Differentiable Fluid Simulation and Rasterization

Xiangyu Kong, School of Computer Science

McGill University, Montreal

August, 2023

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

Master of Computer Science

Abstract

In this work, we present a fully differentiable fluid simulation and rendering framework that propagates gradient from rendered 2D images to 3D simulation and rendering parameters. We implement a differentiable grid-based fluid simulator using both Jax and PyTorch to solve the Navier-Stokes equation, and evaluate and compare between the two packages in the context of fluid simulation. We also present the rendering scheme that transforms the 3D simulated output grid into a mesh and computes the accumulative direct absorption. We conduct experiments to show that even with the loss of information by projecting 3D grids to 2D images, the gradient information can still be propagated throughout the whole pipeline properly. We will show how this gradient information can be used to solve inverse control problems through optimization techniques in a fast and automated fashion.

Abrégé

Dans ce travail, nous présentons un cadre de simulation et de rendu de fluide entièrement différentiable qui propage le gradient des images 2D rendues aux paramètres de simulation et de rendu 3D. Nous implémentons un simulateur de fluide différentiable basé sur une grille utilisant à la fois Jax et PyTorch pour résoudre l'équation de Navier-Stokes, et évaluons et comparons les deux packages dans le contexte de la simulation de fluide. Nous présentons également le schéma de rendu qui transforme la grille de sortie simulée 3D en un maillage et calcule l'absorption directe cumulée. Nous menons des expériences pour montrer que même avec la perte d'informations en projetant des grilles 3D sur des images 2D, les informations de gradient peuvent toujours être propagées correctement dans tout le pipeline. Nous montrerons comment ces informations de gradient peuvent être utilisées pour résoudre des problèmes de contrôle inverse grâce à des techniques d'optimisation de manière rapide et automatisée.

Acknowledgements

First and foremost, I would like to offer my sincerest gratitude to my co-supervisors Prof. Paul G. Kry and Prof. Derek Nowrouzezahrai for their guidance and inspiration in this project and throughout my master's program. I am also grateful to my peers at the McGill Graphics Lab (MGL) for creating an inspiring environment to exchange insightful discussions and feedbacks. I would like to thank Tianyu Cao for her support and encouragement that helped me through my master's program. Finally, I would also like to thank my parents Keran Xing and Geng Kong. This project could not have been completed without their financial and mental support.

Table of Contents

Abstract i
Abrégé ii
Acknowledgements iii
List of Figures viii
List of Tables ix

1 Introduction **1**
1.1 Contributions 2
1.2 Thesis Overview 3

2 Related Work **4**
2.1 Fluid Simulation 5
2.2 Fluid Control 6
2.3 Differentiable Simulation 7
2.4 Differentiable Fluid Rendering 9
2.5 Fully Differentiable Frameworks 10
2.6 Deficiencies of State of the Art 11

3 Background **13**
3.1 Automatic Differentiation 13
3.1.1 AutoDiff 13
3.1.2 AutoDiff Modes 14
3.2 Fluid Simulation 17

3.2.1 The Incompressible Navier-Stokes Equation . . . 17
3.2.2 Advection . . . 19
3.2.3 External Forces . . . 22
3.2.4 Pressure Projection . . . 23
3.3 Rasterization Pipeline . . . 32
3.3.1 Vertex Processing . . . 33
3.3.2 Rasterization . . . 36
3.3.3 Fragment Processing . . . 36
3.4 Smoke Absorption . . . 37
3.4.1 Beer-Lambert Law . . . 37
3.4.2 Absorption Coefficient . . . 39

4 Methodology **40**
4.1 Differentiable Fluid Simulator . . . 40
4.1.1 Data Structures . . . 41
4.1.2 Simulation Parameters . . . 42
4.1.3 Advection . . . 43
4.1.4 Inflow Injection . . . 44
4.1.5 External Forces . . . 46
4.1.6 Pressure Projection . . . 48
4.1.7 Jax and PyTorch Backends . . . 54
4.2 Differentiable Renderer . . . 57
4.2.1 Mesh Conversion . . . 57
4.2.2 Absorption Interpolation . . . 58
4.2.3 Smoke Rendering . . . 59
4.2.4 Acceleration and Differentiation . . . 60
4.3 Optimization Methods . . . 61

5 Experiments and Results **63**

5.1 Differentiable Fluid Simulation 63

5.1.1 Forward Simulation 64

5.1.2 Simulation Learning 66

5.2 Fully Differentiable Pipeline 70

5.2.1 Simulation and Rendering 70

5.2.2 Simulation and Rendering Learning 71

6 Conclusion **80**

6.1 Advantages and Limitations 80

6.2 Future Work 81

References **83**

List of Figures

3.1 AutoDiff forward and reverse modes 16
3.2 Advection backtracing 21
3.3 Null space problem 24
3.4 Staggered grids 24
3.5 Boundary conditions 27
3.6 Solid boundary example 28
3.7 Light through smoke medium 38

4.1 Differentiable sphere 46
4.2 Conjugate gradient tolerance and iterations comparison 52
4.3 Mesh conversion 58

5.1 Forward simulation 65
5.2 Simulation velocity optimization 68
5.3 Simulation velocity optimization loss 69
5.4 Full pipeline forward 72
5.5 Full pipeline inflow location optimization 74
5.6 Full pipeline inflow optimization and manually adjusted inflow location . . 75
5.7 Full pipeline inflow optimization and manually adjusted losses 76
5.8 Full pipeline inflow and camera learning result 77
5.9 Full pipeline inflow and camera learning history 78
5.10 Full pipeline inflow and camera learning loss 78

5.11 Full pipeline inflow and camera learning location visualization 79

List of Tables

3.1 2D mathematical operations 18

4.1 Non-learnable simulation parameters 42
4.2 Learnable simulation parameters 43
4.3 Conjugate gradient tolerance and iterations comparison 51

5.1 Simulation forward performance 66
5.2 Simulation optimization performance against resolution 68
5.3 Simulation optimization performance against time steps 69
5.4 Full pipeline forward performance 71

Chapter 1

Introduction

Fluid simulation is a crucial aspect to many fields, including robotics, engineering, gaming, and cinematography. It allows engineers and artists to simulate the flow of fluids such as liquids and gases accurately and realistically. A common task in the field of fluid simulation is fluid control. By manipulating the fluid simulation parameters, the user can produce different simulated outcomes. For example, in robotics, fluid control can be used to simulate the flow of liquid around a creature to optimize its shape for swimming performance [23]. Traditionally, gradient-based optimization methods such as keyframing [14] and adjoint method [15] have been reliable solutions for solving the fluid control problem. In the recent years, with the advances in machine learning and automatic differentiation, data-driven solutions have been gaining popularity [26, 36, 55]. Among which, one effective solution is to incorporate the physical knowledge of the fluid solver into a machine learning pipeline by making the fluid simulator differentiable. By defining an appropriate loss function between the simulated 3D state and the desired 3D state, the differentiable simulator can propagate the gradient information to the 3D simulation parameters for learning. Incorporating the differentiable simulator into the learning pipeline not only accelerates the learning process, but also produces physically realistic results.

Another common workflow that involves fluid simulation is to render the 3D simulated fluid states into 2D images for visualization and presentation. This process is usually done with rasterization or ray tracing. Similar to the advances in differentiable simulation, differentiable rendering [39, 44] has also been an active research area these years as well. By making the rendering process differentiable, the gradient information can be propagated from the rendered 2D images to the 3D rendering parameters such as camera positions and lighting information.

In the past few years, there has been attempts in combining differentiable simulators and renderers to form a fully differentiable pipeline, so that users can optimize 3D simulation and rendering parameters using information from 2D images. However, these methods are typically applied to rigid or soft body simulations [47, 60], and to our knowledge, there are very few existing systems of fully differentiable pipelines for fluid simulations. The main challenges of building such frameworks are the full differentiability and the scalability. The simulator and renderer must be differentiable themselves, and on top of that, the connection between the two components must be differentiable as well, so that the gradient can propagate backwards without any loss of information. Furthermore, the pipeline must be scalable so that it handles not only significant amount of time steps in the simulation, but also large enough simulation and rendering resolutions to preserve the details. In this work, we explore the viability of the a fully differentiable fluid simulation and rendering pipeline, and present our implementation that overcomes the challenges described above, along with the results for learning tasks using our framework.

1.1 Contributions

We are interested in optimizing 3D parameters using 2D rendered image. We build and evaluate a fully differentiable pipeline by implementing a differentiable and physically accurate grid-based fluid simulator using both Jax and PyTorch to solve the Navier-Stokes

equation, and crafting a rendering scheme that renders the 3D simulated output grid using an existing differentiable renderer. With the fully differentiable fluid pipeline, we demonstrate the accuracy and efficacy of our framework through forward and learning experiments. We will show that even with the loss of information by projecting 3D grids to 2D images, the gradient information can still be propagated throughout the whole pipeline properly.

1.2 Thesis Overview

Chapter 2 conducts a literature review on existing work regarding fluid simulation, fluid control problems, differentiable simulation, differentiable renderer, and existing fully differentiable pipelines. Chapter 3 introduces the background knowledge required to build our differentiable fluid simulation that solves for the Navier-Stokes equation. We introduce the rendering scheme for rendering the 3D simulated grid using direct absorption to the renderer. We will also give an overview on automatic differentiation. Chapter 4 describes the implementation details of our fully differentiable pipeline. We discuss the implementation of our version of the stable fluids algorithm [10] and the rendering scheme introduced in Chapter 3. We will also discuss the techniques used for performing learning tasks and accelerating the pipeline. Chapter 5 presents the results of our experiments. We perform both forward and learning experiments on both the differentiable simulator alone and the full pipeline. We also perform benchmarks on our simulator to compare the implementation using different backends. Finally, Chapter 6 will conclude with discussions and future work ideas.

Chapter 2

Related Work

In this work, we combine differentiable fluid simulation and differentiable fluid rendering into a fully differentiable pipeline. We then use this framework for optimization tasks such as fluid control problems. In this section, we will conduct a brief literature review on the topics related to our work.

First, since our work is based on grid-based fluid simulation, we will give a brief history of fluid simulation in Section 2.1. Our method aims to solve inverse problems such as fluid control. For this reason, we then review how techniques prior to the advent of differentiable simulation methods are used for solving this problem in Section 2.2. After that, we review the recent progress on differentiable simulation and how it can be used to solve the fluid control problem using machine learning techniques in Section 2.3. Then, we shift our focus to discuss previous work on fluid rendering and differentiable rendering methods in Section 2.4. Finally, we review the recent progress on fully differentiable frameworks that combine differentiable simulation and differentiable rendering in Section 2.5.

2.1 Fluid Simulation

Simulating the flow of fluids such as water and smoke has a long history in computer graphics, and it continues to be an active research area due to its importance in the movie and video game industries.

In the 1980s, early fluid simulation techniques focus on simulating particle systems to achieve visually compelling results [3, 6]. While these methods produce satisfactory effects, they do not accurately reflect the physical model of fluid flows, which is described by the Navier-Stokes equation.

As early as the 1960s, scientists have developed methods that solve the Navier-Stokes equation, including Lagrangian (particle-based) and Eulerian (grid-based) approaches [1, 5, 9]. These methods use explicit solvers to solve the Navier-Stokes equation and produce physically accurate fluid simulations. The problem with these explicit solvers is that numerical instabilities will occur as the time steps become larger, and the simulations will become unstable.

To address these problems, methods that combine the two approaches were proposed. Particle-in-cell (PIC) and fluid-implicit-particle (FLIP) [4, 18] are popular methods that advect the Lagrangian particles, transfer velocities to Eulerian grids and then project the velocities in the grid. This approach works exceptionally well with free surface boundary conditions, where there is a clear boundary between the fluid and the air, because advecting and tracking particles explicitly allows the algorithm to preserve more details locally.

Stable fluids [10], proposed by Stam in the early 2000s, is another physically accurate grid-based fluid simulation algorithm, and this algorithm remains fundamental to many modern grid-based fluid simulation to this day. Stam proposes the Semi-Lagrangian scheme that combines the two approaches above. The algorithm advects the fluid using a Lagrangian view point and stores and computes the velocity in an Eulerian grid. The

semi-Lagrangian scheme allows the solver to solve the Navier-Stokes equation implicitly and accurately, even for larger time steps.

After stable fluids was proposed, researchers have continuously worked on improving and extending this algorithm. Under certain constraints, the algorithm's run time can be sped up extensively. For example, for periodic boundary conditions, using Fast Fourier Transform (FFT) for the pressure projection solve significantly improves the algorithm's run time [13]. Other research has been focused on improving the physical accuracy of the algorithm further. For instance, the vortex particle method [17, 25] mitigates the numerical dissipation problem, and the physics-based energy model [22] enhances fluid turbulence. The algorithm has also been extended to highly viscous Smoothed Particle Hydrodynamics (SPH) fluids by reconstructing the velocity field from target velocity gradients [27]. Recently, other improved advect-projection schemes have been proposed to remove artificial viscosity and preserve the vorticity in fluids [38, 57].

In our work, we simulate a scene of smoke in a box, where the air is the fluid, and there is no free surface boundary condition. Instead, we are only concerned with solid boundary conditions, since the fluid is bounded by solid walls. For this reason, in this thesis, we extend an improved version of the stable fluids algorithm explained in the *Fluid Simulation* book by Bridson [21]. Our method is described in Section 4.1. Though our algorithm is relatively simple, our approach should easily be adapted to other more complex algorithms described above.

2.2 Fluid Control

Fluid control allows artists and engineers to manipulate the fluid simulation using control parameters such as velocity and forces to achieve a target flow without losing physical accuracy. Extending the physics-based fluid simulation and the stable fluid algorithm, several optimization methods have been explored to solve the fluid control problem.

The idea of controlling physics-based fluid simulations can be traced back to as early as the late 1990s by Foster et al. [8, 12]. In their work, Foster et al. suggested controlling the simulation outcome by modifying the fluid parameters and imposing velocities at different locations on the grid. By tweaking the initial conditions, users can modify the simulation results. However, to have the simulation achieve a specific desired state, users must go through extensive experiments and trial-and-error to figure out the correct control parameters.

To mitigate this problem, Treuille et al. [14] proposed a fluid control method that achieves a user-specified state was proposed by. By defining an objective function that measures the difference between the simulation state and the user-provided keyframes and mathematically deriving the gradients for each fluid simulation operation, the method turns the fluid control problem into a quasi-Newton optimization problem. By solving the optimization problem, the method computes the external forces acting on the fluid that minimizes the difference between the simulated and keyframe states. This method was later extended and improved by McNamara et al. by optimizing the gradient computation step using the adjoint method [15]. These methods were proposed before the popularization of differentiable simulation frameworks, but later frameworks highly benefitted from these earlier research.

2.3 Differentiable Simulation

With the recent boom of automatic differentiation (AutoDiff) frameworks and machine learning (ML) techniques, differentiable simulations have been gaining popularity in the field of research. Depending on the extent of the involvement of physical knowledge in the differentiable framework, these methods can be classified and put on a spectrum.

On the extreme side of the spectrum, scientists have attempted to replace the traditional simulation entirely using neural networks (NNs) and ML methods [55]. The convincing results and superior runtime performance demonstrated the potential of ML

in this field. However, because these methods lack the knowledge of the physical model, extensive parameter tuning and NN architecture designing are required, and the trained networks cannot be easily generalized to adapt to different boundary conditions while remaining physically accurate.

One approach to including physical knowledge in the training pipeline is incorporating physics-based constraints in the loss functions. Tompson et al. [32] and Xiao et al. [37] included the divergence-free constraint in the loss function and used NNs to replace the iterative PDE solver to infer the pressure term, increasing the speed of the pressure solve while maintaining the accuracy. However, this approach is limited by the complexity of the solution manifold. Incorporating the physics knowledge in the loss function alone is insufficient to capture a more complex range of solutions.

Yet another approach to combining classical numerical methods with machine learning techniques is to integrate the numerical solvers for the PDEs into the ML pipeline during training. Instead of having NNs replacing the fluid solver, the gradients for the fluid solvers are computed during backpropagation [52, 53]. This method is closely related to and derived from the gradient-based fluid control methods described earlier [14, 15]. Compared to pure-NN-based models, the accuracy of the models is significantly improved, and compared to the physics-based loss methods, a lot more details of the simulation are preserved after training. Recent research improves this idea by introducing more advanced learning techniques. For instance, Pan and Manocha [31] accelerated the training by using an alternating direction method of multiplier (ADMM) optimizer and relaxing constraints in intermediate training iterations, and only enforcing the strong constraint at the end of the training iterations.

Thuerey et al. have written an excellent textbook, *Physics-based Deep Learning (PBDL)*, introducing the ideas above [52]. Along the book, Holl et al. implemented *PhiFlow* [42], a differentiable fluid simulator implementing the ideas in the PBDL book, with supporting backends including NumPy [41], PyTorch [40], TensorFlow [24] and Jax [33, 34]. Other open-source differentiable simulation implementations exist, such as *DiffTaichi* [43]. Still,

DiffTaichi tries to generalize to other physical simulations, such as rigid body simulations, instead of specializing in fluid simulation.

2.4 Differentiable Fluid Rendering

In industries such as gaming and engineering, after simulation, a common task is to project the 3D fluid states onto 2D screens for visualization and presentation. There are two general approaches to rendering fluids: ray tracing-based and rasterization-based methods. On top of the rendering solutions, to tackle inverse problems, which propagate gradients from the 2D screens back to the 3D rendering inputs, various differentiable rendering methods are proposed as well.

Ray tracing-based rendering methods, such as the ray marching algorithm [7, 48, 50], provide the most physically accurate visualization. However, these methods are usually slow at runtime and require more complex models than rasterization-based methods. In recent years, significant progress has been made on differentiable ray marching algorithms [35, 54, 59]. These methods provide accurate gradients for the 3D parameters, but they also inherit the problem from forward ray-tracing-based methods that the rendering speed is relatively slow. Because our work mainly focuses on the full differentiability of the pipeline, instead of pursuing high-quality visual effects, we opted to use rasterization-based methods.

In rasterization-based fluid rendering, one of the most commonly used methods is billboarding [16, 19, 20]. These methods project the 3D particle-based fluid states onto 2D screens and render them as 2D sprites. The main advantage of these methods is that they are fast and easy to implement, but with a cost that the quality of the rendered images is inferior and may have trouble encapsulating the density information accurately. Instead of this option, we will convert the 3D fluid grid into a mesh, interpolate the density values at the vertices, and render the absorption image. The detail of our method is explained in Section 4.2.

In recent years, differentiable rasterization pipelines have been researched extensively. OpenDR [58] is one of the first general-purpose differentiable rendering systems, but it has a relatively limited shading model. Neural 3D Mesh Renderer [29] provides a more generalized differentiable rendering system, but its backward pass hallucinates on the triangle edges and thus produces inconsistent gradients. Soft Rasterizer [39] attempted to fix the gradient accuracy by blurring the rasterized triangles, but the blur also makes the triangle edges less sharp, trading image accuracy for gradient consistency. Neural Radiance Fields (NeRF) [46] uses deep neural networks to create 3D scenes from a few 2D photos, and it has been one of the most successful differentiable rendering systems so far.

In our work, we use an existing differentiable rasterization framework named *NvDiffRast* [45], which focuses on fast, GPU-based differentiable rendering for meshes with support to PyTorch [40] and TensorFlow [24]. The implementation treats basic rendering operations such as rasterization and attribute interpolation as individual modules and provides custom gradient computations. This decreases the memory requirement for backpropagation while maintaining the accuracy of the gradients.

2.5 Fully Differentiable Frameworks

In recent years, researchers have tried to combine the differentiable simulation and differentiable rendering into a fully differentiable pipeline. Without the differentiability of both components, connecting them and forming the 2D image will cause a loss of information. This loss of information makes the inverse problem ill-posed and unsolvable. However, the fully differentiable frameworks aim to solve these ill-posed problems using the extra gradient information. In 2021, Murthy et al. [47] were the first to propose a fully differentiable framework named *GradSim* that combines differentiable simulation and differentiable rasterization. *GradSim* used a loss function defined on the rendered frame buffer and mainly focused on soft body and cloth simulations and visual motor control problems. Later research [60] extended this idea to use a depth-based

rendering loss function to remove the dependencies of colours, lighting conditions and textures, but still on soft body simulations.

However, this idea of the fully differentiable pipeline has barely been explored in the area of grid-based fluid simulations. Guan et al. [56] and Li et al. [49] have explored such ideas with particle-based fluid simulations with NeRFs [46]. Liu et al. [61] combined physics-based Eulerian fluid simulations and NeRFs. Their work focuses on liquid simulations and uses a convolutional neural network (CNN) to replace the Poisson solver, accelerating the simulation and training runtime. Our work adopts a similar idea but differs in that we use a fully iterative Poisson solver and focus our work on smoke simulations.

2.6 Deficiencies of State of the Art

As we have seen in the previous sections, differentiable simulation and differentiable rendering have been researched extensively in recent years. However, there are still some deficiencies in the current state of the art, which our work aims to address.

Firstly, although there are plenty of research and implementations on grid-based differentiable fluid simulations, there has not been many comprehensive benchmarks between implementations using different differentiable frameworks, specifically between Jax and PyTorch. Holl et al. provided the PhiFlow framework [42], which supports both the Jax and PyTorch backends, but since they aim to provide a generalized facade that abstracts away the underlying implementation, a lot of unnecessary overhead is introduced, thus causing the benchmarks to be inaccurate. While introducing the DiffTaichi [43] framework, Hu et al. provided benchmarks of the framework against implementations using Jax. However, their implementation did not fully utilize the JIT (Just In Time) compilation feature of Jax, which is one of the main advantages of using Jax. For this reason, the benchmarks are not accurate either. In our work, we fill this gap

by providing both runtime and memory benchmarks using both Jax [33] and PyTorch [40] implementations.

Secondly, as mentioned in Section 2.5, there has not been many research on fully differentiable frameworks that combine grid-based differentiable fluid simulation and rasterization-based differentiable fluid rendering. Existing implementations [39, 49, 56] use either particle based fluid simulations, neural network as fluid solvers, or NeRFs as differentiable renderers. Our work aims to fill this gap by combining a grid-based fluid simulation that uses a Poisson solver and a rasterization-based fluid renderer using the NvDiffRast [44] rasterizer.

Finally, memory consumption has been a long-lasting issue in learning-based methods due to the construction of the computational graph during backpropagation. This is especially significant in the context of grid-based differentiable fluid simulation since the memory consumption increases non-linearly with the grid resolution. In our work, we show how we tackle this problem using the checkpointing technique, and present the memory consumption result using the Jax and PyTorch implementations for both the differentiable simulation alone and the fully differentiable pipeline.

