Controlling 2D Laplacian Eigenfluids

Barnabás Börcsök*
Supervised by: László Szécsi †

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Műegyetem rkp. 3., H-1111 Budapest, Hungary

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, enabling neural network agents to learn to control the physical process to achieve desired outcomes.

Keywords: Computer Graphics, Modeling and Simulation, Fluid Simulation, Neural Networks

1 Introduction

Data helps us model and understand our world more truthfully. Enabling the processing of the ever-increasing volume of data, and running more precise simulations necessitate continuous engineering efforts.

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. More concretely, many real-world applications require us to optimize for some parameters of a physics-based problem. Although such inverse problems have been around for quite some time in engineering applications, recent work showed remarkable results utilizing physical gradients to solve such problems. Examples include finding the best shape to minimize airfoil drag [2] and finding cloth simulation parameters for yielding desired simulation outcomes [9].

We present a novel method for controlling fluid simulations. We show that implementing a differentiable reduced-order physics simulation yields gradients that allow us to achieve speed-ups in the optimization process characteristic of reduced-order models, resulting in fast convergence times. We investigate different possibilities for control. After directly optimizing for parameters already present in the simulation technique (such as initial velocity and external force), we build up to adding a neural network (NN) for predicting control forces, and optimizing for its internal parameters. When optimizing for advection dynamics, we achieve significant speed-ups by utilizing point-wise samples: we keep the advection dynamics in the reduced-dimensional space, instead of reconstructing the velocity field on an $N \times N$ grid in each time step. Thus, we essentially decouple the optimization from the grid resolution, which can be reconstructed in any desired resolution without increasing the complexity of the optimization.

The source code of this project is available

*bborcsok@iit.bme.hu
†szecsi@iit.bme.hu

This work was supported by OTKA K-124124, and the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory.

at `https://github.com/bobarna/controlling-2d-laplacian-eigenfluids`.

2 Previous Work

2.1 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 advecting marker density in our fluid, as well as a comparative "baseline" simulation, we use Eulerian simulation techniques, mostly as described by Stam [13]. For an overview of fluid simulation techniques in computer graphics, see Bridson [1].

Reduced Order Modeling of Fluids. Dimension reduction-based techniques have been applied to fluid simulation in multiple previous works. Wiewel et al. [17] demonstrated that functions of an evolving physics system can be predicted within the latent space of neural networks (NNs). 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. [16] 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 Wiewel et al. [16] was the subdivision of the learned latent space, allowing interpretability, as well as external control over quantities such as velocity and density.

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. [4] 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. [4], multiple papers proposed improvements to the use of Laplacian eigenfunctions for the simulation of incompressible fluid flow. Liu et al. [10] extended the technique to handle arbitrarily-shaped domains. Jones et al. [7] used Discrete Cosine Transform (DCT) on the eigenfunctions for compression. Cui et al. [3] improved scalability of the technique, and modified the method to handle different types of boundary conditions. Cui et al. [3] refer to the simulation technique as *eigenfluids*, which we also adhere to in the following.

2.2 Differentiable Solvers

Differentiable solvers have shown tremendous success lately for optimization problems, including training neural network models [5, 6, 11]. Holl et al. [5] address grid-based solvers. They put forth Φ_{Flow}, an open-source simulation toolkit built for optimization and machine learning applications, written mostly in Python. After trying out multiple recent frameworks aimed at differentiable simulations [11, 6], we implement all of our experiments using Φ_{Flow} [5].

Physics-based Deep Learning. Despite being a topic of research for a long time [12], 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 [2, 9]. Integrating physical solvers in such methods have been shown to outperform previously used learning approaches [15]. Drawing on a wide breadth of current research, Thuerey et al. [14] give an overview of deep learning methods in the context of physical simulations.

3 Background

In this section, we introduce the techniques, theory and notation underlying our methods for controlling eigenfluids in Section 4.

3.1 Fluid Simulation

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

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

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. $\nabla \cdot \mathbf{u} = 0$. There are multiple established ways to simulate fluids, the most widespread being Eulerian (i.e. grid-based) and Lagrangian (i.e. particle-based) methods. We use established Eulerian methods [1] for advecting and comparison purposes.

