Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Controlling Laplacian Eigenfluids

Scientific Students' Association Report

Author:

Barnabás Börcsök

Advisor:

dr. László Szécsi

2022

Contents

Kivonat

Fizikai környezetünk megértése és modellezése egy régóta fennálló kihívás, ami rengeteg tudományterületet érint az időjárás előrejelzéstől kezdve, járművek tervezésén át egészen a számítógépes grafikáig. Fizikai rendszereket általában parciális differenciálegyenletek segítségével írunk le, amiket meglévő numerikus módszerekkel tudunk közelíteni. A szimuláció mellett fontos feladat lehet egy fizikai folyamat irányítása is.

Dolgozatom központi témája, hogy hogyan tudunk gradiens-alapú optimalizálási módszerek számára meglévő tudást átadni fizikai folyamatok működéséről. A folyamat gradiensei a felügyelt tanításban megszokott hibafüggvény értéke mellett arról is tudást adnak át az optimalizációnak („ágensnek"), hogy egy adott pillanatban hozott döntése hogyan befolyásolja nemlineáris fizikai rendszerek lefolyását.

Több kutatási irány összekapcsolásával azt járom körbe, hogyan tudjuk folyadékok viselkedését leírni és irányítani egy csökkentett dimenziójú módszer segítségével. Sűrűségfüggvények advekcióját mintapontokkal közelítem, amiket részecskeként szimulálok a folyadék sebességmezőjében. A módszer előnye, hogy a Laplace-operátor sajátfüggvényeinek lineáris kombinációjaként a sebességmező zárt alakban mintavételezhető. Így a folyadékot a benne áramló anyagokkal együtt anélkül tudjuk szimulálni, hogy a teljes tartományt számon kellene tartani.

Dolgozatomban különböző megközelítésekkel egyre összetettebb problémákat modellezek. Először egyes esetek megoldására nyújtok megoldást gradiens alapú optimalizálás segítségével, majd általánosítva a problémát neurális hálókat tanítok be a fizikai folyamat kívánt módon történő irányítására.

Abstract

Understanding and modeling our environment is a great and important challenge, spanning many disciplines from weather and climate forecast, through vehicle design to computer graphics. Physical systems are usually described by Partial Differential Equations (PDEs), which we can approximate using established numerical techniques. Next to predicting outcomes, planning interactions to control physical systems is also a long-standing problem.

In our work, we investigate the use of Laplacian eigenfunctions to model and control fluid flow. We make use of an explicit description of our simulation domain to derive gradients of the physical simulation. By leveraging current advances in physics-based deep learning, we provide knowledge to the optimization methods on the underlying model equations governing non-linear physical problems.

We introduce different approaches to model physics-based optimization problems of increasing complexity. First, we provide a solution to solve individual problems by gradient-based optimization techniques, building up to a generalized solution by training neural networks to control the physical process to achieve desired outcomes.

Chapter 1

Introduction

Positioned at the crossroads of physical simulation, and deep learning techniques, our work is inspired by current advances in physics-based deep learning. We investigate the general problem of controlling simulation parameters to achieve target outcomes.

Our novel method solves problems in fluid simulation by utilizing gradients of a differentiable physics simulation. More specifically, we use gradients from a reduced order fluid simulation technique based on eigenfunctions of the vector Laplacian operator. Instead of simulating the full domain, a reduced dimensional space is utilized, resulting in significant speed-ups. We show some of the possibilities arising when differentiating a Laplacian eigenfluids simulation, and investigate how the resulting gradients can drive optimization in different scenarios.

In this chapter, we briefly discuss previous research that served as inspiration as well as a base for our current work. We also give an overview of the overall structure of our thesis.

1.1 Previous Work

1.1.1 Fluid Simulation

Most simulation methods are based on either an Eulerian (i.e. grid-based), or Lagrangian (i.e. particle-based) representation of the fluid. For an overview of fluid simulation techniques in computer graphics, see Bridson and Müller-Fischer [3] and Bridson [2].

For advecting marker density in our fluid, as well as a comparative "baseline" simulation, we will use Eulerian simulation techniques, mostly as described by Stam [21].

Reduced Order Modeling of Fluids

Dimension reduction-based techniques have been applied to fluid simulation in multiple previous works. Wiewel et al. [27] demonstrated that functions of an evolving physics system can be predicted within the latent space of neural networks. Their efficient encoder-decoder architecture predicted pressure fields, yielding two orders of magnitudes faster simulation times than a traditional pressure solver.

Recently, Wiewel et al. [26] predicted the evolution of fluid flow via training a convolutional neural network (CNN) for spatial compression, with another network predicting the temporal evolution in this compressed subspace. The main novelty of [26] was the subdi-

vision of the learned latent space, allowing interpretability, as well as external control of quantities such as velocity and density of the fluid.

Eigenfluids

Instead of *learning* a reduced-order representation, another option is to analytically derive the dimension reduction, and its time evolution. De Witt et al. [7] introduced a computationally efficient fluid simulation technique to the computer graphics community. Rather than using an Eulerian grid or Lagrangian particles, they represent fluid fields using a basis of global functions defined over the entire simulation domain. The fluid velocity is reconstructed as a linear combination of these bases.

They propose the use of Laplacian eigenfunctions as these global functions. Following their method, the fluid simulation becomes a matter of evolving basis coefficients in the space spanned by these eigenfunctions, resulting in a speed-up characteristic of reduced-order methods.

Following up on the work of De Witt et al. [7], multiple papers proposed improvements to the use of Laplacian eigenfunctions for the simulation of incompressible fluid flow. Liu et al. [16] extended the technique to handle arbitrarily-shaped domains. Jones et al. [14] used Discrete Cosine Transform (DCT) on the eigenfunctions for compression. Cui et al. [6] improved scalability of the technique, and modified the method to handle different types of boundary conditions. Cui et al. [6] refer to the fluid simulation technique using Laplacian eigenfunctions as *eigenfluids*, which we also adhere to in the following.

1.1.2 Differentiable Solvers

Differentiable solvers have shown tremendous success lately for optimization problems, including training neural network models [10, 11, 18].

Holl et al. [10] address grid-based solvers, noting them as particularly appropriate for dense, volumetric phenomena. They put forth Φ_{Flow}, an open-source simulation toolkit built for optimization and machine learning applications, written mostly in Python.

Physics-based Deep Learning

Despite being a topic of research for a long time [20], the interest in neural network algorithms is a relatively new phenomenon. This is especially true for the use of learning-based methods in physical and numerical simulations, which is a rapidly developing area of current research. Recently, integrating physical solvers in such methods have been shown to outperform previously used learning approaches [25].

Drawing on a wide breadth of current research, Thuerey et al. [24] give an overview of deep learning methods in the context of physical simulations.

1.2 Structure

In chapter 2, we give an overview of mathematical foundations, introducing notation used throughout the text, as well as discuss preliminaries we base later chapters on.

In chapter 3, the basics of physical simulations are introduced. Building on the multivariable calculus introduced in chapter 2, the concept of differentiable physics simulation is introduced.

Diving deeper into a more specific area of physical simulations, chapter 4 gives a deep dive into fluid simulation techniques, discussing the Laplacian eigenfluids method more in-depth in section 4.1.

Finally, as a culmination of all that came before, chapter 5 describes our proposed method of Controlling Laplacian Eigenfluids.

We close our work with a short discussion and possible future directions in chapter 6.

Chapter 2

Mathematical Foundations

This chapter gives a short overview of the mathematical foundations for the techniques we discuss later on, while also establishing the notation used in later sections.

In the following,

$$\mathbf{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \mathbf{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \mathbf{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

will denote the canonical basis vectors.

2.1 Basic Notation

$\mathbf{x} \in \mathbb{R}^n$ is considered a column-matrix, i.e. $\mathbb{R}^n = \mathbb{R}^{n \times 1}$. This also means that $\mathbf{x}^T$ (the transpose of $\mathbf{x}$) is a row-matrix.

We denote the scalar components of a vector $\mathbf{x} \in \mathbb{R}^n$ as $(x_1, x_2, \ldots, x_n)^T$. When $\mathbf{x}$ denotes a position in 3D or 2D space, we also use $\mathbf{x} = (x, y, z)^T$, and $\mathbf{x} = (x, y)^T$, respectively.

Bold uppercase letters denote matrices: $\mathbf{A} \in \mathbb{R}^{n \times m}$, and its elements are indexed with $\mathbf{A}_{i,j}$.

A function $f(x_1, \ldots, x_n)$ is a scalar-valued function $\mathbb{R}^n \to \mathbb{R}$. When $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is a vector field, we will denote it as

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(x_1, \ldots, x_n) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))^T$$

In keeping with the conventions of fluid mechanics literature, we use the letter $\mathbf{u}$ to denote the velocity of a fluid. It is hard to say where this notation came from, but it fits another convention to call the three components of 3D velocity $(u, v, w)^T$ (dropping w for the 2D case).

2.2 Multivariable Calculus

Gradient

The gradient $\boldsymbol{\nabla}$ is a generalization of the derivative to multiple dimensions. The symbol $\boldsymbol{\nabla}$ is called *nabla*, and typically denotes taking partial derivatives along all spatial dimensions.

In three dimensions,

$$\boldsymbol{\nabla} f(x,y,z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right),$$

and in two dimensions,

$$\boldsymbol{\nabla} f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right).$$

It can be helpful to think of the gradient operator as a symbolic vector, e.g. in three dimensions:

$$\boldsymbol{\nabla} = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right).$$

Taking the gradient of vector-valued functions results in a matrix of all its first-order partial derivatives, called the *Jacobian (matrix)*. With $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, its Jacobian takes the form:

$$\boldsymbol{\nabla}\mathbf{f} = \mathbf{J}(\mathbf{f}) = \mathbf{J}_{\mathbf{f}} = \begin{bmatrix} \boldsymbol{\nabla} f_1 \\ \boldsymbol{\nabla} f_2 \\ \vdots \\ \boldsymbol{\nabla} f_n \end{bmatrix}^T = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \tag{2.1}$$

When $\mathbf{f}(x_1, \dots x_n) = (f_1, \dots, f_n)^T$, i.e. $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, mapping the n dimensional Euclidean space onto itself, its determinant is called the *Jacobian determinant*.

Divergence

The divergence operator measures how much the values of a vector field are converging or diverging at any point in space. In three dimensions:

$$\boldsymbol{\nabla} \cdot \mathbf{u}(x,y,z) = \boldsymbol{\nabla} \cdot (u(\mathbf{x}), v(\mathbf{x}), w(\mathbf{x}))^T = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}.$$

Note that the input is a vector, and the output is a scalar, i.e. $\mathbf{u} : \mathbb{R}^3 \to \mathbb{R}^3$, $\boldsymbol{\nabla} \cdot \mathbf{u} : \mathbb{R}^3 \to \mathbb{R}$. Heuristically, in the case of a fluid velocity field $\mathbf{u}$, this translates to a measure of whether a given point acts as a source, or a sink, i.e. whether particles are created or lost in that infinitesimal region. (Later on, we will come back to the notion of a divergence-free fluid.)

Curl

The curl operator measures how much a vector field is rotating around any point. In three dimensions, it is given by the vector

$$\boldsymbol{\nabla} \times \mathbf{u}(x,y,z) = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \\ \partial/\partial z \end{pmatrix}^T \times \begin{pmatrix} u(x,y,z) \\ v(x,y,z) \\ w(x,y,z) \end{pmatrix} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \partial/\partial x & \partial/\partial y & \partial/\partial z \\ u & v & w \end{vmatrix} = \begin{pmatrix} \partial w/\partial y - \partial v/\partial z \\ \partial u/\partial z - \partial w/\partial x \\ \partial v/\partial x - \partial u/\partial y \end{pmatrix}.$$

We can reduce this formula to two dimensions by taking the third component of the expression above, as if we were looking at the three-dimensional vector field $(u, v, 0)$.

Thus, the two-dimensional curl is a scalar:

$$\boldsymbol{\nabla} \times \mathbf{u}(x,y) = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \end{pmatrix} \times \begin{pmatrix} u(x,y) \\ v(x,y) \end{pmatrix} = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}.$$

We can also interpret this value as the third component of a three-dimensional vector, perpendicular to the vector field $(u, v, 0)$:

$$\boldsymbol{\nabla} \times \mathbf{u}(x,y) = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \end{pmatrix} \times \begin{pmatrix} u(x,y) \\ v(x,y) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \end{pmatrix}.$$

Material Derivative

For a velocity $\mathbf{u}(t,x,y,z) = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$, we define the material derivative as

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u},$$

a special case of the total derivative. As x, y, z describe the spatial position of a particle traveling through space over time, they depend on time t themselves, i.e. $\mathbf{u}(t, x(t), y(t), z(t))$. Utilizing the chain rule, we can arrive on the above definition by taking the total derivative of $\mathbf{u}(t, x(t), y(t), z(t))$, and rearranging the terms:

$$\begin{aligned}
\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \frac{\partial \mathbf{u}}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}t} + \frac{\partial \mathbf{u}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathbf{u}}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{\partial \mathbf{u}}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}t} \\
&= \frac{\partial \mathbf{u}}{\partial t}1 \quad + \frac{\partial \mathbf{u}}{\partial x}u \quad + \frac{\partial \mathbf{u}}{\partial y}v \quad + \frac{\partial \mathbf{u}}{\partial z}w \\
&= \frac{\partial \mathbf{u}}{\partial t}1 \quad + u\frac{\partial \mathbf{u}}{\partial x} \quad + v\frac{\partial \mathbf{u}}{\partial y} \quad + w\frac{\partial \mathbf{u}}{\partial z} \\
&= \frac{\partial \mathbf{u}}{\partial t} + \left(\mathbf{u} \cdot \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right]\right) \cdot \mathbf{u} \\
&= \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla}) \cdot \mathbf{u}.
\end{aligned}$$

Laplacian

The Laplacian operator is defined as the divergence of the gradient of a scalar function f. In general, for $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, it is given by

$$\Delta f = \boldsymbol{\nabla}^2 f = \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}.$$

In three dimensions, this reduces to

$$\boldsymbol{\nabla} \cdot \boldsymbol{\nabla} f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2},$$

and in two dimensions,

$$\boldsymbol{\nabla} \cdot \boldsymbol{\nabla} f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

Taking the divergence of the gradient corresponds to an averaging of how much a value at a given position differs from its neighborhood. As an example, we can look at a non-static scalar field, with a rate of change proportional to its Laplacian with proportionality α:

$$\frac{\partial \Phi}{\partial t} = \alpha \boldsymbol{\nabla}^2 \Phi,$$

which is the well-known heat equation, and governs diffusion of the field. It is essentially a smoothing operation that results in an averaging of values at every point in space. Using heat as an analogy, hot and cold spots diffuse throughout the field, resulting in a more uniform distribution of the heat in the field, eventually reaching a uniform distribution of heat.

