Beyond Grids: Neural Networks in Fluid Dynamics

Tooba Rahimnia
McGill University and Mila
Montréal, Canada
tooba.rahimnia@mail.mcgill.ca

Figure 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 [7].

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.

Keywords

Computer Graphics, Animation, Deep Neural Network

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.

Preprint, ArXiv,

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 [6] and Eulerian methods [2]. 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.

The behavior of numerous physical phenomena is dictated by the incompressible Navier-Stokes equations. 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 [4]. The second approach is the adoption of Eulerian methods, which approximate quantities on a fixed grid [3]. 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 [10] and pose challenges when minor adjustments are required in fluid simulation.

In recent years, 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. 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 propose a machine learning-based approach, inspired by the work done by Tompson et al. (2017), to accelerate the linear projection process [9]. 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.

2 Background

Computer animation often focuses on Newtonian fluid dynamics, which are primarily described by the incompressible Navier-Stokes equations, a set of partial differential equations:

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

$$\nabla \cdot (\mathbf{u}) = 0. \tag{2}$$

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. 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. 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 The Momentum Equation

Equation (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 $\mathbf{F} = m\mathbf{a}$. 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 $\mathbf{a}$:

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

Since acceleration is the derivative of velocity, we can express the equation in the following form:

$$\mathbf{F} = m\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} \tag{3}$$

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. 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:

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

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:

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

With this understanding, equation (3) can now be reformulated to incorporate the additional insights as follows:

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

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

$$m\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -V\nabla p + m\mathbf{g}. \tag{4}$$

When performing fluid calculations using a *finite* number of particles (eq. 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. However, a drawback of this approach arises when the mass and volume of each particle approach zero and rendering the equations of motion becomes 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:

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

2.2 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 illustrates two of such methods:

The first method is known as the Lagrangian method which is commonly used in particle systems. In this method, points in space have a position $\mathbf{x}$ and velocity $\mathbf{u}$. Smoothed Particle Hydrodynamics (SPH) is an example of an approach that utilizes the Lagrangian method. On the other hand, the Eulerian method 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. This approach offers the advantage of facilitating easier approximation of spatial derivatives, such as pressure and temperature.

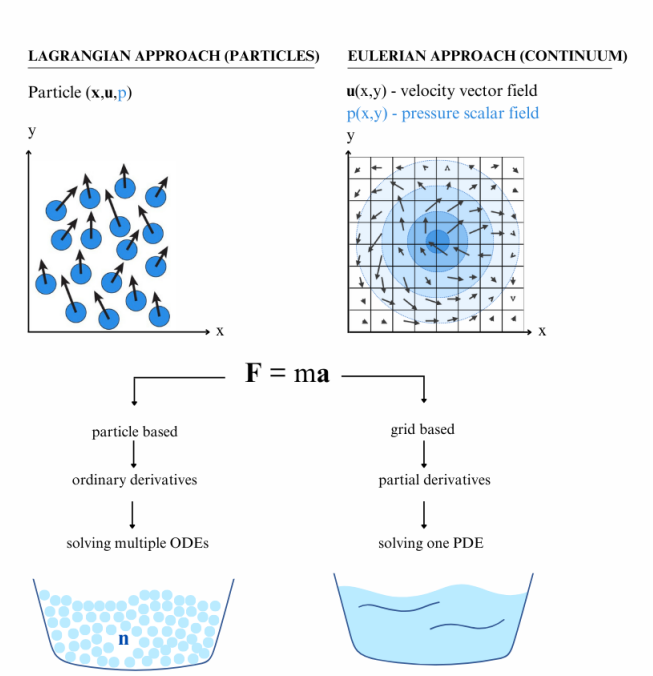


Figure 2: Comparing Lagrangian and Eulerian perspectives in fluid dynamics.