The Laplacian Eigenfunction Method. A velocity field $\mathbf{u}(\mathbf{x})$ can be expressed via the linear combination of N global functions:

$$\mathbf{u}(\mathbf{x}) = \sum_{k=0}^{N} w_k \Phi_k(\mathbf{x}), \tag{2}$$

where the elements of $\mathbf{w} = [w_0, \dots, w_N]$ are called *basis coefficients* and Φ_k are *basis functions*. In the following, we use $\mathbf{u} = \mathscr{R}\mathbf{w}$ to notate a velocity field $\mathbf{u}$ reconstructed from $\mathbf{w}$. De Witt et al. [4] propose the use of eigenfunctions of the vector Laplacian operator $\Delta = \nabla^2$. If we further require our basis fields Φ_k to be divergence-free and to satisfy a free-slip boundary condition, then our basis functions are fully characterized by

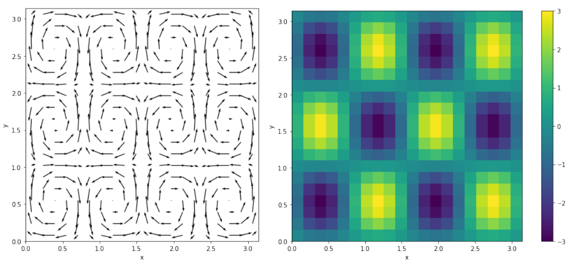
$$\nabla^2 \Phi_k = \lambda_k \Phi_k \tag{3}$$

$$\nabla \cdot \Phi_k = 0 \tag{4}$$

$$\Phi_k \cdot \mathbf{n} = 0 \text{ at } \partial D, \tag{5}$$

where $\mathbf{n}$ is the normal vector at the boundary ∂D of our domain D. On some domains, closed-form expressions exist. Denoting the two scalar components in the x and y directions $\Phi_\mathbf{k} = (\Phi_{\mathbf{k},x}, \Phi_{\mathbf{k},y})$, on the two dimensional $D \in [0,\pi] \times [0,\pi]$ square domain, $\Phi_\mathbf{k}$ can be written as

$$\Phi_{\mathbf{k},x}(x,y) = \eta_\mathbf{k}\big(k_2 \sin(k_1 x)\cos(k_2 y)\big) \tag{6}$$
$$\Phi_{\mathbf{k},y}(x,y) = -\eta_\mathbf{k}\big(-k_1 \cos(k_1 x)\sin(k_2 y)\big),$$



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

Figure 1: Visualizing $\Phi_{(4,3)}$ sampled on a 20×20 grid in our simulation domain $D = [0,\pi] \times [0,\pi]$.

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 $\eta_\mathbf{k} = (-\lambda_\mathbf{k})^{-1}$ is a normalization parameter. As an example, $\Phi_{(4,3)}(x,y)$ is visualized in Figure 1.

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.

For the simulation technique, we further require the vorticity field $\omega = \nabla \times \mathbf{u}$ and a set of vorticity basis functions $\phi = \nabla \times \Phi$. Taking the curl gives us the vorticity basis fields:

$$\phi_\mathbf{k} = \nabla \times \Phi_\mathbf{k} = \begin{pmatrix} 0 \\ 0 \\ \sin(k_1 x)\sin(k_2 y) \end{pmatrix}. \tag{7}$$

As the velocity field $\mathbf{u}$ and vorticity field ω are orthogonal, the vorticity basis functions $\phi_\mathbf{k}$ have

Note: We use the wave length vector $\mathbf{k} = (k_1, k_2)$, as well as a single (non-vector) k for indexing over all of the basis fields – a slight, but very useful abuse of notation. This stems from the fact that a suitable mapping from vector wave length (k_1, k_2) to positive integers is necessary in an implementation.

only a normal component at the boundary and satisfy

$$\nabla^2 \phi_{\mathbf{k}} = \lambda_{\mathbf{k}} \phi_{\mathbf{k}} \tag{8}$$

$$\phi_{\mathbf{k}} \times \mathbf{n} = 0 \text{ at } \partial D. \tag{9}$$

From these properties, De Witt et al. [4] show a velocity-vorticity duality, letting us use the same $\mathbf{w}$ vector to reconstruct $\mathbf{u}$ and ω from Φ_k and ϕ_k, respectively. Furthermore, as both Φ_k and ϕ_k are divergence-free by construction (i.e. $\nabla \cdot \phi_{\mathbf{k}} = 0$), there is no need for a pressure projection in each time step, otherwise often present in fluid simulation techniques.

They also show that on Laplacian eigenfunctions, the inverse operator curl^{-1} takes the simple form $\Phi_k = -\lambda_k^{-1} \cdot \mathrm{curl}(\phi_k)$, making the reconstruction of $\mathbf{u}$ from ω efficient.