Chapter 3

Background

In this chapter, we will introduce the background knowledge required to build our fully differentiable framework. We will first give a brief overview of automatic differentiation, along with some common frameworks in Section 3.1 After that, we introduce fluid simulation in Section 3.2. Finally, we describe the rasterization pipeline in Section 3.3 and how smoke can be visualized and rendered using direct absorption in Section 3.4.

3.1 Automatic Differentiation

With the boom of machine learning, automatic differentiation (AutoDiff) [28] has become a popular technique for computing the gradient of a differentiable function. In this work, we use AutoDiff to make our fluid simulator backwards differentiable to compute the gradients of the loss with respect to simulation parameters. In this section, we will introduce AutoDiff, and one of the main approaches for computing the gradient of a function: reverse-mode AutoDiff.

3.1.1 AutoDiff

Traditionally, there are other methods in computer science that are used for computing gradients. One of which is numerical differentiation. This method approximates the

gradient by taking the central difference of the function at two points that are close to each other. For a function f, the gradient at a point x can be approximated by

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon},$$

where ϵ is a small number.

However, this method is not accurate because of numerical errors, and it is not scalable to complex functions with multiple input variables because the number of evaluations is proportional to the number of input variables.

Another method used for computing the gradients of a function is symbolic differentiation. This method uses expression manipulation and works by applying the chain rule to expand the function into a composition of simpler elementary functions. This solves the problem of numerical errors, but it can often result in lengthy and cryptic expressions, known as "expression swells", and only works with closed-form functions.

AutoDiff provides an alternative to these methods that is automated, accurate, and scalable. It works by decomposing a function into a sequence of elementary operations and applying the chain rule to compute the gradients of the function. The gradients for each elementary function, such as addition, multiplication, trigonometric functions, etc., are defined manually through operator overloading. Depending on the order and direction of the AutoDiff, multiple passes are traversed in the computation graph to compute the gradients of the function.

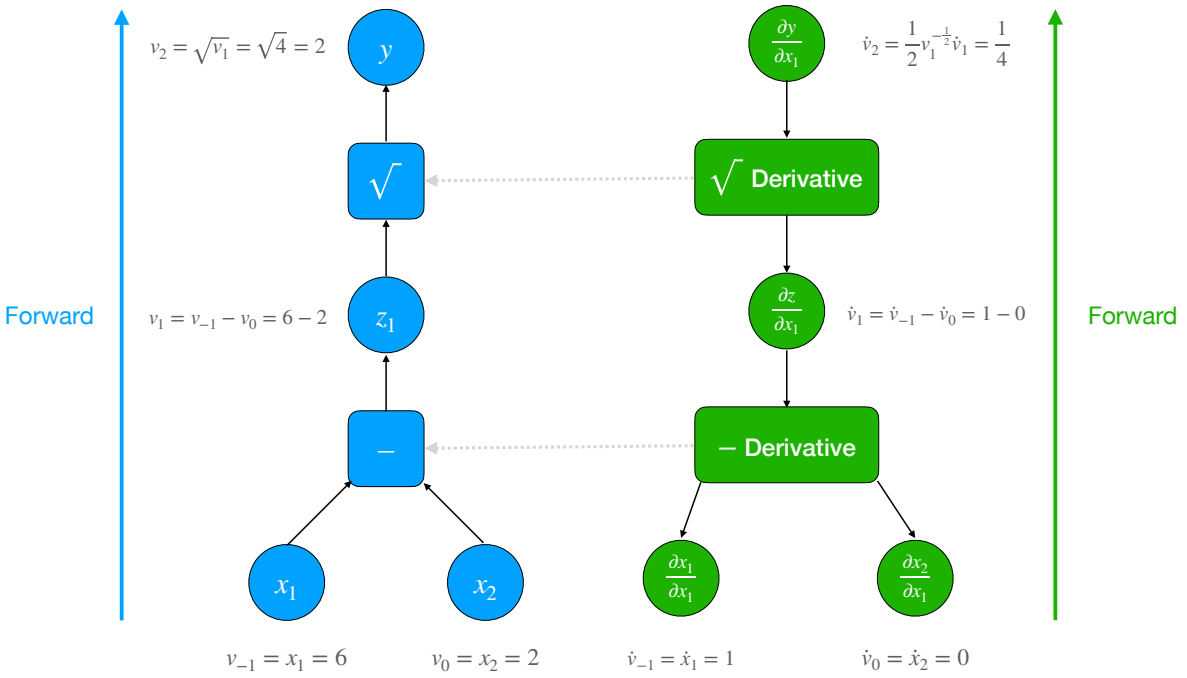
3.1.2 AutoDiff Modes

There are two main approaches for computing the gradient of a function: forward-mode AutoDiff and reverse-mode AutoDiff. Both approaches work by recording the elementary operations in a computation graph and applying the chain rule to compute the gradients of the function. The difference between the two modes is that they traverse the computation graph in different order and will require different passes to compute the

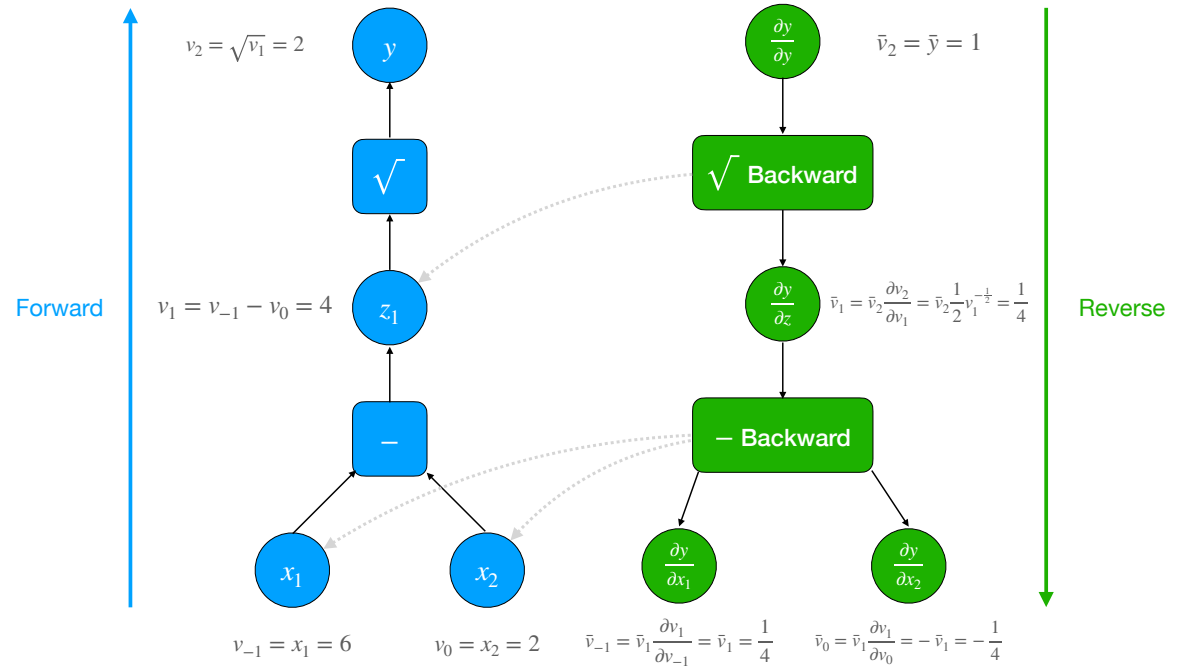
Jacobian (the matrix of all first-order partial derivatives of a vector-valued function where there are multiple outputs and inputs). To illustrate the difference, we work with a toy example that computes the gradient of the function $y = \sqrt{x_1 - x_2}$.

Forward-mode AutoDiff is also known as the tangent linear method. After computing the function value and recording the elementary operations in the computation graph, subsequent forward passes are traversed to compute the gradient of the function with respect to each input variable. For each gradient pass, only one input variable is marked as active, so to compute the Jacobian, the gradient pass needs to be repeated for each input variable. In our toy example, this is illustrated in Figure 3.1a. After recording the computation graph, when computing the gradient of the function with respect to x_1, we mark x_1 as active by setting the tangent trace $\dot{x}_1 = 1$ and others to 0. Then, we traverse the computation graph in the forward direction to compute the gradient of the intermediate results with respect to x_1 after each operation. Finally, we reach the output of the function and we obtain the gradient of the function against x_1. We repeat the same computation for x_2 for the its gradient as well. We can see that in order to compute the full Jacobian, we need 2 gradient forward passes. This mode is useful particularly in scenarios where the number of input variables is smaller than the number of output variables.

On the other hand, reverse-mode AutoDiff is also known as the backpropagation algorithm. Different from the forward-mode, reverse-mode AutoDiff computes the gradient backwards, starting from the output of the function. For each gradient pass, one output variable is marked as active, and the gradient is computed with respect to all the inputs. In our example, this is illustrated in Figure 3.1b. After recording the computation graph, we set the reverse adjoint for the output of the function $\bar{y} = 1$. Then, we traverse in the backward direction to compute the gradient of the function with respect to each intermediate variable after each operation. Finally, we arrive at the inputs of the function and we obtain the gradient of the function against x_1 and x_2. If there were multiple outputs to the function, we repeat the same computation for the rest of the outputs for their gradient as well. We can see that in order to compute the full Jacobian, in this case,



(a) Forward-mode autodiff



(b) Reverse-mode autodiff

Figure 3.1: An example of computing gradient of a function with forward-mode and reverse-mode AutoDiff. The computation graph is shown in blue on the left, and the gradient computation pass is shown in green on the right.

we only require 1 gradient reverse pass. The reverse-mode Autodiff is useful when the number of output variables is smaller than the number of input variables.

In our work, we use a scalar-valued loss function for learning, and there are a lot more degrees of freedom for the input variables, including both simulation and rendering parameters. Therefore, we use reverse-mode AutoDiff to compute the gradients of the loss function with respect to these parameters because of the efficiency of the algorithm in this scenario.

3.2 Fluid Simulation

In this section, we will introduce the basic concepts of fluid simulation. We will explain the Navier-Stokes equation along with the assumptions we make in our work. After that, we describe how the stable fluids [10] algorithm is used to solve the Navier-Stokes equation using the splitting method, and we will cover each part of the fluid solve operations separately along with how these operations affect the differentiability of the simulator.

3.2.1 The Incompressible Navier-Stokes Equation

A fluid simulation composes of multiple time steps, where each time step is a simulation of the fluid's motion over a small time interval Δt. These fluid motions are governed by the Navier-Stokes equation. The Navier-Stokes equation is given by

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \tag{3.1}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{3.2}$$

where $\mathbf{u}$ is the velocity as a vector field, p is the pressure as a scalar field, ρ is the density of the fluid as a scalar, ν is the kinematic viscosity of the fluid as a scalar, and $\mathbf{f}$ is the external force as a vector field.

Table 3.1: 2D mathematical operations used in the Navier-Stokes equation

Notation	Expanded formula	Explanation
∇p	$\begin{bmatrix} \partial p/\partial x \\ \partial p/\partial y \end{bmatrix}$	The gradient of the scalar field p
$\nabla \mathbf{u}$	$\begin{bmatrix} \nabla u & \nabla v \end{bmatrix}$	The gradient of the vector field u
$\nabla^2 \mathbf{u}$	$\nabla \cdot \nabla \mathbf{u}$	The Laplacian of the vector field u
$\nabla \cdot \mathbf{u}$	$\partial u/\partial x + \partial v/\partial y$	The divergence of the vector field u

During the simulation, we are also interested in other quantities that are not included in the equation above. Specifically, we use s to represent the smoke marker density as a scalar field for visualization purposes. Note that this is different from the density ρ of the fluid. We also use T to represent the temperature of the fluid as a scalar field. Later, we will use $\mathbf{q}$ to generalize for quantities (both scalars and vectors) carried by a fluid; we will use the superscript $\mathbf{q}^t$ to represent the quantity at time step t. For 2D vector field $\mathbf{u} = (u, v)$, and scalar field p, the mathematical operations used in the Navier-Stokes equations are defined in Table 3.1. The 3D mathematical operations follow naturally by extending both the spatial dimension and the vector dimension by one.

The Navier-Stokes equation is derived from Newton's second law of motion. The derivation is beyond the scope of this thesis. However, it is covered in detail in the *Fluid Simulation* book by Bridson [21].

In the Navier-Stokes equation, Equation (3.1) encapsulates the conservation of momentum. It describes that in the limit, as the volume of the fluid particles get infinitesimally small, the rate of change of the particles' velocities over time, also known as the material derivative of the velocity, $\frac{D\mathbf{u}}{Dt}$, is equal to the sum of the forces acting on that particle. The definition of material derivative and its meaning will be explained in more detail when we introduce advection in Section 3.2.2.

Equation (3.2) specifies the incompressibility constraint. This means that for a patch of fluid, the amount of fluid flowing into the patch equals the amount of fluid flowing out of the patch. The constraint is enforced by the force caused by pressure, which we will explain in Section 3.2.4.

For our work, we make the assumption that the fluid is inviscid, meaning the fluid is non-sticky and has zero viscosity. We make this assumption because we get diffusion partly for free due to numerical dissipation. Mathematically, this means that we assume $\nu = 0$. This assumption simplifies the Navier-Stokes Equation to

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho} \nabla p + \mathbf{f}, \tag{3.3}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{3.4}$$

The Navier-Stokes Equation does not have an analytical closed-form solution because it contains a partial differential equation (PDE). To solve the Navier-Stokes equation, we use the stable fluids algorithm [10, 21]. The algorithm takes a semi-Lagrangian approach, and it is based on the splitting method, which is a technique for solving a PDE by decomposing it into a set of simpler equations or PDEs. Specifically, in our case, the equation is decomposed into three smaller operations - advection, external forces, and projection. This is useful because it allows different parts to be solved separately with methods that are best suited to them.

3.2.2 Advection

The first step in the stable fluids algorithm is advection. This step describes the fluid particles and the quantities $\mathbf{q}$ they carry being moved by the velocity field. In the stable fluids algorithm, this is done by using a semi-Lagrangian approach, which combines the Lagrangian and Eulerian approaches.

In a continuum, the Lagrangian approach tracks each fluid particle individually. The observer is attached to the fluid particle, and the fluid particle moves with the velocity field. The particles carry quantities $\mathbf{q}(t)$ that change over time t. These quantities include the position $\mathbf{x}(t)$ and velocity $\mathbf{u}(t)$.

The Eulerian approach uses a fixed grid, and the observer is fixed at a location $\mathbf{x}$ in the grid. Depending on the location $\mathbf{x}$, as time t changes, the quantities $\mathbf{q}(t, \mathbf{x})$ changes. Note that in this case, the position $\mathbf{x}$ is an independent variable unrelated to time.

The semi-Lagrangian approach combines the Lagrangian and Eulerian approaches. It uses a fixed grid, while the observer is attached to the fluid particle. At time t, for an observer at location $\mathbf{x}(t)$, the changes of quantities $\mathbf{q}$ can be parametrized as $\mathbf{q}(t, \mathbf{x}(t))$.

The material derivative, $\frac{D\mathbf{q}}{Dt}$, which can be think of as the total derivative, captures the rate of change of the quantities carried by the fluid in the semi-Lagrangian view. It is derived by combining the two viewpoints into

$$\begin{aligned} \frac{D\mathbf{q}(t, \mathbf{x})}{Dt} &= \frac{d}{dt}\mathbf{q}(t, \mathbf{x}(t)) \\ &= \frac{\partial \mathbf{q}}{\partial t} + \frac{d\mathbf{x}}{dt} \cdot \nabla \mathbf{q} \\ &= \frac{\partial \mathbf{q}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{q}. \end{aligned} \tag{3.5}$$

In fluid simulation, in the advection step, because there are no external forces acting on the fluid, the quantity carried by fluid particles will move around in the fluid grid, but should not change in the Lagrangian viewpoint. This means the material derivative should be set to 0. This is equivalent to solving the quantity $\mathbf{q}$ for

$$\frac{\partial \mathbf{q}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{q} = 0. \tag{3.6}$$

Note the velocity is a quantity carried by the fluid particles, and it can be self-advected.

In a semi-Lagrangian viewpoint, since advection is the transportation of quantity by the velocity field, to obtain the quantity $\mathbf{q}_G$ at target location $\mathbf{x}_G$ at time step t, we simply back trace the velocity field to the source location $\mathbf{x}_P$, and use the quantity $\mathbf{q}_P$ carried by the same imaginary particle at the source location at one time step before, $t - 1$. This is illustrated in Figure 3.2.

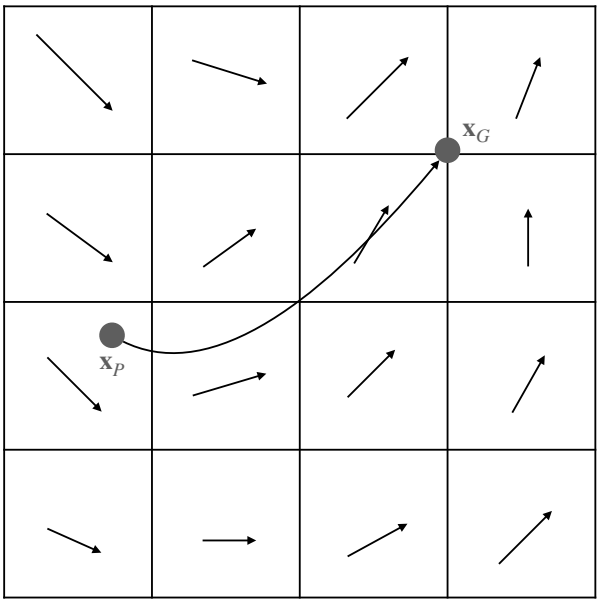


Figure 3.2: Advection by back tracing the particle through the velocity field

With this intuition, solving for advection can be decomposed into two steps. First, we need to find the source location $\mathbf{x}_P$ given the target location $\mathbf{x}_G$, and second, we need to properly determine the quantity $\mathbf{q}_P$ carried by that particle at the source location.

The problem of finding the source location can be categorized as a transient problem. Given the target location $\mathbf{x}_G$ and the ordinary differential equation (ODE) that describes the rate of change of location

$$\frac{\partial x}{\partial t} = \mathbf{u}(\mathbf{x}), \tag{3.7}$$

we would like to go in the reverse direction for Δt amount of time to find the source location $\mathbf{x}_P^{t-1}$.

These transient problems are very well-studied and there are many existing solutions. One of the simplest solutions is to use the "forward" Euler method to go backwards in time. More concretely, this method solves the problem by using the velocity field $\mathbf{u}$ evaluated at $\mathbf{x}_G$ to take one time step backward. Formally, obtaining $\mathbf{x}_P$ using the

"forward" Euler method can be written as

$$\mathbf{x}_P = \mathbf{x}_G - \Delta t \mathbf{u}(\mathbf{x}_G). \tag{3.8}$$

For higher accuracy and stability, we can use the Runge-Kutta method, which is a generalization of the forward Euler method by taking intermediate steps back. Although we did not find this necessary because we focused on low velocity smokes with relatively simple (boxed) boundary conditions, these higher order time integration methods can be integrated into our framework in a straightforward fashion.

The final remaining task is to properly obtain the quantity $\mathbf{q}_P$ carried by the imaginary particle at $\mathbf{x}_P$. Because the particle location may not lie exactly on the discretized grid, we need to interpolate the quantity using the neighbouring particles. In 2D, we achieve this by bi-linear interpolation, and in 3D, we achieve this by tri-linear interpolation.

Finally, combining the time integration (backtracing) step and the interpolation step, we can obtain the quantity $\mathbf{q}_P$ after the fluid is advected. Note that for vector quantities, each component of the vector will be advected separately. This step is completely differentiable as well since all mathematical operations used for both components are differentiable.

3.2.3 External Forces

In fluid simulation, external force fields $\mathbf{f}$ will act on the fluid and change the velocity field $\mathbf{u}$. Depending on the scene and the simulation configuration, the force field is either given as a simulation parameter or computed during each time step. Given the time step size Δt, the amount of change in velocity caused by the external forces can be calculated as $\Delta t \mathbf{f}$, and the velocity update can be written as

$$\mathbf{u}^t = \mathbf{u}^{t-1} + \Delta t \mathbf{f}. \tag{3.9}$$

As for the differentiability, as long as the external force field $\mathbf{f}$ is provided as a constant, or the computation of it is differentiable, the velocity update is differentiable as well.

3.2.4 Pressure Projection

In fluid simulation, the goal of pressure projection is to solve for the pressure field $\mathbf{p}$ that satisfies both the incompressibility constraint and the boundary conditions. Then, the velocity field $\mathbf{u}$ can be updated using the first-order update

$$\mathbf{u}^t = \mathbf{u}^{t-1} - \Delta t \frac{1}{\rho} \nabla \mathbf{p}. \tag{3.10}$$

Staggered Grids

The incompressibility constraint states that the divergence of the velocity field $\mathbf{u}$ should be 0, which means

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{3.11}$$

To compute the divergence of the velocity field, we need to compute the partial derivative of the components of the velocity field with respect to the corresponding spatial coordinates. In a discretized simulation, we compute partial derivatives by central finite differencing. However, if the velocity components are stored at grid centers, we will run into a non-trivial null space problem, where our finite differencing estimation will be 0 while the actual gradient of the velocity field is non-zero. In 1D, this can be illustrated in Figure 3.3. For the piecewise linear function in the figure, the gradient is non zero everywhere and not defined at integers, but at $x = 2$, using finite differencing, we will obtain an incorrect gradient estimate of $\frac{f(3)-f(1)}{2} = \frac{1-1}{2} = 0$.

In order to avoid this non-trivial null space problem, we will adopt a **Staggered Grid**, also known as **Marker and Cell (MAC) Grid**, to store the velocity field [1].