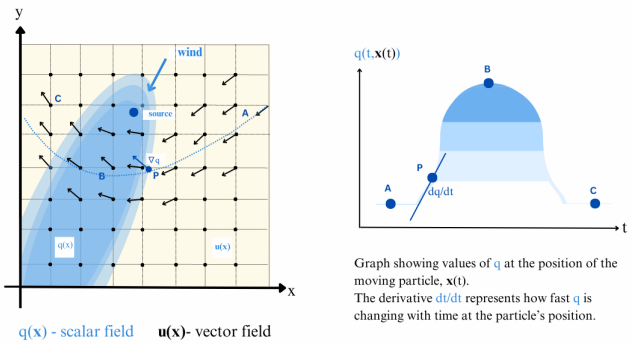


Figure 3: The material derivative.

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 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. 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. To quantify the alignment, we can compute their dot product: $(\mathbf{u} \cdot \nabla q)$ (equivalently represented as $|\mathbf{u}||\nabla q| \cos(\theta)$). 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 3.2). This involves merging the Eulerian and Lagrangian viewpoints to calculate a comprehensive derivative for each vector component:

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

2.3 Fluid Incompressibility

In the real world, fluids generally exhibit some degree of compressibility; however, in the context of computer animation, fluid compressibility can be disregarded, and all fluids can be assumed to be incompressible. 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). 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.

2.4 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. 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 $\hat{t}$ velocity along the solid surface can either be zero for viscous fluids or left unchanged for inviscid fluids. In the current project, the latter condition is adopted.

Lastly, the free surface condition is handled by setting the pressure outside the fluid to zero, without imposing any control on the velocity. This approaches enables two essential processes: 1. interpolating missing data within data range and 2. extrapolating future data outside data range.

3 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.

3.1 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 Δt 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 $\mathbf{g}$ to $\mathbf{u^A}$ to obtain $\mathbf{u^B}$;

Make $\mathbf{u^B}$ divergence-free and enforce incompressibility;

3.2 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 $\mathbf{u} \cdot \nabla q$.

In a practical illustration, Fig. 4 showcases to determine the value of velocity components of a particular point at position $\mathbf{x}_\mathrm{G}$ 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 $\mathbf{x}_\mathrm{P}$. 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 $\mathbf{x}_\mathrm{G}$.

3.3 Pressure Equation

As explained in section 2.1, 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 from the intermediate velocity field $\mathbf{u}$ obtained from the advection step:

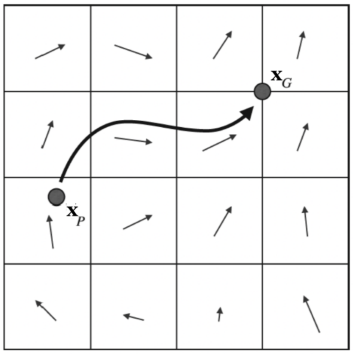


Figure 4: The Semi-Lagrangian Method.

$$\mathbf{u}^{n+1} = \mathbf{u} - \Delta t \frac{\nabla p}{\rho} \tag{5}$$

the resulting velocity field fulfills the requirement of incompressibility:

$$\nabla \cdot \mathbf{u}^{n+1} = 0 \tag{6}$$

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 5 using central differences for ∇p as follows:

$$\begin{aligned} u^{n+1}_{i+\frac{1}{2},j,k} &= u_{i+\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\left(\frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}\right) \\ v^{n+1}_{i,j+\frac{1}{2},k} &= v_{i,j+\frac{1}{2},k} - \frac{\Delta t}{\rho}\left(\frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x}\right) \\ w^{n+1}_{i,j,k+\frac{1}{2}} &= w_{i,j,k+\frac{1}{2}} - \frac{\Delta t}{\rho}\left(\frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x}\right) \end{aligned} \tag{7}$$

Up until now, we have discussed the pressure update (equation 5), but we still need to ensure the satisfaction of the incompressibility condition (equation 6). 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:

$$(\nabla \cdot \mathbf{u})_{i,j,k} \approx \frac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{\Delta x} + \frac{v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k}}{\Delta x} + \frac{w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}}}{\Delta x}. \tag{8}$$