Dynamics. The vorticity formulation of the Navier-Stokes Equations (Equation 1) is

$$\dot{\omega} = \mathrm{Adv}(\mathbf{u}, \omega) + \nu \nabla^2 \omega + \nabla \times \mathbf{f}, \tag{10}$$

where $\omega = \nabla \times \mathbf{u}$ and $\mathbf{f}$ denotes external forces. $\mathrm{Adv}(\mathbf{u}, \omega)$ represents the advection term, defined as $\mathrm{Adv}(\mathbf{u}, \omega) := \mathrm{curl}(\omega \times \mathbf{u})$.

De Witt et al. [4] perform projection to a Laplacian eigenfunction basis by substituting the expansions $\omega = \sum_i w_i \phi_i$, $\mathbf{u} = \sum_j w_j \Phi_j$ and $\dot{\omega} = \sum_k \dot{w}_k \phi_k$ into Equation 10. With rearranging the terms through linearity of operators, they get

$$\begin{aligned} \sum_k^N \dot{w} \phi_k = & \sum_i^N \sum_j^N w_i w_j \mathrm{Adv}(\Phi_i, \phi_j) \\ & + \nu \sum_i^N \nabla^2 w_i \phi_i + \nabla \times \mathbf{f}. \end{aligned} \tag{11}$$

As the $\mathrm{Adv}(\Phi_i, \phi_j)$ terms are constant, we precompute them, and the results are stored in the ϕ_k basis, making up the elements of the $\mathbf{C}_k$ matrices for each basis field, each with $N \times N$ values:

$$\mathbf{C}_k[h, i] = \left(\nabla \times (\phi_h \times \Phi_i)\right) \cdot \phi_k. \tag{12}$$

Thus, the evolution of a fluid's velocity as the time derivative of the kth element of the coefficient vector $\mathrm{d}\mathbf{w}/\mathrm{d}t = \dot{\mathbf{w}}$ can be written as

$$\dot{w}_k = \mathbf{w}^T \mathbf{C}_k \mathbf{w} + \nu \lambda_k w_k + f_{w_k}, \tag{13}$$

where the advection term $\nu \lambda_k w_k$ is a point-wise exponential decay (derived via Equation 8) and f_{w_k} represents the external force $\mathbf{f}$ projected to the given basis. Any standard numerical technique can be used to integrate Equation 13 forward in time. However, De Witt et al. [4] describe a preferred technique that, in order to preserve kinetic energy, renormalizes the energy of the fluid simulation after each integration step. They show that due to the orthogonality of the basis functions, the total kinetic energy can be calculated as a sum of squared coefficients.

Algorithm 1 Eigenfluids: stepping $\mathbf{w}$ by Δt

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

3.2 Neural Networks

The goal of neural networks (NNs) 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

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

where θ is a vector of *weights*, influencing the output of the NN. In the case of a fully-connected NN, we can write its i^{th} layer as

$$\mathbf{o}^i = \sigma\left(\mathbf{W}_i \mathbf{o}^{i-1} + b_i\right), \tag{14}$$

where $\mathbf{o}^i$ is the output of the i^{th} layer, and 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 θ.

Deep learning (DL) is about stacking multiple layers after each other, and finding θ parameters such that the outputs $\mathbf{y}$ 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 $L\big(\mathbf{f}(\mathbf{x},\theta),\mathbf{y}^*\big)$. Using a mean square error for our loss function, we can write the optimization problem as:

$$\arg\min_{\theta} \|\mathbf{f}(\mathbf{x},\theta)-\mathbf{y}^*\|_2^2. \tag{15}$$

The chain rule gives us the derivates of composite functions, letting us calculate the gradients of the loss function L with respect to the weights θ (i.e. $\partial L / \partial \theta$). In Section 4.2.4, we optimize, i.e. *train* our NNs with Adam [8], a stochastic gradient descent (SGD) optimizer.

For the purposes of Section 4, before introducing NNs into the optimization loop, 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. As introduced in Equation 15, for finding the optimal θ parameters of a NN, this is exactly what we need: how to tweak θ to reduce the output of a loss function. More generally, we can optimize w.r.t. any parameter of a function in the same manner, such as the initial velocity field of a fluid simulation. In Section 4, we will do exactly this.

4 Controlling Eigenfluids

In the following, we showcase different optimization scenarios of increasing complexity, investigating different aspects of controlling eigenfluids via differentiable physics (DP) gradients. Making use of the explicit closed-form description of a velocity field (Equation 6) to derive gradients used for optimization, we achieve a speed increase characteristic of reduced-order techniques.