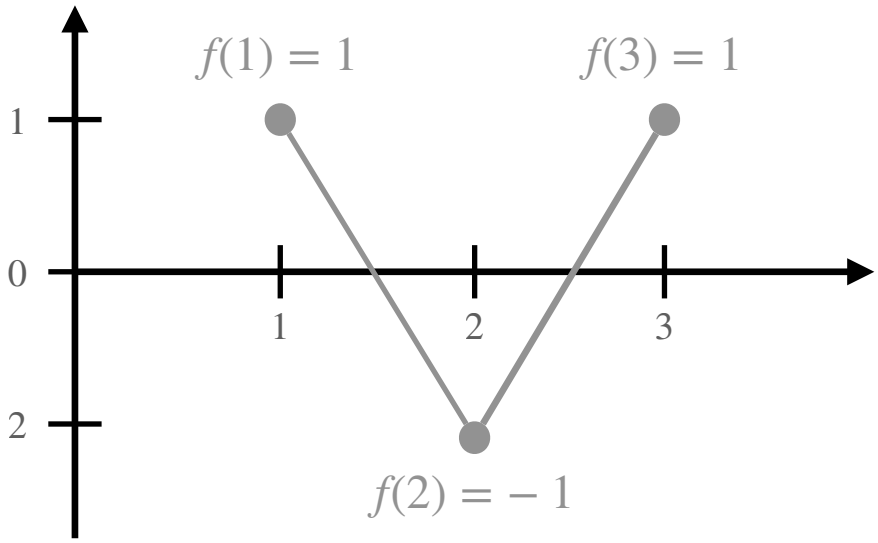
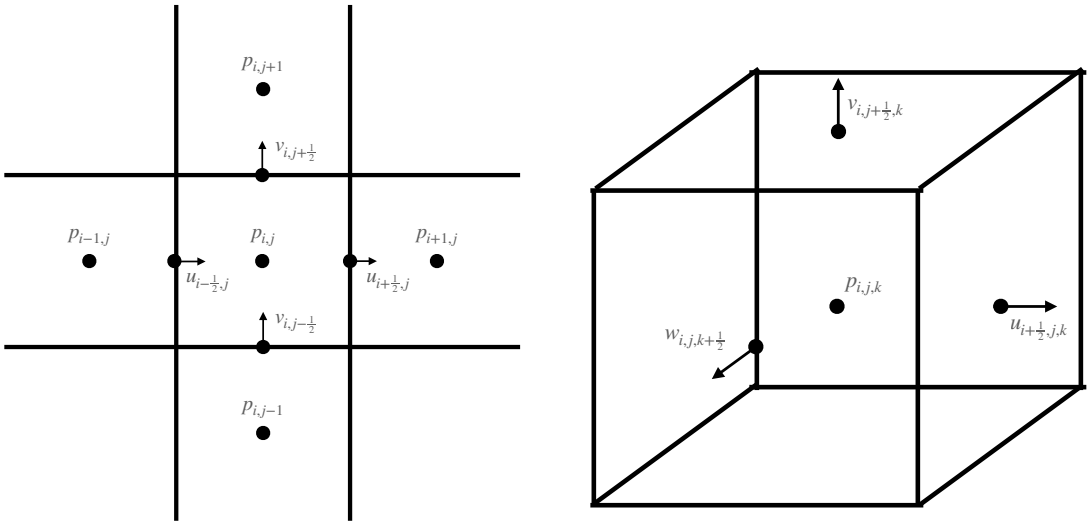


Figure 3.3: Null space problem



(a) 2D staggered grid **(b)** 3D staggered grid

Figure 3.4: Staggered grids

In a 2D staggered grid, the vector components (u, v) are stored at the grid faces with half-indices, while the scalar components such as pressure (p) are stored at the grid centers with integer indices. This is shown in Figure 3.4a.

With staggered grids, taking central differences for the velocity components will give spatial derivatives at the center of the cell. With $\mathbf{u} = (u, v)$, the spatial derivative for the velocity components can be written as

$$\begin{aligned}\left(\frac{\partial u}{\partial x}\right)_{i,j} &= \frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x}, \\ \left(\frac{\partial v}{\partial y}\right)_{i,j} &= \frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta y}.\end{aligned} \tag{3.12}$$

After computing the spatial derivatives, we can evaluate the divergence of the velocities at the center of the grid cells $(\nabla \cdot \mathbf{u})_{i,j}$.

Another advantage of using a staggered grid is that when we evaluate the gradient of the pressure field using central finite differencing, the derivatives will be evaluated at cell face centers, making it easier to compute the velocity update. The spatial derivatives of the pressure field can be written as

$$\begin{aligned}\left(\frac{\partial p}{\partial x}\right)_{i+1/2,j} &= \frac{p_{i+1,j} - p_{i,j}}{\Delta x}, \\ \left(\frac{\partial p}{\partial y}\right)_{i,j+1/2} &= \frac{p_{i,j+1} - p_{i,j}}{\Delta y}.\end{aligned} \tag{3.13}$$

However, the disadvantage of using a staggered grid is that during the advection step, we need to interpolate the velocity components to different locations depending on the quantity being advected. When advecting scalar grids, we need to sample the velocity at grid cell centers, where each sampling point has integer indices $\mathbf{u}_{i,j}$, which can be computed by

$$\mathbf{u}_{i,j} = \left(\frac{u_{i-1/2,j} + u_{i+1/2,j}}{2}, \frac{v_{i,j-1/2} + v_{i,j+1/2}}{2}\right). \tag{3.14}$$

When we use the velocity field to advect itself, we advect each component separately, and for each component, we need to sample the velocity at different locations so that the sampled velocity aligns with the component being advected. For example, when advecting the u component of the 2D velocity field, we need to interpolate the velocity

so that the sampled points align with the data points stored in the u component, which live on the cell face centers where the x component has half-indices and y component has integer indices ($\mathbf{u}_{i+\frac{1}{2},j}$). Similarly, when advecting the v component, we need to sample the velocity at $\mathbf{u}_{i,j+\frac{1}{2}}$. Sampling at these locations can be computed by

$$\begin{aligned}\mathbf{u}_{i+1/2,j} &= \left(u_{i+1/2,j}, \frac{v_{i,j-1/2} + v_{i,j+1/2} + v_{i+1,j-1/2} + v_{i+1,j+1/2}}{4}\right), \\ \mathbf{u}_{i,j+1/2} &= \left(\frac{u_{i-1/2,j} + u_{i+1/2,j} + u_{i-1/2,j+1} + u_{i+1/2,j+1}}{4}, v_{i,j+1/2}\right).\end{aligned} \tag{3.15}$$

In a 3D staggered grid as shown in Figure 3.4b, computing the divergence of the velocity field, the gradient of the pressure field, and sampling the velocity field for advection are very similar to the 2D case. The only difference is that we need to take into account the third dimension.

Boundary Conditions

One of the goals of pressure projection is to enforce boundary conditions. In fluid simulation, if we think about the grid as a voxelized model, as shown in Figure 3.5, then each voxel can be labelled as a fluid (F), solid (S), or empty (E). Then, we can describe two types of boundary conditions: **Dirichlet** and **Neumann** boundary conditions.

The Dirichlet boundary condition is also known as the free surface boundary condition. This boundary condition is enforced on voxel faces between fluid and empty voxels. In Figure 3.5, Dirichlet boundary conditions are marked with blue lines.

The Neumann boundary condition is also known as the solid boundary condition. This boundary condition is enforced on voxel faces between fluid and solid voxels. In Figure 3.5, Neumann boundary conditions are marked with green lines.

Since our work is focused on smoke simulations, the air acts as the fluid, and there will not be empty cells in our simulation, so we do not need to solve for the Dirichlet boundary conditions. Therefore, we will only cover Neumann boundary conditions in this thesis.

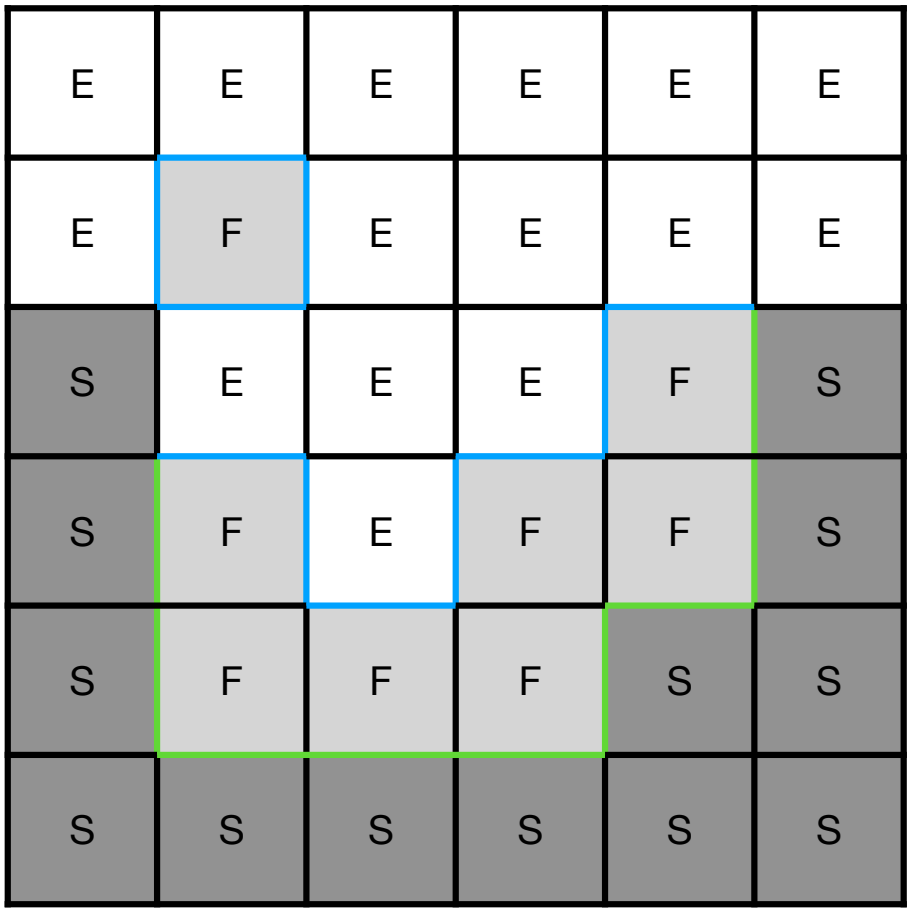


Figure 3.5: Boundary conditions

For solid (Neumann) boundary conditions, we would like to enforce the fact that no fluid should flow into or out of the solid body. Mathematically, this means that in the normal direction, denoted by $\mathbf{n}$, the relative velocity between the fluid and the solid should be zero and can be written as Equation (3.16) and Equation (3.17). Note that for the tangent component, the relative velocity between the fluid and the solid can be non-zero, and the fluid velocity is independent of the solid velocity.

In general, the relationship between the fluid and solid velocities can be written as

$$\mathbf{u} \cdot \mathbf{n} = \mathbf{u}_{solid} \cdot \mathbf{n}. \tag{3.16}$$

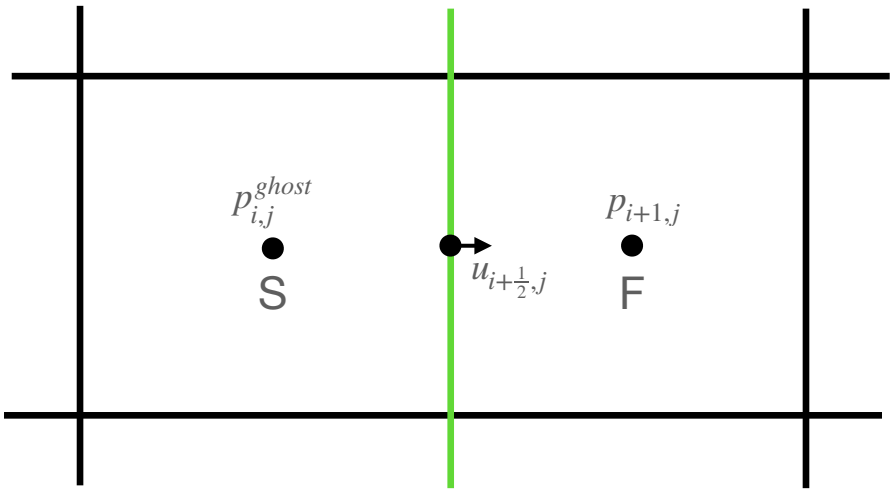


Figure 3.6: Solid boundary example

If we make an additional assumption that the solid is stationary, then the relationship can be simplified to

$$\mathbf{u} \cdot \mathbf{n} = 0. \tag{3.17}$$

To utilize Equation (3.16) and Equation (3.17) to solve for pressure, we will use a **ghost pressure**, p^{ghost} to act as an imaginary pressure value for a solid cell right next to a fluid cell. We will take a 2D solid boundary example of having a solid cell on the left at location (i, j), and a fluid cell on the right at location $(i + 1, j)$. The solid cell will have a ghost pressure of $p_{i,j}^{ghost}$, and we would like to solve for the pressure in the fluid cell $p_{i+1,j}$ so that it satisfies Equation (3.16). This is illustrated in Figure 3.6.

In our example, since the normal direction of the solid boundary is the u component, we can rewrite Equation (3.16), so that in the next time step $(t + 1)$, the updated velocity's u component satisfies

$$u_{i+1/2,j}^{t+1} = u_{i+1/2,j}^{solid}. \tag{3.18}$$

Given the ghost pressure, we can also rewrite the u-component of the velocity update Equation (3.10) as

$$u_{i+1/2,j}^{t+1} = u_{i+1/2,j}^{t} - \Delta t \frac{1}{\rho} \left(\frac{p_{i+1,j} - p_{i,j}^{ghost}}{\Delta x} \right). \tag{3.19}$$

Substituting Equation (3.18) into Equation (3.19) and rearranging, we can derive that the ghost pressure needs to satisfy

$$p_{i,j}^{ghost} = p_{i+1,j} + \frac{\rho \Delta x}{\Delta t} \left(u_{i+1/2,j}^{t} - u_{i+1/2,j}^{solid} \right). \tag{3.20}$$

Note that this ghost pressure constraint can be generalized to the v-component, as well as the 3D case.

The Pressure Equation

In our work, we will assume that the solid is stationary. Therefore, we will proceed under the assumption of Equation (3.17).

We expand the first order velocity update Equation (3.10) into the u-component and v-component as

$$\begin{aligned} u_{i+1/2,j}^{t+1} &= u_{i+1/2,j}^{t} - \Delta t \frac{1}{\rho} \left(\frac{p_{i+1,j} - p_{i,j}}{\Delta x} \right), \\ v_{i,j+1/2}^{t+1} &= v_{i,j+1/2}^{t} - \Delta t \frac{1}{\rho} \left(\frac{p_{i,j+1} - p_{i,j}}{\Delta y} \right). \end{aligned} \tag{3.21}$$

Likewise, we expand the divergence free constraint in Equation (3.2) into the u-component and v-component, and get

$$\nabla \cdot \mathbf{u}_{i,j}^{t+1} \approx \frac{u_{i+1/2,j}^{t+1} - u_{i-1/2,j}^{t+1}}{\Delta x} + \frac{v_{i,j+1/2}^{t+1} - v_{i,j-1/2}^{t+1}}{\Delta y} = 0. \tag{3.22}$$

For simplicity, we assume the grids are squared, meaning $\Delta y = \Delta x$. Substituting Equation (3.21) into Equation (3.22), and rearrange so that the pressure terms are on the left, and the known velocity terms are on the right, we obtain

$$\frac{\Delta t}{\rho} \left(\frac{4p_{i,j} - p_{i+1,j} - p_{i-1,j} - p_{i,j+1} - p_{i,j-1}}{\Delta x^2} \right) = - \left(\frac{u_{i+1/2,j}^{t} - u_{i-1/2,j}^{t}}{\Delta x} + \frac{v_{i,j+1/2}^{t} - v_{i,j-1/2}^{t}}{\Delta x} \right). \tag{3.23}$$

Note that this equation is the finite difference approximation for the Poisson equation for pressure

$$\frac{\Delta t}{\rho}\nabla \cdot \nabla p = -\nabla \cdot \mathbf{u}. \tag{3.24}$$

For cells that are marked as solid, we use the ghost pressure in Equation (3.20) to substitute the corresponding pressure terms in Equation (3.23).

In general, for a cell that is marked as solid, we will modify the pressure equation in Equation (3.23) by reducing the coefficient for the central pressure term by 1 on the left-hand side and removing the pressure term that is solid, and add the velocity term on the right-hand side.

For example, for the central pressure at index (i, j), if the cell on the right $(i + 1, j)$ is marked as solid, then we substitute the ghost pressure in Equation (3.20) into the Equation (3.23), and get

$$\frac{\Delta t}{\rho}\left(\frac{3p_{i,j} - p_{i-1,j} - p_{i,j+1} - p_{i,j-1}}{\Delta x^2}\right) = - \left(\frac{u^t_{i+1/2,j} - u^t_{i-1/2,j}}{\Delta x} + \frac{v^t_{i,j+1/2} - v^t_{i,j-1/2}}{\Delta x}\right) + \left(\frac{u^t_{i+1/2,j}}{\Delta x}\right). \tag{3.25}$$

The Pressure Equation in Vector Form

For each cell in the simulation domain, we can use the above to construct a pressure equation. By combining all these pressure equations, we obtain a system of equations that can be written in vector form as

$$A\mathbf{p} = \mathbf{d}, \tag{3.26}$$

where A is a laplacian coefficient matrix for each cell, and is sparse and symmetric positive definite, as long as the boundary conditions are valid. The vector $\mathbf{p}$ is the unknown pressure values for each cell, and the vector $\mathbf{d}$ is the negative velocity divergences.

This kind of linear systems are often very large, and solving for exact solutions is time consuming. However, in fluid simulation, to obtain a visually pleasing result, we do not need to solve the pressure exactly. Instead, it is desirable to solve for an approximation of the pressure in order to reduce the run time. To achieve this, iterative methods such as the Jacobi, Gauss-Seidel, or conjugate gradient are often used. In our work, we chose to use the conjugate gradient method, which usually converges to an acceptable approximate to the solution within a low amount of iterations. The solver is differentiable as long as the matrix-vector product operator is differentiable as well. However, the convergence tolerance and the number of maximum iterations allowed still needs to be fine-tuned. As the tolerance decreases, the solver will produce a more accurate pressure estimate that enforces the divergence free constraint, but it will also increase the number of iterations it takes to converge. The larger the number of iterations causes more elementary operations to be performed thus increasing the memory requirement for storing a larger computation graph. More importantly, the more iterations, the longer it takes for the solver to run. In our work, we compared different convergence tolerances and maximum iterations in Section 4.1.6, and chose a tolerance of 0.1 and a maximum of 20 iterations for the conjugate gradient solver.

Another important aspect of the iterative solver is the choice of the matrix-vector product operator. The operator should be time-efficient to compute and should not take up too much memory. A naive way to construct the operator is to compute and use the dense matrix A directly. However, for a grid with high resolution of $n \times n \times n$, a dense matrix A will have $n^3 \times n^3$ entries, which is too large to store in memory. One alternative to this is to use the sparse matrix A, where the 0s are not stored in memory. This will reduce the memory usage to around $7n^3$ entries, since there will be n^3 rows, and for each row, the non-zero entries correspond to the neighbors of the cell and the cell itself, and each cell has at most 6 neighbors. This is still very large to store in memory for a high resolution grid, and the matrix-vector product operator will be time-consuming to compute as well. Instead of these options, we take advantage of the fact that the matrix

A is a sparse laplacian matrix, and use convolution filters and paddings to construct the operator. With this approach, the operator is a lot more memory friendly since we only need to store the $3 \times 3 \times 3$ convolution filter, along with the padded simulation grid. The operator is also time-efficient to compute as well since the packages we use, such as PyTorch and Jax, have optimized implementations for convolution operations using vectorization and memory stride management, and even parallelization on GPUs. The details of the convolution implementations are described in Section 4.1.6.

Finally, after solving for the pressure values, we can use them to update the velocities using Equation (3.10). As long as the memory concerns above are addressed, the pressure projection step is differentiable and can be integrated into the pipeline without problem.

3.3 Rasterization Pipeline

In our work, we use a rasterization-based rendering method to render the fluid. For the rasterization and its corresponding gradient computations, we use an existing differentiable rasterizer, NvDiffRast [44]. Prior to rasterization, our work mainly involves vertex transformation and vertex property interpolation. In this section, we will give a quick overview of part of the rasterization pipeline.

In rendering, the 3D scene is usually described by meshes. A mesh is represented using a set of primitives such as triangles, described by vertices and faces. Each vertex will contain information about its position and other attributes such as normals and texture coordinates. Each face will contain information about the indices of the vertices that form the face. The rasterization pipeline takes these object space vertices and the faces as input, and outputs a set of pixels that will be displayed on the screen.

The pipeline consists of the following steps, and each step will be explained in more detail in the following sections.

1. **Vertex Processing** - Transforms object space vertices of the mesh into the clip space by performing a series of transformations including world, look at and projection transformations.

2. **Clipping and Rasterization** - Clips the primitives against the view frustum, and then rasterize them into fragments.

3. **Fragment Processing** - Processes the fragments to determine the visibility and the colour of the pixels.

4. **Display** - Displays the rasterized image onto the screen.

3.3.1 Vertex Processing

In a rasterization pipeline, when a mesh is first defined, the vertex coordinates are defined in model space. The vertex processing stage is in charge of transforming the vertices from the model space into the screen space. The transformation is done by applying a series of transformation matrices including the model/view/projection (MVP) matrices and the viewport transform matrices to the vertices. These transformation matrices are 4x4, and the coordinates of the vertices are represented as 4D homogeneous coordinates, with the fourth component being 1, making the coordinates $[x, y, z, (w = 1)]$. The transformation matrices are defined as follows.

The model matrix M_{model} transforms vertices from the model space into the world space. This matrix is usually arbitrarily defined by artists or programmers.

The view matrix M_{view}, also known as the look-at matrix, transforms vertices from the world space into the camera space. This matrix is defined by the position of the camera $\mathbf{c}$, the position of the target $\mathbf{t}$, and the up vector $\mathbf{u}$. In our work, we use a right-handed coordinate system, where the camera's z-axis points out of the screen, the x-axis points to the right, and the y-axis points up. Using these three vectors, we can define the camera

space coordinate system by

$$\mathbf{z} = \frac{\mathbf{c} - \mathbf{t}}{\|\mathbf{c} - \mathbf{t}\|},$$
$$\mathbf{x} = \frac{\mathbf{u} \times \mathbf{z}}{\|\mathbf{u} \times \mathbf{z}\|},$$
$$\mathbf{y} = \mathbf{z} \times \mathbf{x}.$$

Then, the view matrix is defined as the inverse of the camera-to-world matrix

$$M_{view} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}.$$

The projection matrix M_{proj} transforms the view space into the clip space. Depending on the type of projection, the projection matrix can be defined differently.

