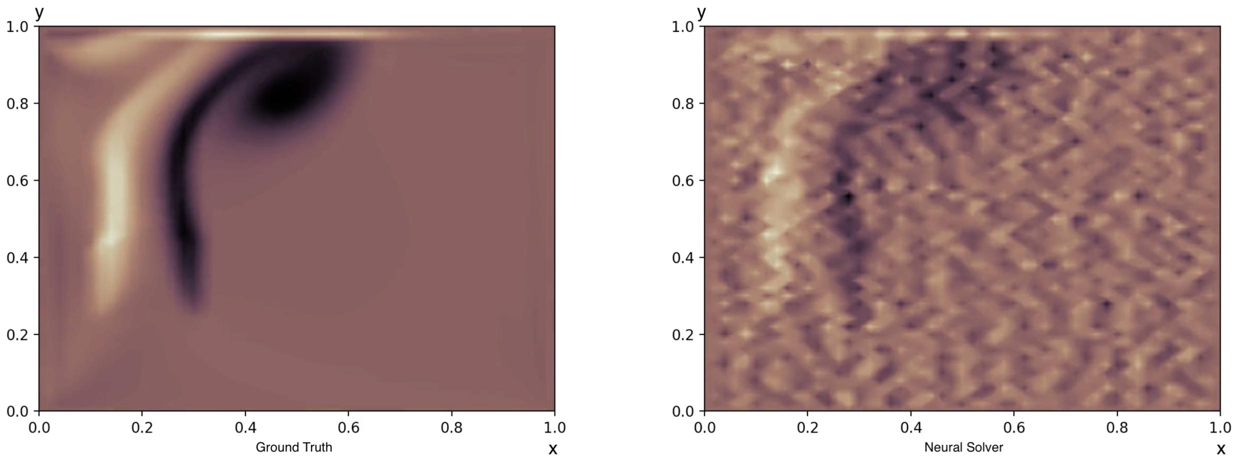


Figure 5.8: Contrasting ground truth simulation and unregularized model performance.

Despite the similar simulation runtimes, our neural network method produced visually noisy results, as illustrated in Fig. 5.8. When examining the pressure prediction graph shown in Fig. 5.9, where we compare the calculated (green) and predicted (blue) results, we observe that our supervised model was able to capture some aspects of the ground truth pattern; however, there were noticeable spikes and considerable fluctuations, contributing to the observed noise in our simulation. It is worth to mention; however, the overall fluid movement was reasonably captured by our neural solver approach, as the model provided predictions within the range of pressure values (0.5-0.61), which did not significantly exceed the bounded limits of the ground truth range (0.5-0.562).

