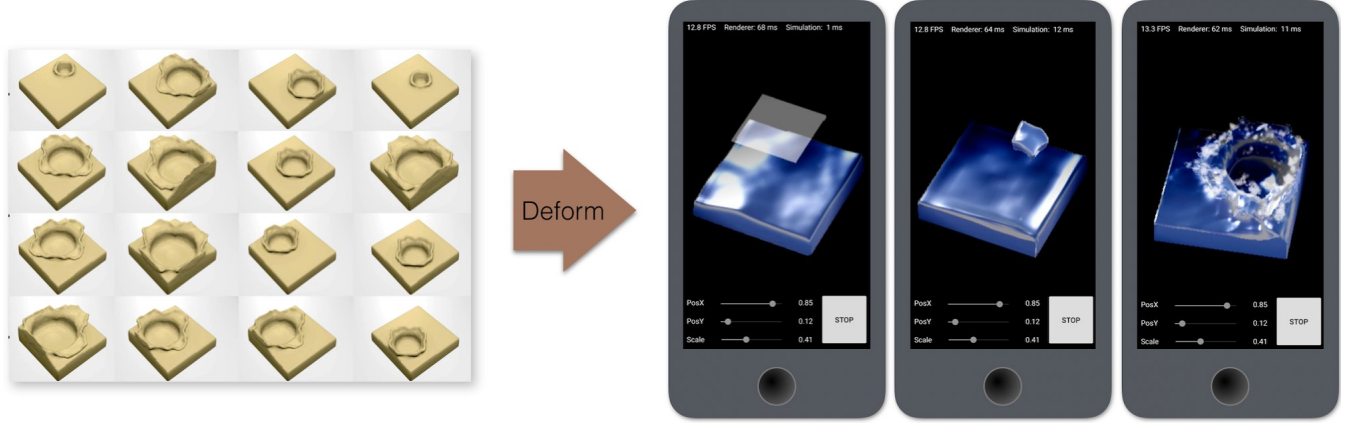# Pre-computed Liquid Spaces with Generative Neural Networks and Optical Flow

Boris Bonev*   Lukas Prantl*   Nils Thuerey*

Technical University of Munich

(* = similar contribution)

Figure 1: *Our method represents a complex space of liquid behavior, such as varying liquid drops shown in the left, with a novel reduced representation based on deformations. In addition to regular space-time deformations obtained with an optical flow solve, we employ a generative neural network to capture the whole space of liquid motions. This allows us to very efficiently generate new versions. As a proof of concept, we demonstrate that it is possible to interactively run this setup on a regular mobile device (captured screens on the right).*

Abstract

Liquids exhibit highly complex, non-linear behavior under changing simulation conditions such as user interactions. We propose a method to map this complex behavior over a parameter range onto a reduced representation based on space-time deformations. In order to represent the complexity of the full space of inputs, we use aligned deformations from optical flow solves, and we leverage the power of generative neural networks to synthesize additional deformations for refinement. We introduce a novel deformation-aware loss function, which enables optimization in the highly non-linear space of multiple deformations. To demonstrate the effectiveness of our approach, we showcase the method with several complex examples in two and four dimensions. Our representation makes it possible to generate implicit surfaces of liquids very efficiently, which allows us to very efficiently display the scene from any angle, and to add secondary effects such as particle systems. We have implemented a mobile application with our full pipeline to demonstrate that real-time interaction is possible with our approach.

Keywords: fluid simulation, convolutional neural networks, optical flow, space-time deformation

1 Introduction

Having the possibility to interact with physical effects allows users to experience and explore their complexity directly. In interactive settings, users can ideally inspect the scene from different angles, and experiment with the physical phenomena using varying modes of interaction. This hands-on experience is difficult to convey with pre-simulated and rendered sequences. While it is possible to perform real-time simulations with enough computing power at hand [Macklin et al. 2014], many practical applications cannot rely on a dedicated GPU at their full disposal. An alternative to these di-

rect simulations are *data-driven methods* that pre-compute the motion and/or the interaction space to extract a reduced representation that can be evaluated in real-time. This direction has been demonstrated for a variety of interesting effects such as detailed clothing [Kim et al. 2013], soft-tissue effects [Xu and Barbič 2016], swirling smoke [Treuille et al. 2006], or for forces in SPH simulations [Ladicky et al. 2015].