Vector Laplacian

The Laplacian can also be applied to vector (or even matrix) fields, and the result is simply the Laplacian of each component.

Essentially, the vector Laplacian is what we have been building towards so far, as this operator is going to be the cornerstone of the eigenfluids simulation technique in section 4.1. As such, we will show some important properties of this operator, and will return to these in later sections.

The vector Laplacian of a vector field $\mathbf{f}$ is defined as

$$\underbrace{\Delta \mathbf{f}}_{\text{vector Laplacian}} = \underbrace{\boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \mathbf{f})}_{\text{gradient of the divergence}} - \underbrace{\boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \mathbf{f})}_{\text{curl of curl=curl}^2} = \text{grad}(\text{div}(f)) - \underbrace{\text{curl}(\text{curl}(f))}_{\text{curl}^2(f)}$$

In Cartesian coordinates, the vector Laplacian simplifies to taking the Laplacian of each component:

$$\Delta \mathbf{f}(x, y, z) = (\boldsymbol{\nabla} \cdot \boldsymbol{\nabla})\mathbf{f} = \begin{bmatrix} \Delta f_1 \\ \Delta f_2 \\ \Delta f_3 \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 f_1}{\partial x^2} + \frac{\partial^2 f_1}{\partial y^2} + \frac{\partial^2 f_1}{\partial z^2} \\ \frac{\partial^2 f_2}{\partial x^2} + \frac{\partial^2 f_2}{\partial y^2} + \frac{\partial^2 f_2}{\partial z^2} \\ \frac{\partial^2 f_3}{\partial x^2} + \frac{\partial^2 f_3}{\partial y^2} + \frac{\partial^2 f_3}{\partial z^2} \end{bmatrix}. \tag{2.2}$$

We can see that these are equivalent by writing out $\text{grad}(\text{div}(\mathbf{f})) - \text{curl}(\mathbf{f})$ explicitly:

$$\boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \mathbf{f}) - \boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \mathbf{f}) = \begin{bmatrix} \frac{\partial^2 f_1}{\partial x^2} + \cancel{\frac{\partial^2 f_2}{\partial x \partial y}} + \cancel{\frac{\partial^2 f_3}{\partial x \partial z}} \\ \cancel{\frac{\partial^2 f_1}{\partial y \partial x}} + \frac{\partial^2 f_2}{\partial y^2} + \cancel{\frac{\partial^2 f_3}{\partial y \partial z}} \\ \cancel{\frac{\partial^2 f_1}{\partial z \partial x}} + \cancel{\frac{\partial^2 f_2}{\partial z \partial y}} + \frac{\partial^2 f_3}{\partial z^2} \end{bmatrix} - \begin{bmatrix} \cancel{\frac{\partial^2 f_2}{\partial x \partial y}} - \frac{\partial^2 f_1}{\partial y^2} - (\frac{\partial^2 f_1}{\partial z^2} - \cancel{\frac{\partial^2 f_3}{\partial x \partial z}}) \\ \cancel{\frac{\partial^2 f_3}{\partial y \partial z}} - \frac{\partial^2 f_2}{\partial z^2} - (\frac{\partial^2 f_2}{\partial x^2} - \cancel{\frac{\partial^2 f_1}{\partial y \partial x}}) \\ \cancel{\frac{\partial^2 f_1}{\partial z \partial x}} - \frac{\partial^2 f_3}{\partial x^2} - (\frac{\partial^2 f_3}{\partial y^2} - \cancel{\frac{\partial^2 f_2}{\partial z \partial y}}) \end{bmatrix}, \tag{2.3}$$

where the mixed second order partial derivatives cancel each other out, giving us equation (2.2).

Differential Identities

It can be shown [13] that for any smooth function $\mathbf{u}$,

$$\begin{aligned} \boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} \times \mathbf{u}) &\equiv 0, \\ \boldsymbol{\nabla} \times (\boldsymbol{\nabla} \mathbf{u}) &\equiv 0. \end{aligned} \tag{2.4}$$

The idea behind the Helmholtz or Hodge decomposition is that any vector field $\mathbf{u}$ can be written as the composition of a divergence-free part, and a curl-free part. Making use of Equations (2.4), we can write the divergence-free part as the curl of something, and the curl-free part can be written as the gradient of something else. In three dimensions,

$$\mathbf{u} = \boldsymbol{\nabla} \times \boldsymbol{\Psi} - \boldsymbol{\nabla} p,$$

where $\boldsymbol{\Psi}$ is a vector-valued potential function, and p is a scalar potential function. In two dimensions, Ψ is also scalar:

$$\mathbf{u} = \boldsymbol{\nabla} \times \Psi - \boldsymbol{\nabla} p.$$

This decomposition technique becomes highly relevant for incompressible fluid flows, where we would like to make our velocity field $\mathbf{u}$ divergence-free (i.e. no particles should be lost or created). Simulation techniques often decompose an intermediate fluid field $\mathbf{u}^{t+1}$ into a divergence-free part, and interpreting p as the pressure that is keeping the fluid flow divergence-free, usually throwing away the values of p immediately.

Rearranging equation (2.3), we can derive another useful identity:

$$\boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \mathbf{u}) \equiv \boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \mathbf{u}) - \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} \mathbf{u}.$$

2.3 Optimization

Iterative optimization algorithms look for a solution by iteratively applying some update step Δ to some starting parameter $\mathbf{x}_0$, giving an estimation of how to approach some optimal parameter $\mathbf{x}^*$, with the goal to continuously lower the error as defined by a loss function L. We address optimization scenarios where the target is to minimize a scalar-valued loss function $L(\mathbf{x}) : \mathbb{R}^N \to \mathbb{R}$ with respect to one of its inputs:

$$\arg\min_{\mathbf{x}} L(\mathbf{x}) = \mathbf{x}^*.$$

Among the vast number of established optimization algorithms, Gradient Descent (GD) is the most basic and straightforward. Making use of the Jacobian matrix as defined in (2.1), it gives us an update step Δ given a parameter $\mathbf{x}$, consisting of the transposed Jacobian matrix of $\mathbf{f}$ scaled by a scalar *learning rate* λ. Repeatedly applying the update step $\Delta(\mathbf{x}) = -\lambda J^T(\mathbf{x})$, the steps of a Gradient Descent (GD) optimization can be written out as:

$$
\begin{aligned}
&\mathbf{x}_0 \\
&\mathbf{x}_1 = \mathbf{x}_0 - \lambda\Delta = \mathbf{x}_0 - \lambda J_L^T(\mathbf{x}_0) = \mathbf{x}_0 - \lambda \boldsymbol{\nabla} L^T(\mathbf{x}_0) \\
&\quad \vdots \\
&\mathbf{x}_t = \mathbf{x}_{t-1} - \lambda J_L^T(\mathbf{x}_{t-1}) \\
&\mathbf{x}^* = \mathbf{x}_t - \lambda J_L^T(\mathbf{x}_t),
\end{aligned} \tag{2.5}
$$

which is exactly what we will be utilizing in our first couple of optimization scenarios in chapter 5. Note that as our loss is scalar-valued, the transposed Jacobian matrix of L has the same dimensionality as the input $\mathbf{x}$, i.e. $J_L^T \in \mathbb{R}^{N\times 1}$; $\mathbf{x} \in \mathbb{R}^{N\times 1}$, making them both a column vector of size N, which means that they can indeed be added together.

We can also think about these optimization steps as continuously moving towards some locally observed lowest point in the error landscape. The gradient $\boldsymbol{\nabla} L$ is giving us the direction of steepest *ascent*, which means that the opposite direction, $-\boldsymbol{\nabla} L$ will be the direction of the steepest *descent* locally, guiding us towards some (potentially only local) error minimum.

We show examples of utilizing the GD steps as written in (2.5) for minimizing the error between a simulated and target state of physical simulations in sections 5.1,5.2.2, and 5.2.3.

2.3.1 Neural Networks

The goal of Deep Learning (DL) is to approximate an unknown function

$$\mathbf{f}^*(\mathbf{x}) = \mathbf{y}^*,$$

where $\mathbf{y}^*$ denotes *ground truth* solutions. $\mathbf{f}^*(\mathbf{x})$ is approximated by a Neural Network (NN) representation

$$\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{y},$$

where $\boldsymbol{\theta}$ is a vector of *weights*, influencing the output of the NN. DL is about finding $\boldsymbol{\theta}$ parameters such that the outputs of the NN match the $\mathbf{y}^*$ outputs of the original function $\mathbf{f}^*$ as closely as possible, as measured by some scalar-valued loss function L:

$$\arg\min_{\boldsymbol{\theta}} L\big(\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}^*\big).$$

In the simplest case, using a mean-square error (also known as L_2 norm):

$$\arg\min_{\boldsymbol{\theta}} L_2\big(\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}^*\big) = \arg\min_{\boldsymbol{\theta}} |\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}) - \mathbf{y}^*|_2^2. \tag{2.6}$$

As discussed in section 2.3, we can use the gradients of the loss function L with respect to the weights $\boldsymbol{\theta}$ (i.e. $\partial L / \partial \boldsymbol{\theta}$) to solve equation (2.6), yielding the optimal $\boldsymbol{\theta}$ parameters. We optimize, i.e. *train* our NN with a Stochasitc Gradient Descent (SGD) optimizer, such as Adam [15].

In the case of a fully-connected NN, we can write its i^{th} layer as

$$\mathbf{o}^i = \sigma\left(\mathbf{W}_i \mathbf{o}^{i-1} + b_i\right), \tag{2.7}$$

where $\mathbf{o}^i$ is the output of the i^{th} layer, σ is a non-linear activation function, such as the rectified linear unit (ReLU) function, and $\mathbf{W}_i$ and b_i are the weight matrix and the bias of layer i, respectively. We call $\mathbf{W}_i$ and b_i the parameters of the NN, and collect their values from all layers in $\boldsymbol{\theta}$.

It is worth explicitly noting that the term Deep Learning (DL) is referring to a "deep" Neural Network (NN), meaning that many layers are stringed after each other. People usually refer to a NN as DL, when its architecture consists of more than three layers. In turn, both NNs and DL are a subset of the more general paradigm of Machine Learning (ML). Finally, Artificial Intelligence (AI) is the broadest term used to classify any and all techniques that aim to create a form of (human-like) intelligence in computers.

In the context of DL, it is helpful to think of the derivative as *function sensitivity*, denoting how a small change in an input variable changes the output of the function. For finding the $\boldsymbol{\theta}$ parameters of a NN, this is exactly what we need: how to tweak $\boldsymbol{\theta}$ to reduce the output of a loss function $L(\mathbf{x}, \theta)$.

As we already showed in equation (2.2), the chain rule gives us the derivatives of composite functions. For a multivariable function $f : \mathbb{R}^2 \to \mathbb{R}$, it can be summarized as:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} \tag{2.8}$$

and expressed with vector notation

$$= \frac{\partial}{\partial t} f \circ \mathbf{x}(t) = \frac{\partial}{\partial t} f(\mathbf{x}(t)) = \boldsymbol{\nabla} f \cdot \mathbf{x}(t) \tag{2.9}$$

As we will use it in section 5.2.4, a fully-connected NN can be written as a function composition of layers and activation functions σ. In the case of a simple fully-connected NN with layer weights $\mathbf{W}_i$ and $\mathrm{ReLU}(\mathbf{x}) = \max(0, \mathbf{x})$ activation functions, this becomes:

$$\begin{aligned}
\text{linear function:} \quad & f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{W}\mathbf{x} \\
\text{2 layers:} \quad & f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x}) \\
\text{3 layers:} \quad & f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{W}_3 \max(\mathbf{0}, \mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x})) \\
\text{4 layers:} \quad & f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{W}_4 \max(\mathbf{0}, \mathbf{W}_3 \max(\mathbf{W}_2 \max(\mathbf{0}, \mathbf{W}_1 \mathbf{x}))) \\
& \dots
\end{aligned} \tag{2.10}$$

Note that, as in equation (2.7), there is an additional $+b_i$ scalar addition at each layer that we did not write out in (2.10) for clarity.

Given some target values $\mathbf{y}^*$, also known as *ground truth data*, we can use a scalar (also known as *regression*) loss to measure the error of the predictions of the NN:

$$\text{L1 loss} \quad L(\mathbf{x}, \mathbf{y}^*, \boldsymbol{\theta}) = \frac{1}{N} \sum_i^N \|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}) - \mathbf{y}^*\|_1 \tag{2.11}$$

$$\text{MSE (or } L_2^2\text{) loss} \quad L(\mathbf{x}, \mathbf{y}^*, \boldsymbol{\theta}) = \frac{1}{N} \sum_i^N \|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}) - \mathbf{y}^*\|_2^2, \tag{2.12}$$

where the Mean Square Error (MSE) is the squared L_2 loss. The L_2 norm of a vector is its length in Euclidean space. In 3D: $L_2(\mathbf{x}) = \sqrt{x^2 + y^2 + z^2}$, and we will also denote it later on as $|\cdot|_2^2$.

In the context of Physics-based Deep Learning (PBDL), it is especially important to understand the flow of gradients, as in the next chapter, we will integrate differentiable physical simulations into this learning setup.

Regarding nomenclature, in classical literature, *adjoint method* and *reverse mode differentiation* are equivalent names for backpropagation.

Chapter 3

Physical Simulations

Modeling the world around us is a longstanding problem of science. For many physical processes, model equations exist, describing how a given system evolves through time. From weather and climate forecasts [22] over quantum physics [19], to the control of plasma fusion [12], or optimizing the shape of vehicles [23], it has become an integral part of engineering applications to use numerical methods to derive solutions from model equations.