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 (7) is substituted into equation (8) as follows:

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

$$\begin{aligned}
\frac{1}{\Delta x}\Bigg[& \left(u_{i+\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}\right) \\
& - \left(u_{i-\frac{1}{2},j,k} - \frac{\Delta t}{\rho}\frac{p_{i,j,k} - p_{i-1,j,k}}{\Delta x}\right) \\
& + \left(v_{i,j+\frac{1}{2},k} - \frac{\Delta t}{\rho}\frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x}\right) \\
& - \left(v_{i,j-\frac{1}{2},k} - \frac{\Delta t}{\rho}\frac{p_{i,j,k} - p_{i,j-1,k}}{\Delta x}\right) \\
& + \left(w_{i,j,k+\frac{1}{2}} - \frac{\Delta t}{\rho}\frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x}\right) \\
& - \left(w_{i,j,k-\frac{1}{2}} - \frac{\Delta t}{\rho}\frac{p_{i,j,k} - p_{i,j,k-1}}{\Delta x}\right)\Bigg] = 0
\end{aligned} \tag{9}$$

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):

$$-\frac{\Delta t}{\rho}\begin{pmatrix} \frac{-6p_{i,j,k} + p_{i+1,j,k} + p_{i,j+1,k} + p_{i,j,k+1}}{\Delta x^2} \\ + \frac{p_{i-1,j,k} + p_{i,j-1,k} + p_{i,j,k-1}}{\Delta x^2} \end{pmatrix} = -\begin{pmatrix} \frac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{\Delta x} \\ + \frac{v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k}}{\Delta x} \\ + \frac{w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}}}{\Delta x} \end{pmatrix}. \tag{10}$$

3.4 Finding and Applying Pressure

Once equation (10) 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 b:

$$Ax = b. \tag{11}$$

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 (7). The resulting velocity field, denoted as $\mathbf{u}^{n+1}$, is divergence-free and satisfies the specified boundary conditions.

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:

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla \mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f}, \tag{12}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{13}$$

In the equation 12 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 [5]. 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 [2].

To solve equation 12 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 12, followed by a projection onto fields that are divergence-free. The overall procedure is illustrated in the figure below:

$$\mathbf{u}_0(\mathbf{x}) \xrightarrow{\text{add force}} \mathbf{u}_1(\mathbf{x}) \xrightarrow{\text{advect}} \mathbf{u}_2(\mathbf{x}) \xrightarrow{\text{project}} \mathbf{u}_3(\mathbf{x})$$

Figure 5: 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 2**, inspired by the work on accelerating Eulerian fluid simulation by Tompson and colleagues [9].

Algorithm 2: 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 2** deliberately excludes the pressure term ($-\nabla p$) from equation 12. 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 13. 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 [8]. 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.

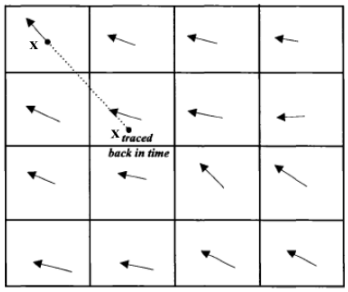


Figure 6: Semi-Lagrangian advection scheme.

In **Algorithm 2**, step 2.1 is by far the most computationally demanding component. It involves solving the following Poisson equation:

$$\nabla^2 p_t = \frac{1}{\Delta t}\nabla \cdot \mathbf{u}_t^*, \tag{14}$$

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 14, 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.

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. 7) 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.