Liquids are an especially tough candidate in this area, as the constantly changing boundary conditions at the liquid-gas interface result in a strongly varying and complex space of surface motions and configurations. A notable exception here is the *StateRank* approach [Stanton et al. 2014], which enables interactive liquids by carefully pre-computing a large graph of pre-rendered video clips to synthesize the desired liquid behavior. The algorithm by Ladicky et al. also successfully proposes a pre-computation scheme for liquids, which however targets the accelerations in a Lagrangian approach in order to accelerate a full dynamic simulation.

In the following, we will present a novel approach to realize interactive liquid simulations: we treat a (potentially very large) collection of input simulations as a smoothly deforming set of space-time signed-distance functions (SDF). An initial implicit surface is deformed in space and time to yield the desired behavior as specified by the initial collection of simulations. To calculate and represent the deformations efficiently, we take a two-step approach: First, we span the sides of the original collection with multiple pre-computed deformations. In a second step, we refine this rough surface with a generative neural network. Given the pre-computed deformations, this network is trained to represent the details within inner space of the initial collection.

For data-driven liquids, working with implicit surfaces represents a sweet spot between the raw simulation data, which for liquids often consists of a large number of particles, and a pre-rendered video. The deformed implicit surface can be rendered very efficiently from
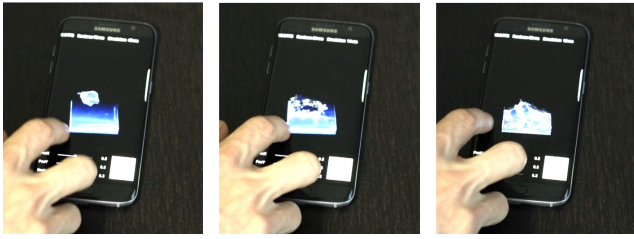
Figure 2: *A user placing drops in our mobile implementation.*

any viewpoint, and can be augmented with secondary effects such as splashes or foam floating on the surface. At the same time, it makes our method general in the sense that we can work with implicit surfaces generated by any simulation approach.

All steps of our pipeline, from applying the pre-computed deformations to evaluating the neural network and rendering the surface, can be performed very efficiently. To demonstrate the very low computational footprint of our approach, we run our final models on a regular mobile device at interactive frame rates. Our demo application can be obtained online in the *Android Play* store under the title *Neural Liquid Drop* [Prantl et al. 2017], and several screenshots of it can be found in Fig. 2. In this app, generating the implicit surface for each frame requires only 30ms on average. The central contributions of our work are a novel *deformation-aware neural network approach* to efficiently represent large collections of space-time surfaces, an *improved deformation alignment* step in order to apply the pre-computed deformations, and a *real-time synthesis framework* for mobile devices. Our contributions make it possible to represent the deformation space for more than 1000 input simulations in a 30MB mobile executable.

2 Related Work

Representing complex physics with reduced models has a long and successful history in computer graphics. Initially, works focused particularly on deformable models, e.g., demonstrating fast simulations with modal dynamics [Pentland and Williams 1989] or using tabulation techniques for deformable scenes [James and Fatahalian 2003] (capturing illumination effects at the same time). Subsequent works targeted more expressive and general constitutive models [Barbič and James 2005] and related effects such as sound generation [James et al. 2006]. More recently, researchers have also demonstrated flexible character animations with reduced models [Kim and James 2012], even including interactions with collisions not captured in the pre-computed basis [Teng et al. 2015] or coupling Lagrangian and Eulerian reduced representations [Teng et al. 2016]. Pre-computed models were shown to be especially useful in constrained and repetitive settings, e.g. for secondary soft-tissue effects [Xu and Barbič 2016]. In the following, we will focus on fluid effects, for which the commonly used physical model is given by the *Navier-Stokes* (NS) equations: $D\mathbf{u}/Dt = -\nabla p + \nu\nabla\cdot\nabla\mathbf{u} + \mathbf{f}_{ext}$, in conjunction with the constraint $\nabla\cdot\mathbf{u} = 0$. Here $\mathbf{u}$ is the flow velocity, p the pressure, and $\mathbf{f}_{ext}$ denotes the external body forces. ν parametrizes viscosity, and is typically set to zero for graphics applications, yielding the incompressible Euler equations. Model reduction techniques have also been applied successfully to single-phase flows, i.e. wind and smoke effects [Treuille et al. 2006]. This approach was generalized to work with modular bases [Wicke et al. 2009], to include rendering effects [Gupta and Narasimhan 2007], and to accurately model the semi-Lagrangian transport commonly employed in graphics solvers [Kim and Delaney 2013]. Other researchers have explored specialized basis constructions [De Witt et al. 2012], and compression techniques [Jones

et al. 2016]. The gradient calculation of our learning approach also shares similarities with the adjoint method for fluid flow [McNamara et al. 2004], where gradients of the advection operator likewise play an important role. There, the optimization works on sequences of three-dimensional, divergence free motions with comparably small motions.