In this section, we build up an understanding of modeling physical phenomena with Partial Differential Equations (PDEs). We also introduce the notion of Differentiable Physics (DP) after a brief introduction to classical numerical methods.

3.1 Partial Differential Equations

PDEs are the most fundamental description of evolving systems from quantum mechanics to turbulent flows. PDEs are equations relating the partial derivatives of some unknown function. For a physical system $\mathbf{u}(\mathbf{x}, t)$, the governing PDE can be written as

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{P}\left(\mathbf{u}, \frac{\partial \mathbf{u}}{\partial \mathbf{x}}, \frac{\partial^2 \mathbf{u}}{\partial \mathbf{x}^2}, \ldots, \mathbf{y}(t)\right), \tag{3.1}$$

where $\mathcal{P}$ models the physical behavior of the system, and $\mathbf{y}(t)$ denotes an (optional) external force factor.

3.1.1 Numerical Methods

Analytic solutions (i.e. closed-form expressions) can be found only for a very small subset of Partial Differential Equations (PDEs). The main idea behind numerical methods is to discretize an equation, reducing a continuous equation (such as 3.1) to a finite number of unknowns. We discretize the temporal dimension by introducing a Δt step size. For spatial dimensions, a typical solution is discretization by assigning values to grid cells or particles: we call these the Eulerian, and the Lagrangian perspective, respectively.

Numerical Integration with Explicit Schemes

As we introduce only a small subset of numerical methods, we refer to Baraff and Witkin [1] to give an introduction to numerically approximating Partial Differential Equations (PDEs) from a computer graphics perspective for use in physically based modeling. They discuss the respective shortcomings of the techniques in a visual way.

Euler's method (introduced back in 1768) starts with an initial value, and steps along the tangent of the function. Given a first order Ordinary Differential Equation (ODE)

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = f(x, y)$$

describing the derivative of a function $x(t) = y$, and starting with an initial condition x_0, y_0, and step size Δt at time t_0, an Euler step

$$y_1 = y_0 + \Delta t f(x_0, y_0)$$

gives the y_1 estimation for $x(t_0 + \Delta t)$. With this, we can compute $f(x_1, y_1)$ and so on, giving us an estimated trajectory for $x(t)$. Note that $y_n \neq x(t_0 + n\Delta t)$, as the function $x(t)$ is unknown.

Euler's method gives us a first order approximation of the function. The *midpoint method* achieves second order accuracy by evaluating the derivative at an intermediate half step:

$$\begin{aligned} \tilde{y} &= y_0 + \frac{\Delta t}{2} f(x_0, y_0) \\ y_1 &= y_0 + \Delta t f(x_0 + \Delta t/2, \tilde{y}). \end{aligned}$$

The midpoint is actually a 2^{nd} order Runge-Kutta (RK) method. The RK family of integrators can be used to construct integrators of arbitrary order. In practice, the 4^{th} order RK gives a good compromise between accuracy and computation cost:

$$\begin{aligned} k_1 &= f(x_0, y_0) && \text{(compute a first estimate of the slope)} \\ k_2 &= f(x_0 + \Delta t/2, y_0 + \Delta t/2 k_1) && \text{(predict the tangent at midpoint)} \\ k_3 &= f(x_0 + \Delta t/2, y_0 + \Delta t/2 k_2) && \text{(correct the estimate)} \\ k_4 &= f(x_0 + \Delta t, y_0 + \Delta t k_3) && \text{(predict slope with full step)} \\ y_0 &= y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) && \text{(finally, perform step with weighted slopes).} \end{aligned}$$

3.2 Differentiable Physics

Given $\mathbf{u}(\mathbf{x}, t)$, described by a PDE as in Equation (3.1), a regular solver can move the system forward in time via Euler steps:

$$\mathbf{u}(\mathbf{x}, t) = \text{Solver}\left[\mathbf{u}(t_i), \mathbf{y}(t_i)\right] = \mathbf{u}(t_i) + \Delta t \cdot \mathcal{P}\left(\mathbf{u}(t_i), \ldots, \mathbf{y}(t_i)\right), \tag{3.2}$$

computing a solution trajectory $\mathbf{u}(t)$, that approximates a solution to the PDE. Although (3.2) is differentiable, it is not well-suited to solve optimization problems, as gradients can only be approximated by finite differencing, and (especially for high-dimensional or continuous systems), this method would become computationally expensive, because a full trajectory needs to be computed for each optimizable parameter.

Holl et al. [10] address this issue via the use of differentiable solvers, backpropagating through the chain of operations via analytic derivatives. Differentiable solvers can efficiently compute the derivatives with respect to any of the inputs $\partial \mathbf{u}(t_{i+1})/\partial \mathbf{u}(t_i)$ and $\partial \mathbf{u}(t_{i+1})/\partial \mathbf{y}(t_i)$.

3.2.1 Loss Functions for Differentiable Physics

In chapter 5, we will introduce optimization scenarios, where we take a physical simulation, and by tweaking its parameters, our goal is to reach a target state. To measure "reaching a target state", and how good we are doing, we have to introduce a measure for the goodness of a solution. More specifically, this measure will be given as a loss function, measuring how *bad* we are doing, conventionally called the *error*. Knowing how to decrease this error lets us find a better solution. A gradient is something that tells us exactly this: how a change in an input variable (i.e. simulation parameter) changes the output (i.e. the error from the loss function). With this idea in the back of our mind, we first lay some groundwork for setting up these kinds of problems.

Given a model $\mathcal{M}$ of a physical system with initial state $\mathcal{M}^0$, and observed part $o(\mathcal{M})$, our goal is to match some target observation o^* at time t. With $\mathcal{P}$ describing the time evolution of $\mathcal{M}$, we can write the resulting loss function as

$$L(\mathcal{M}^0, o^*) = \left| o\Big(\mathcal{P}^t(\mathcal{M}^0)\Big) - o^* \right|, \tag{3.3}$$

where $|\cdot|$ denotes some distance metric between the observations.

As a toy example, in section 3.2.2, $\mathcal{M}$ describes a projectile traveling in the 2D (x, y) plane. Here, $\mathcal{P}$ describes the projectile flying through the air, colliding with the ground floor at $y = 0$.

In our experiments in chapter 5, $\mathcal{M}$ describes some particles $\mathbf{p}$ advected in a fluid flowing with velocity $\mathbf{u}$, and $\mathcal{P}$ solving the Navier-Stokes equations 4.1.

By phrasing equation 3.3 in a general way, our intent is to highlight that there is a vast applicability of physics-based loss functions well beyond our discussion and examples. The general requirement of solving a problem with DP is a differentiable solver for $\mathcal{P}$, which is often not readily available.

3.2.2 Comparison with Supervised Learning

Figure 3.1 shows the results of a comparison between a supervised and Differentiable Physics (DP) manner of teaching two networks for solving a ballistic problem. The main setting is an object being thrown from position (x, y), with velocity v and angle α. The point of impact x_{final} at ground level $y = 0$ is given by the function $f(x, y, v, \alpha)$. Both the supervised and the DP network approximate the inverse function $f^{-1}(x_{\text{final}}) : \mathbb{R} \mapsto \mathbb{R}^4$, mapping the final position to some initial values the object was thrown with. We can already see that this problem is multimodal, i.e. it has multiple solutions.

In both cases, the same network architecture (and initialization)is used, with the same number of training examples. Both supervised and DP teaching uses an L_2 norm to measure the error between the point of impact resulting from the predicted initial values and the intended position.

Looking at figure 3.1, it is evident that the DP network is able to get orders of magnitude closer than the supervised network, which has no knowledge of the underlying physical

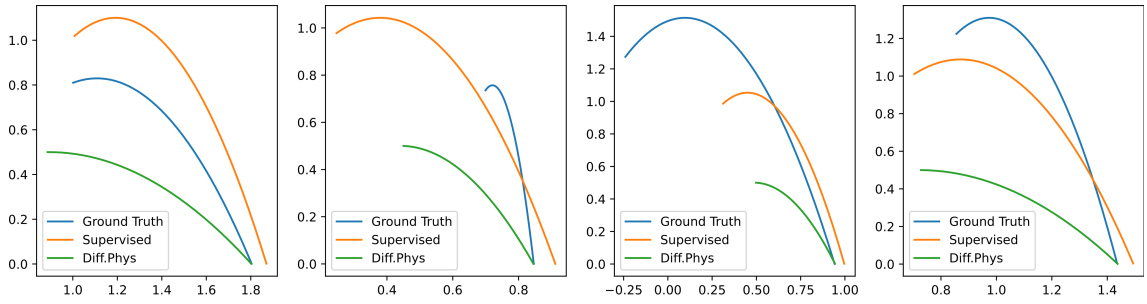


Figure 3.1: *Learning to throw. The goal is to give an initial velocity v, angle α, and position* $\mathbf{x}$ *for a projectile, that hits a target at the ground floor when simulated. The supervised network is outperformed by the DP approach, as it always hits closer to the target by orders of magnitude than its supervised counterpart. The only difference between the two models is the way they derive their gradients from the same L_2 error: while the DP network gains an understanding of the underlying physical system via gradients of the simulation, the supervised network only sees examples of input-output pairs, where multimodality becomes an inherent problem. (Figure recreated after Holl [9].)*

system, and it's best guess is to interpolate between the closest data points it has seen during training, which results in a coarse approximation. Also, as the result space to this problem is not unimodal, the supervised model is further thrown off, and will give values that are the results of an averaging of examples seen during training. This means that even if we increase the training data, the supervised model struggles to properly solve this problem.

Following the notation introduced in section 2.3.1, we can write the loss functions as

$$L(x_{\text{target}}) = \left| \mathcal{P}\Big(\mathbf{f}(x_{target}, \boldsymbol{\theta})\Big) - x_{target} \right|_2^2 \tag{3.4}$$

for the differentiable physics loss, and

$$L(x_{\text{target}}, y^*) = \left| \mathbf{f}\Big(x_{target}; \boldsymbol{\theta}\Big) - y^* \right|_2^2 \tag{3.5}$$

for the supervised loss. Notice how (3.4) does not require ground truth y^* initial parameters, but measures how close its predictions were via running a differentiable physics simulation $\mathcal{P}$ at training time to calculate the point of impact. The gradients of $\mathcal{P}$ are then part of the backpropagation chain. This type of *solver-in-the-loop* technique was introduced by Um et al. [25], outperforming previously used learning approaches. For more details and examples on Physics-based Deep Learning (PBDL) in general, also see Thuerey et al. [24].

In summary, gradients of a differentiable physics simulation give a learning method an inductive physical bias, i.e. existing knowledge to solve a problem at hand. Although a further investigation of the modality of our experiments remains a direction for future research, in section 5.1, we show how our optimization makes use of an inductive physical bias to find solutions of an inverse problem.

Chapter 4

Fluid Simulation

Simulating convincing fluid dynamics is a continuing challenge of computer graphics, especially when considering real-time applications. There are multiple established ways to simulate fluids, the most widespread being Eulerian (i.e. grid-based) and Lagrangian (i.e. particle-based) methods. Here, we restrict ourselves to discussing Eulerian methods, and also introduce the Laplacian Eigenfunction method, introduced by De Witt et al. [7].

The dynamics of fluids are governed by the Navier-Stokes Equations:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} = -\frac{1}{\rho}\boldsymbol{\nabla} p + \nu \boldsymbol{\nabla}^2 \mathbf{u} + \mathbf{f}, \tag{4.1}$$

where $\mathbf{u}$ is the velocity of the fluid, ρ is the density, p is the scalar pressure field, ν is the viscosity constant, and $\mathbf{f}$ denotes external forces. For incompressible fluids, the divergence-freeness also has to hold, i.e. $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$

Even though equation (4.1) already describes the evolution of the fluid as a Partial Differential Equation (PDE), it is too complex for simply stepping it forward in time with Euler steps. Instead, a technique called *operator splitting* is applied for numerical simulations, where each term is treated individually, and their effect is combined to fully approximate the original equation. We give a short overview of each term to get a general understanding of fluid simulation, first treating the problem in an Eulerian way (i.e. sampling $\mathbf{u}$ on a grid), building up our way towards the Laplacian Eigenfunction method discussed in section 4.1. For a more complete overview of established fluid simulation techniques, see [3] and [2].

Equation (4.1) is usually split by separating out the advection part, the external force part, and the pressure/incompressibility part. When viscosity is important, that can also be separated. This means, we work out methods for solving these simpler equations:

$$\begin{aligned} \frac{\mathrm{d}q}{\mathrm{d}t} &= 0 \quad \text{(advection)} \\ \frac{\partial \mathbf{u}}{\partial t} &= \mathbf{f} \quad \text{(external forces)} \\ \frac{\partial \mathbf{u}}{\partial t} + \frac{1}{\rho}\boldsymbol{\nabla} p &= 0 \\ \text{such that} \quad \boldsymbol{\nabla} \cdot \mathbf{u} &= 0. \quad \text{(pressure, enforcing incompressibility)} \end{aligned}$$

A generic quantity q is used in the advection equation, because as we also show later on in our experiments, we may be interested in advecting other field quantities than just the

velocity $\mathbf{u}$. For the advection part, the Advect$(\mathbf{u}, \Delta t, q)$ algorithm is introduced: it advects quantity q through the velocity field $\mathbf{u}$ for a time interval Δt.

For the external forces, any traditional numerical integration approach, such as a simple Euler step can be used: $\mathbf{u}^{t+1} = \mathbf{u}^t + \Delta t \mathbf{f}$. (See section 3.1.1.)

For calculating the pressure, an algorithm Project$(\Delta t, \mathbf{u})$ calculates and applies just the right amount of pressure to the velocity field to make it divergence-free, and also enforce any solid wall boundary conditions. The term "project" comes from the fact that the algorithm essentially projects $\mathbf{u}$ to the closest divergence-free velocity field, and interpreting the difference between these two fields as a pressure resulting from "particles" being too close together. We do not go into the details of a pressure solve here, as our velocity field $\mathbf{u}$ from the Laplacian Eigenfunction method (section 4.1) will already be divergence-free by construction.