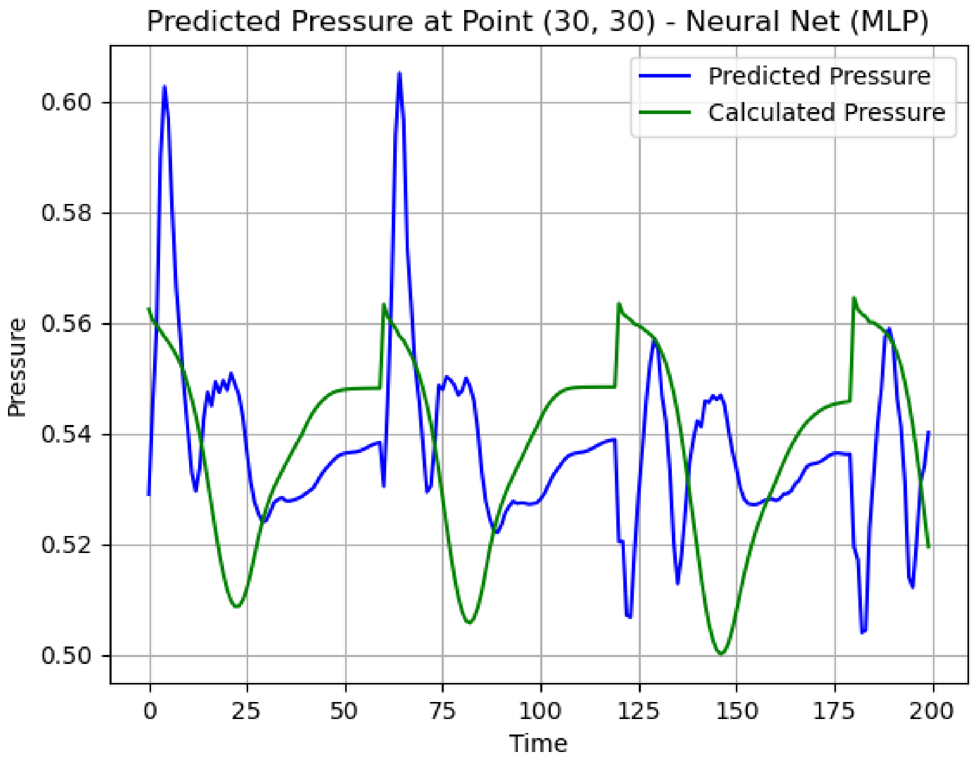


Figure 5.9: Comparison of predicted pressure and calculated pressure (unregularized model).

Lastly, we need to highlight that the calculated pressure data combines results from separate simulations, each characterized by unique fluid properties and spanning approximately 60 timeframes. This explains the presence of abrupt, spiky lines in the green calculated pressure graph.

- With Regularization:

  As pointed out earlier, in our simulation results we encountered a situation where the number of parameters in our model appeared excessively large relative to the size of our training dataset. As an initial step to address this, we decided to significantly reduce the number of parameters, scaling it down by a factor of 1000 to a more manageable number 4,280,851; however, even with this reduction, the number of parameters remained relatively high for the dataset's scale. This led us to suspect that overfitting might be a concern with our model.

To mitigate the potential overfitting issue arising from our model with many parameters and limited data, we opted to employ regularization techniques. Specifically, we applied Dropout to the hidden layers, which serves to prevent the model from fitting noise within the dataset. The impact of this regularization can be observed in Fig. 5.10, where we can see a notable improvement in the model's performance. The results on the right closely resemble those of the benchmark on the left. Our supervised model successfully captured the fluid properties and produced accurate pressure values.

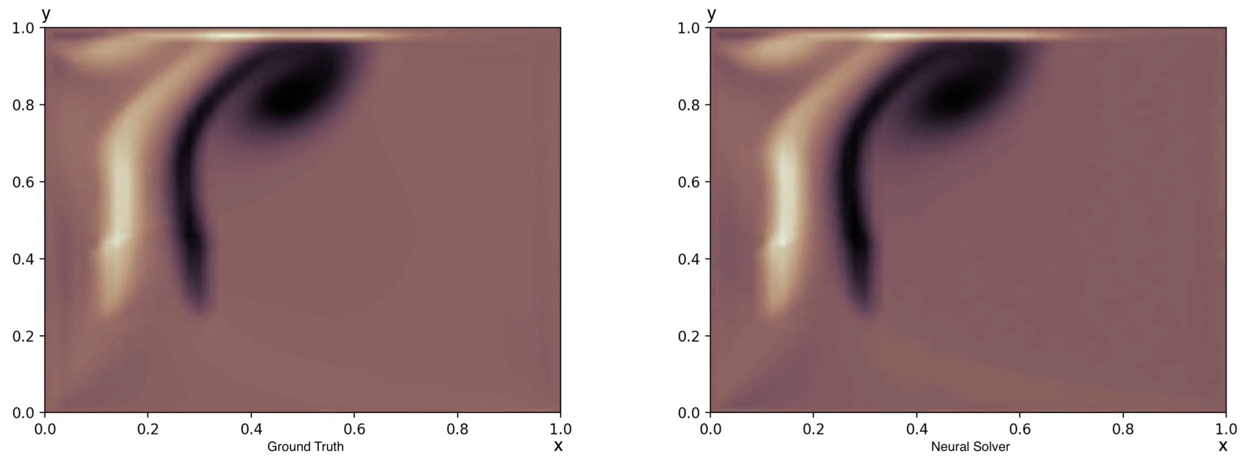


Figure 5.10: Contrasting ground truth simulation and regularized model performance.