For orthographic projection, the projection matrix is defined by the left (l), right (r), bottom (b), and top (t) planes of the view frustum and the near (n) and far (f) planes, and is given by

$$M_{proj\_ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For perspective projection, the projection matrix is defined by the field of view (θ) in radians, aspect ratio (α), and the near (n) and far (f) planes. Using the field of view, we can calculate the top (t), bottom (b), left (l), and right (r) planes using

$$t = n \tan \frac{\theta}{2},$$
$$b = -t,$$
$$r = t\alpha,$$
$$l = -r.$$

Using these planes, we construct the perspective projection matrix using

$$M_{proj\_persp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Note that the model, view and projection matrices can be combined into a single MVP matrix M_{MVP} and be used to transform the vertices $\mathbf{v}$ from the model space into the clip space by

$$\mathbf{v}_{clip} = M_{MVP}\mathbf{v}_{model} = M_{proj}M_{view}M_{model}\mathbf{v}_{model}.$$

Then, by dividing the x, y, and z components of the vertices by the w component, called the perspective division, we can transform the vertices from the clip space into the canonical view volume space or normalized device coordinates (NDC) space. The NDC space is defined by the range $[-1, 1]$ for all three axes.

Finally, the view port transform transforms the NDC space into the screen space. Given the width (n_x) and height (n_y) of the screen in pixels, the viewport transform matrix is defined as

$$M_{view\_port} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

In summary, a 3D coordinate of a vertex $\mathbf{v}$ in the model space can be transformed into the 2D coordinates in screen space by combining all the transformations

$$\mathbf{v}_{screen} = \frac{M_{view\_port}M_{MVP}\mathbf{v}_{model}}{w}.$$

Note that the $\mathbf{v}_{screen}$ will have 4 coordinates, $(x_s, y_s, z_c, 1)$, where (x_s, y_s) are coordinates in the screen space, and z_c is the depth value of the vertex in the clip space that is stored for the depth buffer.

3.3.2 Rasterization

The rasterization step converts the continuous geometric primitives into discrete pixel fragments. For each primitive in the scene, the rasterization process computes the pixels covered by the primitive and generates a fragment for each pixel. Barycentric coordinates are used to determine whether a pixel is within the primitive. For each pixel, interpolated attributes are computed based on the barycentric coordinates and the attributes of the vertices of the primitive. In our case, the interpolated attribute mainly refers to the absorption coefficient of the smoke.

3.3.3 Fragment Processing

Fragment processing composes of two main tasks: visibility determination and shading.

The visibility problem determines whether a fragment is visible or not. A z-buffer, as known as the depth buffer, is used to perform the depth test. The depth value is computed in the vertex transformation stage and stored as the depth coordinate z_c in the screen space coordinates. When drawing the primitives onto the screen, for each pixel, the depth value of the pixel is compared with the depth value stored in the depth buffer. If the depth value of the pixel is smaller than the depth value stored in the depth buffer, the pixel is visible and the depth value in the depth buffer is updated. Otherwise, the pixel is occluded and is discarded.

Another important task of the fragment processing stage is shading, which means computing the colour of the pixel. In our case, we compute a gray-scale colour for each pixel based on the absorption of the smoke; see Section 4 for more detail. Hence, we will not cover shading in this thesis.

Due to its if-statement like branching logic when performing the depth test, the task of determining the visibility of a fragment is discontinuous and thus non-differentiable. This is one of the main challenges in designing a differentiable renderer. As described in Section 2.4, there exists different techniques and frameworks to address this problem. Among them, we use an existing differentiable rasterizer, NvDiffRast [44], in our framework. The rasterizer avoids this problem and provides gradient against the vertex information by analytically post-process edge antialiasing, which means computing and interpolating the pixel colour blending depending on the pixel location of the triangle edges during the antialiasing operation.

3.4 Smoke Absorption

In this section, we will introduce the absorption of the smoke, which is used to compute the outgoing light radiance after a light beam traverses the smoke medium. We will first introduce the Beer-Lambert law for computing the outgoing light radiance that traverses out of a smoke medium, then we introduce the absorption coefficient used in the Beer-Lambert law and how it is computed.

3.4.1 Beer-Lambert Law

When a beam of light traverses through a smoke medium with incoming radiance L_i, it will get absorbed by the smoke, and the outgoing radiance L_o will decrease due to this absorption. This is illustrated in Figure 3.7.

This phenomenon is modelled by the Beer-Lambert law. The Beer-Lambert law defines the transmittance of light through a medium. The transmittance T is defined as the ratio of the outgoing radiance L_o to the incoming radiance L_i.

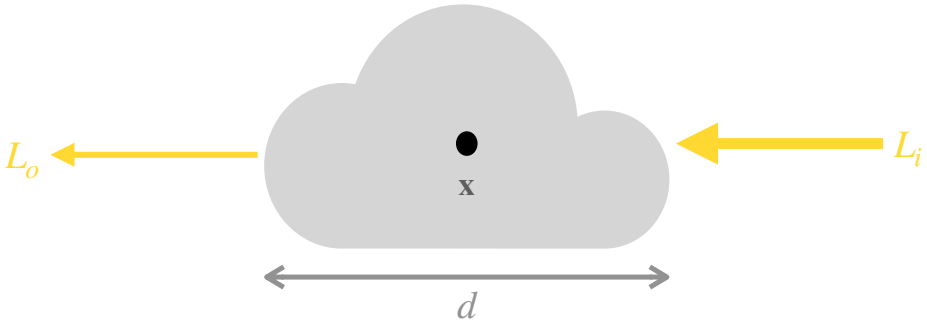


Figure 3.7: Illustration of light traversing through a smoke medium

For a homogeneous medium, the absorption coefficient does not change throughout the medium, and the Beer-Lambert law is defined as

$$T = \frac{L_o}{L_i} = e^{-\sigma_a d}, \tag{3.27}$$

where σ_a is the absorption coefficient, and d is the length of the light path in the medium. This can be rearranged and solved for the outgoing radiance L_o by

$$L_o = L_i e^{-\sigma_a d}. \tag{3.28}$$

For a heterogeneous smoke medium, the smoke absorption coefficient varies depending on the sampling location in the medium. The Beer-Lambert law can be written in the continuous form

$$L_o = L_i e^{\int_0^d -\sigma_a(x)dx}, \tag{3.29}$$

where x is the a point on the light path and σ_a is a function of absorption coefficient that depends on x.

By assuming that the absorption coefficient σ_a is constant along each small segment of the light path, we can discretize and rewrite the integral form in Equation (3.29) as

$$\begin{aligned} L_o &= L_i e^{\sum_{i=0}^{n} -\sigma_{a,i}\Delta x} \\ &= L_i \prod_{i=0}^{n} e^{-\sigma_{a,i}\Delta x}, \end{aligned} \tag{3.30}$$

where $\sigma_{a,i}$ is the absorption coefficient at the i-th segment of the light path, and Δx is the length of each segment.

3.4.2 Absorption Coefficient

The Beer-Lambert law uses the absorption coefficient σ_a to compute the transmittance of the light through the smoke medium. In our work, because we work with a simulated smoke grid, and at each cell, the amount of smoke s is different, the absorption coefficient for each cell will be different as well. We will work with a discretized heterogeneous smoke medium and need to compute the absorption coefficient for each segment of the light path.

To compute the absorption coefficient σ_a given the smoke density s for a patch of smoke in ppm, we can compute the absorption coefficient σ_a for the patch using the light extinction coefficient equation

$$\sigma_a = K_m s, \tag{3.31}$$

where K_m is the extinction coefficient of the medium, which is a constant value defined by the user. In Section 4.2, we detail how we convert the smoke grid into a triangle mesh and how the smoke density stored in each grid cell is converted into the per-vertex absorption coefficient.

Chapter 4

Methodology

We will present our methods of implementing the fully differentiable fluid pipeline in this section. The goal of this framework is to solve for simulation inverse problems by optimizing the 3D parameters of the fluid simulator and the renderer using the gradient of 2D image losses. First, we will introduce our implementation of the differentiable fluid simulator, including the data structures we use and our variation of the stable fluids [10] algorithm. Then, we will describe the differentiable fluid renderer. Since NvDiffRast [44] does most of the heavy lifting for the rasterization pipeline, we will focus on discussing how the simulated 3D smoke grid is converted into geometry mesh and how the absorption of light is computed. Finally, we describe the optimization process used in the framework to solve for inverse problems.

4.1 Differentiable Fluid Simulator

Our project implements the differentiable fluid simulator based on the stable fluids [10] algorithm described in Section 3.2. We use the Python language and implement both a Jax [33] version and a PyTorch [40] version of the simulator. Both implementations can perform forward simulation and backward differentiation. In the following sections, we will first describe the data structure used in the simulator, and then the

specific implementation of the stable fluids algorithm, and finally we will explain some acceleration techniques used and give a brief comparison between the Jax and PyTorch implementations.

4.1.1 Data Structures

Before detailing the implementation of the algorithm, we will first introduce the data structures used for storing the grid data in the simulator.

For the simulation, we define the spatial resolution in the x, y and z axes as nx, ny and nz respectively. They represent the number of cells in the grid along each axis. On top of the spatial resolution, we also include a batch size B for our simulation to support batched operations.

For scalar grids such as the smoke density grid and the temperature grid, we will use a centered grid that stores values at the center of the cell. The Centered Grid is represented by a 4D tensor of shape (B, nx, ny, nz).

For vector grids such as the velocity grid and the force grid, we will use a staggered grid that stores values at the cell face centers, described in Section 3.2.4. The Staggered Grid is represented by a tensor of shape $(B, 3, nx+1, ny+1, nz+1)$. The second dimension corresponds to the (u, v, w) components of the vector field. For each component, the corresponding spatial dimension will have one more value stored compared to the rest of the two because the values are stored at the cell face centers. Then, for the rest of the two spatial dimensions, empty values are padded to fit the dimension of the tensor. For example, given a vector field $\mathbf{u}$ of shape $(B, 3, nx + 1, ny + 1, nz + 1)$, we query the u-component by setting $u = \mathbf{u}[:, 0, :, :, :]$. Then, the u-component of the vector field $\mathbf{u}$ is of shape $(B, nx + 1, ny + 1, nz + 1)$. For the u-component, index $[b, i, j, k]$ represents the $(b, i - 1/2, j, k)$ coordinate. Hence, the valid shape of the u-component is $(B, nx + 1, ny, nz)$. Programmatically, stacking the vector components together will bring significant speedup to later computations. To do so, the shapes of the components must be consistent, so in the u-component example, to make the shape consistent with

Table 4.1: Non-learnable configuration parameters

Parameter	Functionality
$time\_steps$ (int)	The number of time steps to simulate
Δt (float)	The time step size
$size\_x, size\_y, size\_z$ (floats)	The physical size of the grid
nx, ny, nz (integers)	The spatial resolution of the grid
$T_{ambient}$ (float)	The ambient temperature
g (vector of 3 floats)	The gravitational acceleration
k_max (integer)	The maximum iterations for the pressure solve
ϵ (float)	The convergence tolerance for the pressure solve

the rest of the two components, we take the ceiling of each spatial dimension of the shapes, and pad the u-component with an extra column and an extra channel so that it has shape $(B, nx+1, ny(+1), nz(+1))$. The similar logic applies to the v-component and the w-component as well.

An alternative to the using arrays to store staggered grids is to use custom objects and store each component as a separate array as the object's property. This approach might sound more intuitive and easier to debug, but since we would like to use Jax's and PyTorch's JIT compilation to speed up the simulation, and only functional programming works with JIT and object-oriented programming is not supported, we have to use arrays to store the staggered grids.

4.1.2 Simulation Parameters

The parameters for our simulation can be separated into non-learnable configuration parameters and learnable simulation parameters. Table 4.1 summarizes the list of non-learnable parameters used in the simulator. These parameters are fixed throughout the simulation and are not optimized. Table 4.2 summarizes the list of learnable simulation parameters used in the simulator. During learning, any subset of these parameters can be optimized.

Table 4.2: Learnable simulation parameters

Parameter	Functionality
s (Centered Grid of floats)	The smoke marker concentration grid
$\mathbf{u}$ (Staggered Grid of floats)	The velocity grid
$\mathbf{x}_{inflow}$ (vector of 3 floats)	The spatial location of the inflow source
r_{inflow} (float)	The radius of the inflow source
s_{inflow} (float)	The concentration of the inflow source
T_{inflow} (float)	The temperature of the inflow source
$\mathbf{f}$ (vector of Staggered Grids of floats)	The acceleration caused by external forces

4.1.3 Advection

The first step of each fluid solve is the advection step. As described in Section 3.2.2, the advection step consists of backtracing the velocity field and interpolating the values from the previous time step.

First, the velocity field $\mathbf{u}_{sampled}$ is sampled at different locations depending on the type of grid being advected. For scalar fields such as smoke concentration s and temperature T, the velocity is sampled at the center of the cell. For vector fields such as velocity $\mathbf{u}$, for each component of the vector field, the velocity is sampled at different cell face centers. The sampling scheme follows Equation (3.15) in Section 3.2.4.

Then, we backtrace the velocity field to find the locations at the previous time step. A naive implementation of this is to use a for loop to iterate through each cell and backtrace the velocity field. However, this is extremely inefficient because Python is an interpreted language and its for-loops are very slow. Instead, throughout our implementation, we would like to use vectorized and library built-in operations as much as possible to speed up our simulation.

In the context of backtracing, we first create a grid coordinate system representing the locations of each cell $\mathbf{x}_G$. For this step, we use functions such as `torch.linspace` and `torch.meshgrid`. These coordinate systems are represented as a 5D tensor of shape $(B, 3, nx, ny, nz)$, where the second dimension corresponds to the x, y, z components. Using the grid coordinate system and the sampled velocity $\mathbf{u}_{sampled}$, the backtraced

Algorithm 1 Advection

1: **procedure** ADVECTION($\mathbf{q}^t, \mathbf{u}^t, \Delta t$)
2: $\quad \mathbf{u}_{sampled} = \text{SAMPLE\_VELOCITY}(\mathbf{q}^t, \mathbf{u}^t)$
3: $\quad \mathbf{x}_G = \text{CREATE\_COORDINATE\_SYSTEM}(\mathbf{q}^t)$
4: $\quad \mathbf{x}_P = \mathbf{x}_G - \Delta t \mathbf{u}_{sampled}$
5: $\quad \mathbf{q}^{t+1} = \text{TRILINEAR\_INTERPOLATION}(\mathbf{q}^t, \mathbf{x}_P)$
6: $\quad$ **return** $\mathbf{q}^{t+1}$

locations $\mathbf{x}_P$ are computed using the forward Euler integration scheme

$$\mathbf{x}_P = \mathbf{x}_G - \Delta t \mathbf{u}_{sampled}.$$

With the backtraced locations $\mathbf{x}_P$, the values corresponding to these locations $\mathbf{q}_P$ are then interpolated using tri-linear interpolation. Different packages have different vectorized implementations for tri-linear interpolation. In PyTorch, this is achieved using `torch.nn.functional.grid_sample`. In Jax, the equivalent function for tri-linear interpolation is `jax.scipy.ndimage.map_coordinates`.

In general, to advect a quantity $\mathbf{q}^t$ (either a scalar grid or the components of a vector grid) at time step t to get the quantity at the next time step $t + 1$, the advection step can be described by Algorithm 1.

4.1.4 Inflow Injection

After advection, the next step is to inject smoke and temperature into the scene, so that the smoke forms a continuous plume throughout the simulation time steps. The inflow is injected in the form of a sphere mask described by the inflow source location $\mathbf{x}_{inflow} = (x_{inflow}, y_{inflow}, z_{inflow})$ and the inflow radius r_{inflow}. The amount of inflow and the temperature being injected into the scene is described by the inflow concentration s_{inflow} and the inflow temperature T_{inflow} respectively.

One important note is that we need to be careful of how the inflow sphere mask is created so that the differentiability of the inflow parameters is not broken, as described bellow.

One common method to create a sphere mask is to use a sphere function

$$f_{sphere}(x,y,z) = \begin{cases} 1 & \text{if } (x-x_{inflow})^2 + (y-y_{inflow})^2 + (z-z_{inflow})^2 \leq r_{inflow}^2 \\ 0 & \text{otherwise} \end{cases},$$

where if (x, y, z) lies inside the sphere, the mask value is 1, otherwise the value is 0. The problem with this method is that the function is not differentiable at the boundary of the sphere because of the discontinuity where the function jumps from 0 to 1. This causes the gradient to be undefined at the boundary, and the optimization algorithm will not be able to learn the inflow location parameter.

To solve this problem, we use a differentiable sphere function with the edge smoothed out using the $\tanh$ function. The function is defined as

$$f_{sphere}(x,y,z) = \frac{1}{2}\left(1 + \tanh\left(\sqrt{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2} - r\right)\right). \tag{4.1}$$

Note that the part within the $\tanh$ function is very similar to the discontinuous sphere function above, and it encompasses information regarding whether a point in space lies within the sphere. The $\tanh$ function is a well-known smoothing function that is commonly used in machine learning to deal with discontinuities. In our case, it helps make the sphere edges continuous. Finally, we rectify the range of the function to between 0 and 1, so that it is easier to apply it to the smoke or temperature directly. This function is differentiable everywhere, and the gradient can be computed using the chain rule. A visualization of the function in a 2D slice can be shown in Figure 4.1. Note that an alternative to the $\tanh$ function is the sigmoid function, which is also commonly used in machine learning. However, given the same function outputs, the $tanh$ function has a

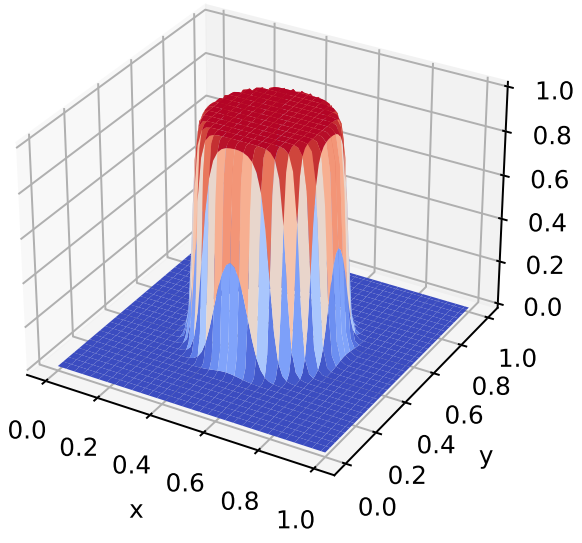


Figure 4.1: Visualization of a differentiable sphere function in a 2D slice

Algorithm 2 Inflow Injection

procedure INFLOW_INJECTION($s^t, T^t, \mathbf{x}_{inflow}, r_{inflow}, s_{inflow}, T_{inflow}$)
  I = CREATE_CIRCLE_MASK($\mathbf{x}_G, \mathbf{x}_{inflow}, r_{inflow}$)
  $s^{t+1} = s^t + s_{inflow} I$
  $T^{t+1} = T^t + T_{inflow} I$
  return s^{t+1}, T^{t+1}

steeper gradient than the sigmoid function, which in our case, helps with the optimization process.

With the differentiable sphere mask creation function, the operation of injecting smoke and temperature into the scene is differentiable as well, since the operation includes only addition and multiplication. The inflow injection step can be described by Algorithm 2.

4.1.5 External Forces

In our work, the external forces applied to the smoke body consist of two parts. One is the user-defined external forces, and the other is the buoyancy force that makes the smoke rise and form a plume. The two forces are combined and applied to the velocity grid at the external forces computation step.

The user-defined external forces take the form of a tensor of Staggered Grids throughout the temporal dimension. At each time step t in the simulation, the corresponding external force $\mathbf{f}^t$ is applied directly to the velocity grid $\mathbf{u}^t$ using the formula

$$\mathbf{u}^{t+1} = \mathbf{u}^t + \Delta t \mathbf{f}^t.$$

The buoyancy force is computed using the Boussinesq approximation. It takes two factors into account - temperature and density.

At the beginning of our simulation, we assume an ambient temperature $T_{ambient}$ in our scene. As smoke is being injected, the inflow location will also introduce heat into the scene. This is represented by the inflow temperature T_{inflow}. We track the temperature field as a scalar field that is being advected in the advection step. After advection and injection, we also want to update the temperature field to account for the heat dissipation or diffusion modelled by the Newton law of cooling [2, 51], which can be expressed as

$$T^{t+1} = T^t + (T^t - T_{ambient})(1 - \exp(-\Delta t)).$$

The difference between the temperature carried by the smoke and the ambient temperature causes hot smoke to float and cool smoke to sink, thus the higher the temperature carried by the smoke, the greater the smoke will experience a buoyancy force in a positive y-direction.

On the other hand, because the air carrying smoke is denser than the air without any smoke, they will tend to fall down due to gravity. This means that for a constant smoke volume, the more concentrated the smoke, the heavier it is, and the more gravitational force will be applied to the smoke in a negative y-direction.

The buoyancy force is computed by combining the two factors together. It is then applied to the velocity grid $\mathbf{u}$ using the Boussinesq approximation [9, 11]

$$\mathbf{u}^{t+1} = \mathbf{u}^t + \Delta t \left(\alpha s - \beta T^t\right) \mathbf{g},$$

Algorithm 3 External Forces

1: **procedure** EXTERNAL_FORCES($\mathbf{u}^t, \mathbf{f}^t, s^t, T^t, \mathbf{g}, \Delta t$)
2: $\quad \mathbf{u} = \mathbf{u}^t + \Delta t \mathbf{f}^t$
3: $\quad T^{t+1} = T^t + (T^t - T_{ambient})(1 - \exp(-\Delta t))$
4: $\quad \mathbf{u}^{t+1} = \mathbf{u} + \Delta t \left(\alpha s - \beta T^t\right) \mathbf{g}$
5: $\quad$ **return** $\mathbf{u}^{t+1}$

where α and β are user defined constants.

The external forces step can be described by Algorithm 3. Note that in Section 3.2.3, we mentioned that the differentiability of the external force step is dependent on how the force is generated. In our method of generating the external forces, both the Newton law of cooling and the Boussinesq approximation are continuous and differentiable, thus making the external forces step differentiable as well.

4.1.6 Pressure Projection

After advection and applying external forces, the final step of the stable fluids algorithm is to apply pressure projection to make the fluid incompressible and conform with the boundary conditions. In this section, we describe our implementation of the pressure projection step. We will first describe the boundary conditions for our framework and how we construct our linear system. Then, we will describe our implementation of the conjugate gradient solver that is used to solve the linear system.

Boundary Conditions

In our framework, since we work with smoke, we will only work with Neumann boundary conditions or solid boundary conditions. In addition, we will also assume that we are working with a closed cubed domain with no obstacles in the scene. This assumption is made to simplify the implementation of the linear system construction.