4.1 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 $\mathscr{R}\mathbf{w}_t = \mathbf{u}_t$ velocity field will match some precalculated $\mathbf{u}^* : [0,\pi]\times[0,\pi] \to \mathbb{R}^2$ target velocity field:

$$L(\mathbf{w}) = \left\|\mathscr{R}\mathscr{P}^t(\mathbf{w}) - \mathbf{u}^*\right\|_2^2, \tag{16}$$

where $\mathscr{P}^t(\mathbf{w}) = \mathscr{P}\circ\mathscr{P}\cdots\circ\mathscr{P}(\mathbf{w})$ is a composite function: the physical simulation of base coefficients $\mathbf{w}$, t times.

For the optimization, we initialize a $\mathbf{w}_{\text{init}} \in \mathbb{R}^N$ vector with random numbers (from a normal distribution), and run the eigenfluid simulation for t time steps, after which we measure the error as given by loss function 16. Relying on backpropagation [1] to derive the necessary gradients, we use the gradient descent (GD) optimization method to iteratively find a vector $\mathbf{w}_{optim}$, yielding a low scalar loss:

$$\mathbf{w} \leftarrow \mathbf{w} - \lambda\nabla L^T(\mathbf{w}). \tag{17}$$

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 $32\times32\times2$ velocity values of $\mathbf{u}^* = \mathscr{R}\mathbf{w}^*$ 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.

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

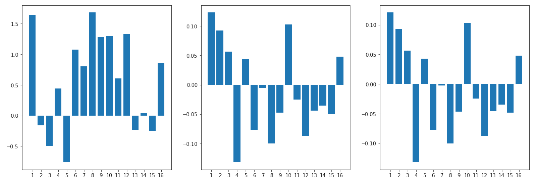
[1]By backpropagation, we refer to the reverse mode automatic differentiation technique of deriving the gradients w.r.t. any given parameter(s) of a composite function by applying the chain rule.

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 $4 \cdot 10^{-4}$.

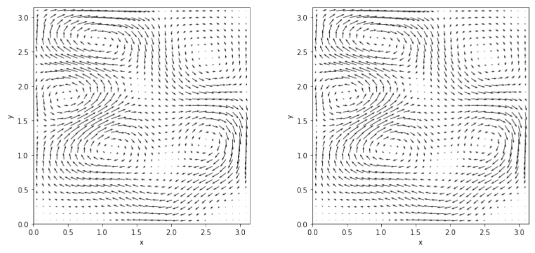
Naturally, this very basic method has its limits. Optimizing for initial coefficients for a target velocity field after 100 steps proved to be a substantially harder problem, as even a relatively small error can accumulate into major deviations over these longer time steps, resulting in much less stable 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 2. 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 the target time step, the optimized $\mathbf{w}_{optim}$ vector already starts to look similar to $\mathbf{w}^*$. Keep in mind that this is not guaranteed at all. 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 minimum of the parameter space.

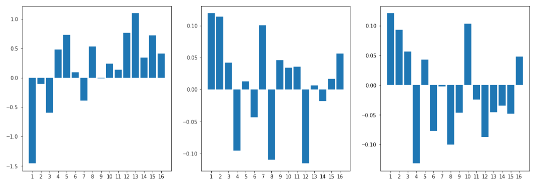
Although there are a number of ways to tweak this setup, we can already verify from these results that the flow of the gradients is working, and is ready to be tested in more advanced scenarios.



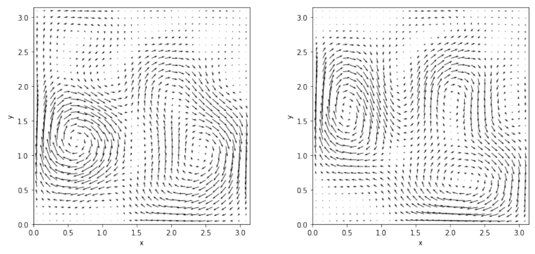
(a) $\mathbf{w}_{init}$, $\mathbf{w}_{optim}$, and $\mathbf{w}^*$, optimizing for velocity field after 16 time steps



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



(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 $\mathscr{P}^{100}(\mathbf{w}_{optim})$

Figure 2: 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.