Fig. 5.11 provides further insights into the outcomes of our regularization efforts. Notably, the predicted pressure plot (depicted in blue) exhibits reduced amplitude compared to the non-regularized approach discussed earlier. Despite this reduction in amplitude, it yields visually accurate results. This observation aligns with the Helmholtz-Hodge decomposition theorem, which suggests that in the simulation, we primarily require an updated divergence-free velocity field, which depends on the gradient of pressure rather than its magnitude. As a result, the dampened pressure values yield the same updated velocity field as those with higher amplitudes, thus contributing to the overall success of our model.

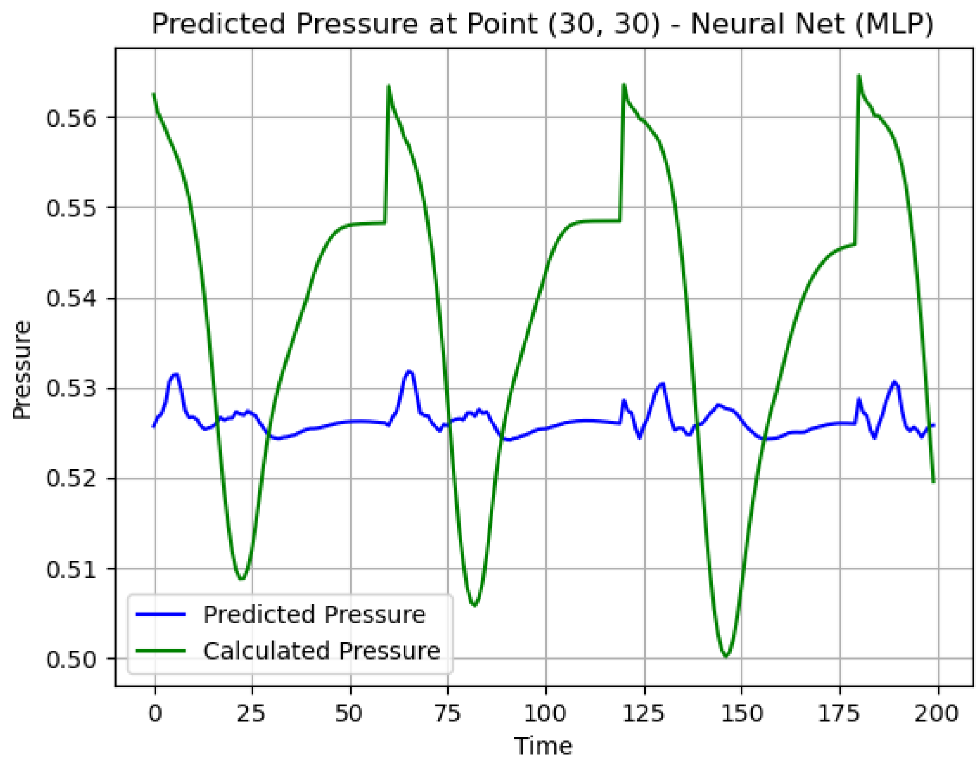


Figure 5.11: Comparison of predicted pressure and calculated pressure (regularized model).

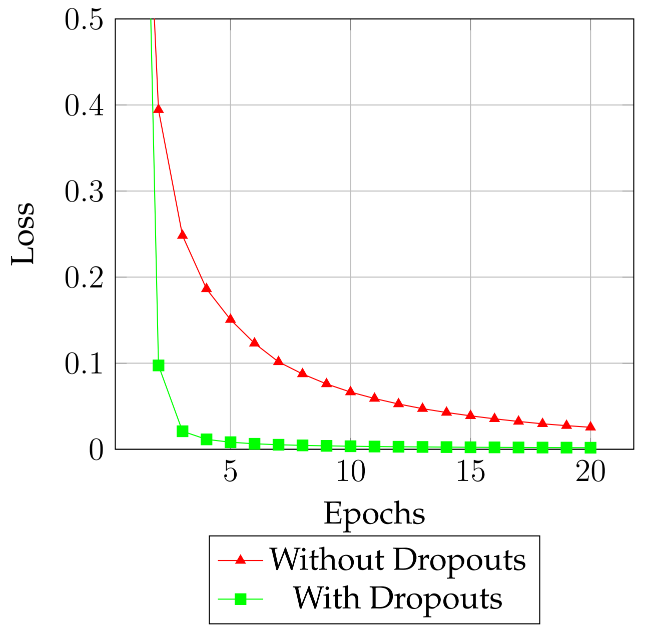


Figure 5.12: Comparison of training loss functions.