4.1.2 Classes and Functions. The code for this Master's thesis paper 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

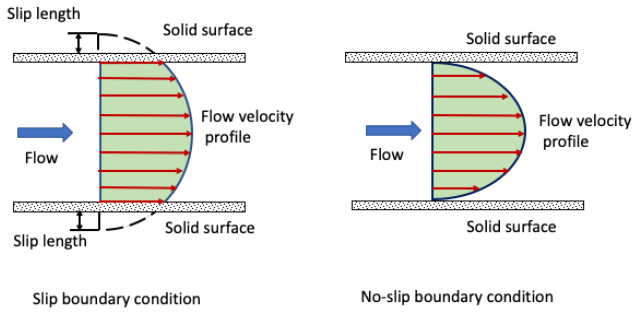


Figure 7: Slip and no-slip boundary conditions: in no-slip boundary conditions, the speed of the fluid at the wall is zero, whereas in slip boundary conditions there is relative movement between the wall and the fluid. Image by [12].

- `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 multilayer 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.

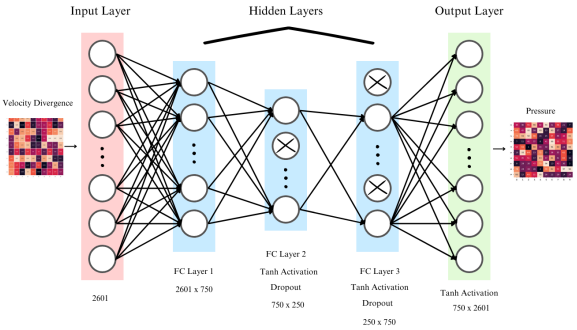


Figure 8: Our convolutional network for pressure solve.

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 8.

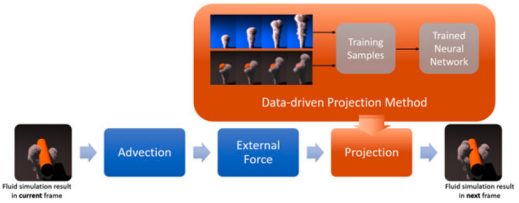


Figure 9: Here, our approach closely aligns with the methodology employed by Yang et al. (2016) Specifically, our data-driven projection method seamlessly integrates into the established framework of traditional grid-based fluid simulations. By using a basic dataset for training, our data-driven projection method significantly speeds up solving the Poisson equation compared to traditional methods. Image by [11].

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.

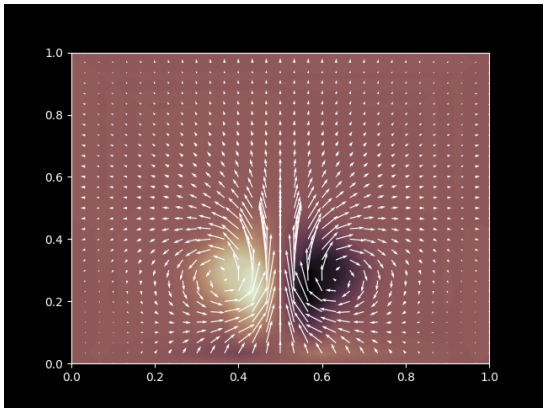
(6) Saving the Model and Loss Plot: After training, the trained model's state dictionary is saved to a .pth file. Additionally, a loss plot is generated and saved as an image file, aiding in analyzing the model's learning progress.

Testing Setup

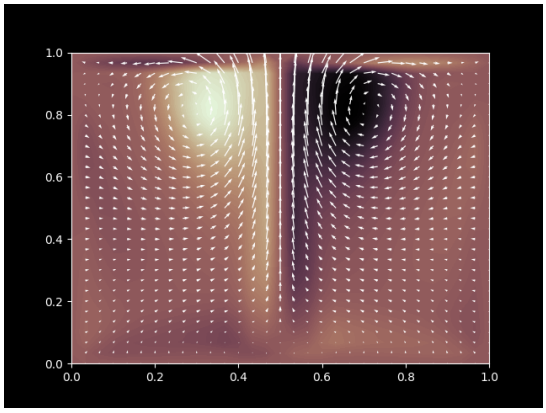
The final code segment involves the testing and evaluation of our trained neural network model. It starts by preparing the testing input data, reshaping it into a suitable format, and performing min-max scaling normalization. The same preprocessing steps are applied to the testing output data. Subsequently, 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.