Extending the 2D per-cell pressure equation Equation (3.23) from in Section 3.2.4 to 3D, to construct the linear system, for each pressure $p_{i,j,k}$, using the neighbouring pressure values and the divergence of the velocity field $\mathbf{u} = (u, v, w)$, we can construct a linear

system

$$\frac{\Delta t}{\rho}\left(\frac{6p_{i,j,k}-p_{i+1,j,k}-p_{i-1,j,k}-p_{i,j+1,k}-p_{i,j-1,k}-p_{i,j,k+1}-p_{i,j,k-1}}{\Delta x^3}\right)= \\ -\left(\frac{u^t_{i+1/2,j,k}-u^t_{i-1/2,j,k}}{\Delta x}+\frac{v^t_{i,j+1/2,k}-v^t_{i,j-1/2,k}}{\Delta x}+\frac{w^t_{i,j,k+1/2}-w^t_{i,j,k-1/2}}{\Delta x}\right). \tag{4.2}$$

In the matrix-vector form, this can be expressed as

$$\mathbf{Ap}=\mathbf{d}. \tag{4.3}$$

Note that for cell (i,j,k), the coefficient of the pressure terms on the left is the equivalent of convolving a 3D Laplacian kernel with the $3\times3\times3$ neighbouring pressure terms, where the kernel is defined by

$$kernel=\left[\begin{bmatrix}0&0&0\\0&-1&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&-1&0\\-1&6&-1\\0&-1&0\end{bmatrix}\begin{bmatrix}0&0&0\\0&-1&0\\0&0&0\end{bmatrix}\right]. \tag{4.4}$$

Then, the operation of $\mathbf{Ap}$ can be expressed as by a function that encodes the convolution between $kernel$ and the pressure field $\mathbf{p}$.

For the boundary conditions, as described in Section 3.2.4, we need to make modifications to both the left-hand side and the right-hand side of Equation (4.2).

For the left-hand side, because we assume a closed boxed domain as our solid boundary condition when we apply the ghost particle updates for the solid surfaces at the edges, it is equivalent to replacing the solid boundary cells with the same value as the center cell that is next to the boundary. By doing so, the equation on the left will not contain the pressure term for the solid cells, and the coefficient of the center pressure will be reduced by one.

As for the right-hand side of the equation, before computing the divergences, we need to account for velocity in the normal direction to the boundary surfaces. To do so, for

the boundary cells, we simply set the velocity component that is perpendicular to the boundary to be zero. In our case, this is equivalent to setting the first and the final element of the velocity component that is perpendicular to the edges of our domain box to zero. Take the u-component for example, the boundary cell faces are vertical cell faces that point perpendicular to the u-direction, and they are located at the cells with $i = 0$ and $i = n_x$. Then, for all $j \in [0, n_y]$ and $k \in [0, n_z]$, we set $u_{0,j,k} = u_{n_x,j,k} = 0$. In Python, this can be done with `u[:, [0, -1], :, :] = 0`.

After the edge normal velocities have been set to zero, computing the divergence of the velocity field with the Staggered Grid data structure is fairly straightforward, as described in Section 4.1.1. We will denote the computed divergence vector as variable $\mathbf{d}$.

Given the left-hand side matrix-vector multiplication operator `A_operator` and the right-hand side velocity divergence vector, we can now solve the linear system using the conjugate gradient solver. With Jax and PyTorch, the convolution operation comes built-in and differentiable, and hence the `A_operator` is differentiable as well.

Conjugate Gradient Solver

The conjugate gradient solver is a well-known iterative solver that is used to solve linear systems of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a symmetric positive definite matrix. With a good pre-conditioner, it can usually converge to a pressure estimate that produces a visually acceptable simulation in just a few iterations. In our work, we did not use any pre-conditioner since we put more focus on the differentiability instead of the performance of the solver. As described in Section 3.2.4, choosing the residual tolerance and the maximum number of iterations for the solver is a trade-off between the accuracy of the solution and time along with memory requirement. Without fine tuning the maximum iteration or the residual tolerance, if the residual tolerance is too low or the maximum iterations allowed is too high, the solver converges to an accurate solution, but the runtime becomes too long and the memory required to store the computation graph also increases. On the other hand, if the residual tolerance is too high or the maximum

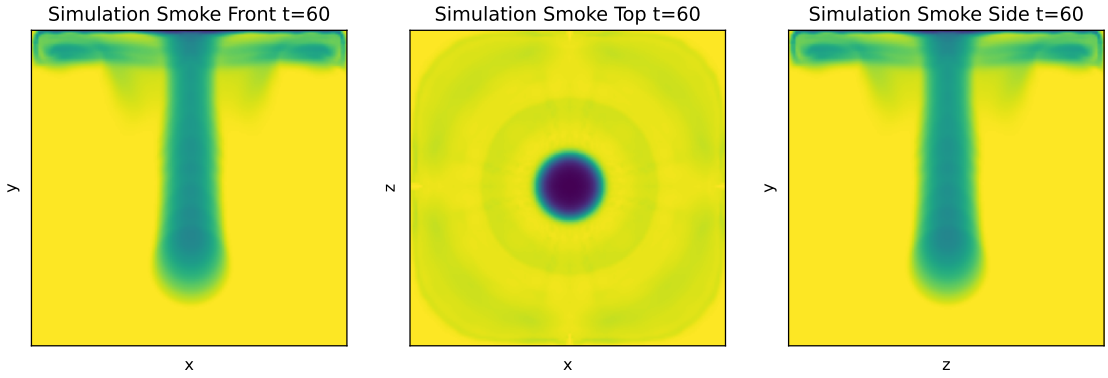
Table 4.3: A comparison of the simulation performance produced using different conjugate gradient solver settings combination.

Iterations	Tolerance	Runtime (s)	Memory (GB)
10	1.00	14.4	2.0588
20	0.10	29.3	2.2248
50	0.01	40.9	2.5014

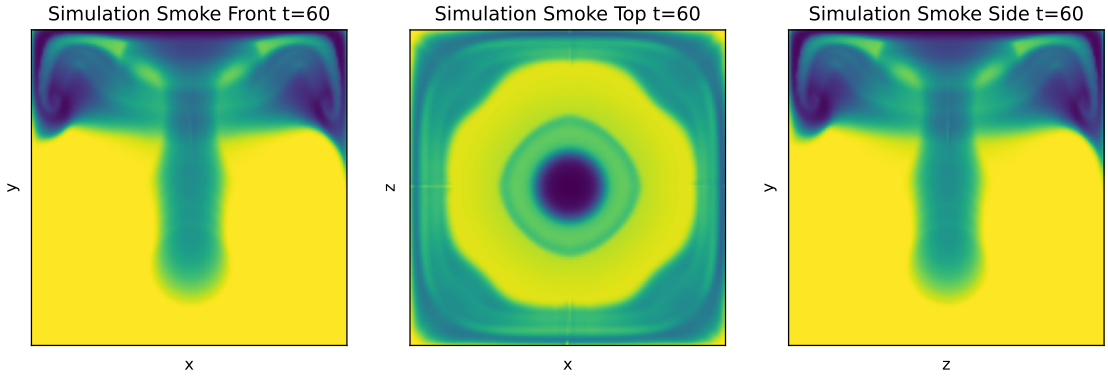
iterations allowed is too low, the solution will be inaccurate and a lot of local details in the simulation will be lost.

To determine the optimal setting for the residual tolerance and the maximum number of iterations, we ran the simulation on a $128 \times 128 \times 128$ grid for 60 time steps with different settings combinations and record both the visual results and the performance of the simulator. We will illustrate with a few examples in this section. Figure 4.2 and shows the result of the simulated final states rendered using a simple absorption scheme described in Section 5.1.1. Table 4.3 records the corresponding performance of the simulator with these different settings combinations. Figure 4.2a shows the result of the simulation using 10 iterations and a residual tolerance of 1.0. We can see that the simulation has a relatively poor result compared to the other configurations because there are less details of the smoke and a lot of unwanted dissipation exists. The simulation took relatively less time to converge because of this low iteration allowance and the high tolerance. Figure 4.2b shows the result of the simulation using 20 iterations and a residual tolerance of 0.1. We observe that compared to the previous result, the simulation produced much more details and the smoke looks more realistic. Compared to before, the simulation took longer to converge and also requires more memory. Figure 4.2c shows the result of the simulation using 50 iterations and a residual tolerance of 0.01. This is the most detail-rich simulation among all the configurations. However, it took the longest time to converge and the memory required to store the computation graph is also the largest.

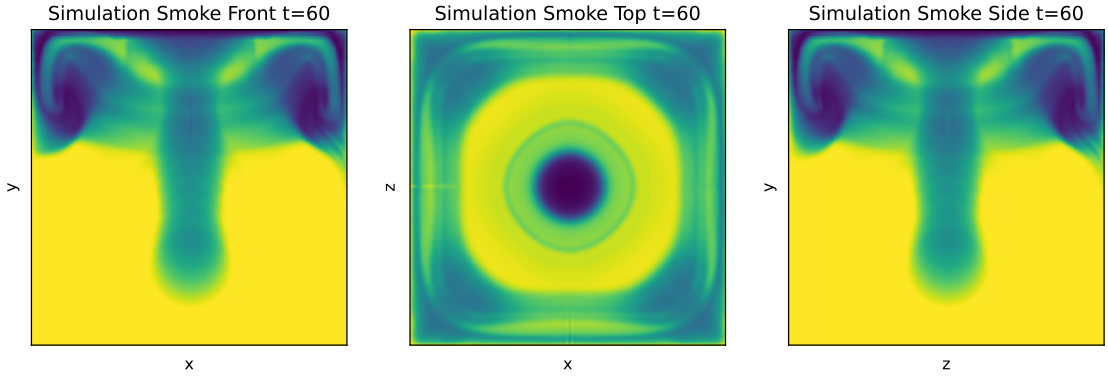
After experimenting with these different conjugate gradient solver settings, we chose to use a tolerance of $\epsilon = 0.1$ and a maximum iteration of 20, because this setting



(a) Rendered final frame of the simulation using 10 iterations and a residual tolerance of 1.0



(b) Rendered final frame of the simulation using 20 iterations and a residual tolerance of 0.1



(c) Rendered final frame of the simulation using 50 iterations and a residual tolerance of 0.01

Figure 4.2: A comparison of the simulation result produced by different conjugate gradient solver settings combination. Yellow indicates light and blue indicates absorption by the smoke

combination gives a visually convincing result while keeping the number of iterations required for the solver to reach the tolerance low. Naturally, as the grid resolution increases, the runtime and memory requirements also increase due to the larger degrees of freedom.

For the Jax package, the conjugate gradient solver comes built-in with the package. The advantages of the Jax implementation are that it is both fast and memory-friendly. The implementation is written with XLA (accelerated linear algebra) support and the iterations are executed in low-level C++-like for-loops, making the implementation faster. It also comes bundled with a custom gradient computation for the solver, so that the gradient of the pressure with respect to the inputs can be computed by solving the gradient using the same iterative solver pass. By doing so, there is no need to store all elementary operations used in the CG solver in the operation tape, thus saving memory.

As for the PyTorch package, the conjugate gradient solver does not come with the package, and we coded our version of the implementation. The implementation uses Python for-loops, so it is considerably slower compared to the Jax built-in implementation. In addition, because the gradient computation is not custom, the gradient must be computed by storing all operations in the CG solver in the tape and traversing the tape backwards when computing gradients. This makes the gradient computation more memory-costly than Jax.

Pressure Projection Algorithm

With the boundary conditions and the conjugate gradient solver described, we can now describe the pressure projection step along with the velocity update. The pressure projection step can be described by Algorithm 4.

Algorithm 4 Pressure Projection

procedure PRESSURE_PROJECTION($\mathbf{u}^t, A\_operator, \Delta t$)
    $\mathbf{u} =$ ADJUST_VELOCITY($\mathbf{u}^t$)
    $\mathbf{d} =$ COMPUTE_DIVERGENCE($\mathbf{u}$)
    $\mathbf{p} =$ CONJUGATE_GRADIENT($A\_operator, \mathbf{d}, k\_max, \epsilon$)
    $\mathbf{u}^{t+1} = \mathbf{u}^t - \frac{\Delta t}{\rho} \nabla \mathbf{p}$
    return $\mathbf{u}^{t+1}$

4.1.7 Jax and PyTorch Backends

In this section, we compare the implementation of the fluid simulation using Jax and PyTorch backends. We describe the similarity and differences in both the differentiability and acceleration techniques used by the two packages.

Differentiability

As described in Section 3.2 and the sections above, each step of the simulation, including advection, inflow injection, external forces and pressure projection are all differentiable. The main concerns regarding the differentiability and the memory consumption, such as the sphere mask creation, external force computation and pressure projection have all been addressed. This means that with a autodiff package such as Jax and PyTorch, with their elementary operation overloading and back propagation abilities, we are able to compute the gradients of the simulation outcome with respect to the inputs.

One main difference between the two packages is that Jax provides built-in conjugate gradient solver, along with custom gradient computations for the solver. They utilize the fact that to compute the gradient is equivalent to passing the upstream gradient into the same iterative solver pass. The custom gradient definition eliminates the need to store all elementary operations in the operation tape and reduces memory consumption. Hence, regardless of the number of iterations it takes for the solver to converge, the amount of memory used for Jax to compute the gradient is the same. On the other hand, defining custom gradient computations in PyTorch is less accessible, and we were not able to implement this due to time constraints. For this reason, the gradients must be computed

by storing all operations in the CG solver in the tape and traversing the tape backwards when computing gradients.

Acceleration

Python is an interpreted language, and it is not as fast as compiled languages such as C++. This disadvantage impacts the runtime when we use loops for repeating multiple simulation time steps and iteratively solving for the pressure in the CG solver. One mitigation to this is the just-in-time (JIT) compilation method. JIT works by tracing or scanning the Python functions to convert them into lower-level interpreted representations (IRs) and compiling the IRs into machine code before the next execution. This method significantly speeds up the simulation. However, even though both Jax and PyTorch support JIT compilation, the performance and the extent of JIT support for the two packages are still very different.

For Jax, both forward and backward passes can be JIT compiled, thus accelerating the learning process the most. For PyTorch, only forward passes can be JIT compiled, and the backward process must be executed in uncompiled Python code. This makes the learning process much slower than Jax's. However, compared to Jax, PyTorch requires fewer modifications to the plain Python code to make it JIT compilable. Jax requires special XLA syntax to substitute for for-loops and while-loops, making the implementation effort non-trivial.

To illustrate the Jax XLA syntax, we show the top level simulation pseudocode written in both Python and Jax's JIT compatible syntax in Algorithm 5. We abstract away a function STEP that takes in the current velocity field $\mathbf{u}^t$ and returns the velocity field at the next time step $\mathbf{u}^{t+1}$. In the Python implementation, we use a for-loop to iterate over the simulation time steps. When JIT compiling with PyTorch, this syntax works without problem. However, in the Jax implementation, we use the JAX.LAX.SCAN function to substitute for the for-loop, so that the code can be JIT compiled. On top of substituting the for-loop with JAX.LAX.SCAN, the STEP function must also conform with

Algorithm 5 Simulation in Python and Jax code

```
procedure SIMULATE_PYTHON(u^0)
    for t in range(0, N) do
        u^{t+1} = STEP(u^t)
    return u^N

procedure SIMULATE_JAX(u^0)
    u^N, {u^0, ..., u^N} = JAX.LAX.SCAN(STEP, u^0, N)
    return u^N
```

Jax's required function signatures, making the conversion from plain Python code to Jax's JIT compatible syntax complex.

Another acceleration technique we used for our simulation is the use of GPUs. Because the simulation is highly parallelizable, we can take advantage of the parallel computing power of GPUs to speed up the simulation. Both Jax and PyTorch support GPU acceleration and have built-in methods for device management. Programmatically, PyTorch is much more mature in this aspect with better documentation and less complexity.

Conversion between Jax and PyTorch

As described above, the Jax and PyTorch packages specialize and excel in different aspects. The fluid simulation that we described is also modular as it is split into different steps. For this reason, a reasonable idea is to implement the simulation using both packages at the same time. For example, writing the for-loops with PyTorch and using Jax's built-in CG solver with custom gradient definition will greatly increase not only the performance but also the maintainability of the simulator.

Unfortunately, to our knowledge, this idea is not feasible because the types in the two packages are not compatible. Although there are ways to convert Jax data types to PyTorch data types, there is not a straightforward way to do so while also keeping the gradient information. Without preserving the gradient information, backpropagation

will break and the learning framework will not work. For this reason, we could not build a third version of the simulator that incorporates both packages.

4.2 Differentiable Renderer

This section describes the differentiable renderer and the smoke rendering scheme we used in our framework. We first describe how the simulated 3D smoke grid is converted to meshes; then we describe the rendering process used to render the smoke from 3D mesh to 2D image.

4.2.1 Mesh Conversion

From the smoke simulation described in Section 4.1, as an output, we obtain a 3D scalar grid of smoke density stored as a 4D tensor of shape (B, nx, ny, nz). The goal is to convert this 3D grid into a 3D mesh that can be rendered by NvDiffRast.

For our 3D grid, each grid cell can be viewed as a voxel containing 8 vertices and 6 faces. We triangulate each grid cell faces into 2 triangles to form 12 triangles for a single voxel. However, we do not want the triangle faces to be duplicated and overlap for neighbouring grid cells. Otherwise, when computing the absorption in the next step, the amount of absorption will be double-counted and will not result in an accurate image.

The mesh in the scene is defined by an array of vertices and an array of faces. We will first describe how the mesh vertices are constructed. For a 3D grid, the vertices are simply the corners of each grid cell. For a grid with shape (nx, ny, nz), there are $(nx + 1) \times (ny + 1) \times (nz + 1)$ vertices. The coordinates of these vertices should be in the model space. For our framework, for convenience, we arbitrarily decided that the model space origin starts from the center of the grid instead of the corner of the grid. For a grid-space coordinate $\mathbf{x}_{grid} = (i, j, k)$, we convert it to model-space coordinate $\mathbf{x}_{model}$ using

$$\mathbf{x}_{model} = \mathbf{x}_{grid} - \frac{\max(\mathbf{x}_{grid}) - \min(\mathbf{x}_{grid})}{2}. \tag{4.5}$$

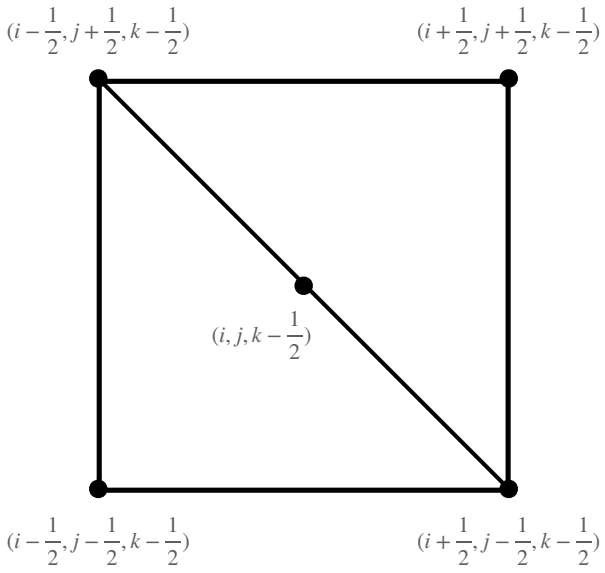


Figure 4.3: Mesh conversion for faces normal to the z-axis.

For the mesh faces, we construct the faces separately according to the cell face normal directions. For each cell face, there will be 2 triangles. Take faces normal to the z-axis for example, if the cell center coordinate is $\mathbf{x}_{center} = (i, j, k - 1/2)$, the two triangles will be constructed by splitting squared the cell face diagonally. The vertices constructing the triangle faces will be $\{(i-1/2, j-1/2, k-1/2)$, $(i+1/2, j-1/2, k-1/2)$, $(i-1/2, j+1/2, k-1/2)\}$ and $\{(i+1/2, j-1/2, k-1/2)$, $(i+1/2, j+1/2, k-1/2)$, $(i-1/2, j+1/2, k-1/2)\}$. This can be shown in Figure 4.3.

For the faces normal to the z-axis direction, there will be $2 \times (n_x + 1) \times ny \times nz$ faces in total. The conversion for faces normal to the x-axis and y-axis follow a similar logic, and there will be $2 \times nx \times (ny + 1) \times nz$ and $2 \times nx \times ny \times (nz + 1)$ faces respectively.

4.2.2 Absorption Interpolation

After converting the grid into a mesh consisting of triangles defined by vertices and faces, we now need to compute the absorption coefficient carried by each vertex.

From the simulation, we obtain a 3D scalar grid with smoke density s at each grid cell center, with unit ppm (parts per million). We first compute the smoke density at

each vertex. Since the mesh's vertices are just the grid cell corners, we can use tri-linear interpolation to take the average smoke density of the 8 neighbouring smoke density values stored at grid cell centers. With these per-vertex smoke densities, we can use the extinction coefficient parameter K_m to compute the absorption coefficient σ_a at each vertex using the absorption coefficient Equation (3.31) described in Section 3.4.2.

4.2.3 Smoke Rendering

With the mesh defined and the per-vertex absorption coefficient value computed, we can now render the smoke. For our renderer, we assume an orthographic camera and compute the direct absorption for each ray of light hitting the camera. We assume an incoming backlight of radiance L_i down the z-axis and put the smoke between the backlight and the camera. The incoming light L_i traverses through the smoke and gets partially absorbed, and finally forms the outgoing light L_o that reaches the camera, and we will render the outgoing light L_o onto the screen. The outgoing light can be computed using the Beer-Lambert law described in Section 3.4.1. Specifically, we will use the discretized heterogeneous version of the Beer-Lambert law, described by Equation (3.30).

At a high-level, our renderer first assumes that there is no smoke medium in the scene and initialize the outgoing light L_o to be equal to the incoming light source L_i. Then, incrementally, starting from the camera, the algorithm marches along the light path towards the light source. For each segment, we compute the absorption coefficient of the smoke and reduce the amount of the outgoing light radiance after it has been absorbed by the smoke in the segment. We accumulate for each light path, and finally, after iterating over the entire smoke medium, we will obtain the final outgoing light radiance L_o after absorption.