- All Together:

  Let's conduct a comprehensive comparison of our two distinct approaches. In Fig. 5.12 we place side-by-side the training loss functions of our model with Dropout (depicted in green) and without Dropout (in red). The striking observation here is that with the incorporation of Dropout, our model rapidly converges to a stable point, and the loss consistently remains lower than that of the non-regularized approach. This signifies that applying Dropout as a regularization technique has a substantial impact on the training process, enhancing its efficiency and effectiveness.

  Moving beyond the training results, Fig. 5.13 delves into the model's performance on unseen data for both approaches. Interestingly, it reveals that our model excels in terms of generalization; the validation losses for both approaches are remarkably low and exhibit minimal divergence after just five epochs. This suggests that, regardless of the regularization technique applied, our model demonstrates strong generalization capabilities, indicating its ability to perform well on previously un-

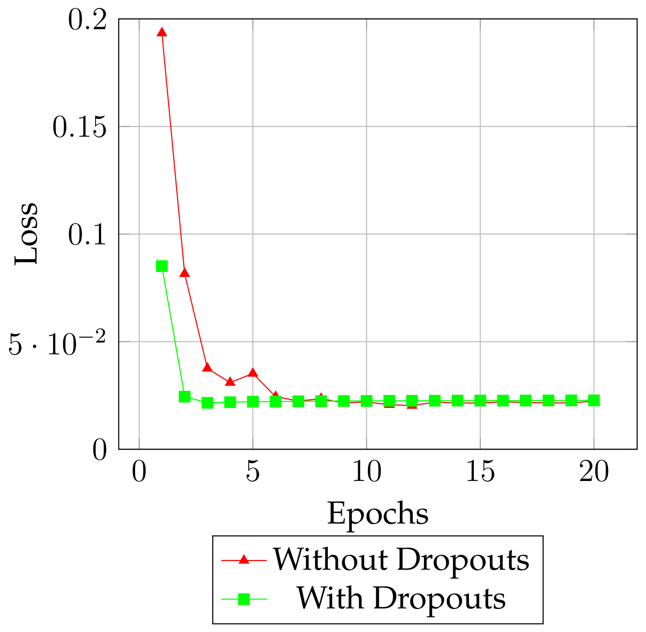


Figure 5.13: Comparison of validation loss functions.

Table 5.1: Testing result of models with different resolutions.

Model	**Test Loss**	**MSE**
No Dropout	1.021e-3	1.022e-3
With Dropout	0.959e-3	0.961e-3

seen data.

When we delve into the evaluation of testing results, as documented in Table 5.1 a noteworthy trend emerges. The model without Dropout, which yielded relatively poorer training results and exhibited less accuracy during the training phase, experiences a higher testing error in comparison to the model with applied regularization. This finding supports the importance of using regularization techniques, such as Dropout, in improving a model's ability to generalize seamlessly well to unseen data, ultimately leading to better overall performance and accuracy.

Chapter 6

Conclusion

6.1 Limitations and Future Work

In this thesis, our primary objective was to develop a simplified 2D grid-based fluid simulation that substitutes the most computationally intensive calculation step used in it with a neural network. In essence, the core of this project involved addressing a partial differential equation through the framework of supervised learning. The significance of this PDE lies in its application for creating realistic animations of natural phenomena. To accomplish this, we designed a straightforward feedforward neural network as our PDE solver.

As we delved into animation for this thesis, we encountered substantial challenges. The demand for high computing power surpassed our current setup, prompting us to explore alternative solutions. Choosing the appropriate platform and coding language proved critical, introducing complexity, particularly when features necessitated unfamiliar languages. Moreover, acquiring high-quality datasets for training deep learning models posed a significant challenge. To mitigate this, we opted for a simpler dataset that we created ourselves, albeit at the cost of sacrificing the complexity of test scenes. Despite these hurdles the results were successful, not only in terms of visual quality, which closely matched the outcomes of traditional numerical approaches, but also in terms of computational efficiency when dealing with unknown variables.