Note that the order in which these algorithms are being applied matters a lot, as the advection must be done on a divergence-free field. Putting all of these together, a basic fluid simulation algorithm can be written as:

$\mathbf{u}^0 \leftarrow$ an initial divergence-free velocity field
for $t = 0, 1, 2 \ldots$ **do**
  $\Delta t \leftarrow$ a suitable time step to go from t_n to t_{n+1}
  $\widetilde{\mathbf{u}} \leftarrow \text{Advect}(\mathbf{u}^n, \Delta t, \mathbf{u}^n)$
  $\widetilde{\mathbf{u}} \leftarrow \widetilde{\mathbf{u}} + \Delta t \mathbf{f}$
  $\mathbf{u}^{n+1} \leftarrow \text{Project}(\widetilde{\mathbf{u}}, \Delta t)$
end for ▷ $[\mathbf{u}^0, \ldots, \mathbf{u}^t]$ is the simulated fluid flow for t time steps.

Advect

Before moving on, we briefly discuss the Advect algorithm on grids. As already shown in section 2.2, we can write out the advection $\frac{\mathrm{d}q}{\mathrm{d}t}$ in 3D as

$$\frac{\partial \mathbf{u}}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}t} + \frac{\partial \mathbf{u}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathbf{u}}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{\partial \mathbf{u}}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}t}$$

A simple and physically-motivated advection approach, called the semi-Lagrangian method was introduced by Stam [21]. Advection in a Lagrangian frame is trivial: moving particles through a velocity field $\mathbf{u}$ automatically solves $\frac{\mathrm{d}q}{\mathrm{d}t} = 0$. (Which is something we will fundamentally base our experiments on in chapter 5.)

To find a new value q at some point $\mathbf{x}$ in space, the semi-Lagrangian method conceptually finds the particle, that ended up there from the previous time step. As we know that the "particle" ended up at $\mathbf{x}$ from the previous time step, we can trace it backwards through the velocity field to find where it came from, grabbing the old value of q at that previous point. When simulating on a grid, but the start point was not on the grid, then interpolation is applied.

For tracing a particle at position $\mathbf{x}$ *backwards* in time by Δt through a velocity field $\mathbf{u}$, we can utilize integration schemes introduced in section 3.1.1. The simplest way is an Euler step:

$$\mathbf{x}_{\text{old}} = \mathbf{x} - \Delta \mathbf{u}(\mathbf{x}).$$

In practice, it is advised to apply at least a midpoint method:

$$\tilde{\mathbf{x}} = \mathbf{x} - \frac{1}{2}\Delta t \mathbf{u}(\mathbf{x})$$
$$\mathbf{x}_{\text{old}} = \mathbf{x} - \Delta t \mathbf{u}(\tilde{\mathbf{x}}).$$

Once we calculated $\mathbf{x}_{\text{old}}$, we can now simply grab the value of q from the previous time step from there, and assign it to our new position in the current time step, i.e. $q^t(\mathbf{x}) = q^{t-1}(\mathbf{x}_{\text{old}})$.

For our smoke simulation examples in chapter 5 , we are using a MacCormak advection scheme [17] that uses a forward as well as a backward lookup to estimate and correct the error of the semi-Lagrangian advection.

4.1 The Laplacian Eigenfunction Method

De Witt et al. [7] introduced the method of using Laplacian eigenfunctions for fluid simulation. Cui et al. [6] addressed scalability and generalization issues, and referred to the technique as *eigenfluids*, which we also adhere to.

The main idea is to express the velocity field $\mathbf{u}(\mathbf{x})$ via the linear combination of N global functions:

$$\mathbf{u}(\mathbf{x}) = \sum_i^N w_i \Phi_i(\mathbf{x}), \tag{4.2}$$

where the elements of $\mathbf{w} = [w_0, \ldots, w_N]$ are called *basis coefficients*, and Φ_i are *basis functions*.

As our basis functions, we choose eigenfunctions of the vector Laplacian operator $\Delta = \boldsymbol{\nabla}^2 = \text{grad}(\text{div}) - \text{curl}^2$ (see section 2.2), which further simplifies to $\boldsymbol{\nabla}^2 = -\text{curl}^2$ for divergence-free fields.

Besides being eigenfunctions of the vector Laplacian operator, if we further require our basis fields $\Phi_{\mathbf{k}}$ to be divergence-free, and to satisfy a free-slip boundary condition, our basis functions are fully characterized by

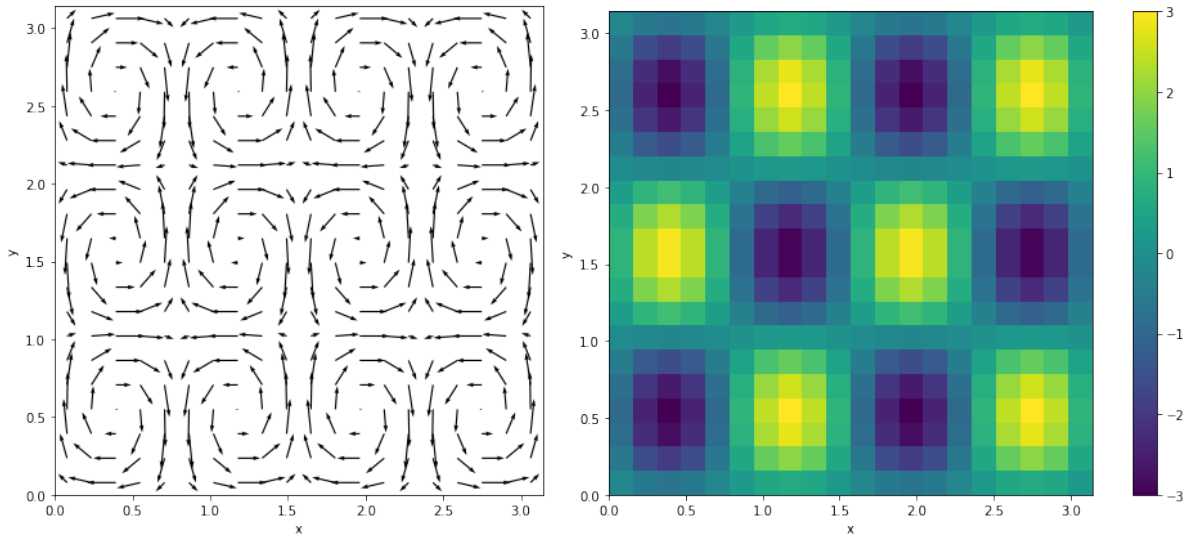
$$\nabla^2 \Phi_{\mathbf{k}} = \lambda_{\mathbf{k}} \Phi_{\mathbf{k}}$$
$$\nabla \cdot \Phi_{\mathbf{k}} = 0$$
$$\Phi_{\mathbf{k}} \cdot \mathbf{n} = 0 \text{ at } \partial D,$$

where $\mathbf{n}$ is the normal vector at boundary ∂D.

Closed-form expressions of $\Phi_{\mathbf{k}}$ exist on the two dimensional $D \in [0, \pi] \times [0, \pi]$ square domain [5]. Notating the two scalar components in the x and y directions $\Phi_{\mathbf{k}} = (\Phi_{\mathbf{k},x}, \Phi_{\mathbf{k},y})$, we can write them as

$$\Phi_{\mathbf{k},x}(x, y) = \eta_{\mathbf{k}}(k_2 \sin(k_1 x) \cos(k_2 y)) \tag{4.3}$$
$$\Phi_{\mathbf{k},y}(x, y) = -\eta_{\mathbf{k}}(-k_1 \cos(k_1 x) \sin(k_2 y)),$$

where $\mathbf{k} = (k_1, k_2) \in \mathbb{Z}^2$ is the *vector wave number*, $\lambda_{\mathbf{k}} = -(k_1^2 + k_2^2)$ is the *eigenvalue*, and $\eta_{\mathbf{k}} = \frac{1}{-\lambda_{\mathbf{k}}} = \frac{1}{k_1^2 + k_2^2}$ is a normalization parameter. Cui et al. [6] use $\frac{1}{\sqrt{-\lambda}} = \frac{1}{\sqrt{k_1^2 + k_2^2}}$



(a) *Velocity field* $\Phi_{(4,3)}$. **(b)** *Curl field* $\nabla \times \Phi_{(4,3)} = \phi_{(4,3)}$.

Figure 4.1: *Visualizing* $\Phi_{(4,3)}$ *sampled on a* 20×20 *grid in our simulation domain* $D = [0, \pi] \times [0, \pi]$.

for normalization, but we keep the non-root version, as we did not observe any noticeable difference between the two during implementation.

As an example, $\Phi_{(4,3)}(x, y)$ is visualized in figure 4.1. In appendix A.1, we also plot the first 16 basis fields.

This is a good time to mention that higher wave lengths corresponding to smaller scales of vorticity has a very literal meaning in our simulation. As we choose to truncate the spectrum of Φ_k at some number N, the error we incur is well defined: we lose the ability to simulate vortices smaller than a given scale. Also, as we will see later on, this correspondence to spatial scales of vorticity lets us control the viscosity (i.e. energy decay) in relation to the scales of vortices by modifying the base coefficients. By setting the magnitude of each basis coefficient to decay with a time constant equal to the eigenvalue, we get the physically correct behavior that small vortices dissipate faster than large vortices.

Vorticity Basis Fields

For the simulation technique, we further require the vorticity field $\boldsymbol{\omega} = \nabla \times \mathbf{u}$ and a set of vorticity basis functions $\phi = \nabla \times \Phi$. Taking the curl (as introduced in section 2.2) of the velocity basis fields $\Phi_{\mathbf{k}}$ from equation (4.3) gives us the vorticity basis fields:

$$\begin{aligned}
\phi_{\mathbf{k}} = \nabla \times \Phi_{\mathbf{k}} &= \frac{\partial \Phi_{\mathbf{k},y}}{\partial x} - \frac{\partial \Phi_{\mathbf{k},x}}{\partial y} \\
&= -\mu_{\mathbf{k}} k_1 \sin(k_2 y) k_1 (-sin(k_1 x)) - \mu_{\mathbf{k}} k_2 \sin(k_1 x) k_2 (-\sin(k_2 y)) \\
&= \mu_{\mathbf{k}} \sin(k_1 x) \sin(k_2 y)(k_1^2 + k_2^2) = \sin(k_1 x) \sin(k_2 y)
\end{aligned}$$

We can also interpret this value as the third component of a 3D vector,

$$\phi_{\mathbf{k}} = \nabla \times \Phi_{\mathbf{k}} = \begin{pmatrix} 0 \\ 0 \\ \sin(k_1 x) \sin(k_2 y) \end{pmatrix}. \tag{4.4}$$

As the velocity field $\mathbf{u}$ and vorticity field $\boldsymbol{\omega}$ are orthogonal, the vorticity basis functions $\phi_{\mathbf{k}}$ have only a normal component at the boundary, and satisfy

$$\boldsymbol{\nabla}^2 \phi_{\mathbf{k}} = \lambda_{\mathbf{k}} \phi_{\mathbf{k}} \tag{4.5}$$

$$\phi_{\mathbf{k}} \times \mathbf{n} = 0 \text{ at } \partial D. \tag{4.6}$$

Dynamics

The vorticity formulation of the (4.1) Navier-Stokes equation is

$$\dot{\boldsymbol{\omega}} = \text{Advect}(\mathbf{u}, \boldsymbol{\omega}) + \nu \boldsymbol{\Delta} \boldsymbol{\omega} + \boldsymbol{\nabla} \times \mathbf{f}, \tag{4.7}$$

where $\boldsymbol{\omega} = \boldsymbol{\nabla} \times \mathbf{u}$, and $\mathbf{f}$ denotes external forces.

We now apply the basic building blocks of fluid simulation introduced in the beginning of chapter 4 to derive the time evolution of a fluid's velocity by the continuous change of the coefficient vector $\mathbf{w}$. We derive the time derivative $\frac{\mathrm{d}\mathbf{w}}{\mathrm{d}t} = \dot{\mathbf{w}}$ in terms of only the coefficient vector $\mathbf{w}$.

The velocity $\mathbf{u}$ formed by the eigenfunctions $\Phi_{\mathbf{k}}$ is intrinsically divergence free, and hence no pressure projection step is needed, when simulating the fluid dynamics in this coordinate system.

Damping due to viscosity is given by the first-order differential equation $\dot{w}_k = \nu \lambda_k w_k$, which corresponds to a point-wise exponential decay similar to [21]:

$$w_k^{t+1} = w_k^t e^{\nu \lambda_k \Delta t}.$$

External forces $\mathbf{f}$ given on the original domain $D \in [0, \pi] \times [0, \pi]$ can be incorporated by projecting $\mathbf{f}$ to the velocity basis, representing them as a base coefficient vector $\mathbf{f}_w$ in the coordinate system with basis $\{\Phi_k\}$. The contribution of the external forces is then defined as:

$$\dot{\mathbf{w}} = \mathbf{f}_w.$$

Advection

Looking at equation (4.7), we can see that after dealing with both viscosity and the external forces, the only right-hand term left is the advection.

Following De Witt et al. [7], we perform projection to a Laplacian eigenfunction basis by substituting the expansions $\boldsymbol{\omega} = \sum_i w_i \phi_i, \mathbf{u} = \sum_j w_j \Phi_j$, and $\dot{\boldsymbol{\omega}} = \sum_k \dot{w} \phi_k$ into equation (4.7). With rearranging the terms through linearity of operators, we get

$$\sum_k \dot{w} \phi_k = \underbrace{\sum_i^N \sum_j^N w_i w_j \text{Advect}(\Phi_i, \phi_j)}_{\text{Advection}} + \underbrace{\nu \sum_i^N \boldsymbol{\nabla}^2 w_i \phi_i}_{\text{Viscosity}} + \underbrace{\text{curl}(\mathbf{f})}_{\text{External forces}}. \tag{4.8}$$

The $\text{Advect}(\Phi_i, \phi_j)$ terms represent the non-linear advection of basis fields. As the terms are constant, we precompute them, and the basis coefficients of the results are stored in "a set of $\mathbf{C}_k$ matrices" (De Witt et al. [7]), resulting in N number of $N \times N$ matrices, or equivalently, a "3^{rd} order advection tensor $\mathfrak{C}$" (Cui et al. [6]). The dimensions of $\mathbf{C}$ and

$\mathfrak{C}$ are both $N \times N \times N$. In the following, we will refer to these precomputed advection values as N $\mathbf{C}_k$ matrices. The respective works also propose different ways to precompute these values.