4.2 Controlling Shape Transitions

Advection of some scalar quantity in a fluid is an abstract problem that describes many real-world phenomena, like ink in water or smoke in the air. We define a density function $\psi(\mathbf{x})$ over a simulation domain D. In a fluid with velocity $\mathbf{u}$, and $\nabla \cdot \mathbf{u} = 0$ holding (i.e. the fluid is incompressible), the advection is governed by the equation

$$\frac{\partial \psi}{\partial t} + \mathbf{u} \cdot \nabla \psi = 0. \qquad (18)$$

We define each shape with

$$\psi(\mathbf{x}) = \begin{cases} 1, & \text{inside the shape} \\ 0, & \text{outside the shape.} \end{cases} \quad (19)$$

In Eulerian fluid simulation methods [1], both $\mathbf{u}$ and ψ are sampled on grids, numerically approximating the evolution of the field quantities. The work of Holl et al. [5] formulated the shape transition problem in an Eulerian representation, with explicitly simulating the shapes as scalar marker densities being advected by the velocity field of the simulated fluid. Instead, playing to the strengths of an eigenfluids simulation, 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 $\mathbf{u}$ (Equations 2 and 6) 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.

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

- In Section 4.2.2, similarly to the problem statement in Section 4.1, we are looking for an initial coefficient vector $\mathbf{w}_0$, such that when simulated for t time steps, the reconstructed velocity field $\mathscr{R}\mathbf{w}_t = \mathbf{u}_t$ advects some initial shape into some target shape.

- In Section 4.2.3, 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, in Section 4.2.4, 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 $\mathbf{f}(\mathbf{p}_t, \mathbf{p}_{t+1}, \mathbf{w}_t, \theta)$, also passing in the current basis coefficient vector $\mathbf{w}_t$, and optimizing for its parameters θ to yield the desired outcome (as introduced in Section 3.2).

In each of these tasks, a velocity field $\mathbf{u} = \mathscr{R}\mathbf{w}$ advects a set of initial points $\mathbf{p}_0 = \left[\mathbf{p}_0^0, \ldots, \mathbf{p}_0^i\right]$ to line up with target positions $\mathbf{p}_t = \left[\mathbf{p}_t^0, \ldots, \mathbf{p}_t^i\right]$ after time t. Using a mean-square error, our loss function becomes

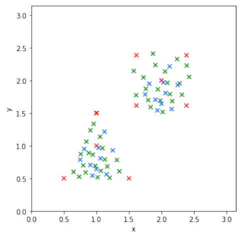
$$L(\mathbf{w}, \mathbf{p}_0, \mathbf{p}_t) = \left\| \mathscr{P}^t(\mathbf{p}_0, \mathbf{w}) - \mathbf{p}_t \right\|_2^2. \quad (20)$$

4.2.1 Sampling

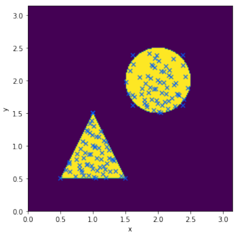
As we neither want to lose too much information about our original function nor do we want to keep track of an unnecessary number of points, the feasibility of our method necessitates an efficient sampling of $\psi(\mathbf{x})$. We use a simple rejection-based sampling technique. We generate random points $\mathbf{p}_{\text{sample}} \in [0,1] \times [0,1]$, 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 $\mathbf{p}_{\text{sample}} \in S_0 \cup S_{\text{t}}$, i.e. we reject points that are not inside both shapes. For the *unique* points, we sample a different set of points for each shape. To generate low-discrepancy, quasi-random 2D coordinates, we use Halton sequences.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 shapes.

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 sets of sample points $\mathbf{p}_0, \mathbf{p}_t \in \mathbb{R}^{2(O+U+T)}$. Figure 3 shows the result of our sampling strategy for a triangle and a circle shape.



(a) $O = 30$ (blue), $U = 30$ (green), and $T = 5$ (red) points.



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

Figure 3: 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.

4.2.2 Optimizing for Initial Velocity

As Equation 20 introduced the problem, our goal is to find an initial velocity field $\mathscr{R}\mathbf{w} = \mathbf{u}$ 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:

$$\arg\min_{\mathbf{w}} \left\| \mathscr{P}^t(\mathbf{p}_0, \mathbf{w}) - \mathbf{p}_t \right\|_2^2. \qquad (21)$$

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

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

Finally, we simply iterate a GD optimizer to find a (good enough) solution for the minimization problem of Equation 21:

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

where L is the same as in Equation 20:

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

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 4, with the details of both experiments described in Table 1.

Table 1: Details of the two optimization scenarios shown in Figure 4.