Programmatically, we use a rendering technique called depth peeling, which is provided by the NvDiffRast package. This is a common technique used to render semi-transparent objects in a scene. The method iteratively "peels" the mesh layer by layer, where each layer is rasterized and processed separately. In our framework, for each

Algorithm 6 Absorption Accumulation Rendering

procedure ABSORPTION_ACCUMULATION_RENDERING(s, L_i, K_m)
    $V, F =$ CONVERT_GRID_TO_MESH(s)
    $V\_\sigma_a =$ PER_VERTEX_COEF(s, K_m)
    $L_o = L_i$
    while depth peeling not finished **do**
        $rast\_out =$ RASTERIZE_NEXT_LAYER(V, F)
        $pixel\_\sigma_a =$ INTERPOLATE(V_σ, $rast\_out$, F)
        $L_o = L_o e^{-\sigma_a \Delta x}$
        $L_o =$ ANTI_ALIAS(L_o, $rast\_out$, V, F)
    return L_o

layer, we first rasterize the mesh layer to obtain the u, v coordinates and the depth value. Then, we use the rasterization output and the per-vertex absorption coefficients to interpolate the per-pixel absorption coefficient. We use the per-pixel absorption coefficient to accumulate the absorption for the outgoing light L_o using Equation (3.31). After that, we apply anti-aliasing to our computed L_o for a better-looking output and, most importantly, to allow the gradients to be propagated properly in the backward pass. Finally, we update the outgoing light accumulated for each pixel.

This absorption accumulation rendering process is shown in Algorithm 6.

4.2.4 Acceleration and Differentiation

NvDiffRast automatically provides GPU-accelerated forward and backward computations for the rendering algorithm above. One important note is that although NvDiffRast provides elementary operations including `Rasterization`, `Interpolation` and `AntiAlias`, these operations are not bundled with their gradient computations. The `Rasterization` operation does not propagate gradients related to occlusion and visibility, because the `AntiAlias` operation provides these gradients by smoothing out the discontinuous silhouette edges.

NvDiffRast and PyTorch

We make an important note that, unfortunately, NvDiffRast supports only the PyTorch and TensorFlow backends. To our knowledge, there is no official support for the Jax backend. As mentioned in Section 4.1.7, there is no official support for converting between Jax and PyTorch while keeping the gradient information. For this reason, we only use the PyTorch version of our implementation when we conduct experiments on the fully differentiable framework. However, we will still run experiments to compare the performances of Jax and PyTorch backends of the fluid simulators.

4.3 Optimization Methods

In this section, we give a quick overview of the optimization methods used in our work for solving inverse problems.

For our experiments, we use L2 pixel-wise image losses to compute the loss between the rendered image L and the target image $\hat{L}$. The loss is defined as

$$\mathcal{L}_{img} = \frac{1}{N} \sum_{i=1}^{N} \left\| L_i - \hat{L}_i \right\|_2^2, \tag{4.6}$$

where N is the number of pixels in the image.

For the optimization method, we use the Adam optimizer [30] with varying epochs and learning rates depending on the experiment cases. We also implemented a custom learning rate scheduling scheme that decreases the learning rate over time to prevent the optimization from overshooting at later time steps. Our learning rate scheduler is defined by

$$\alpha(t) = \alpha_0 10^{-\frac{t}{1000}}, \tag{4.7}$$

where α_0 is the initial learning rate, and t is the number of epochs.

In order to save GPU memory, we also adapted the checkpointing technique to our simulation. The checkpointing technique works by saving the intermediate results to the disk, and only loading them back to the GPU memory when needed. In our case,for each time step of the simulation, we save the simulated states and their gradients to the disk. This allows us to run the simulation and compute the gradients in smaller batches of time steps. This technique alleviates the GPU memory bottleneck and allows us to run the simulation with a larger resolution, but at the cost of slower simulation time because operations need to be re-evaluated, and the gradient and results need to be saved to the disk and loaded back to the GPU memory.

Chapter 5

Experiments and Results

This chapter details the experiments we ran on our framework and their results. The experiments were run on both the simulation and the full pipeline for forward and backward tasks. For each experiment, we will discuss the setup, results, runtime, and memory performances.

For all our experiments, unless otherwise specified, we run the JIT compiled simulation with Intel Gold 6148 Skylake CPU and Nvidia V100SXM2 (16GB) GPU on the Compute Canada Beluga compute cluster. Each runtime and memory measurement is computed by taking 3 independent runs and averaging the results.

5.1 Differentiable Fluid Simulation

First, since one of our contributions is the implementation of the differentiable fluid simulator using both the Jax and PyTorch packages and the evaluation between them, we run our simulator on a few scenes to verify its correctness and performance, and also perform some learning tasks to show case how the gradients can be propagated from 3D simulated states to the 3D initial states. For these experiments, the differentiable renderer is not included. The rendering follows a similar logic to the direct absorption computation

described in Section 4.2.2, but without rasterization, using a non-differentiable Python library Matplotlib.

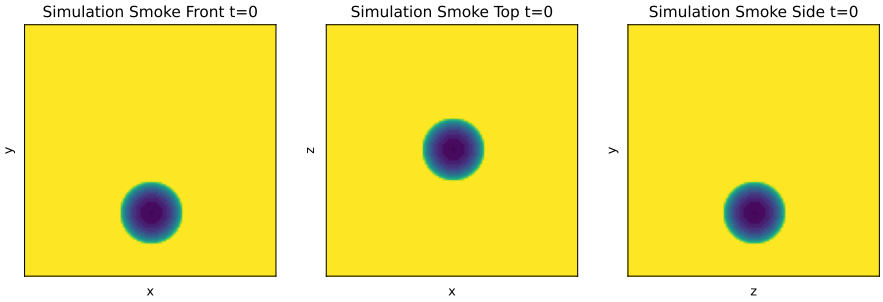
5.1.1 Forward Simulation

First, we run the forward simulation without any external force or initial velocities. We run the experiment using both Jax and PyTorch backends on different resolutions, and we compare the results and their performances. All simulations are run for 60 time steps, with time step size Δt of 0.5. The smoke is injected at the center bottom of the grid with a fixed radius.

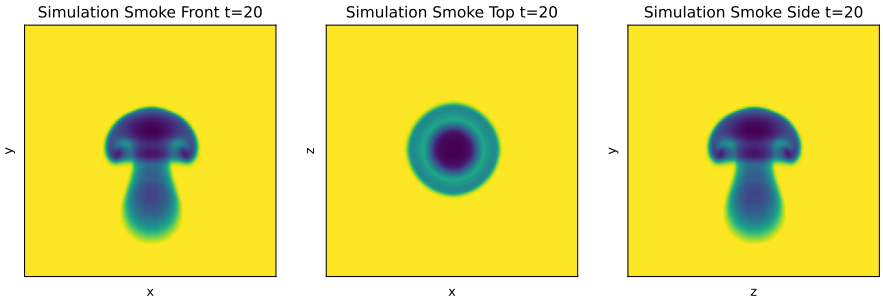
A few key frames of the PyTorch simulation with resolution $128 \times 128 \times 128$ can be seen in Figure 5.1. Each row corresponds to the starting frame of the smoke, the frame where the smoke forms a plume, the frame where the smoke plume hits the top of the grid, and the end frame of the plume. The left column corresponds to the absorption accumulation viewed from the front (xy-plane), the middle column corresponds to the absorption accumulation viewed from the top (xz-plane), and the right column corresponds to the absorption accumulation viewed from the side (zy-plane). From the results, we can see that the smoke rises naturally and forms a plume with a considerable amount of detail. As the plume hits the top and side of the grid, it does not go past the domain, and the velocity is divergence free and also conforms with the boundary conditions. The results are stable and physically accurate.

The Jax version of the simulation produces the same results as the PyTorch version because the simulation logic is the same despite the package-specific semantic differences.

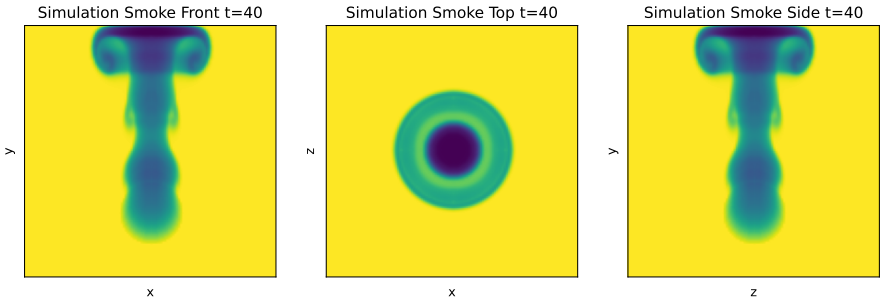
As for runtime and memory performances, we compare the performances between different resolutions and between the Jax and PyTorch backends in Table 5.1. The measurements are made by taking the average for multiple simulation passes in order to account for the initial JIT compilation run. In general, as the resolution doubles, the number of cells in the grid increases by a factor of 8, and the runtime and memory requirements increase by roughly the same magnitude as well.



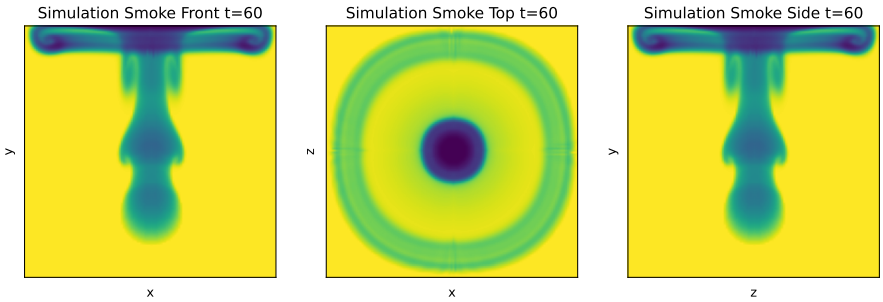
(a) 1st frame of the simulation



(b) 20th frame of the simulation



(c) 40th frame of the simulation



(d) 60th (final) frame of the simulation

Figure 5.1: Forward simulation with resolution $128 \times 128 \times 128$. Yellow indicates low absorption and high light intensity, and blue indicates high absorption and low light intensity.

Table 5.1: Forward Simulation Runtime and Memory Performance

Resolution	PyTorch		Jax	
	Runtime (s)	Memory (GB)	Runtime (s)	Memory (GB)
$32 \times 32 \times 32$	4.0	0.0302	3.8	0.0288
$64 \times 64 \times 64$	4.2	0.2308	4.1	0.2250
$128 \times 128 \times 128$	29.1	1.8055	22.3	1.7617
$256 \times 256 \times 256$	215.4	14.2843	160.3	13.7764

Comparing the two backends, the Jax version of the implementation runs significantly faster than the PyTorch version. For JIT compilation, Jax compiles the simulation much faster than PyTorch, and the compiled Jax function also runs faster than the compiled PyTorch function. As for memory, the two implementations do not differ significantly.

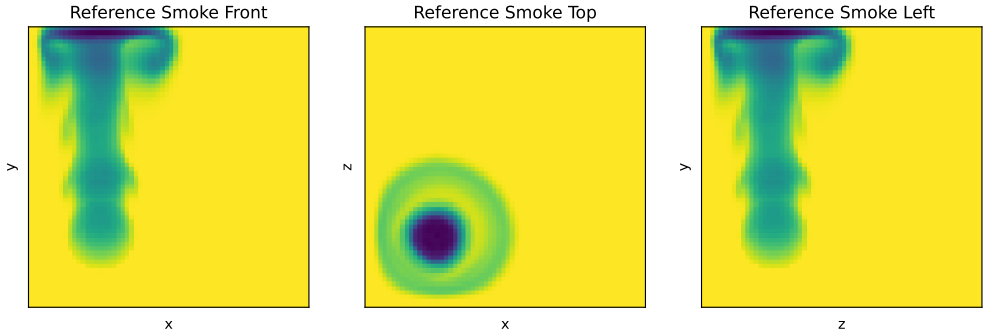
5.1.2 Simulation Learning

We conduct a learning example to verify the gradients are propagated from the 3D simulated smoke state to the simulation parameters correctly. The optimization goal is to make the final frame of the simulated smoke state match a target smoke state as much as possible by adjusting the initial velocity parameter while keeping all other parameters the same. The target smoke state is generated by running the forward simulation with a different inflow injection location. Note that because the injection location is different, the learned simulation state will never be able to match the target state perfectly.

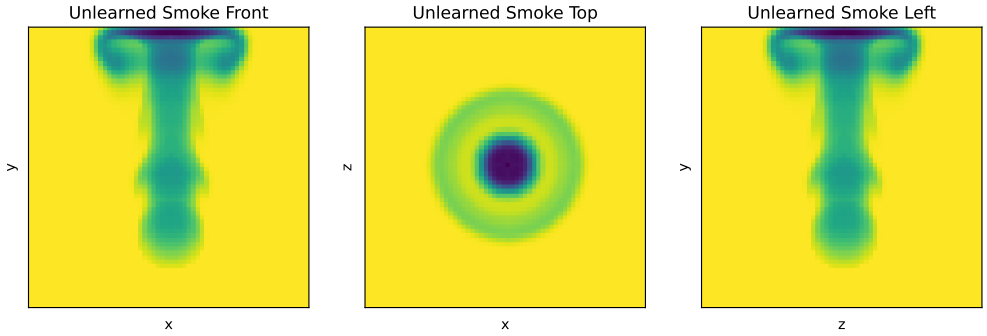
The simulation is run on different resolutions (32^3, 64^3 and 96^3) for the same physical domain size. The simulation is run for 30 time steps, with time step size Δt of 0.5, so that the smoke reaches the top of the box to form a plume and does not fill the box excessively. There is no external force applied to the smoke. Training is done with the Adam optimizer with learning rate scheduling described in Section 4.3. The loss function is the 3D voxel-wise L2 loss between the final frame of the simulated and reference smoke state grids. In order to benchmark the implementations' performances, we run the learning tasks on different resolutions. Because the resolution of the experiments is different, the learning rate and epochs are tuned and adjusted for each resolution.

Figure 5.2 shows the optimization results of the Jax implementation with a simulation resolution of $64 \times 64 \times 64$. The PyTorch version produces similar results. Figure 5.2a shows the reference smoke state. We place the reference inflow location at the corner of our simulation domain. The smoke rises at the corner and hits the top while conforming with the boundary conditions. Figure 5.2b shows the smoke state of the simulation with a different inflow location and unlearned initial velocity. The inflow location is placed at the bottom center of the grid. Figure 5.2c shows the of the simulated smoke state using the learned initial velocity for the center inflow location. From the results, we can see that even though the inflow location is different, the learned velocity still produces a final smoke state that tries to match the reference smoke state as much as possible. For this experiment, a learning rate of 0.5 was used, and the initial velocity was trained for 500 epochs. The loss plot can be seen in Figure 5.3.

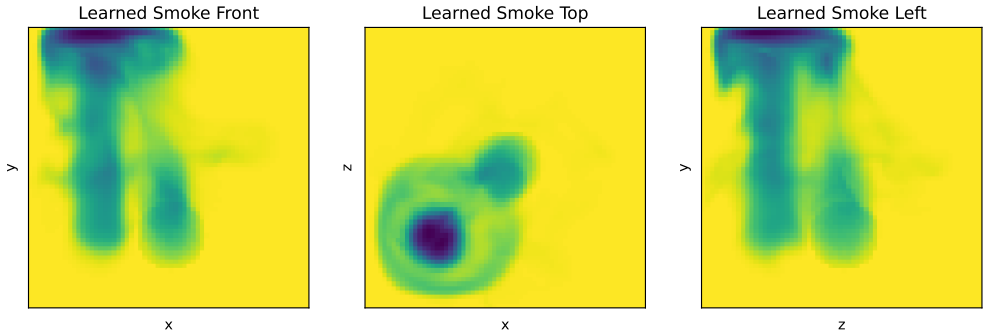
As for the performance for the learning task, we compare the runtime and memory performances between the PyTorch and Jax backends in Table 5.2 and Table 5.3. Because the epochs differ for different resolutions, the runtime is computed as the per epoch runtime. The general trend is similar to the forward performance. The Jax implementation is faster than the PyTorch implementation, and the runtime and memory requirements increase as the resolution and number of time steps increase. We make an observation that compared to the runtime, the memory requirements for the learning task are much higher than the forward task. This is because the backward computation requires the computation graph or operation tape to be stored in memory. Depending on the learning task, the size of the computation graph increases non-linearly with respect to the resolution. Note that a resolution of $128 \times 128 \times 128$ is not included in the table because the GPU ran out of memory.



(a) Final state of the reference simulation



(b) Final state of the simulation with unlearned initial velocity



(c) Final state of the simulation with optimized initial velocity

Figure 5.2: Optimizing initial velocity with resolution $64 \times 64 \times 64$

Table 5.2: Optimization runtime and memory performance against resolution for 30 time steps

Resolution	PyTorch		Jax	
	Runtime (s)	Memory (GB)	Runtime (s)	Memory (GB)
$32 \times 32 \times 32$	1.3	0.2644	1.2	0.2630
$64 \times 64 \times 64$	1.6	2.1159	1.5	2.0271
$96 \times 96 \times 96$	2.7	6.8530	2.3	6.3560

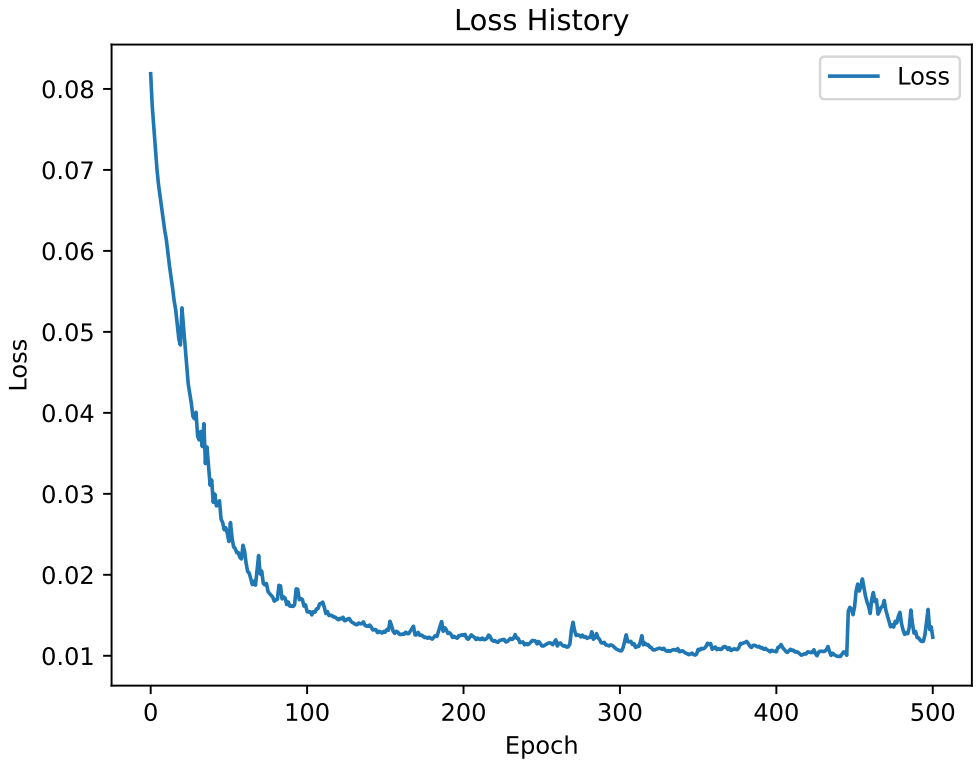


Figure 5.3: Loss values over training epochs

Table 5.3: Optimization runtime and memory performance against time steps for $64 \times 64 \times 64$ resolution

Time steps	PyTorch		Jax	
	Runtime (s)	Memory (GB)	Runtime (s)	Memory (GB)
15	1.0	1.1543	0.9	1.1388
30	1.6	2.1159	1.5	2.0271
60	3.1	3.9420	2.7	3.7338

5.2 Fully Differentiable Pipeline

This section presents the experiments and results of the fully differentiable simulation and rasterization pipeline described in Section 4. We first present the results of the forward simulation and rendering experiments in Section 5.2.1. Then we present the results of the simulation and rendering optimization experiments in Section 5.2.2. Since the differentiable renderer we use (NvDiffRast) does not support the Jax backend, we will only use the PyTorch backend for the full pipeline experiments.

For our scene setup, we use an orthographic projection with a near plane of 1 unit away from the camera for simplicity. We would like to place the center of the smoke grid at the center of the viewing frustum. We'd also like the viewing frustum to be $2nx \times 2ny \times 2nz$ large so that no matter how we rotate the camera, the smoke grid will always be fully contained in the viewing frustum and the rendered image will have some paddings around the smoke box. To satisfy the requirements above, we first place the smoke grid center at the world space origin. Then, we place the camera at nz units away in the negative z axis in the world space. To position the grid center at the viewing frustum center, we set the far plane $2n_z + 1$ units away from the camera, the left and right planes n_x units away from the camera in the x-axis, and the top and bottom planes n_y units away from the camera in the y-axis. As for the light scenario, for simplicity, we use a first order lighting setting, where we have a directional back light that points towards the camera, and we only compute the direct absorption described in Section 3.4. To simplify matters further, We assume the backlight $L_i = 1.0$ and render the outgoing light reaching the camera, L_o, after the light traverses through the smoke.

5.2.1 Simulation and Rendering

In this experiment, we test the rendered result of the fully connected differentiable simulation and rendering pipeline and analyze the runtime and memory performance.

Table 5.4: Full pipeline forward performance

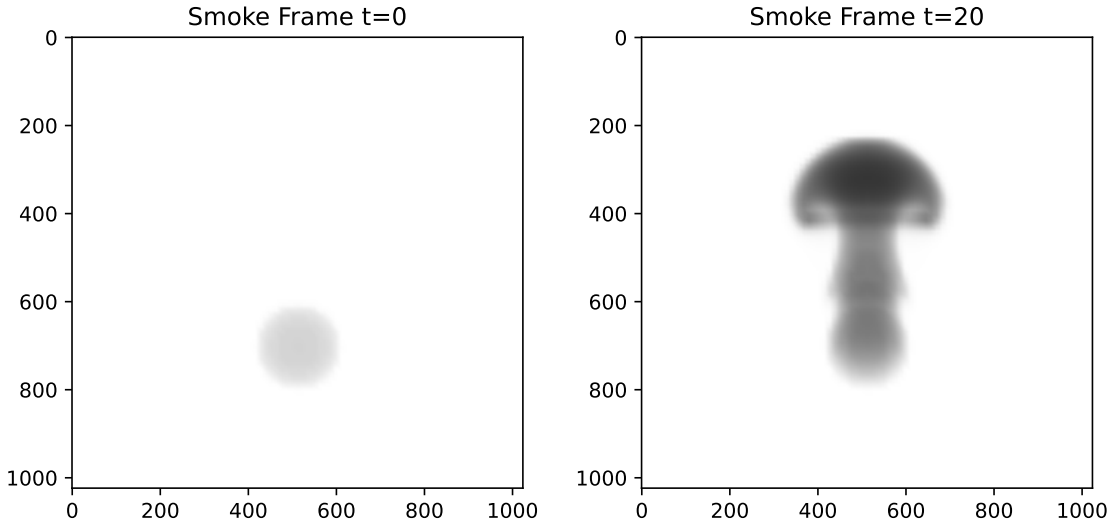
Simulation Resolution	Rendering Resolution	Runtime (s)	Memory (GB)
$32 \times 32 \times 32$	512×512	3.8	0.1628
	1024×1024	4.5	0.1688
$64 \times 64 \times 64$	512×512	6.3	0.6241
	1024×1024	8.4	0.6431
$128 \times 128 \times 128$	512×512	15.4	4.5265
	1024×1024	18.5	4.5617

For rendering, we use the setup described above. For simulation, we use the same simulation setup as described in Section 5.1.1. We experimented with combinations of different grid resolutions and image resolutions. Figure 5.4 shows the result of different stages of a $64 \times 64 \times 64$ grid simulation rendered on a 1024×1024 resolution screen. Compared to Figure 5.1, the rasterization renders the image in a much higher resolution with less pixelation and aliasing artifacts. All details of the fluid simulation are preserved, and the rendered image is stable and physically accurate.