De Witt et al. [7] propose

$$\mathbf{C}_g[h,i] = \Big(\boldsymbol{\nabla} \times (\phi_h \times \Phi_i)\Big) \cdot \phi_g. \tag{4.9}$$

Cui et al. [6] improve on (4.9) by using the method introduced by Liu et al. [16]:

$$\mathfrak{C}(g,h,i) = \int_D (\boldsymbol{\nabla} \times \Phi_i) \cdot (\Phi_g \times \Phi_h) \mathrm{d}D, \tag{4.10}$$

noting improvements such as preserving the anti-symmetry of the tensor by construction, i.e. $\mathfrak{C}(g,h,i) = -\mathfrak{C}(h,g,i)$.

In our implementation, we keep with computing the basis coefficients according to equation (4.9), as it was working well enough for our purposes of differentiable physics simulation. Also note that as these are just constant values of the same dimension and used the same way, it is trivial to change them out at will in an implementation.

Time Evolution Equation

Putting it all together, the time derivative of each basis coefficient is

$$\dot{w_k} = \mathbf{w}\mathbf{C}_k\mathbf{w} + \nu\lambda_k w_k + f_{w,k}, \tag{4.11}$$

governing all of our reduced-order fluid simulations in chapter 5.

Time Integration

Any standard numerical technique (such as the ones discussed in section 3.1.1) can be used to integrate equation (4.11) forward in time. However, De Witt et al. [7] describe a preferred technique that in order to preserve kinetic energy, renormalizes the energy of the fluid simulation after each integration step. De Witt et al. [7] show that due to the orthogonality of the basis functions, the total kinetic energy can be calculated as a sum of squared coefficients. With this final addition, the final algorithm that we implemented for our experiments in chapter 5 can be described as:

$e_1 = \sum_i^N \mathbf{w}[i]^2$ ▷ store kinetic energy of velocity field
for $k = 1 \dots N$ **do**
  $\dot{\mathbf{w}}[k] = \mathbf{w}^T\mathbf{C}_k\mathbf{w}$ ▷ matrix-vector products for advection
end for
$\mathbf{w}$ += $\dot{\mathbf{w}}\Delta t$ ▷ explicit Euler integration step
$e_2 = \sum_i^N \mathbf{w}[i]^2$ ▷ calculate energy after time step
$\mathbf{w}$ *= $\sqrt{e_1/e_2}$ ▷ renormalize energy
for $k = 1 \dots N$ **do**
  $\dot{\mathbf{w}}[k]$ *= $e^{\lambda_k \Delta t}$ ▷ dissipate energy for viscosity
  $\dot{\mathbf{w}}[k]$ += $\mathbf{f}[k]$ ▷ add external forces
end for.

Precalculating the Advection Matrices

Before moving on, we discuss how to compute each element of the $\mathbf{C}_k$ matrices in an implementation.

For finding the structure coefficients of the $\mathbf{C}_k$ matrices, we can start with writing out the advection operator $\text{Advect}(\Phi_i, \phi_j) = \boldsymbol{\nabla} \times (\phi_j \times \Phi_i)$ as

$$\begin{aligned}\boldsymbol{\nabla} \times (\phi_j \times \Phi_i) = \Big(&\frac{1}{\lambda_i} i_1 j_2 \cos(i_1 x)\cos(j_2 y)\sin(j_1 x)\sin(i_2 y) \\ &- \frac{1}{\lambda_i} i_2 j_1 \cos(j_1 x)\cos(i_2 y)\sin(i_1 x)\sin(j_2 y)\Big).\end{aligned}$$

The trigonometric identity $\cos(\alpha)\sin(\beta) = \frac{1}{2}\sin(\alpha+\beta) - \frac{1}{2}\sin(\alpha-\beta)$ enables factoring to a suitable expression which is in the span of $\{\phi_k\}$:

$$\begin{aligned}\text{Advect}(\Phi_i, \phi_j) = \frac{1}{4\lambda_i}\Big[&(i_1 j_2 - i_2 j_1)\phi_{i_1+j_1, i_2+j_2} \\ &- (i_1 j_2 + i_2 j_1)\phi_{i_1+j_1, i_2-j_2} \\ &+ (i_1 j_2 + i_2 j_1)\phi_{i_1-j_1, i_2+j_2} \\ &- (i_1 j_2 - i_2 j_1)\phi_{i_1-j_1, i_2-j_2}\Big].\end{aligned}$$

The resulting coefficients are[1]

$$\begin{aligned}\mathbf{C}_{i_1+j_1, i_2+j_2}[i,j] &= -\frac{1}{4(i_1^2 - i_2^2)}(i_1 j_2 - i_2 j_1) \\ \mathbf{C}_{i_1+j_1, i_2-j_2}[i,j] &= \frac{1}{4(i_1^2 + i_2^2)}(i_1 j_2 + i_2 j_1) \\ \mathbf{C}_{i_1-j_1, i_2+j_2}[i,j] &= -\frac{1}{4(i_1^2 + i_2^2)}(i_1 j_2 + i_2 j_1) \\ \mathbf{C}_{i_1-j_1, i_2-j_2}[i,j] &= \frac{1}{4(i_1^2 - i_2^2)}(i_1 j_2 - i_2 j_1).\end{aligned} \tag{4.12}$$

Note: We are using $\mathbf{k} = (k_x, k_y) = (k_1, k_2)$ interchangeably. A single (non-vector) k is also used for indexing over all of the basis fields – a slight, but very useful abuse of notation, stemming from the fact that a suitable remapping from vector wave lengths (k_1, k_2) to positive integers is necessary in an implementation, as seen in the indexing of the $\mathbf{C}_k$ values in (4.12) above.

[1] Deriving the underlying equations by hand, and consulting the original implementation by De Witt et al. [7], one of the signs is purposefully different from the appendix in [7].

Chapter 5

Controlling Laplacian Eigenfluids

Many real world applications require us to optimize for some parameters of a physics-based problem. A toy example would be to optimize for some initial angle and velocity of a projectile to hit some target (see figure 3.1). As a more involved example, Chen et al. [4] address finding the best shape to minimize drag. These kinds of inverse problems have been around for quite some time in engineering applications.

Building on all of the previous ideas, we now introduce our investigation into the use of eigenfluids in fluid control problems, making use of their explicit closed-form description of a velocity field (equation (4.3)) to derive gradients used for optimization. Our main proposition is to achieve reduced-order modeling-like speed increase: in lieu of representing the fluid on a grid, we reconstruct the velocity field only at discrete points in space, while simulating the dynamics of the fluid in a reduced dimensional space as in equation (4.2).

In the following, we showcase different optimization scenarios, where we try out different aspects of controlling eigenfluids via Differentiable Physics (DP) gradients. (See section 3.2.1.)

We start with examples of "traditional" optimization scenarios. By "traditional", we mean finding individual solutions to problems via some optimization technique – in our case, Gradient Descent (GD). Moving further, we look for generalized solutions to a set of problems by training Neural Networks (NNs).

After trying out multiple recent frameworks aimed at differentiable simulations [18, 11], we implemented all of our experiments using Φ_{Flow} [10].

5.1 Matching Velocities

To verify the feasibility of our technique before moving on to more involved setups, our most straightforward optimization scenario is finding an initial basis coefficient vector $\mathbf{w}_0 \in \mathbb{R}^N$ for an eigenfluid simulation using $N = 16$ basis fields, such that when simulated for t time steps, the reconstructed $\mathcal{R}\mathbf{w}_t = \mathbf{u}_t$ velocity field will match some precalculated $\mathbf{u}^* : [0, \pi] \times [0, \pi] \to \mathbb{R}^2$ target velocity field:

$$L(\mathbf{w}) = \left| \mathcal{R}\mathcal{P}^t(\mathbf{w}) - \mathbf{u}^* \right|_2^2, \tag{5.1}$$

where $\mathcal{P}^t(\mathbf{w}) = \mathcal{P} \circ \mathcal{P} \cdots \circ \mathcal{P}(\mathbf{w})$ is the physical simulation of base coefficients $\mathbf{w}$ (in the reduced dimension) t times.

For the optimization, we initialize a $\mathbf{w}_{\text{init}} \in \mathbb{R}^N$ vector with random numbers (from a normal/Gaussian distribution), and run the eigenfluid simulation for t time steps, after which, we measure the error as given by loss function (5.1). Relying on backpropagation to derive the necessary gradients, we use the GD optimization method as introduced in equations (2.5) to iteratively find a vector $\mathbf{w}_{optim}$, yielding a low scalar loss $L(\mathbf{w}_{optim})$.

To be able to make some further evaluation of the end results possible, we step an eigenfluid solver for time t to precalculate the target $\mathbf{u}^*$ velocity field, sampled on a 32×32 grid. We denote the initial base coefficient vector of this reference simulation $\mathbf{w}^*$, but keep in mind that the optimization has absolutely zero knowledge of this value, as it sees only the $32 \times 32 \times 2$ velocity values of $\mathbf{u}^*$ at time t. Also, these values could have been precalculated from any other kind of fluid simulation as well, or even just initialized randomly. Deriving $\mathbf{u}^*$ as the result of an eigenfluid simulation has the added benefit of exposing to us a solution $\mathbf{w}^*$ that we can use to compare with the solution of the optimizer.

We test this setup on two scenarios, with differing the number of time steps simulated: first with $t = 16$, and then with $t = 100$.

For $t = 16$ simulation steps, starting from a loss of around 400, the first 100 GD optimization steps with $\lambda = 10^{-3}$ reduced the loss to under 1.0, while 200 steps further decreased it to under $4 * 10^{-4}$, with each further step still continuously decreasing the error.

Naturally, this very basic method has its limits. Optimizing for initial coefficients based solely on that when reconstructed on a 32×32 grid after 100 steps of a non-linear simulation, they should match a given velocity field, proved to be a substantially harder problem, as even a relatively small error can accumulate into major deviations over these longer time steps, resulting in much less stable gradients. With using the same learning rate, the optimization diverged almost instantly. With some tuning of the learning rate λ in the range of $[10^{-4}, 10^{-8}]$, we were able to get the loss below 0.14. (Starting from an initial loss of 320 from the random initialization.)

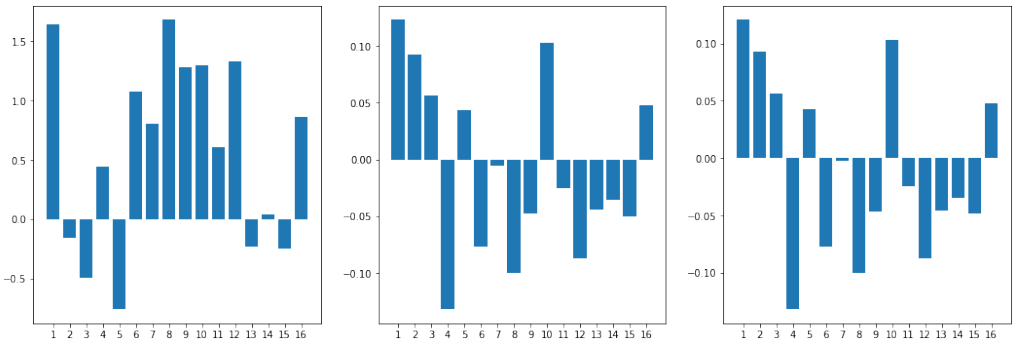
We visualize the results of these two scenarios in Figure 5.1. It is interesting to observe that even though the optimization had absolutely no knowledge of $\mathbf{w}^*$, only a comparison with a precomputed $\mathbf{u}^*$ velocity field at time step 100, the optimized $\mathbf{w}_{optim}$ vector already starts to look similar to $\mathbf{w}^*$. Keep in mind that this is not guaranteed at all, as highlighted with the learning to throw example on figure 3.1. In some other cases of running this optimization setup, we also observed $\mathbf{w}_{optim}$s that are completely different from $\mathbf{w}^*$. Due to the physical constraints of the eigenfluids simulation, in these cases the optimization could not change any of the 16 values of $\mathbf{w}_{optim}$ locally in a way that would further reduce the loss below some small number, and was stuck in a local minima of the parameter space.

Although there are a number of ways to tweak this setup, we can already verify from these results that the flow of the gradients is working, and is ready to be tested in more advanced scenarios.

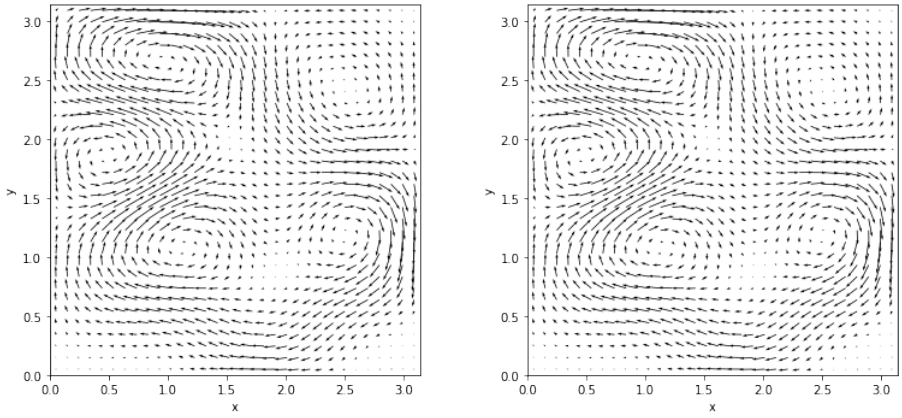
5.2 Controlling Shape Transitions

In the following, we showcase an optimization scenario, with the target of controlling the transition between two marker shapes in a fluid simulation setup. [1] The work of Holl et al. [10] formulated this problem in an Eulerian representation, with explicitly simulating the

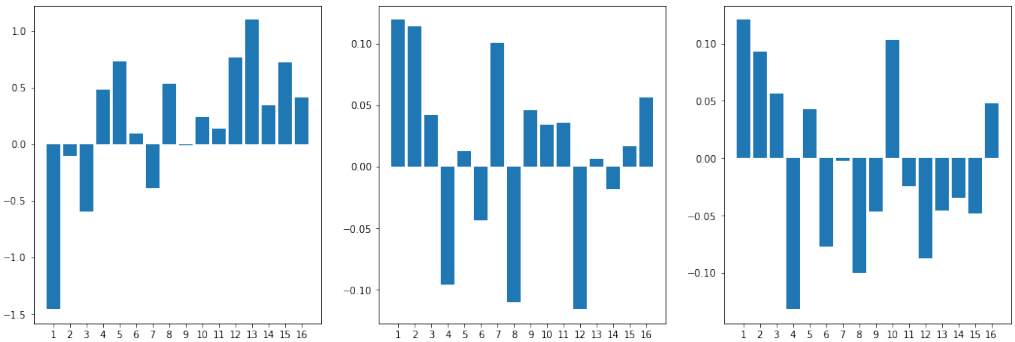
[1] Note that we use the terms *smoke*, *marker*, *density*, and *scalar valued density/marker function* interchangeably throughout the text.