It's also essential to note that mathematical methodologies underpin the foundations of machine learning models, providing structured pathways and formulas to decipher the behavior of our world. These traditional methods, with centuries of application, will continue to be pivotal in advancing research. However, the ascent of artificial intelligence presents an equally compelling trajectory. We believe that the integration of deep learning models will open new horizons in the field of animation, pushing the boundaries of human imagination to achieve what was once deemed unattainable. While our project has concluded successfully, the journey need not end here.

Numerous promising avenues await exploration and expansion. Some potential directions for future work and collaborations include implementing the results in a 3D setting, utilizing more complex or real-world datasets, exploring Physics-Informed Neural Networks (PINN) as a more advanced model, and experimenting with unsupervised learning for PDE solving networks.

Bibliography

[1] Ahrens, C. D. *Meteorology Today: An Introduction to Weather, Climate, and the Environment*, 9th ed. Brooks/Cole, Cengage Learning, Belmont, CA, 2009.

[2] Bargteil, A. W., Goktekin, T. G., O'Brien, J. F., and Strain, J. A. A semi-lagrangian contouring method for fluid simulation. In *SIGGRAPH '05: Proceedings of the ACM SIGGRAPH 05 Electronic Art and Animation Catalog* (2005), ACM Press, pp. 238–238.

[3] Batty, C. Physics-based animation/sci-tech oscars. Tech. rep., 2007. Available at: `https://www.physicsbasedanimation.com/sci-tech-oscars/#:~:text=DOUG%20ROBLE%2C%20NAFEES%20BIN%20ZAFAR,achieve%20large%2Dscale%20water%20effects`.

[4] Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems* (2015), pp. 1171–1179.

[5] Bridson, R. *Fluid Simulation for Computer Graphics*, 2nd ed. A K Peters/CRC Press, New York, 2015.

[6] Cai, Z., Chen, J., Liu, M., and Liu, X. Deep least-squares methods: An unsupervised learning-based numerical method for solving elliptic PDEs. *Journal of Computational Physics 420* (nov 2020), 109707.

[7] CHENG, L., ILLARRAMENDI, E. A., BOGOPOLSKY, G., BAUERHEIM, M., AND CUENOT, B. Using neural networks to solve the 2D Poisson equation for electric field computation in plasma fluid simulations, 2021.

[8] CHENTANEZ, N., AND MÜLLER, M. Real-time simulation of large bodies of water with small scale details. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '10)* (2010), pp. 197–206.

[9] CHENTANEZ, N., AND MÜLLER, M. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics 30* (Year), 1–10.

[10] CHRISTEN, F., KIM, B., AZEVEDO, V. C., AND SOLENTHALER, B. Neural smoke stylization with color transfer. *CoRR abs/1912.08757* (2019).

[11] CHU, M., LIU, L., ZHENG, Q., FRANZ, E., SEIDEL, H.-P., THEOBALT, C., AND ZAYER, R. Physics informed neural fields for smoke reconstruction with sparse data. *ACM Transactions on Graphics 41*, 4 (jul 2022), 1–14.

[12] CHU, M., THÜEREY, N., SEIDEL, H.-P., THEOBALT, C., AND ZAYER, R. Learning meaningful controls for fluids. *ACM Transactions on Graphics 40* (08 2021), 1–13.

[13] DONG, W., LIU, J., XIE, Z., AND LI, D. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (nov 2019), ACM.

[14] ECKERT, M.-L., UM, K., AND THÜEREY, N. Scalarflow: a large-scale volumetric data set of real-world scalar transport flows for computer animation and machine learning. *ACM Transactions on Graphics 38*, 6 (nov 2019), 1–16.

[15] ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. Animation and rendering of complex water surfaces. *ACM Transactions on Graphics (TOG) 21*, 3 (2002), 736–744.

[16] FEDKIW, R., STAM, J., AND JENSEN, H. W. Visual simulation of smoke. *ACM SIGGRAPH 2001 Papers 35*, 3 (2001), 15–22.