	Figure 4a	Figure 4b
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

4.2.3 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 t of an eigenfluid simulation, initial positions $\mathbf{p}_0$ will be advected to target positions $\mathbf{p}_t$ after t time steps:

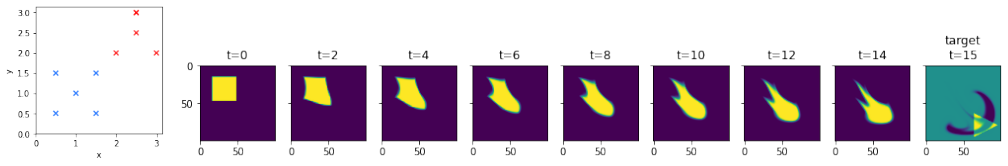
$$\arg\min_{\mathbf{f}} \left\| \mathscr{P}^t(\mathbf{p}_0, \mathbf{w}, \mathbf{f}) - \mathbf{p}_t \right\|_2^2,$$

where $\mathscr{P}^t(\mathbf{p}_0, \mathbf{w}, \mathbf{f}) = \mathscr{P} \circ \cdots \circ \mathscr{P}(\mathbf{p}_0, \mathbf{w}, \mathbf{f})$ 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.

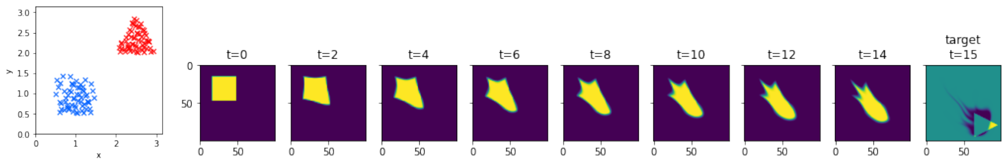
4.2.4 Neural Network Training

We generalize the control force estimation (CFE) problem by defining a function $\mathbf{f}(\mathbf{p}_t, \mathbf{p}_{t+1}, \mathbf{w}_t) : \mathbb{R}^{2 \cdot 2(O+U+T)+N} \rightarrow \mathbb{R}^N$, that given particles at positions $\mathbf{p}_t$ and velocity field $\mathbf{w}_t$ at time t, gives a force that advects the particles to positions $\mathbf{p}_{t+1}$ in the next time step. Its inputs are the (x, y) coordinates of the points, and the N basis coefficients, 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 4.2.1.

We approximate function $\mathbf{f}$ with a CFE NN $\mathbf{f}(\mathbf{p}_t, \mathbf{p}_{t+1}, \mathbf{w}_t, \theta)$. Each layer is constructed as described in Equation 14 with ReLU non-linearities in-between. Figure 6 gives an overview of our NN architecture. As the input size of 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



(a) Using $O = 0$, $U = 0$, $T = 5$ sampling points and $N = 16$ basis fields.



(b) Using $O = 30$, $U = 30$, $T = 0$ sampling points and $N = 36$ basis fields.

Figure 4: In the least complex scenario, we solve the shape transition problem by optimizing for an initial coefficient vector **w** without any further control over the simulation. Time evolution over 16 time steps of two different simulation set-ups are shown, with different number of sample points and basis fields used. Initial (blue) and target (red) sample points are shown.

sample points, the NN has 337 392 trainable parameters. Testing the setup, we overfit the NN to a single training sample. Plotting the results of the time evolution in Figure 7, 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 [8] with learning rate 10^{-3}, the results shown in Figure 7 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. At the end of the training, we generate further data the NN has not seen during training to further test generalization. Using an Adam [8] optimizer with learning rate 10^{-3}, the results shown in Figure 8 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.

5 Results & Discussion

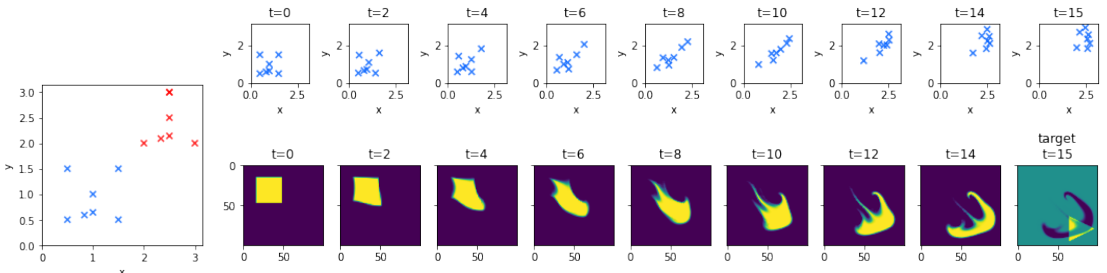
After introducing gradient-based optimization in the context of eigenfluids (Section 4.1), we proposed a novel approach to control shape transitions (Section 4.2) in the reduced-dimensional fluid simulation. Starting with individual optimization problems (Figures 4 and 5), we showed that NNs can not only give comparable results to a set of problems, but they also generalize beyond the examples seen during training (see Figure 8).