The runtime and memory performance for the forward simulation and rendering pipeline is shown in Table 5.4. The runtime is computed taking into account both the simulation and rendering. The memory is computed as the peak memory usage during the simulation and rendering process. Note that NvDiffRast supports batched rendering, and the memory used for rendering depends on the batch size. For our experiment, we fixed a batch size of 16 for all our test cases. The results show that the most significant factor of the runtime and memory requirement is the simulation resolution. Increasing the rendering resolution has a relatively smaller effect on the runtime and memory requirements compared to increasing the simulation resolution.

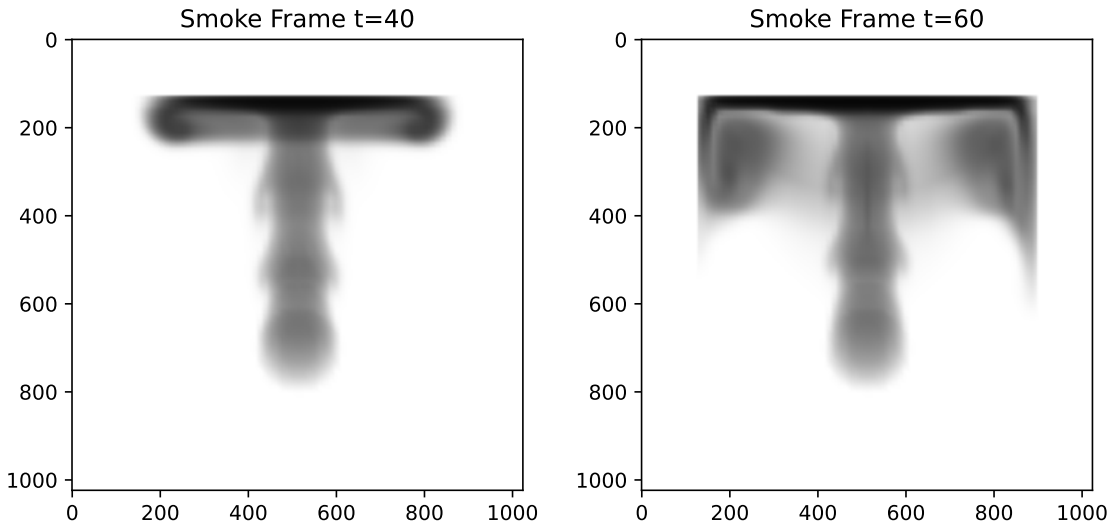
5.2.2 Simulation and Rendering Learning

In this section, we present learning experiments using the fully differentiable pipeline. We will show how the gradient information can be propagated from 2D rendered image back to 3D simulation and rendering parameters. We will first show the result of a learning



(a) Rendered 1st frame

(b) Rendered 20th frame



(c) Rendered 40th frame

(d) Rendered Final frame

Figure 5.4: Experiment 3: simulation and rendering with grid resolution $64 \times 64 \times 64$ and image resolution 1024×1024

task that uses the gradient information to learn 3D simulation parameters. Then, we will show a more complex learning task that optimizes for both simulation and rendering parameters.

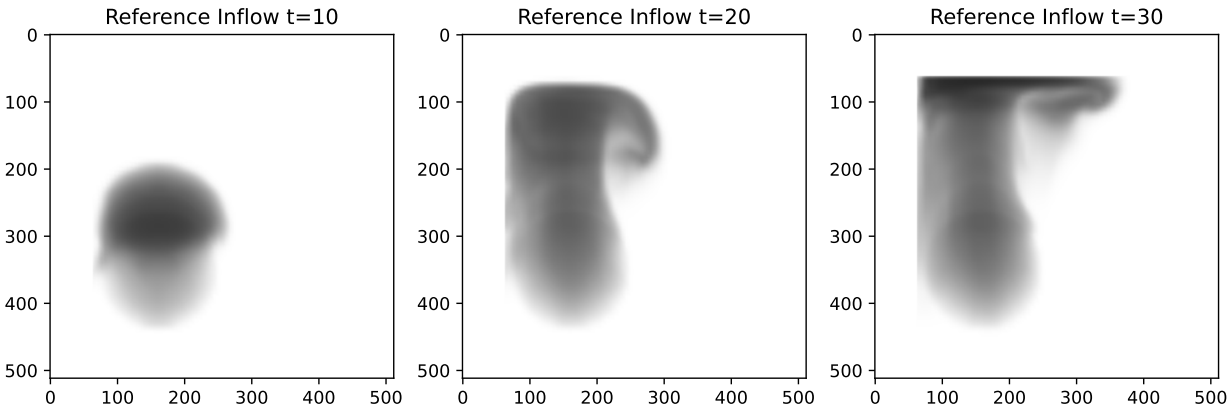
Inflow Location Optimization

In this experiment, we try to learn the 3D coordinate of the inflow location of the smoke, but base on only the 2D rendered image. This task requires the gradient to be propagated from the rendered 2D image through the renderer and the simulator and finally to the input inflow location. The learning goal is to find an inflow location that gives an image as close to the rendered image using the reference inflow location as possible.

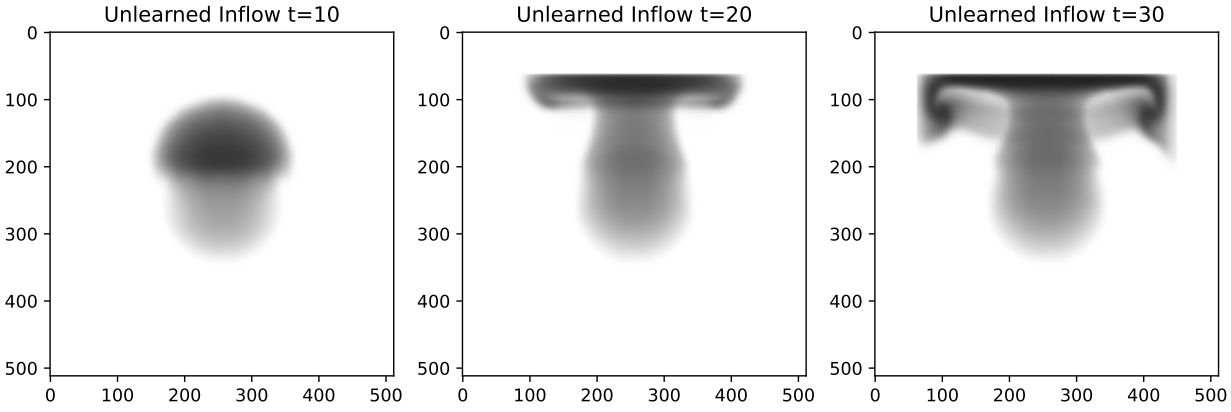
The initial setup is similar to that of Section 5.1.2. The simulation is run for 30 time steps with time step size Δt of 0.5, with no external forces or initial velocity applied to the smoke. The rendering process is the same as described in Section 5.1.1, and the resolution used in this experiment is 512×512. The loss function this time is the L2 pixel-wise difference between the rendered 2D final images of the simulations with the learned and reference inflow locations. The learning method is the same as described in Section 5.1.2, and the learning rate is set to 1.0, and trained for 100 epochs.

Figure 5.5 shows the learning results. Each row shows the rendered simulation using the reference, unlearned, learned inflow locations respectively. Each column shows the 10th, 20th and 30th (final) time step of the simulation respectively. We can see that the learned inflow location is very close to the reference inflow location, and the rendered images are almost identical.

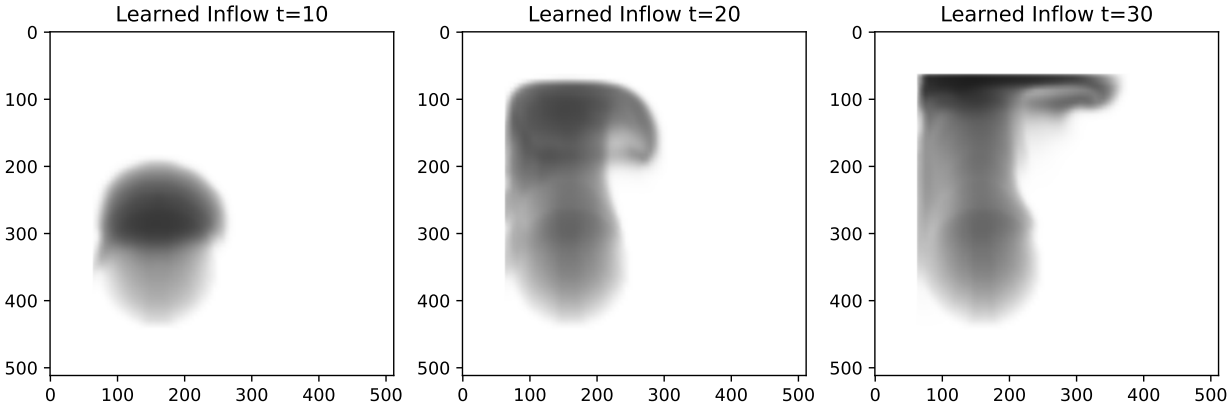
Figure 5.6a shows the history of the learning process. We make an observation that the learned inflow location moves in the upper-left direction first before moving down to reach the reference location, instead of intuitively moving directly in the bottom left direction towards the reference location, as suggested by Figure 5.6b. This happens due to the non-linear nature of fluid simulation. Moving the inflow location in the negative y direction will reduce the size of the plume reaching the top, thus reducing the absorption



(a) Rendered final state of the reference simulation

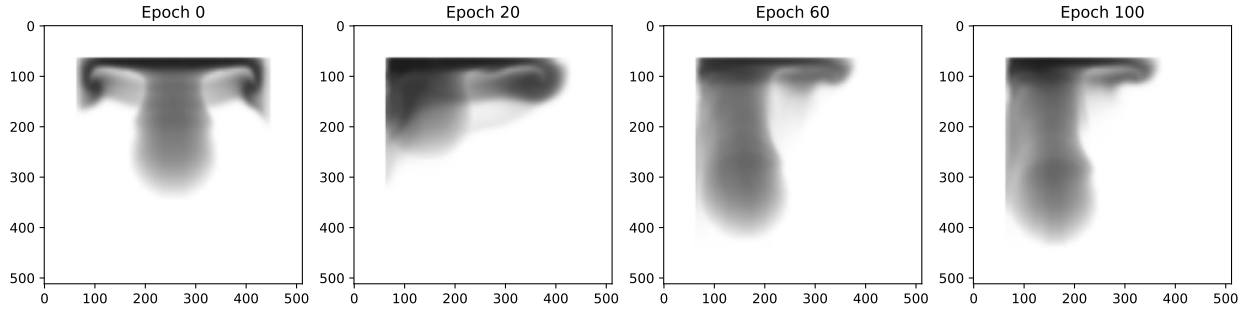


(b) Rendered final state of the unlearned simulation

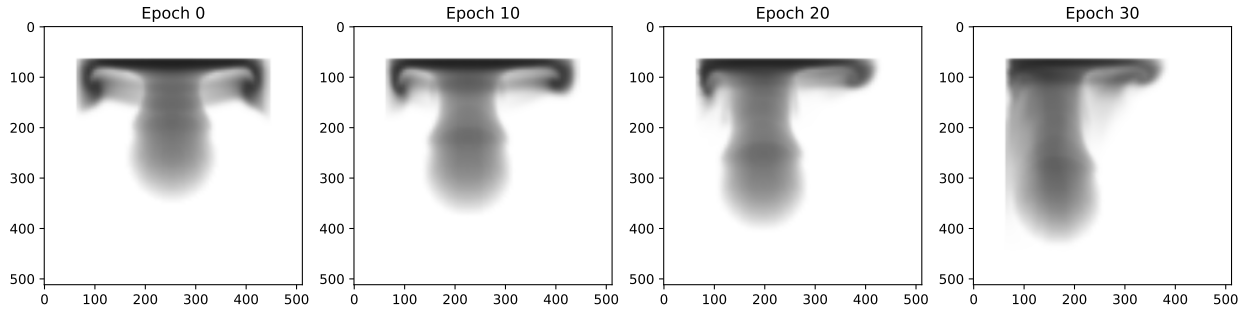


(c) Rendered final state of the learned simulation

Figure 5.5: Optimizing inflow location



(a) Learning history at different epochs



(b) Manually Adjusted Inflow Location at Different Epochs

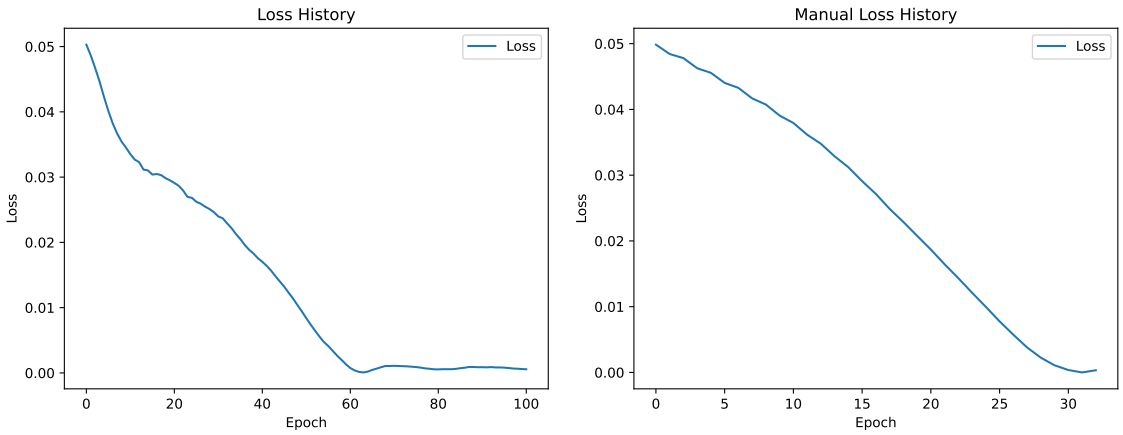
Figure 5.6: A comparison between the learning history and the manually forced "linear" history

image. Because the images overlaps less, the loss will also not decrease effectively, even though the inflow location is moving towards the correct location.

Figure 5.7a shows the loss plot comparison of the learning process against the manually adjusted inflow location. We can clearly see that although the starting and ending losses for the two plots are the same, the loss plot for the learning process makes a faster descent initially by moving the inflow location to the top left corner to reduce the pixel-wise L2 loss.

Inflow and Camera Location Learning

Finally, we present a more complex experiment that optimizes both the inflow location and the camera location. The goal of this experiment is to show that the proposed



(a) Optimization Loss **(b)** Manual Loss

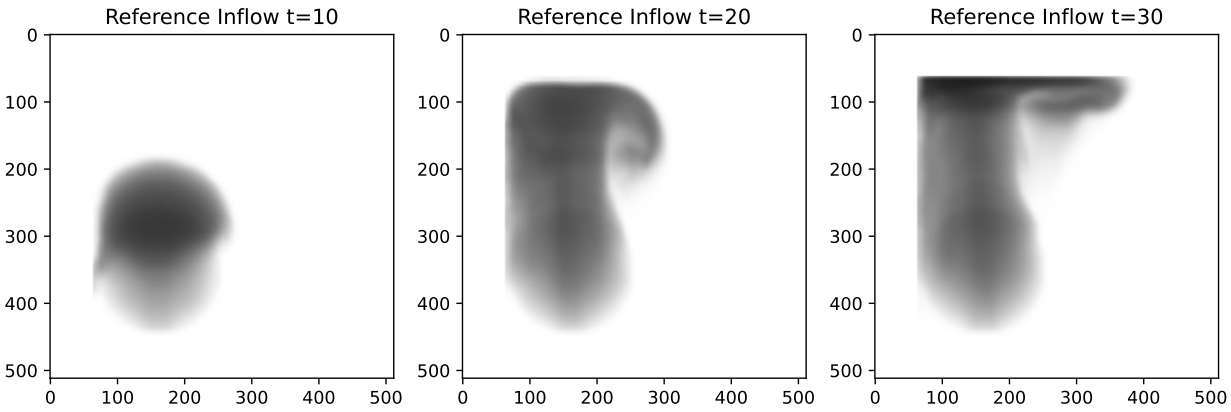
Figure 5.7: A comparison of the loss plot between the learning history and the manually forced "linear" history

method can be used to optimize multiple parameters, including simulation and rendering parameters, simultaneously.

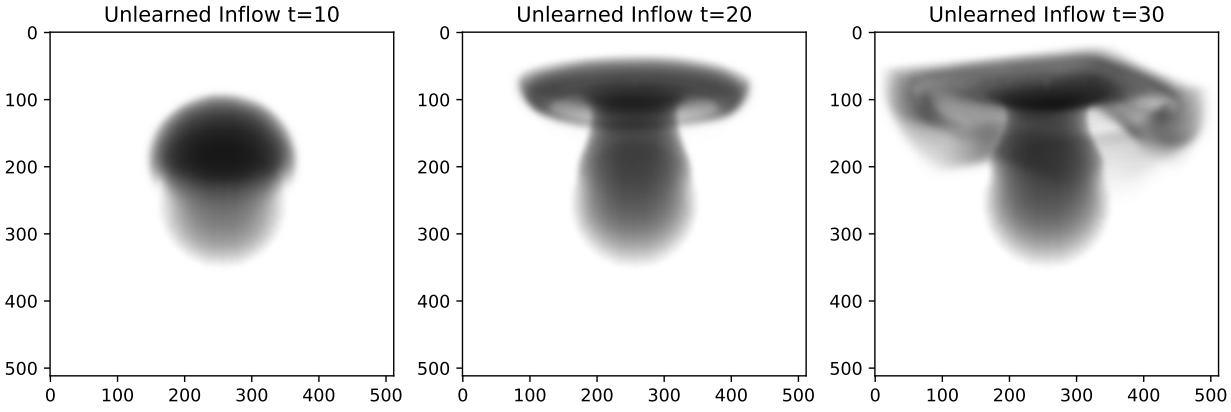
We use the same setup as the previous experiment, except that on top of the different inflow locations, we also initialize the learning process with a different camera location. The reference and unlearned inflow locations are initialized the same as the previous experiment. The reference camera position sits on the negative z-axis in the world space, where as the unlearned initial camera position is shifted off-axis. For the learning process, we continue to use the L2 pixel loss of the rendered final frame. We use a learning rate of 1.0 and train for 100 epochs.

Figure 5.8 shows the results of the learning experiment. Similar to Section 5.2.2, each column represent the rendered smoke at a different time step, and each row represent the simulation using the reference, unlearned and learned inflow and camera parameters. We can see that the learned inflow and camera locations are able to produce a rendered image that is very similar to the reference image.

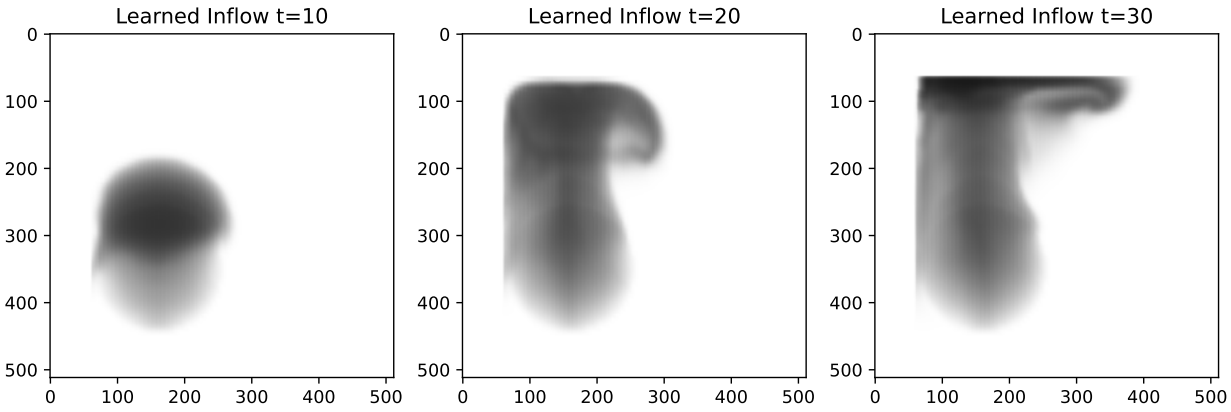
Figure 5.9 shows the learning history at different epochs.