While suitable for single phase flows, the aforementioned techniques become problematic for free-surface flows. In this setting, the commonly used free surface boundary conditions strongly influence the dynamics of the liquid phase and require prohibitively large numbers of basis functions to be captured with sufficient detail. The *StateRank* approach proposed by Stanton et al. [2014] takes a different viewpoint to make pre-computations for liquid simulations possible. It pre-computes a state graph based on user interactions to capture the liquid, and refines the graph at runtime. In contrast to our approach, the data in this state graph is static. While temporal transitions employ blending, all other pre-rendered video sequences are used without modification. On the other hand, our approach uses a single data point, which is then transformed smoothly into a desired space-time motion. As such, we believe our contributions could complement such a state graph to significantly reduce its size.

The non-linear model we employ for our reduced representations is centered around concatenated deformations. A central building block for our approach is the calculation of deformation fields from surface correspondences established by close proximity. Previous works employed non-rigid iterative closest point methods for Lagrangian surfaces [Raveendran et al. 2014], and variational formulations for Eulerian data sets [Thuerey 2017]. We will likewise take an Eulerian viewpoint and employ the latter method. However, we outline two changes when computing and applying Eulerian deformations to increase accuracy and robustness. What sets our method apart from these two previous approaches is that both of them focused on computing single deformations, instead of dealing with optimization problems that target a whole spaces spanned by deformations. These previous approaches always work with two single data points, source and target, while our approach considers the whole collection of target data sets at once. It is worth pointing out that the deformations typically have long range, vary strongly in space and time and that applying them is a non-linear operation. While this makes our representation very small and expressive, the high degree of non-linearity makes them difficult to handle in optimization frameworks. This is one of the central issues we will address in Sec. 4.2.

There are various methods in graphics that have targeted real-time simulations with specialized techniques. Especially for larger open water scenes, reducing the dimensionality with two-dimensional models allows for very fast simulations [Kass and Miller 1990; Wang et al. 2007]. Interesting variations include erosion effects [Stava et al. 2008] or incorporate three-dimensional effects near the surface [Chentanez and Müller 2011]. Particle-based liquid simulations are also highly interesting for fast simulations [Solenthaler and Pajarola 2009; Macklin and Müller 2013] Likewise, many recent techniques focus on novel representations to allow for fast simulations, e.g. to represent highly detailed surfaces with meshes [Chentanez et al. 2015] or to represent detailed cut surfaces [Manteaux et al. 2015]. We employ a typical NS solver from the graphics area [Stam 1999; Enright et al. 2003], using ghost-fluid boundaries [Enright et al. 2003], and the FLuid-Implicit Particle method [Zhu and Bridson 2005]. However, our method is not restricted to inputs from this particular choice of solver. Liquid surfaces from particle based simulators [Ihmsen et al. 2014] or any other type of physical simulator would likewise be suitable inputs, as long as they are available as implicit representation for each time step.

Figure 3: *Numerical inaccuracies over time lead to unpredictable changes, even when the input parameters only change very slightly, as can be seen for the right sheet in each image.*

While the fundamentals of machine learning have been established at the advent of computer technology, the field of machine learning with *neural networks* (NNs) has recently experienced a steep increase in interest due to technological advances [Werbos 1974; Rumelhart et al. 1988]. Impressive results have been achieved in many disciplines of computer vision, such as image classification [Krizhevsky et al. 2012] and object detection and segmentation [Girshick et al. 2014]. In the field of computer graphics, researchers have also employed neural networks for diverse tasks such as robust human correspondences from depth maps [Wei et al. 2015], visual similarity for product design [Bell and Bala 2015], automatic image colorization [Iizuka et al. 2016] and photo retrieval from sketches [Sangkloy et al. 2016]. The general architecture of our networks follows other generative approaches [Goodfellow et al. 2014], with the central difference that we focus on learning deformations, and do not employ an adversarial network architecture.