(a) Flow motion at time step 10.



(b) Flow motion at time step 50.

Figure 10: Ground truth: sample fluid simulation with velocity field.

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.

Figures in Appendix A illustrate 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 A 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.

In the context of our analysis, when examining both out color-schemed and gray-schemed figures in Fig. A, 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 (24a, 24b) and (25a, 25b). 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.

6 Model Performance

6.1 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:

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

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

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.
- 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.

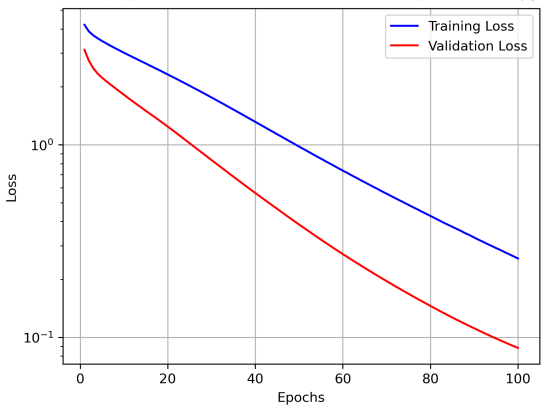


Figure 11: Training and validation loss over epochs (learning rate = 0.0001).

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.

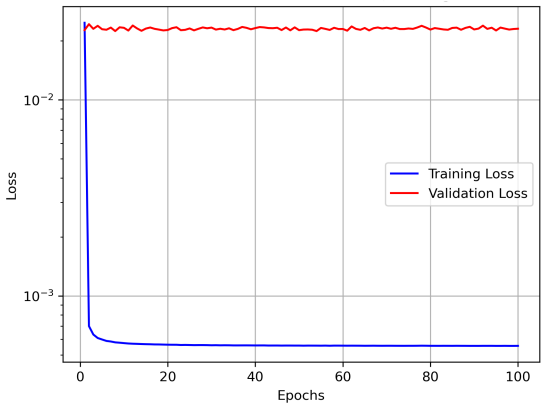


Figure 12: Training and validation loss over epochs (learning rate = 0.99).

- 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.

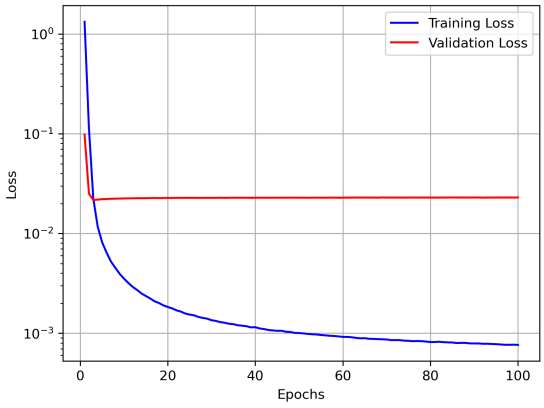


Figure 13: Training and validation loss over epochs (learning rate = 0.01).

6.2 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.

  Despite the similar simulation runtimes, our neural network method produced visually noisy results, as illustrated in Fig. 14. When examining the pressure prediction graph shown in Fig. 15, 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,

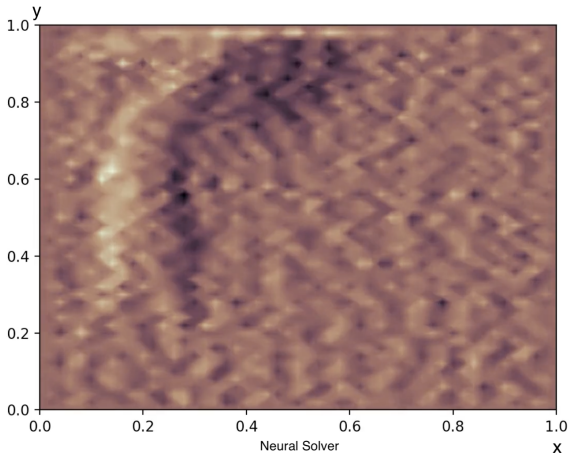


Figure 14: Contrasting ground truth simulation and unregularized model performance.