[17] FOSTER, N., AND METAXAS, D. Realistic animation of liquids. *Graphical Models and Image Processing 58*, 5 (1996), 471–483.

[18] FRANZ, E., SOLENTHALER, B., AND THÜEREY, N. Global transport for fluid reconstruction with learned self-supervision. *CoRR abs/2104.06031* (2021).

[19] GATYS, L. A., ECKER, A. S., AND BETHGE, M. A neural algorithm of artistic style. *CoRR abs/1508.06576* (2015).

[20] GAUTSCHI, W. *Leonhard Euler: His Life, the Man, and His Works*. Springer, 2017.

[21] GINGOLD, R. A., AND MONAGHAN, J. J. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society 181*, 3 (1977), 375–389.

[22] HANSEN, D., ROBINSON, D. M., ALIZADEH, S., GUPTA, G., AND MAHONEY, M. Learning physical models that can respect conservation laws. In *ICML 2023* (2023).

[23] HARLOW, F. H., AND WELCH, J. E. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids 8*, 12 (1965), 2182–2189.

[24] HOLL, P., KOLTUN, V., AND THÜEREY, N. Learning to control PDEs with differentiable physics. *CoRR abs/2001.07457* (2020).

[25] JR., J. D. A. *A History of Aerodynamics and Its Impact on Flying Machines*. Cambridge University Press, 1997.

[26] KOHL, G., UM, K., AND THÜEREY, N. Learning similarity metrics for numerical simulations. In *Proceedings of the 37th International Conference on Machine Learning* (13–18 Jul 2020), H. D. III and A. Singh, Eds., vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 5349–5360.

[27] Ladický, L., Jeong, S., Solenthaler, B., Pollefeys, M., and Gross, M. Data-driven fluid simulations using regression forests. *ACM Trans. Graph. 34*, 6 (October 2015), 199:1–199:9.

[28] Mathieu, M., Couprie, C., and LeCun, Y. Deep multi-scale video prediction beyond mean square error, 2016.

[29] McAdams, A., Sifakis, E., and Teran, J. A parallel multigrid poisson solver for fluids simulation on large grids. In *Eurographics/ACM SIGGRAPH Symposium on Computer Animation* (2010), Z. Popovic and M. Otaduy, Eds., The Eurographics Association.

[30] McCartney, M., Whitaker, A., and Wood, A., Eds. *George Gabriel Stokes: Life, Science and Faith*. World Scientific, 2019.

[31] Min, C., and Gibou, F. A second order accurate projection method for the incompressible Navier-Stokes equations on non-graded adaptive grids. *Journal of Computational Physics 219*, 2 (2006), 912–929.

[32] Müller, M. Fast and robust tracking of fluid surfaces. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2009), pp. 237–245.

[33] Müller, M., Charypar, D., and Gross, M. Particle-based fluid simulation for interactive applications. vol. 2003, pp. 154–159.

[34] Navier, C. L. M. H. Mémoire sur les lois du mouvement des fluides. *Mémoires de l'Académie Royale des Sciences de l'Institut de France* (1823), 389–440.

[35] Palmer, S., Garcia, J., Drakeley, S., Kelly, P., and Habel, R. The ocean and water pipeline of disney's moana. In *SIGGRAPH 2017 Talks (Sketches)*.

[36] Polyak, B. *Introduction to Optimization*. Optimization Software, New York, 1987.

[37] RAO, C., SUN, H., AND LIU, Y. Physics-informed deep learning for incompressible laminar flows. *Theoretical and Applied Mechanics Letters 10*, 3 (2020), 207–212.

[38] RICHARDSON, L. F. *Weather Prediction by Numerical Process*, 2 ed. Cambridge Mathematical Library. Cambridge University Press, 2007.

[39] RONNEBERGER, O., FISCHER, P., AND BROX, T. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)* (2015), vol. 9351, Springer, LNCS, pp. 234–241.

[40] SAAD, N., GUPTA, G., ALIZADEH, S., AND ROBINSON, D. M. Guiding continuous operator learning through physics-based boundary constraints. In *ICLR 2023* (2023).

[41] SAAD, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2003.