Closer to our area of application, researchers have also investigated computing image deformations, i.e. solving the optical flow problem, with neural networks. While many works target localized regions and feature descriptors to retrieve depth and correspondences [Bailer et al. 2016], several works have investigated retrieving dense image-space deformations. The so called *FlowNet* architecture has proven to be efficient and accurate [Dosovitskiy et al. 2015] and has been extended to compute large-range motions with the help of multi-scale approaches [Ranjan and Black 2016; Ilg et al. 2016]. These papers require explicit annotations with deformation data during the learning phase. In contrast, our networks learn to deform the data given only a target data set. This is crucial for applications where reference motions are not readily available for all inputs.

Despite the success stories in many areas of computer science, machine learning techniques have rarely been applied in the area of physically-based animations. Specifically for liquid simulations, a recent approach was proposed to efficiently model response forces for particle-based simulations using regression forests [Ladicky et al. 2015]. Only few papers exist up to now employing neural networks for Eulerian simulations: one work aims for learning localized, divergence-free velocity updates [Yang et al. 2016], while the another work learns to remove divergent motions for whole grids using convolutional neural networks [Tompson et al. 2016]. In addition, neural networks have recently been used to learn the similarity of coarse and fine smoke simulations [Chu and Thuerey 2017], and to generate small-scale splash effects [Um et al. 2017]. All of these works employ machine learning to speed up or augment direct simulations, while we aim to represent clearly defined regions of liquid behavior with our approach. We will employ convolutional neural networks in the following, as their generative capabilities were shown to be powerful tools [Goodfellow et al. 2014]. After spanning the sides of a chosen region with pre-computed long range deformations, our networks learn to generate deformations to respect the content of these regions.

3 Problem Description

Given a Navier-Stokes boundary value problem with a liquid-gas interface, we consider the interface over time as a single space-time surface. We start with a set of these space-time surfaces, which we will call Γ, defined over a convex region of our simulation parameters $\boldsymbol{\alpha}$. We assume all parameter dimensions to be normalized, i.e. $\boldsymbol{\alpha} \in [0,1]^N$. In practice, $\boldsymbol{\alpha}$ could be any parameter of the simulation, e.g. initial positions, velocity boundary conditions, or physical parameters such as viscosity. Looking at this set of surfaces as a whole, $\Gamma(\boldsymbol{\alpha})$ represents the complex manifold that we want to capture and represent. We choose implicit functions $\phi(\boldsymbol{\alpha}) \in \mathbb{R}^3 \times \mathbb{R}^+ \to \mathbb{R}$ to represent specific instances of the manifold, such that $\Gamma(\boldsymbol{\alpha}) = \left\{ \mathbf{x} \in \mathbb{R}^3 \times \mathbb{R}^+; \phi(\boldsymbol{\alpha})(\mathbf{x}) = 0 \right\}$. In the following, ϕ and ψ will denote four-dimensional signed distance functions, where the first three dimensions are space, while the fourth dimension represents time. We will abbreviate $\phi(\boldsymbol{\alpha})$ with ϕ_α in the following to simplify notation, and to indicate that this set of surfaces represents a collection of constant reference surfaces in our setting.

Representing the whole set ϕ_α is a challenging task. In theory, the original surfaces of Γ are continuous, and by choosing the implicit representation ϕ_α, we regularize the space of inputs. Due to bifurcations and noise-like artifacts (as seen in Fig. 3) from discretization errors there can, however, be very significant differences even for small changes of $\boldsymbol{\alpha}$. Despite these challenges, our goal is to find a manageable and reduced representation. We use our prior knowledge of the problem, and embed ϕ_α into a space spanned by warping a single input surface with a set of deformation fields. Thus, we generate an approximation of ϕ_α by applying a space-time deformation $F : \mathbb{R}^4 \to \mathbb{R}^4$ to an initial surface $\psi(\mathbf{x})$, where $F(\mathbf{x}, U, \boldsymbol{\alpha})$ again depends on the simulation parameters $\boldsymbol{\alpha}$. In our case F is constructed from a set of auxiliary deformation fields $U : \{\mathbf{u}_1, ...\mathbf{u}_N\}$, where $\mathbf{u}_i : \mathbb{R}^4 \to \mathbb{R}^4$. Note that even the simplest choice for F with a single deformation, i.e. $F = \mathbf{x} - \alpha_1 \mathbf{u}_1$, is already non-linear. For now we will assume that all functions are continuous, and sufficiently differentiable. We will later on discretize all functions with Cartesian grids and employ linear interpolation, although other Eulerian representations and higher order basis functions could likewise be used.