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).

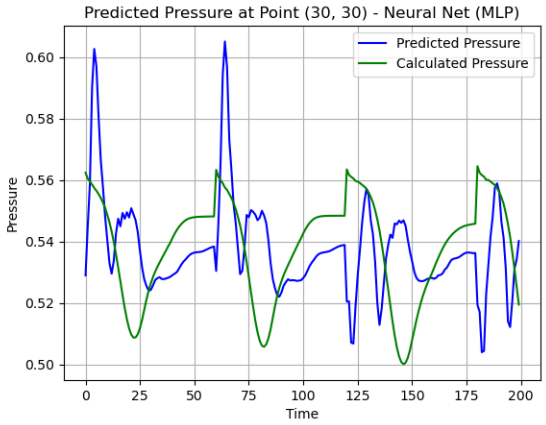


Figure 15: Comparison of predicted pressure and calculated pressure (unregularized model).

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. 16, 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.

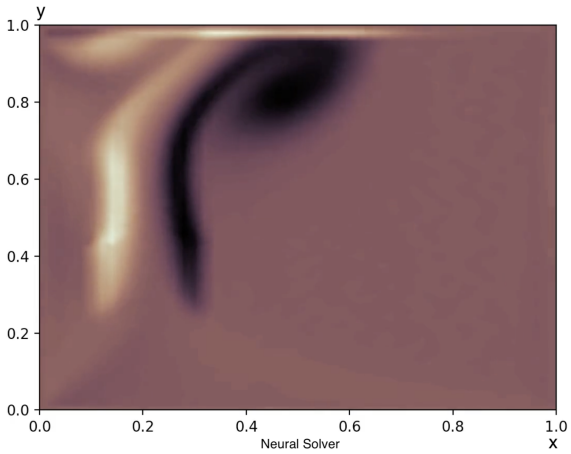


Figure 16: Contrasting ground truth simulation and regularized model performance.

Fig. 17 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.

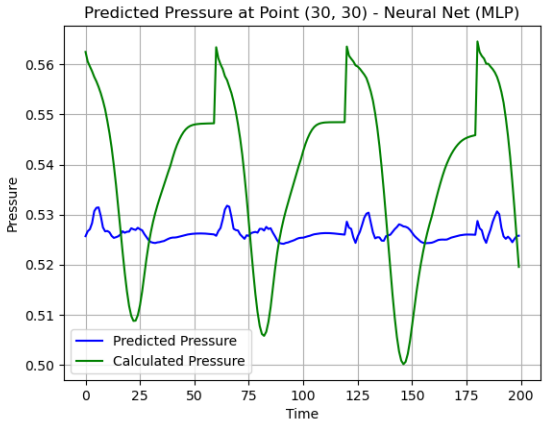


Figure 17: Comparison of predicted pressure and calculated pressure (regularized model).

- All Together:
  Let's conduct a comprehensive comparison of our two distinct approaches. 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. 19 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 unseen data.

  When we delve into the evaluation of testing results, as documented in Table 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.

Table 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

7 Conclusion

In this paper, 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.

References

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

[2] Robert Bridson. 2015. *Fluid Simulation for Computer Graphics* (2nd ed.). A K Peters/CRC Press, New York. 276 pages. https://doi.org/10.1201/9781315266008

[3] Nicholas Foster and Dimitri Metaxas. 1996. Realistic Animation of Liquids. *Graphical Models and Image Processing* 58, 5 (1996), 471–483.

[4] R. A. Gingold and J. J. Monaghan. 1977. Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars. *Monthly Notices of the Royal Astronomical Society* 181, 3 (1977), 375–389. https://doi.org/10.1093/mnras/181.3.375

[5] Francis H. Harlow and James E. Welch. 1965. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *Physics of Fluids* 8, 12 (1965), 2182–2189. https://doi.org/10.1063/1.1761178

[6] Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-Based Fluid Simulation for Interactive Applications. *Fluid Dynamics* 2003, 154–159.