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).

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 NNs as universal function approximators.

Generalizing to 3D. All of the introduced methods generalize to 3D in a straightforward way. As shown by Cui et al. [3], the Laplacian eigenfluids technique is a viable option for simulating three-dimensional incompressible fluid flow.

Target Trajectory. We estimate the trajectory as a linear interpolation between start and end positions. Recalculating the trajectory based on the actual path taken after applying the control forces at each time step might lead to more natural transition paths. Alternatively, the effects of implementing a predictor-corrector scheme as introduced by Holl et al. [5] could be investigated.



(a) Sample points.

(b) Optimized trajectory of the sample points underlying the optimization (top). Smoke advection is shown for qualitative comparison, reconstructed on a 100×100 grid (bottom).

Figure 5: Direct force optimization results with 16 time steps, and using $N = 16$ basis fields. We observe that although the sample points (blue) were advected close to their target positions (red), using only 5 sample points was not enough to approximate the underlying higher dimensional advection dynamics.

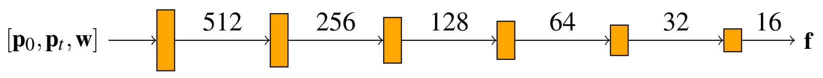


Figure 6: The CFE NN transforms the input vector of size $2 \cdot 2(O+U+T)+N$ into a force vector $\mathbf{f}$ that can be added to the $\mathbf{w}$ coefficients as external force. (The architecture for $N = 16$ fields is shown.) Each layer is linear, with outuput sizes matching each following input size. A ReLU non-linearity is applied after each layer.

References

[1] R. Bridson. *Fluid simulation for computer graphics, Second Edition*. K Peters/CRC Press, 2015. doi: 10.1201/9781315266008. URL `https://doi.org/10.1201/9781315266008`.

[2] 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.

[3] 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. URL `https://doi.org/10.1145/3197517.3201352`.

[4] 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. URL `https://doi.org/10.1145/2077341.2077351`.

[5] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to Control PDEs with Differentiable Physics. In *ICLR*, 2019. URL `https://ge.in.tum.de/publications/2020-iclr-holl/`.

[6] 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.

[7] 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.

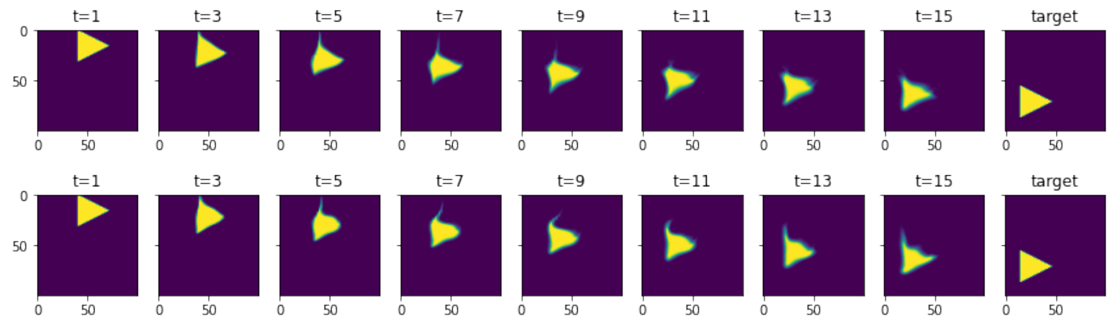


Figure 7: Time evolution of simulating two overfitted CFE NNs to a single shape transition for 16 time steps, using the same 75 sample points, but different number of basis fields (top: $N = 16$, bottom: $N = 36$).

[8] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *ICLR*, 12 2014.

[9] Yifei Li, Tao Du, Kui Wu, Jie Xu, and Wojciech Matusik. DiffCloth: Differentiable Cloth Simulation with Dry Frictional Contact. *ACM Trans. Graph.*, 42(1), oct 2022. ISSN 0730-0301. doi: 10.1145/3527660. URL `https://doi.org/10.1145/3527660`.

[10] 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. URL `https://doi.org/10.1145/2816795.2818130`.

[11] 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).

[12] 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. URL `https://doi.org/10.1038/323533a0`.