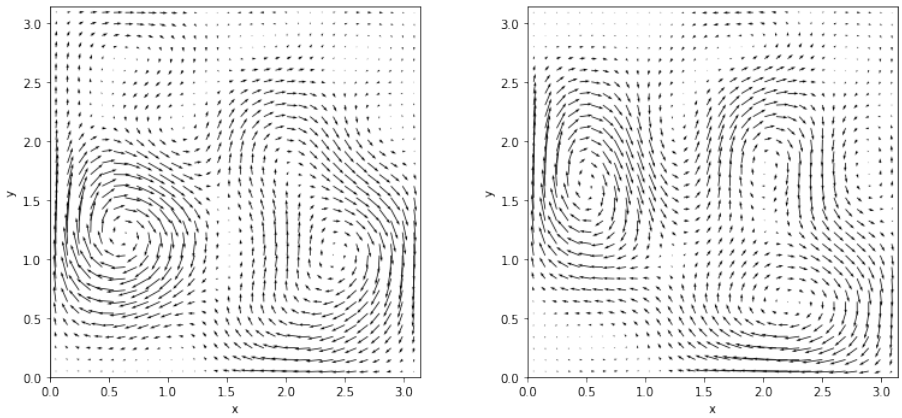
(a) $\mathbf{w}_{init}$, $\mathbf{w}_{optim}$, *and* $\mathbf{w}^*$, *optimizing for velocity field after* 16 *time steps*



(b) *Target* $\mathbf{u}^*$, *and* $\mathbf{u}^{16}$, *reconstructed from* $\mathcal{P}^{16}(\mathbf{w}_{optim})$



(c) *Initial basis coefficients* $\mathbf{w}_{init}$, $\mathbf{w}_{optim}$, *and* $\mathbf{w}^*$, *optimizing for velocity field after* 100 *time steps*



(d) *Target* $\mathbf{u}^*$, *and* $\mathbf{u}^{100}$, *reconstructed from* $\mathcal{P}^{100}(\mathbf{w}_{optim})$

Figure 5.1: *Results of optimizing for an initial* $\mathbf{w}_0$ *basis coefficient vector that matches a target velocity field* $\mathbf{u}^*$ *when reconstructed after simulating for* t *time steps.*

shapes as scalar marker densities being advected by the velocity field of the simulated fluid.

Playing to the strength of the eigenfluids method, our method makes use of an explicit, closed-form description of the fluid velocity $\mathbf{u}$ as in equation (4.3). We stay independent of a grid resolution, and approximate the 2D shapes via a set of sample points. We reconstruct the velocity field only partially at these discrete points as needed for the advection of these particles. This results in both a faster fluid simulation as well as optimization as compared to fully simulating an $N \times N$ grid, advecting a marker density, and backpropagating gradients of a physical simulation with much more degrees of freedom.

We formulate three different control problems, each with a different mean to exert control over the fluid simulation.

- First, in a similar vein to the problem statement in section 5.1, we are looking for an initial coefficient vector $\mathbf{w}_0$ of an eigenfluids simulation, such that when simulated for t time steps, the reconstructed velocity field advects some initial points to the desired positions.
- Second, we optimize for some force vector $\mathbf{f} \in \mathbb{R}^{t \times N}$, such that $\mathbf{f}_t \in \mathbb{R}^N$ applied as external force to each time step of an eigenfluid simulation, it yields the desired outcome.
- Finally, we generalize the problem to looking for a function that exerts the necessary control force at time t, such that particles currently at positions $\mathbf{p}_t$ end up at target positions $\mathbf{p}_{t+1}$ at the next time step. We formulate this third task as a Neural Network (NN) model in the form $\mathbf{f}(\mathbf{p_t}, \mathbf{p_{t+1}}, \mathbf{w}_t, \boldsymbol{\theta})$, also passing in the current basis coefficient vector $\mathbf{w}_t$, and optimizing for its parameters $\boldsymbol{\theta}$ to yield the desired outcome.

In each of these tasks, a velocity field $\mathbf{u} = \mathcal{R}\mathbf{w}$ advects a set of initial points $\mathbf{p}_0 = [\mathbf{p}_0^0, \ldots, \mathbf{p}_0^i]$ to line up with target positions $\mathbf{p}_t = [\mathbf{p}_t^0, \ldots, \mathbf{p}_t^i]$. We formulate this as

$$L(\mathbf{w}, \mathbf{p}_0, \mathbf{p}_t) = \left|\mathcal{P}^t(\mathbf{p}_0, \mathbf{w}) - \mathbf{p}_t\right|_2^2 = \sum_i \left|\mathcal{P}^t(p_0^i, \mathbf{w}) - p_t^i\right|_2^2, \tag{5.2}$$

where $\mathcal{P}^t(\mathbf{w}, \mathbf{p}) = \underbrace{\mathcal{P} \circ \mathcal{P} \circ \cdots \circ \mathcal{P}(\mathbf{w}, \mathbf{p})}_{t \text{ times}}$ denotes the physical simulation of base coefficients $\mathbf{w}$ and points $\mathbf{p}$ in $\mathbf{u} = \mathcal{R}\mathbf{w}$, the velocity field reconstructed from $\mathbf{w}$. We use a simple mean-square error (also known as squared L_2 norm) for measuring the error.

5.2.1 Sampling

Advection of some scalar quantity in a fluid is an abstract problem that describes many real-world phenomena. We can think of the transport of some ink dropped into water, clouds in the air, or some buoyant smoke rising. Phenomena such as these can be modeled as a density function $\psi(\mathbf{x})$ defined over the simulation domain D. In a fluid with velocity $\mathbf{u}$, and $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$ (i.e. the fluid is incompressible), the advection is governed by the equation

$$\frac{\partial \psi}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla} \psi = 0.$$

In Eulerian fluid simulation methods [21], both $\mathbf{u}$ and ψ are sampled on grids, numerically approximating the evolution of the field quantities. Instead, our method proposes sampling the density function at discrete particle positions, thus rephrasing the process in a Lagrangian way.

In the context of Laplacian eigenfluids, a Lagrangian viewpoint is especially inviting, as the explicit description of the fluid velocity $\mathbf{u}$ (equation (4.3)) allows us to reconstruct $\mathbf{u}$ only partially, while keeping the simulation of the fluid dynamics in a reduced dimensional space. In a forward physics simulation, this can already lead to substantial speed-ups, but this formulation seems especially promising when the backpropagation of variables is desired, such as the optimization scenarios introduced herein.

A straightforward way to define a shape is

$$\psi(\mathbf{x}) = \begin{cases} 1, & \text{inside the shape} \\ 0, & \text{outside the shape.} \end{cases} \tag{5.3}$$

Sampled on an $N \times N$ grid, this is equivalent to a binary image with a resolution of $N \times N$. Moreover, when sampled on a grid, and advected, it is straightforward to interpret the resulting grid and its values as a grayscale image with values $[0, \dots, 1]$.

Often used in 3D scanning, reconstruction and scene understanding problems, a Signed Distance Function (SDF) can be defined as the distance to the surface (in 2D, the edge) of an object, with positive values outside, and negative values inside. In our implementation, we define our shapes as SDFs. For example, a circle with radius r and center $\mathbf{o} = (o_x, o_y)^T$ is defined as

$$\mathrm{SDF}_{\mathrm{circle}}(\mathbf{x}, \mathbf{o}, \mathbf{r}) = |\mathbf{x} - \mathbf{o}| - r = \sqrt{(x - o_x)^2 + (y - o_y^2)} - r.$$

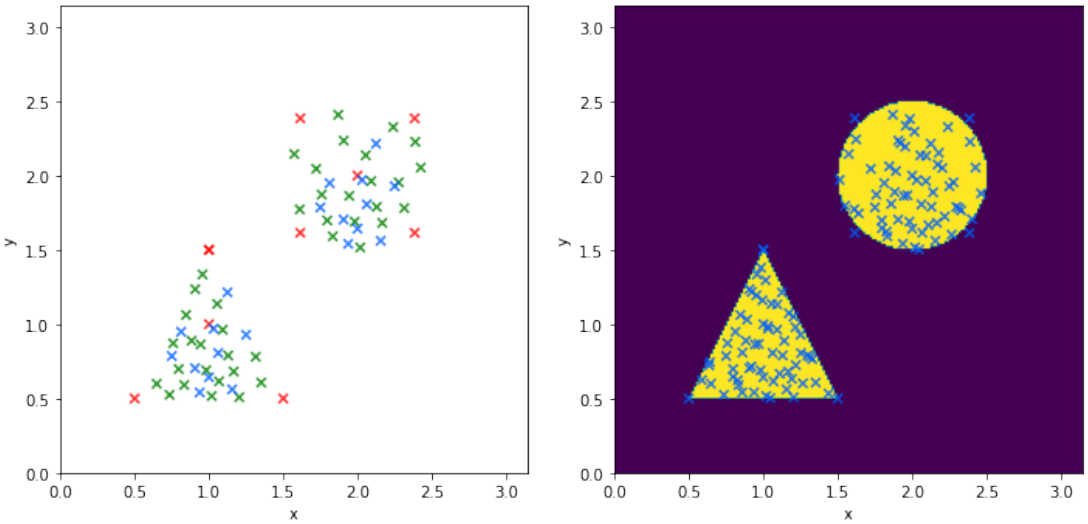
For simulating (and visualizing) the advection dynamics of these shapes, we transform the SDFs to a binary form as in equation (5.3).

As we neither want to lose too much information about our original function, nor want to keep track of an unnecessary number of points, the feasibility of this method necessitates an efficient sampling of $\psi(\mathbf{x})$. We use a simple rejection-based sampling technique. Transforming the shapes to fit inside the unit rectangle $[0, 1] \times [0, 1]$, we generate random points $\mathbf{p}_{\mathrm{sample}} \in [0, 1] \times [0, 1]$, rejecting them if they lie outside the shape.

As we consider shape transitions given start and target shapes S_0 and S_{t}, it is important to take into consideration the connection between these shapes. To balance finding spatial correspondences between the shapes, while still approximating their unique shapes, we sample O overlapping, and U unique points. For the overlapping points, we accept only $\mathbf{p}_{\mathrm{sample}} \in S_0 \cup S_{\mathrm{t}}$, i.e. we reject points that are not inside both shapes (transforming both shapes to fit inside the unit square for the sampling). For the unique points we sample a different set of points for each shape.

To generate low-discrepancy, quasi-random 2D coordinates, we use a Halton sequence [8], giving deterministic coordinates, given coprimes p and q. Using one set of primes for sampling O overlapping points, and another set of primes for sampling U unique points can give us further overlapping points, as the proposed (but potentially rejected) sequence of points will be the same for both shapes.

We further generate $T = 5$ trivial points that are hand-picked to best resemble the given shape, as well as line up between different shapes. We choose these to be the center, upper right, upper left, lower left, and lower right corners of the shape.



(a) *The $O = 30$ overlapping (blue), $U = 30$ unique (green), and $T = 5$ trivial (red) points for each shape.*

(b) *Sample points plotted over $\psi_{triangle} + \psi_{circle}$.*

Figure 5.2: *Sampling strategy for transitioning from a triangle to a circle. Halton series with base $(2, 7)$ and $(3, 11)$ were used to generate the overlapping and unique positions, respectively.*

In conclusion, our final set of $\mathbf{p}_0$ initial, and $\mathbf{p}_t$ target sample positions are given by concatenating the O overlapping, U unique, and T trivial points for each shape, resulting in two set of sample points $\mathbf{p}_0, \mathbf{p}_t \in \mathbb{R}^{O+U+T}$.

Figure 5.2 shows the result of our sampling strategy for a triangle and a circle shape.

5.2.2 Optimizing for Initial Velocity

As introduced the problem in the beginning of the chapter (see equation 5.2), our goal is to find an initial velocity field $\mathcal{R}\mathbf{w} = \mathbf{u}$ that advects points $\mathbf{p}_0$ to line up with target positions $\mathbf{p_t}$ after t steps. We can write optimizing for base coefficients $\mathbf{w}$ as:

$$\arg\min_{\mathbf{w}} \left| \mathcal{P}^t(\mathbf{p}^0, \mathbf{w}) - \mathbf{p}^t \right|_2^2.$$

Making use of the differentiability of our physical simulator $\mathcal{P}$, and the multivariable chain rule for deriving the gradient of the above $\mathcal{P}^t = \mathcal{P} \circ \cdots \circ \mathcal{P}$ function composition, we can derive its gradient with respect to the initial coefficients:

$$\frac{\partial \mathcal{P}^t(\mathbf{w}, \mathbf{p})}{\partial \mathbf{w}}.$$

Finally, as introduced in (2.5), we simply iterate a GD optimizer to find a (good enough) solution for our above minimization problem:

$$\mathbf{w}_{\text{better}} = \mathbf{w} - \lambda \frac{\partial L(\mathbf{w}, \mathbf{p}_0, \mathbf{p}_t)}{\partial \mathbf{w}},$$

where L is the same as in equation (5.2):

$$L(\mathbf{w}, \mathbf{p}_0, \mathbf{p}_t) = \left| \mathcal{P}^t(\mathbf{p}_0, \mathbf{w}) - \mathbf{p}_t \right|_2^2.$$

The main difficulty of this non-linear optimization problem lies in that we have no control over the natural flow of the fluid besides supplying an initial $\mathbf{w}_0$ vector.

We showcase two different setups in Figure 5.3, with the details of both experiments described in Table 5.1.