[7] Sean Palmer, Jonathan Garcia, Sara Drakeley, Patrick Kelly, and Ralf Habel. [n. d.]. The Ocean and Water Pipeline of Disney's Moana. In *SIGGRAPH 2017 Talks (Sketches)*.

[8] Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An Unconditionally Stable MacCormack Method. *J. Sci. Comput.* 35 (06 2008), 350–371. https://doi.org/10.1007/s10915-007-9166-4

[9] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. 2017. Accelerating Eulerian Fluid Simulation With Convolutional Networks. In *ICML'17: Proceedings of the 34th International Conference on Machine Learning - Volume 70*. 3424–3433.

[10] Gabriel D. Weymouth. 2022. Data-driven Multi-Grid solver for accelerated pressure projection. *Computers & Fluids* 246 (Oct. 2022), 105620. https://doi.org/10.1016/j.compfluid.2022.105620

[11] Cheng Yang, Xubo Yang, and Xiangyun Xiao. 2016. Data-driven projection method in fluid simulation. *J. Comput. Phys.* 123, 4 (May 2016), 567–581. https://doi.org/10.1234/jcp.2016.1234

[12] Chen Zhang, Xuming Wang, Jiaqi Jin, Lixia Li, and Jan D. Miller. 2021. AFM Slip Length Measurements for Water at Selected Phyllosilicate Surfaces. *Colloids and Interfaces* 5, 4 (2021). https://doi.org/10.3390/colloids5040044

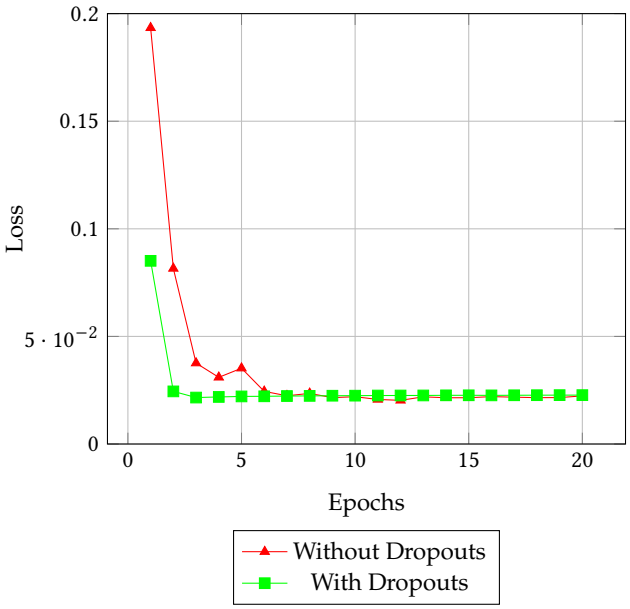


Figure 18: Comparison of validation loss functions.

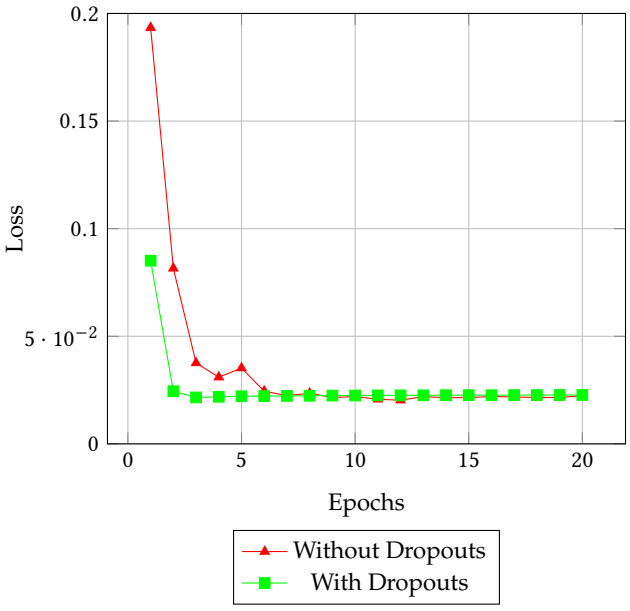
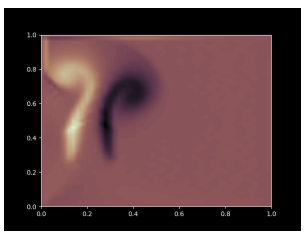
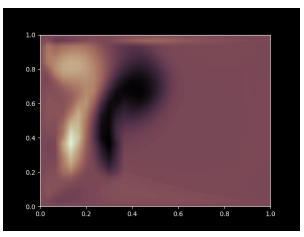