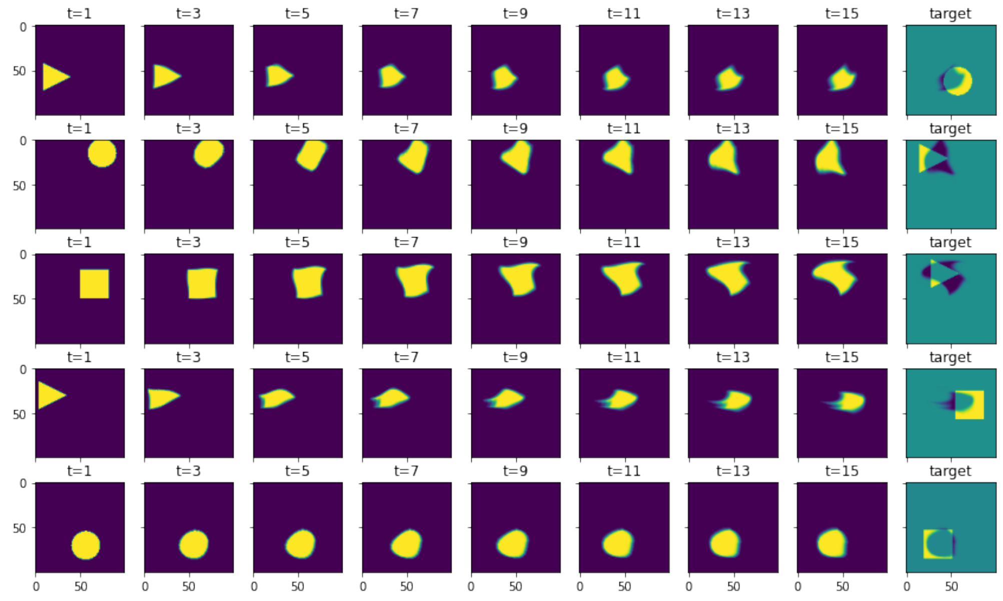
[13] Jos Stam. Stable Fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 121–128, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0201485605. doi: 10.1145/311535.311548. URL `https://doi.org/10.1145/311535.311548`.

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

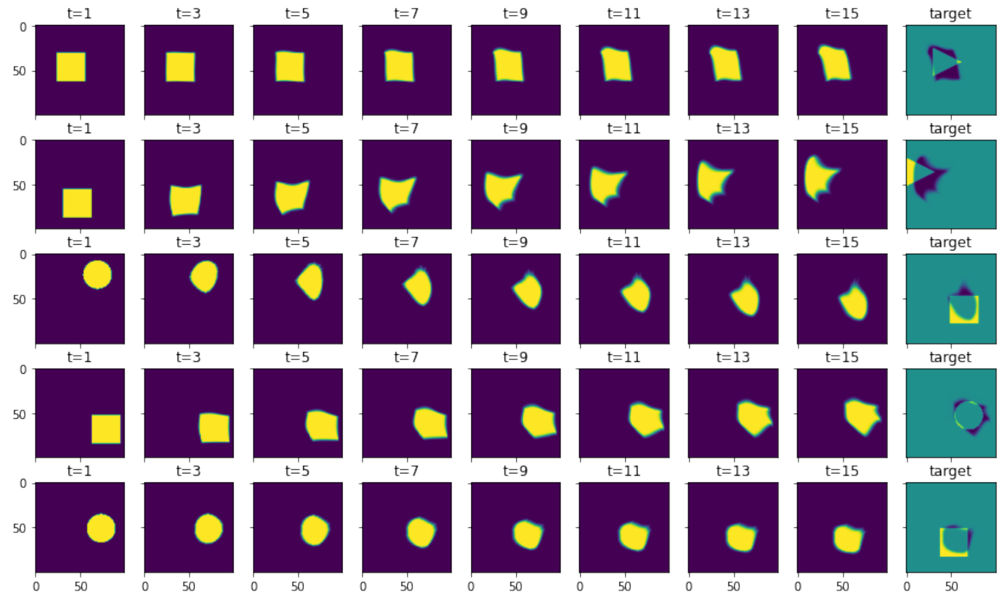
[15] Kiwon Um, Robert Brand, Yun (Raymond) Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. In *Advances in Neural Information Processing Systems*, 2020.

[16] 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. URL `https://doi.org/10.1111/cgf.14097`.

[17] Steffen Wiewel, Moritz Becher, and Nils Thürey. Latent Space Physics: Towards Learning the Temporal Evolution of Fluid Flow. *Computer Graphics Forum*, 38, 2019.



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



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

Figure 8: 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 $t = [0 \dots 15]$.