(a) Rendered final state of the reference simulation



(b) Rendered final state of the unlearned simulation



(c) Rendered final state of the Learned Simulation

Figure 5.8: Optimizing inflow and camera locations

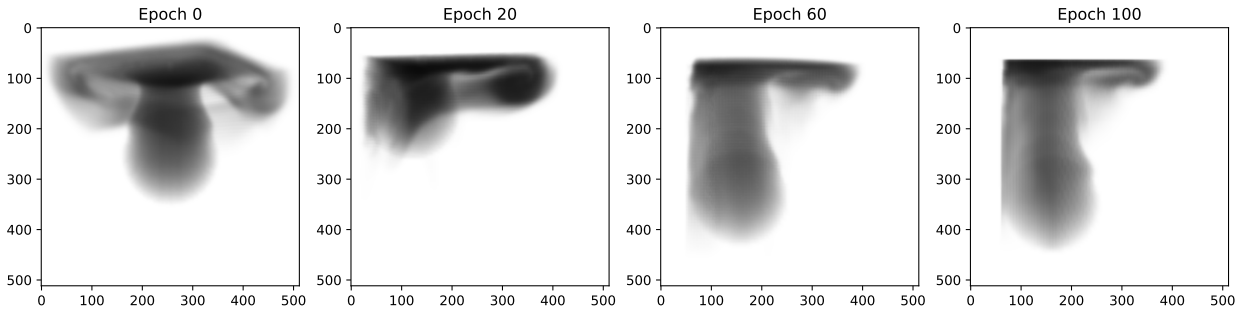


Figure 5.9: Learning history at different epochs

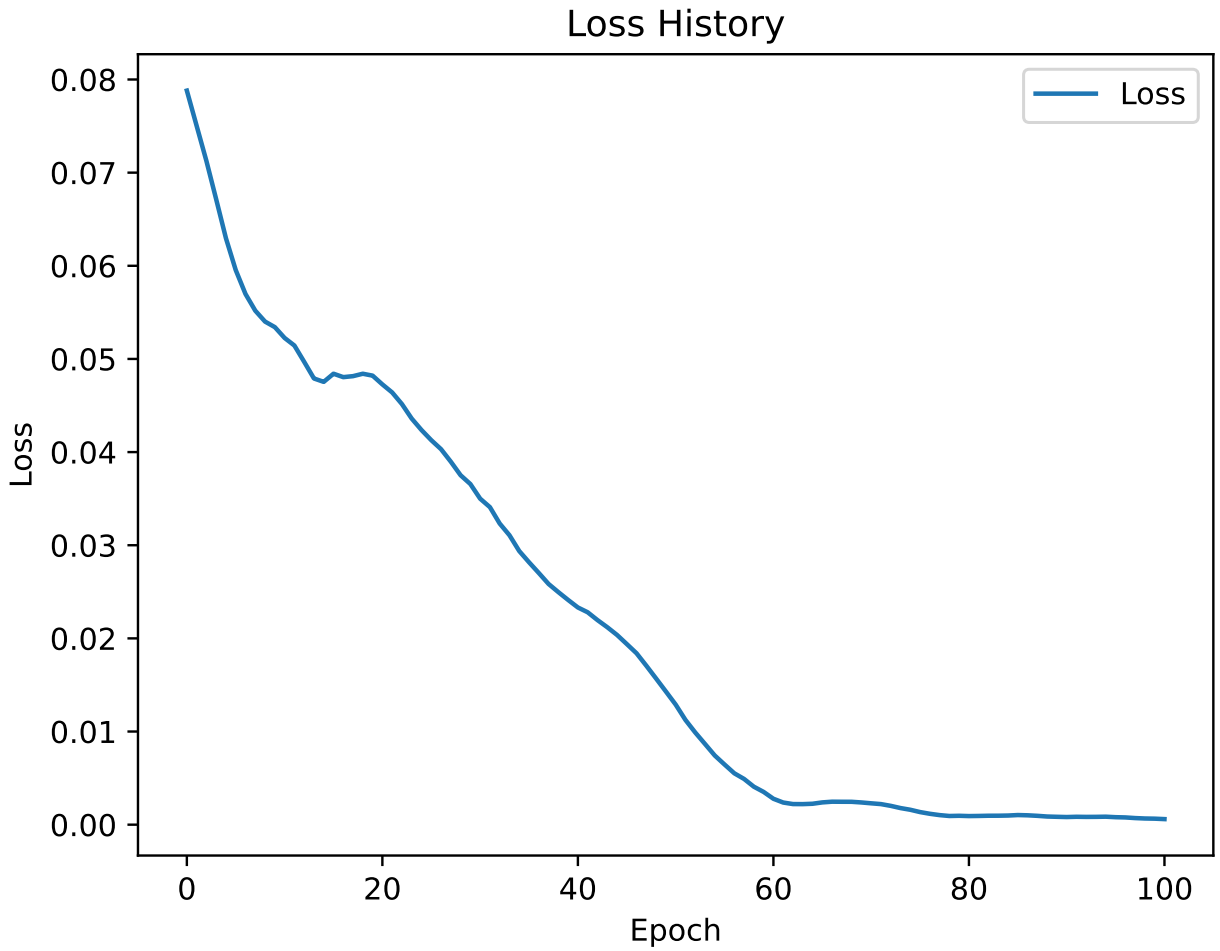


Figure 5.10: Loss curve of the learning process

Figure 5.10 shows the loss curve of the learning process. We can see that the loss decreases quickly at the beginning of the learning process, and then slowly converges to a local minimum.

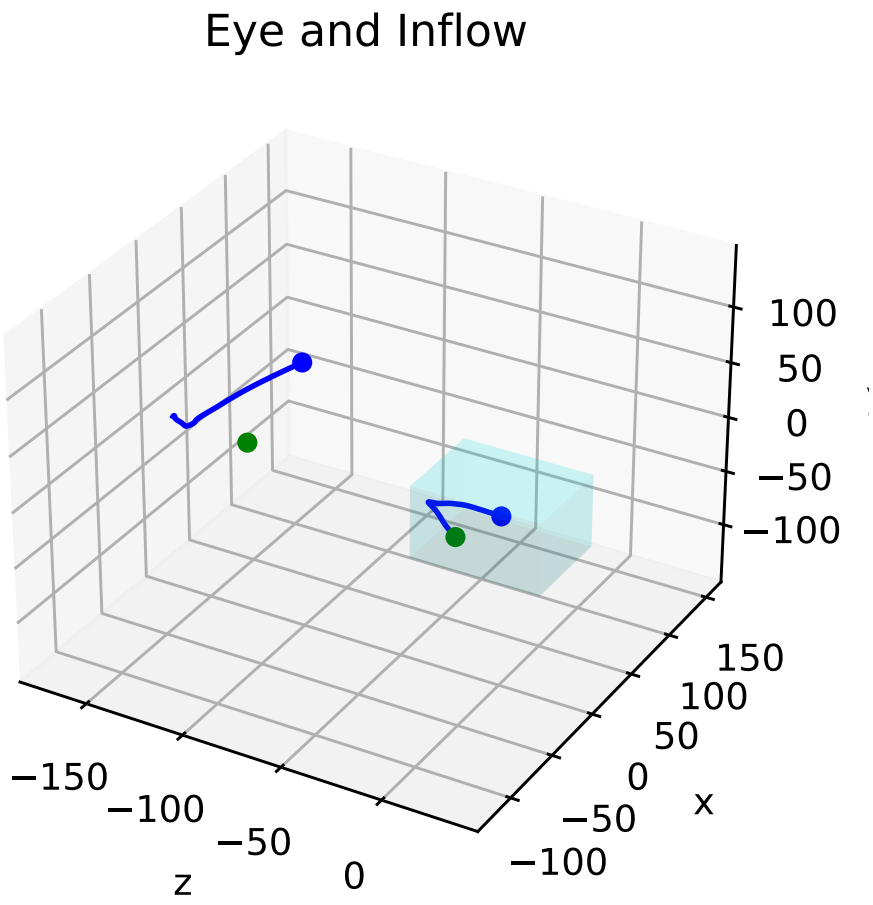


Figure 5.11: A visualization of the learned inflow and camera location trajectories

Figure 5.11 shows the reference and learned inflow and camera locations. The transparent box represents the simulation domain in the world space. The green point in the simulation domain represents the reference inflow location, where as the green point outside the simulation domain represents the reference camera location. The blue points represent the unlearned initial inflow and camera locations. The blue lines represent the learned inflow and camera locations throughout the training process. We can see that the learned inflow location is very close to the reference inflow location, but the learned camera location is not quite the same because the z-axis is different from the reference location. However, because we use an orthographic camera, the rendered image is not affected by the z-axis of the camera location, and the learned rendered image is very similar to the reference rendered image.

Chapter 6

Conclusion

In this work, we built a fully differentiable fluid simulation and rendering pipeline. We implemented a differentiable grid-based fluid simulator using Jax and PyTorch and compared performances for both implementations. We conducted experiments to show that the differentiable fluid simulator can be used to optimize the simulation parameters. By connecting the 3D output from the simulator to a differentiable renderer (NvDiffRast), we showed further that even with the loss of information by projecting 3D states to 2D rendered images, the gradient information can still be propagated to 3D fluid simulation and rendering control parameters.

6.1 Advantages and Limitations

Our framework is fully differentiable, which means that gradients can be passed from the rendered image to both the renderer and the fluid simulator. The framework allows users to optimize multiple parameters for both components simultaneously to achieve a desired rendered image. This replaces the traditional trial-and-error process of manually tuning parameters. Furthermore, our framework is physically accurate. The simulator solves the Navier-Stokes equations, which are the governing equations for fluid dynamics. The renderer also uses a physically accurate scheme to compute and render the

absorption of light by the smoke. Finally, our framework is flexible and easy to use. The implementation is modular, and the simulation is generalized to allow various parameter configurations. Users can easily create physically realistic fluid simulations for different scenarios.

Our work also has limitations. First, the simulator is not optimized for runtime. The main bottleneck is the Conjugate Gradient solver for the PyTorch implementation, which has a runtime a lot slower than the Jax implementation. This is because the implementation uses a Python for-loop, which is extremely slow compared to other lower-level languages such as C++. Secondly, the simulator does not scale well with the simulation resolution. Because the simulation solve is non-linear, the memory requirement for backpropagation increases non-linearly as the resolution increases. This is a common problem for most differentiable physics-based simulators and is an active research field in the community. Also, our simulation currently only supports a box-shaped domain and boundary because of the padding and convolution implementation mentioned in Section 4.1.6. This limits the possibility of learning smoke parameters in a more complex environment or even learning the boundary condition itself.

6.2 Future Work

There are many possible future directions for this work. First, we can improve the runtime of the simulator. As mentioned in Section 6.1, the simulator is not optimized for runtime. We can improve the runtime by implementing the conjugate gradient solver in a lower-level language such as C++ before wrapping it with Python.

We also plan to combine existing optimization and training methods into our framework. The adjoint method [15] can provide custom gradients for the fluid simulation operations instead of storing all elementary operations on the computation tape, thus potentially reducing the computation time and memory requirement. Multi-grid and Alternating direction method of multiplier (ADMM) [31] can improve our

training method by optimizing parameters with reduced spatial and temporal resolutions before up-sampling and optimizing for the full resolution parameters. The lower resolution parameters can be used as a good initialization for the full resolution parameters, potentially reducing the training time.

Finally, we plan on expanding our implementation to support more boundary conditions. By doing so, we can make boundary conditions a parameter and perform optimization tasks. This will give us more degrees of freedom and enable us to explore more applications using our framework.

References

[1] Francis H. Harlow and J. Eddie Welch. "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface". In: *The Physics of Fluids* 8.12 (1965), pp. 2182–2189. DOI: 10.1063/1.1761178.

[2] D. J. Tritton. "Thermal Flows: Basic Equations and Concepts". In: *Physical Fluid Dynamics*. Dordrecht: Springer Netherlands, 1977, pp. 127–134. DOI: 10.1007/978-94-009-9992-3_13.

[3] W. T. Reeves. "Particle Systems—a Technique for Modeling a Class of Fuzzy Objects". In: *ACM Trans. Graph.* 2.2 (Apr. 1983), pp. 91–108. DOI: 10.1145/357318.357320.

[4] J. U. Brackbill and H. M. Ruppel. "FLIP: A Method for Adaptively Zoned, Particle-in-Cell Calculations of Fluid Flows in Two Dimensions". In: *Journal of Computational Physics* 65.2 (Aug. 1, 1986), pp. 314–343. DOI: 10.1016/0021-9991(86)90211-1.

[5] Larry Yaeger, Craig Upson, and Robert Myers. "Combining Physical and Visual Simulation—Creation of the Planet Jupiter for the Film "2010"". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 85–93. DOI: 10.1145/15886.15895.

[6] Karl Sims. "Particle Animation and Rendering Using Data Parallel Computation". In: *SIGGRAPH Comput. Graph.* 24.4 (Sept. 1990), pp. 405–413. DOI: 10.1145/97880.97923.

[7] John Hart. “Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces”. In: *The Visual Computer* 12 (June 13, 1995). DOI: 10.1007/s003710050084.

[8] Nick Foster and Dimitris Metaxas. “Controlling Fluid Animation”. In: *Proceedings of the 1997 Conference on Computer Graphics International*. CGI ’97. USA: IEEE Computer Society, 1997, p. 178.

[9] Nick Foster and Dimitris Metaxas. “Modeling the Motion of a Hot, Turbulent Gas”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 181–188. DOI: 10.1145/258734.258838.

[10] Jos Stam. “Stable Fluids”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’99. USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128. DOI: 10.1145/311535.311548.

[11] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. “Visual Simulation of Smoke”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, Aug. 1, 2001, pp. 15–22. DOI: 10.1145/383259.383260.

[12] Nick Foster and Ronald Fedkiw. “Practical Animation of Liquids”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 23–30. DOI: 10.1145/383259.383261.

[13] Jos Stam. “A Simple Fluid Solver Based on the FFT”. In: *J. Graph. Tools* 6.2 (Sept. 2002), pp. 43–52. DOI: 10.1080/10867651.2001.10487540.

[14] Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. “Keyframe Control of Smoke Simulations”. In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 716–723. DOI: 10.1145/882262.882337.

[15] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. "Fluid Control Using the Adjoint Method". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 449–456. DOI: 10.1145/1015706.1015744.

[16] Andrew Selle, Alex Mohr, and Stephen Chenney. "Cartoon Rendering of Smoke Animations". In: *Proceedings of the 3rd International Symposium on Non-Photorealistic Animation and Rendering*. NPAR '04. Annecy, France: Association for Computing Machinery, 2004, pp. 57–60. DOI: 10.1145/987657.987666.

[17] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. "A Vortex Particle Method for Smoke, Water and Explosions". In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 910–914. DOI: 10.1145/1073204.1073282.

[18] Y. Zhu and R. Bridson. "Animating Sand as a Fluid". In: ACM Transactions on Graphics. Vol. 24. 3. 2005, pp. 965–972. DOI: 10.1145/1073204.1073298.

[19] Morgan McGuire and Andi Fein. "Real-Time Rendering of Cartoon Smoke and Clouds". In: *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering*. NPAR '06. Annecy, France: Association for Computing Machinery, 2006, pp. 21–26. DOI: 10.1145/1124728.1124733.

[20] Tamás Umenhoffer, László Szirmay-Kalos, and Gábor Szijártó. "Spherical billboards and their application to rendering explosions". In: vol. 2006. Jan. 2006. DOI: 10.1145/1143079.1143089.

[21] Robert Bridson. *Fluid Simulation*. USA: A. K. Peters, Ltd., 2008.

[22] Rahul Narain, Jason Sewall, Mark Carlson, and Ming C. Lin. "Fast Animation of Turbulence Using Energy Transport and Procedural Synthesis". In: *ACM SIGGRAPH Asia 2008 Papers*. SIGGRAPH Asia '08. Singapore: Association for Computing Machinery, 2008. DOI: 10.1145/1457515.1409119.

[23] Jie Tan, Yuting Gu, Greg Turk, and C. Karen Liu. "Articulated Swimming Creatures". In: *ACM Trans. Graph.* 30.4 (July 2011). DOI: 10.1145/2010324.1964953.

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[25] Zhanpeng Huang, Ladislav Kavan, Weikai Li, Pan Hui, and Guanghong Gong. "Reducing numerical dissipation in smoke simulation". In: *Graphical Models* 78 (2015), pp. 10–25. DOI: https://doi.org/10.1016/j.gmod.2014.12.002.

[26] Tobias Martin, Nobuyuki Umetani, and Bernd Bickel. "OmniAD: Data-Driven Omni-Directional Aerodynamics". In: *ACM Trans. Graph.* 34.4 (July 2015). DOI: 10.1145/2766919.

[27] Andreas Peer, Markus Ihmsen, Jens Cornelis, and Matthias Teschner. "An Implicit Viscosity Formulation for SPH Fluids". In: *ACM Trans. Graph.* 34.4 (July 2015). DOI: 10.1145/2766925.

[28] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic Differentiation in Machine Learning: A Survey". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 5595–5637.

[29] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. *Neural 3D Mesh Renderer*. Nov. 20, 2017. DOI: 10.48550/arXiv.1711.07566. preprint.

[30] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2017).

[31] Zherong Pan and Dinesh Manocha. "Efficient Solver for Spacetime Control of Smoke". In: *ACM Trans. Graph.* 36.5 (July 2017). DOI: 10.1145/3016963.

[32] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. "Accelerating Eulerian Fluid Simulation with Convolutional Networks". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 3424–3433.

[33] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018.

[34] Roy Frostig, Matthew Johnson, and Chris Leary. *Compiling machine learning programs via high-level tracing*. 2018.

[35] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. "Differentiable Monte Carlo Ray Tracing through Edge Sampling". In: *ACM Transactions on Graphics* 37.6 (Dec. 4, 2018), 222:1–222:11. DOI: 10.1145/3272127.3275109.

[36] Nobuyuki Umetani and Bernd Bickel. "Learning Three-dimensional Flow for Interactive Aerodynamic Design". In: *ACM Transactions on Graphics (SIGGRAPH 2018)* 37.4 (2018). DOI: 10.1145/3197517.3201325.

[37] Xiangyun Xiao, Cheng Yang, and Xubo Yang. "Adaptive learning-based projection method for smoke simulation". In: *Computer Animation and Virtual Worlds* 29.3-4 (2018). e1837 cav.1837, e1837. DOI: https://doi.org/10.1002/cav.1837.

[38] Jonas Zehnder, Rahul Narain, and Bernhard Thomaszewski. "An advection-reflection solver for detail-preserving fluid simulation". In: *ACM Transactions on Graphics* 37 (July 2018), pp. 1–8. DOI: 10.1145/3197517.3201324.

[39] Shichen Liu, Weikai Chen, Tianye Li, and Hao Li. "Soft Rasterizer: A Differentiable Renderer for Image-Based 3D Reasoning". In: *2019 IEEE/CVF International*

Conference on Computer Vision (ICCV) (Oct. 2019), pp. 7707–7716. DOI: 10.1109/ICCV.2019.00780.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[41] Charles R. Harris, K. Jarrod Millman, Stéfan J.van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[42] Philipp Holl, Vladlen Koltun, and Nils Thuerey. *Learning to Control PDEs with Differentiable Physics*. 2020.

[43] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. *DiffTaichi: Differentiable Programming for Physical Simulation*. 2020.

[44] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. "Modular Primitives for High-Performance Differentiable Rendering". In: *ACM Transactions on Graphics* 39.6 (2020).

[45] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. "Modular Primitives for High-Performance Differentiable Rendering". In: *ACM Transactions on Graphics* 39.6 (Nov. 27, 2020), 194:1–194:14. DOI: 10.1145/3414685.3417861.

[46] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*. Aug. 3, 2020. DOI: 10.48550/arXiv.2003.08934. preprint.

[47] Krishna Murthy Jatavallabhula, Miles Macklin, Florian Golemo, Vikram Voleti, Linda Petrini, Martin Weiss, Breandan Considine, Jerome Parent-Levesque, Kevin Xie, Kenny Erleben, Liam Paull, Florian Shkurti, Derek Nowrouzezahrai, and Sanja Fidler. *gradSim: Differentiable simulation for system identification and visuomotor control*. 2021.

[48] Markus Kettunen, Eugene D'Eon, Jacopo Pantaleoni, and Jan Novák. "An Unbiased Ray-Marching Transmittance Estimator". In: *ACM Transactions on Graphics* 40.4 (July 19, 2021), 137:1–137:20. DOI: 10.1145/3450626.3459937.

[49] Yunzhu Li, Shuang Li, V. Sitzmann, Pulkit Agrawal, and A. Torralba. "3D Neural Scene Representations for Visuomotor Control". In: *ArXiv* (July 8, 2021).

[50] Daqi Lin, Chris Wyman, and Cem Yuksel. "Fast Volume Rendering with Spatiotemporal Reservoir Resampling". In: *ACM Transactions on Graphics* 40.6 (Dec. 10, 2021), 279:1–279:18. DOI: 10.1145/3478513.3480499.

[51] Shigenao Maruyama and Shuichi Moriya. "Newton's Law of Cooling: Follow up and Exploration". In: *International Journal of Heat and Mass Transfer* 164 (Jan. 2021), p. 120544. DOI: 10.1016/j.ijheatmasstransfer.2020.120544.

[52] Nils Thuerey, Philipp Holl, Maximilian Mueller, Patrick Schnell, Felix Trost, and Kiwon Um. *Physics-based Deep Learning*. WWW, 2021.

[53] Kiwon Um, Robert Brand, Yun, Fei, Philipp Holl, and Nils Thuerey. *Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers*. 2021.

[54] Cheng Zhang, Zihan Yu, and Shuang Zhao. "Path-Space Differentiable Rendering of Participating Media". In: *ACM Transactions on Graphics* 40.4 (July 19, 2021), 76:1–76:15. DOI: 10.1145/3450626.3459782.

[55] Li-Wei Chen and Nils Thuerey. *Towards high-accuracy deep learning inference of compressible turbulent flows over aerofoils*. 2022.

[56] Shanyan Guan, Huayu Deng, Yunbo Wang, and Xiaokang Yang. *NeuroFluid: Fluid Dynamics Grounding with Particle-Driven Neural Radiance Fields*. June 17, 2022. DOI: 10.48550/arXiv.2203.01762. preprint.

[57] Mohammad Sina Nabizadeh, Stephanie Wang, Ravi Ramamoorthi, and Albert Chern. "Covector Fluids". In: *ACM Trans. Graph.* 41.4 (July 2022). DOI: 10.1145/3528223.3530120.

[58] N. Passalis, S. Pedrazzi, R. Babuska, W. Burgard, D. Dias, F. Ferro, M. Gabbouj, O. Green, A. Iosifidis, E. Kayacan, J. Kober, O. Michel, N. Nikolaidis, P. Nousi, R. Pieters, M. Tzelepi, A. Valada, and A. Tefas. "OpenDR: An Open Toolkit for Enabling High Performance, Low Footprint Deep Learning for Robotics". In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Oct. 2022). DOI: 10.1109/iros47612.2022.9981703.

[59] Sebastian Weiss and Rüdiger Westermann. "Differentiable Direct Volume Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 28.1 (Jan. 2022), pp. 562–572. DOI: 10.1109/TVCG.2021.3114769.

[60] K. Arnavaz, M. Kragballe Nielsen, P. G. Kry, M. Macklin, and K. Erleben. "Differentiable Depth for Real2Sim Calibration of Soft Body Simulations". In: *Computer Graphics Forum* 42.1 (2023), pp. 277–289. DOI: 10.1111/cgf.14720.

[61] Jinxian Liu, Ye Chen, Bingbing Ni, Jiyao Mao, and Zhenbo Yu. *Inferring Fluid Dynamics via Inverse Rendering*. 2023.