Figure 19: Comparison of validation loss functions.

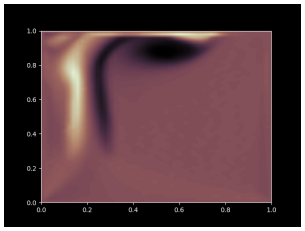
A Appendix



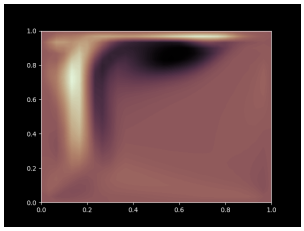
(a) Neural solver at time step 20.



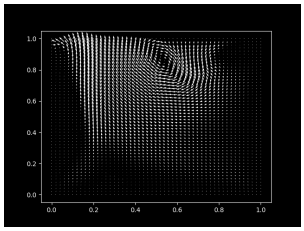
(b) Ground truth at time step 20.



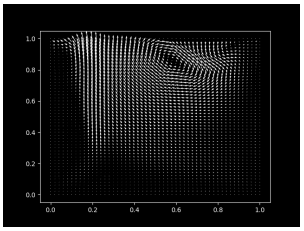
(a) Neural solver at time step 40.



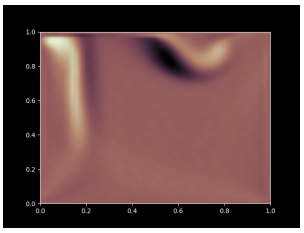
(b) Ground truth at time step 40.



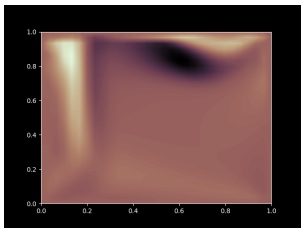
(a) Neural solver at time step 60.



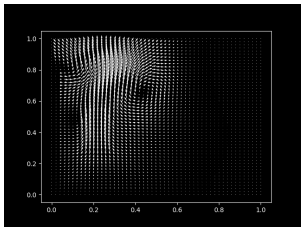
(b) Ground truth at time step 60.



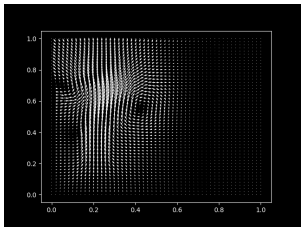
(a) Neural solver at time step 60.



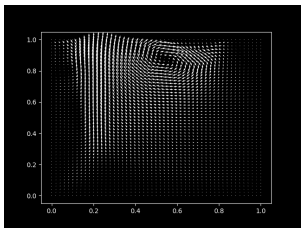
(b) Ground truth at time step 60.



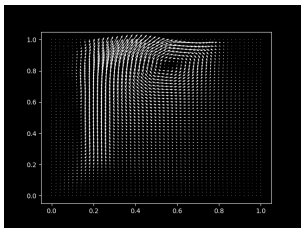
(a) Neural solver at time step 20.



(b) Ground truth at time step 20.



(a) Neural solver at time step 40.



(b) Ground truth at time step 40.