Table 5.1: *Details of the 2 optimization scenarios shown in Figure 5.3*

	Figure 5.3d	**Figure 5.3e**
N	16	36
Sampling size for smoke simulation	32	32
Eigenfluid initialization time	6.19 sec	68.47 sec
Time for 51 optimization steps	108.05 sec	230.48 sec
Initial loss	2.3	2.19
Final loss	0.08	0.09
Number of overlapping points O	0	30
Number of unique points U	0	30
Number of trivial points T	5	0

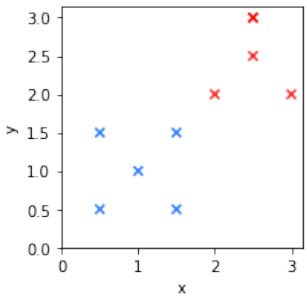
5.2.3 Control Force Estimation

In this scenario, we optimize for a force vector $\mathbf{f} \in \mathbb{R}^{t \times N}$, such that $\mathbf{f}_t \in \mathbb{R}^N$ applied as external force at each time step of an eigenfluid simulation, some initial positions $\mathbf{p}_0$ will be advected to target positions $\mathbf{p}_t$ after t time steps:

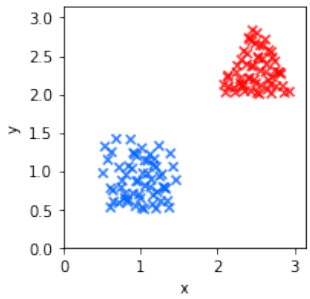
$$\arg\min_{\mathbf{f}} \left| \mathcal{P}^t(\mathbf{p}_0, \mathbf{w}, \mathbf{f}) - \mathbf{p}_t \right|_2^2,$$

where $\mathcal{P}^t(\mathbf{p}_0, \mathbf{w}, \mathbf{f}) = \mathcal{P} \circ \cdots \circ \mathcal{P}^t(\mathbf{p}_0, \mathbf{w}, \mathbf{f})$ denotes simulating the physical system for t time steps, applying $\mathbf{f}_t$ force at each time step.

Results of the optimization are shown in Figure 5.4.

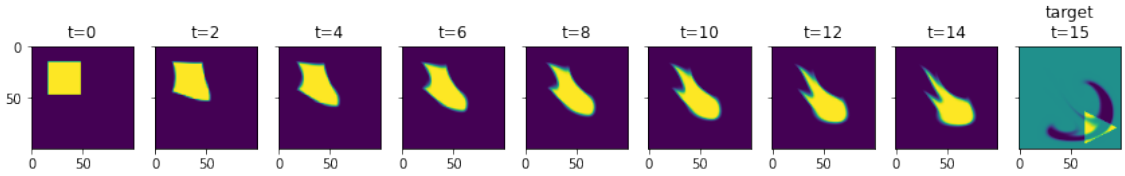


(a) $O = 0$, $U = 0$, $T = 5$

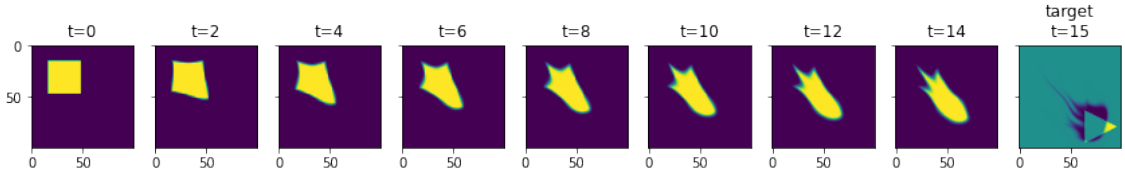


(b) $O = 30$, $U = 30$, $T = 0$

(c) *Initial (blue), and target (red) sample points.*

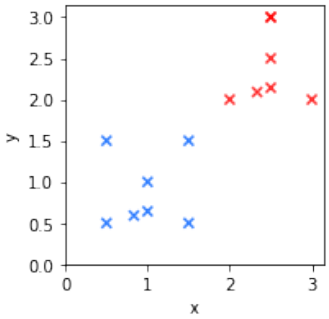


(d) *Using $N = 16$ basis fields, and the $T = 5$ trivial points.*

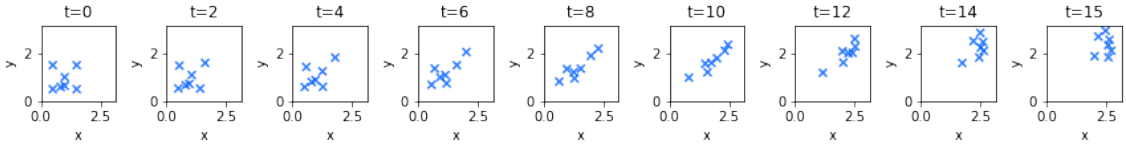


(e) *Using $N = 36$ basis fields, and using a total of 60 sampling points.*

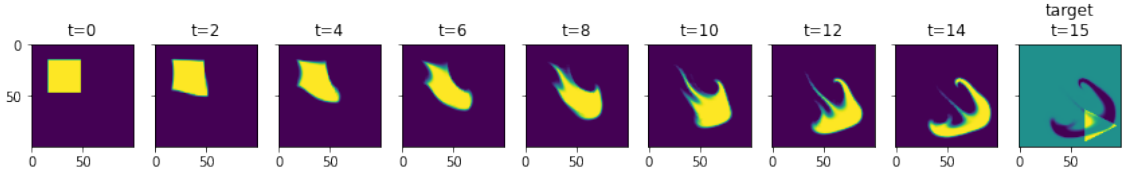
Figure 5.3: *Solving the shape transition problem by optimizing for an initial coefficient vector* **w** *without any further control over the simulation.*



(a) *Initial (blue), and target (red) sample points. ($O = 1$, $U = 1$, $T = 5$)*



(b) *Optimized trajectory of the sample points underlying the optimization.*



(c) *Smoke advection for qualitative comparison, reconstructed on a 100×100 grid.*

Figure 5.4: *Force optimization results with 16 time steps, and using $N = 16$ basis fields.*

5.2.4 Neural Network Training

We generalize the Control Force Estimation (CFE) problem by defining a function $\mathbf{f}(\mathbf{p}_0, \mathbf{p}_t, \mathbf{w}) : \mathbb{R}^{2\cdot 2(O+U+T)+N} \to \mathbb{R}^N$, that gives a force vector $\mathbf{f} \in \mathbb{R}^N$ to be applied at the current time step to move points $\mathbf{p}_0$ to $\mathbf{p}_t$ in the next time step. Its inputs are the (x, y) coordinates of $\mathbf{p}_0$ and $\mathbf{p}_t$, as well as the basis coefficient vector $\mathbf{w}$ at the current time step concatenated after each other, giving $2 \cdot 2(O + U + T) + N$ values, where O, U, and T denote the number of overlapping, unique, and trivial sample points, respectively, as introduced in section 5.2.1.

We approximate the CFE function $\mathbf{f}$ with a Control Force Estimator Neural Network (NN) $\mathbf{f}(\mathbf{p}_0, \mathbf{p}_t, \mathbf{w}, \theta)$.

Each layer is constructed exactly as described in equation (2.7) with ReLU non-linearities, making the resulting concatenation of layers the same as in equation (2.10). Figure 5.5 gives an overview of our NN architecture.

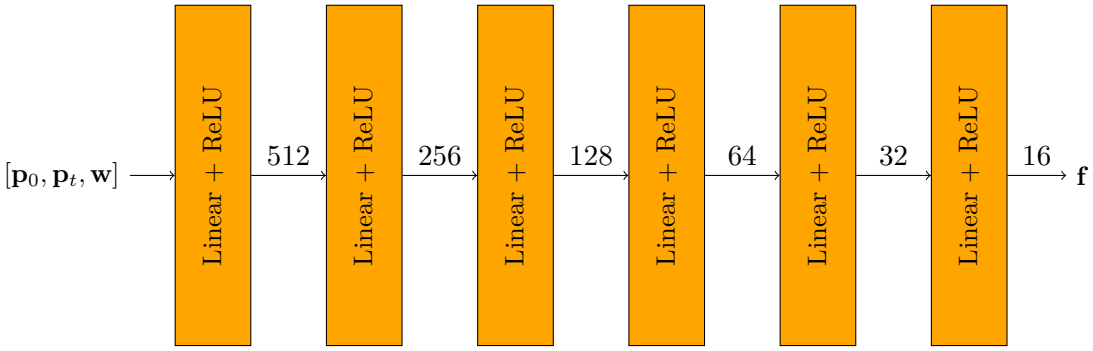


Figure 5.5: *The CFE NN transforms the input vector of size $2 \cdot 2(O + U + T) + N$ into a force vector $\mathbf{f}$ that can be added to the $\mathbf{w}$ coefficients as external force. The output size of each layer matches the input size of the following layer, and a ReLU non-linearity is applied after each layer.*

As the input size to the NN is dependent on the specific problem, the number of trainable parameters also varies, and a new NN has to be trained when using a different number of basis fields, or different number of total sample points. As an example, for $N = 16$ basis fields, and 75 sample points, the NN has 337392 trainable parameters.

Overfitting to a single training sample

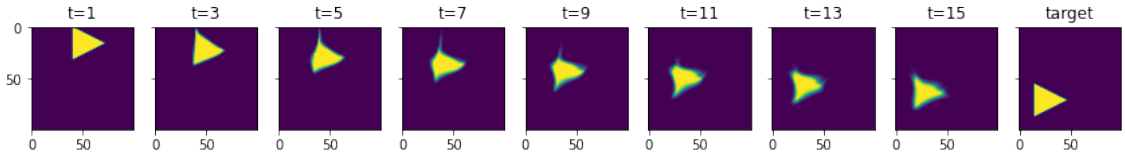
Testing the setup, we overfit the NN to a single training sample. Plotting the results of the time evolution on figure 5.6, we observe that a reduced degrees of freedom can yield comparable, or even better results with the same setup, and training time.

Using an Adam optimizer [15] with learning rate 10^{-3}, the results shown in Figure 5.6 were achieved in 260 epochs. The training took 53.94 seconds.

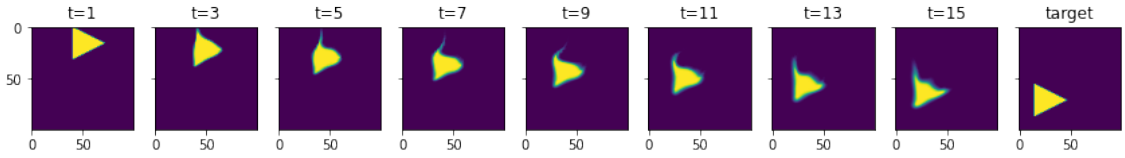
Training

We generate 2000 samples, using 1800 for training, and 200 for validation. Using $N = 16$ basis fields, we train the NN for the CFE problem detailed above. (See Figure 5.7a.)

At the end of the training, we generate further data the NN has not seen during training to further test generalization. (See Figure 5.7b.)



(a) $N = 16$ *basis fields*

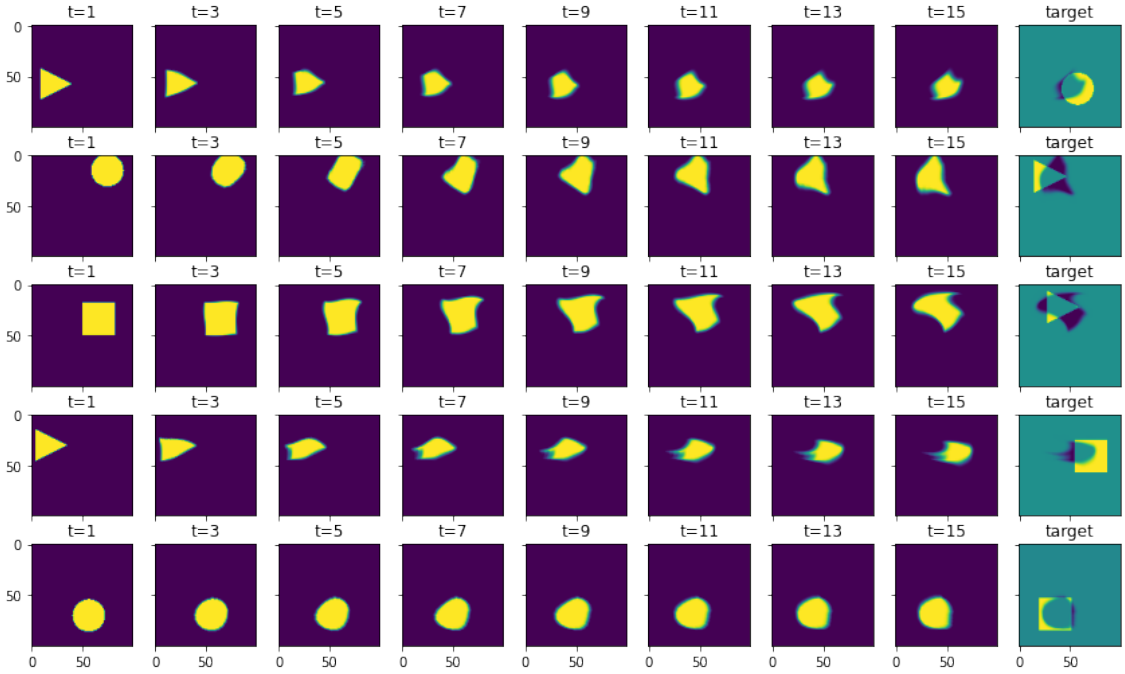


(b) $N = 36$ *basis fields*

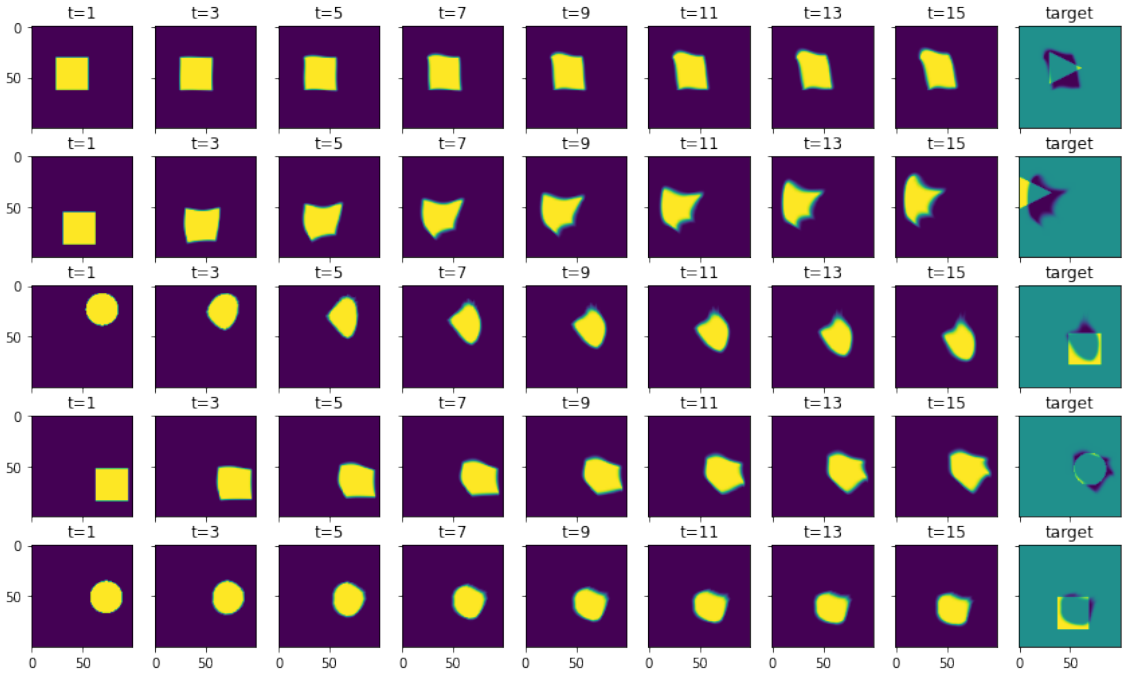
Figure 5.6: *Time evolution of simulating two overfitted CFE NNs to a single shape transition for* 16 *time steps* $t = [0 \dots 15]$. *Using* $O = 30$ *overlapping,* $U = 40$ *unique, and* $T = 5$ *trivial sample points.*