[42] SELLE, A., FEDKIW, R., KIM, B., LIU, Y., AND ROSSIGNAC, J. An unconditionally stable maccormack method. *J. Sci. Comput. 35* (06 2008), 350–371.

[43] STAM, J. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 121–128.

[44] THE WASHINGTON POST. Video games dive deeper into realistic water simulations, 2023. Available at: `https://www.washingtonpost.com/video-games/2023/03/28/video-game-water/`.

[45] THÜEREY, N., WEISSENOW, K., PRANTL, L., AND HU, X. Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows. *AIAA Journal 58*, 1 (jan 2020), 25–36.

[46] THÜREY, N., MÜLLER, M., AND SCHIRM, S. Real-time breaking waves for shallow water simulations. In *Proceedings of the Pacific Conference on Computer Graphics and Applications, Pacific Graphics 2007* (October 2007), pp. 1–9.

[47] THÜREY, N., AND RÜDE, U. Stable free surface flows with the lattice boltzmann method on adaptively coarsened grids. *Comput. Visual Sci. 12* (2009), 247–263.

[48] TOMPSON, J., SCHLACHTER, K., SPRECHMANN, P., AND PERLIN, K. Accelerating eulerian fluid simulation with convolutional networks. In *ICML'17: Proceedings of the 34th International Conference on Machine Learning - Volume 70* (August 2017), pp. 3424–3433.

[49] UM, K., BRAND, R., YUN, FEI, HOLL, P., AND THÜEREY, N. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers, 2021.

[50] UM, K., HU, X., AND THÜEREY, N. Liquid splash modeling with neural networks. *CoRR abs/1704.04456* (2017).

[51] WAKILI, A., AND SADIQ, M. Comparism of quassi-seidel, jacobi and conjugate gradient methods for convergent and speed using matlab for linear system of equations. *IOSR Journal of Mathematics (IOSR-JM) 15*, 3 Ser. III (May–June 2019), 38–44.

[52] WERHAHN, M., XIE, Y., CHU, M., AND THÜEREY, N. A multi-pass gan for fluid flow super-resolution. *Proc. ACM Comput. Graph. Interact. Tech. 2*, 2 (July 2019), Article 10.

[53] WEYMOUTH, G. D. Data-driven multi-grid solver for accelerated pressure projection. *Computers &amp; Fluids 246* (Oct. 2022), 105620.

[54] WHITE, F. M. *Fluid Mechanics*, 8th edition ed. McGraw-Hill Education, 2016.

[55] WIEWEL, S., BECHER, M., AND THÜEREY, N. Latent-space physics: Towards learning the temporal evolution of fluid flow. *Eurographics 2019* (2019).

[56] WIEWEL, S., KIM, B., AZEVEDO, V. C., SOLENTHALER, B., AND THÜEREY, N. Latent space subdivision: Stable and controllable time predictions for fluid flow, 2020.

[57] WIKIPEDIA. Euler method, Accessed on 2023-09-23. Available at: `https://en.wikipedia.org/wiki/Euler_method`.

[58] WIKIPEDIA. Joseph-louis lagrange, Accessed on 2023-09-23. Available at: `https://en.wikipedia.org/wiki/Joseph-Louis_Lagrange`.

[59] XIAO, X., ZHOU, Y., WANG, H., AND YANG, X. A novel cnn-based poisson solver for fluid simulation. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 26*, 03 (2020), 1454–1465.

[60] XIE, Y., FRANZ, E., CHU, M., AND THÜEREY, N. tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow. *ACM Transactions on Graphics (SIGGRAPH 2018) 37*, 4 (July 2018), Article 95.

[61] YANG, C., YANG, X., AND XIAO, X. Data-driven projection method in fluid simulation. *Journal of Computational Physics 123*, 4 (May 2016), 567–581.

[62] ZHANG, C., WANG, X., JIN, J., LI, L., AND MILLER, J. D. Afm slip length measurements for water at selected phyllosilicate surfaces. *Colloids and Interfaces 5*, 4 (2021).

[63] ZHU, X., LIANG, J., AND HAUPTMANN, A. G. Msnet: A multilevel instance segmentation network for natural disaster damage assessment in aerial videos. *CoRR abs/2006.16479* (2020).