Using an Adam [15] optimizer with learning rate 10^{-3}, the results shown in Figure 5.7 were achieved in 260 epochs. The training took 1201.74 seconds (20 minutes).

As we did not experience any overfitting issues during training, no additional regularization schemes were applied.



(a) *Performance after training on the* ***training data****. (Randomly sampled.)*



(b) *Testing on previously unseen* ***test data****. (Randomly sampled.)*

Figure 5.7: *Randomly sampled time evolution of controlled shape transition tasks. Using* $N = 16$ *basis fields, sampling the smokes on a* 32×32 *grid, approximating them with* $O = 30$ *overlapping,* $U = 40$ *unique and* $T = 5$ *trivial sample points, through* 16 *time steps* $t = [0 \dots 15]$.

Chapter 6

Discussion and Future Work

In this work, after assessing established techniques and current research advancements in the related fields, we introduced a novel approach to control shape transitions by using the gradients of a fluid simulation technique based on the eigenfunctions of the vector Laplacian operator.

Owing to the reduced-order nature of the approach, we achieved speed-ups that usually result in convergence times of minutes even in the case of more advanced setups (and sub-minute, or seconds in the more straight-forward ones).

At multiple points while connecting different areas to form our proposed solution, we resorted to baseline methods. Moving forward, our method could benefit from incorporating a number of state-of-the-art solutions.

Although not a silver bullet, we believe that this approach complements and connects existing techniques in a new and exciting way, offering a fresh perspective on thinking about Neural Networks as universal function approximators. In the last part of our thesis, we consider some of the possible future research directions.

Generalizing to 3D

All of the introduced methods generalize to 3D in a very straightforward way. As shown by Cui et al. [6], the Laplacian Eigenfluids technique is a viable simulation for three dimensional incompressible fluid flow. The exponential increase of simulation variables is a problem not only in forward simulations, but especially when computing gradients for optimizing.

General Improvements to the NN

After introducing a simple training process, and purposefully keeping our architecture simple, a number of improvements from the continuously expanding literature on DL and AI techniques could be incorporated to improve our solution.

Improving the Loss Function

The loss function for the shape transition problem could also be improved in a number of ways. In our solution, we estimate the trajectory as a linear interpolation between

start and end positions. Recalculating the trajectory based on the actual path taken by applying the control forces could potentially lead to more natural transition paths.

Moving further, our solution could also be improved by implementing predictor-scheme as introduced by Holl et al. [10].

Point Sampling

In general, estimating functions by sampling discrete points fits into a vast body of existing literature. The sampling strategies introduced in section 5.2.1 could be expanded upon in a number ways, among which improving on the correspondences between the initial and target shapes is a noteworthy option.

Bibliography

[1] David Baraff and Andrew P. Witkin. Physically based modeling: Principles and practice. 1997, URL https://www.cs.cmu.edu/~baraff/sigcourse/.

[2] R. Bridson. *Fluid simulation for computer graphics, Second Edition*. 2015. DOI: 10.1201/9781315266008.

[3] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: Siggraph 2007 course notesvideo files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 1–81, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781450318235. DOI: 10.1145/1281500.1281681.

[4] Li-Wei Chen, Berkay A. Cakal, Xiangyu Hu, and Nils Thuerey. Numerical investigation of minimum drag profiles in laminar flow using deep learning surrogates. *Journal of Fluid Mechanics*, 919:A34, 2021. DOI: 10.1017/jfm.2021.398.

[5] D.K. Cheng. *Field and Wave Electromagnetics*. Addison-Wesley series in electrical engineering. Addison-Wesley, 1989. ISBN 9780201528206, URL https://books.google.hu/books?id=hjhBAQAAIAAJ.

[6] Qiaodong Cui, Pradeep Sen, and Theodore Kim. Scalable laplacian eigenfluids. *ACM Trans. Graph.*, 37(4), jul 2018. ISSN 0730-0301. DOI: 10.1145/3197517.3201352.

[7] Tyler De Witt, Christian Lessig, and Eugene Fiume. Fluid simulation using laplacian eigenfunctions. *ACM Trans. Graph.*, 31(1), feb 2012. ISSN 0730-0301. DOI: 10.1145/2077341.2077351.

[8] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, dec 1964. ISSN 0001-0782. DOI: 10.1145/355588.365104.

[9] Philip Holl. Learn to throw example, URL https://tum-pbs.github.io/PhiFlow/Learn_to_Throw_Tutorial.html.

[10] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. 2019, URL https://ge.in.tum.de/publications/2020-iclr-holl/.

[11] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *ICLR*, 2020.

[12] D.A. Humphreys, Peter De Vries, F. Felici, Kim Sangjin, G. Jackson, A. Kallenbach, Egemen Kolemen, J. Lister, Didier Moreau, A. Pironti, G. Raupp, O. Sauter, E. Schuster, J. Snipes, W. Treutterer, M.L. Walker, A. Welander, Axel Winter, and L. Zabeo. Novel aspects of plasma control in iter. *Physics of Plasmas*, 22:021806, 02 2015. DOI: 10.1063/1.4907901.

[13] E. B. Wilson J. W. Gibbs. *Vector analysis*. 1901, URL https://archive.org/details/117714283/page/236/mode/2up?view=theater. (page 237).

[14] Aaron Demby Jones, Pradeep Sen, and Theodore Kim. Compressing Fluid Subspaces. In Ladislav Kavan and Chris Wojtan, editors, *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. The Eurographics Association, 2016. ISBN 978-3-03868-009-3. DOI: 10.2312/sca.20161225.

[15] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[16] Beibei Liu, Gemma Mason, Julian Hodgson, Yiying Tong, and Mathieu Desbrun. Model-reduced variational fluid simulation. *ACM Trans. Graph.*, 34(6), nov 2015. ISSN 0730-0301. DOI: 10.1145/2816795.2818130.

[17] Robert W MacCormack. The effect of viscosity in hypervelocity impact cratering. *Journal of spacecraft and rockets*, 40(5):757–763, 2003.

[18] Miles Macklin. Warp: A high-performance python framework for gpu simulation and graphics. https://github.com/nvidia/warp, March 2022. NVIDIA GPU Technology Conference (GTC).

[19] Peter O'Malley, Ryan Babbush, Ian Kivlichan, Jonathan Romero, Jarrod McClean, Rami Barends, Julian Kelly, Pedram Roushan, Andrew Tranter, Nan Ding, Brooks Campbell, Yu Chen, Zijun Chen, Ben Chiaro, Andrew Dunsworth, Austin Fowler, Evan Jeffrey, Anthony Megrant, Josh Mutus, Charles Neil, Chris Quintana, Daniel Sank, Ted White, Jim Wenner, Amit Vainsencher, Peter Coveney, Peter Love, Hartmut Neven, Alán Aspuru-Guzik, and John Martinis. Scalable quantum simulation of molecular energies. *Physical Review X*, 6:031007, 2016, URL https://journals.aps.org/prx/abstract/10.1103/PhysRevX.6.031007.

[20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. DOI: 10.1038/323533a0.

[21] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 121–128, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0201485605. DOI: 10.1145/311535.311548.

[22] Thomas Stocker. *Climate change 2013: the physical science basis: Working Group I contribution to the Fifth assessment report of the Intergovernmental Panel on Climate Change*. Cambridge university press, 2014.

[23] Dominique Thévenin and Gábor Janiga. Optimization and computational fluid dynamics. 2008.

[24] Nils Thuerey, Philipp Holl, Maximilian Mueller, Patrick Schnell, Felix Trost, and Kiwon Um. *Physics-based Deep Learning*. WWW, 2021, URL https://physicsbaseddeeplearning.org.

[25] 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, 2020.

[26] S. Wiewel, B. Kim, V. C. Azevedo, B. Solenthaler, and N. Thuerey. Latent space subdivision: Stable and controllable time predictions for fluid flow. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '20, Goslar, DEU, 2020. Eurographics Association. DOI: 10.1111/cgf.14097.

[27] Steffen Wiewel, Moritz Becher, and Nils Thürey. Latent space physics: Towards learning the temporal evolution of fluid flow. *Computer Graphics Forum*, 38, 2019.

Appendix

A.1 Plotting the First 16 Basis Fields

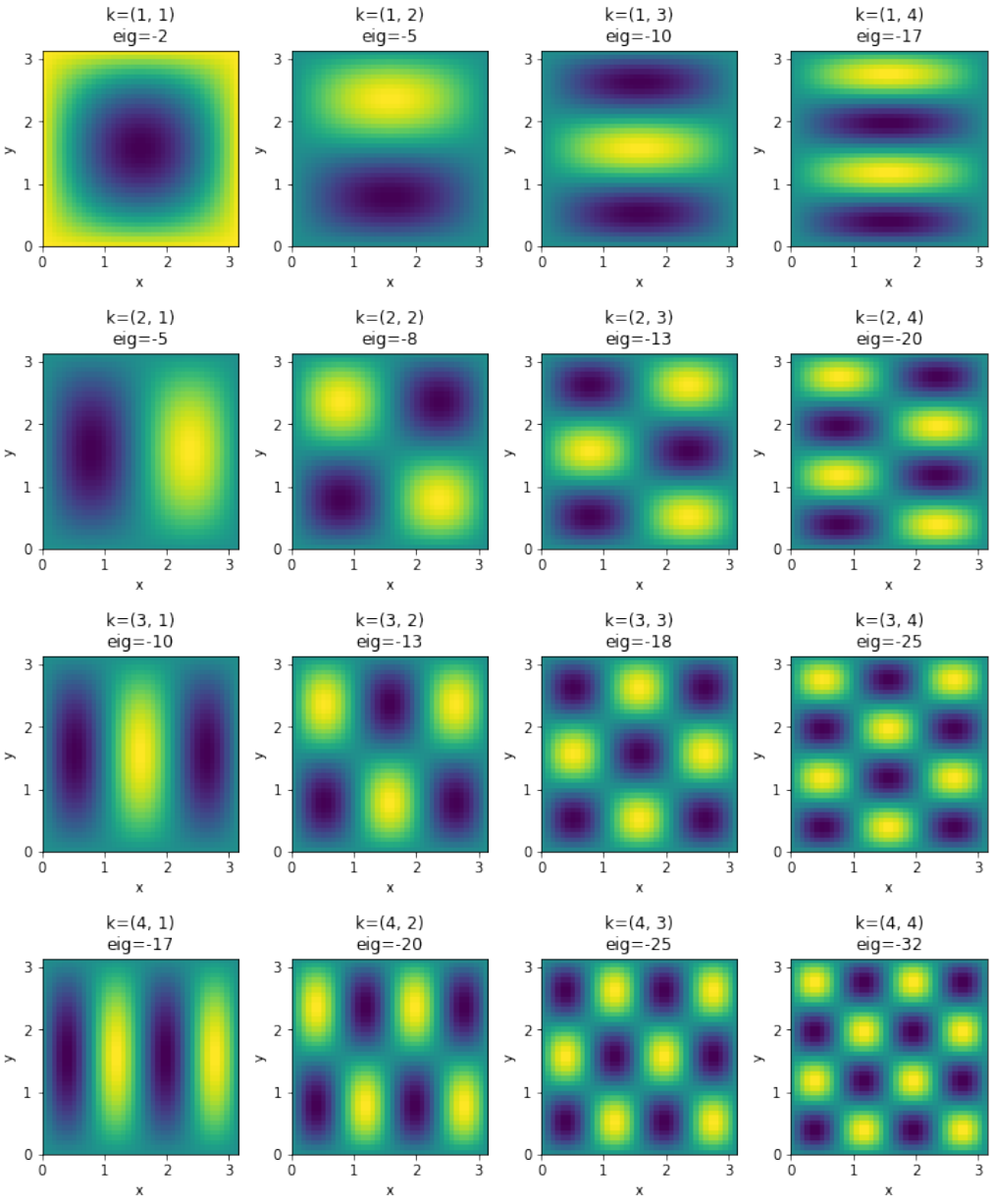


Figure A.1.1: *Visualizing the curl of the first* 16 $\Phi_{\mathbf{k}}$ *basis fields (i.e.* $\phi_{\mathbf{k}}$*), sampled on a* 40×40 *grid in our simulation domain* $D = [0, \pi] \times [0, \pi]$*. Larger magnitudes of eigenvalues correspond to smaller scale vortices.*