We have two goals when calculating the reduced representation: the first goal is staying true to the original simulations by minimizing $\int \|\psi(F(\mathbf{x}, U, \boldsymbol{\alpha})) - \phi_\alpha(\mathbf{x})\|_2^2 \, d\alpha$, where we used the L^2 norm $\|.\|_2$ to measure the distance between the two implicit functions. Although we will present our algorithm with the use of said norm, the concepts described here generalize well and can be adapted to other metrics. As a secondary goal, we would like to minimize the function above with a small number of deformation fields $|U|$. The size of this set[1] not only directly represents the amount of data we have to store, but also the amount of computations we have to perform to generate an output. We will focus on single sets of N deformations in the following, but naturally multiple sets for adjacent parameter regions could be combined to cover even larger spaces of behavior.

For our method, we span the outer sides of the $\boldsymbol{\alpha}$ simplex with pre-computed end-point deformations, and then refine the insides using a generative neural network with a deformation-aware loss function. As the end-point deformations only consider source and target, they do not necessarily represent the space between the two extremes well. This region is where our generative neural network will be active to refine the surfaces produced by the end-point deformations. In the following, we will use an optical flow solver to

---

[1]Note that the set of deformations can not be considered a basis, as the application of a single, arbitrary deformation is already a non-linear process.

compute the end-point deformations $\mathbf{u}_i$ [Thuerey 2017]. However, our approach is not limited to inputs from optical flow. Any algorithm that computes deformations between two implicit surfaces could be used for this pre-processing stage [Solomon et al. 2015]. As we will point out below, we require these deformations to be smooth. This is usually directly enforced with corresponding regularization when computing the end-point deformations.

In line with previous work [Raveendran et al. 2014; Thuerey 2017] we concatenate spatial surfaces as slices of each 4D volume, and then generate a 4D SDF from this initial surface. Without loss of generality, we assume similar resolutions in space in time in the following. Otherwise, this straight forward concatenation could lead to a space vs. time distance bias in the volumes, which however could be corrected for with a suitable scaled distance metric, if necessary.

3.1 Deformation Alignment

As outlined above, ϕ_α will denote the reference signed distance functions (SDFs) of our input parameter space, while ψ will denote instances of a single input surface deformed by our algorithm. We will denote the single initial surface with ψ_0 in the following. Here, we typically use the zero point of our parameter space, i.e., $\psi_0 = \phi_{\alpha_0}$, with $\alpha_0 = \mathbf{0}$. Hence, we aim for deforming ψ_0 such that it matches all instances of ϕ_α as closely as possible. Before we formalize this in terms of a loss function in the next section, we will explain our approach for applying the deformation.
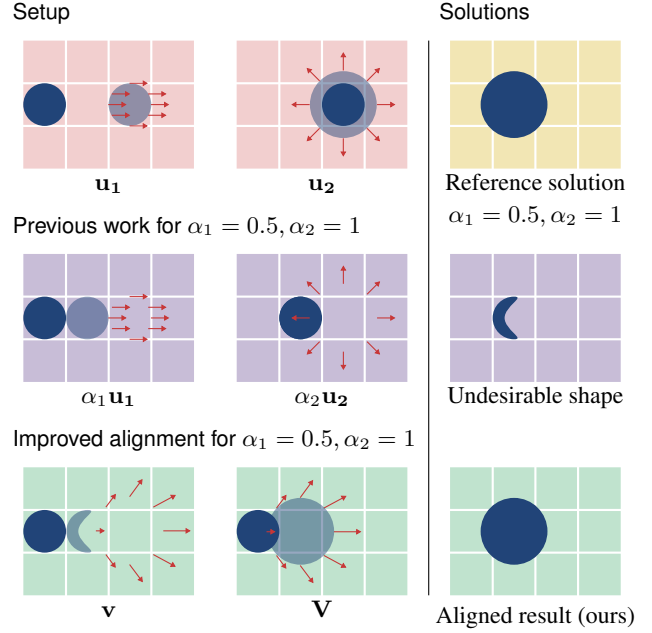
For the end-point deformations, it is our goal to only use a single deformation for each dimension of the parameter space $\boldsymbol{\alpha}$. Thus $\mathbf{u}_1$ will correspond to α_1, and we can apply $\alpha_1\mathbf{u}_1$ to compute a deformation for an intermediate point along this dimension. E.g., the deformed initial surface for a chosen point $\tilde{\alpha}$ on the first dimension α_1 can be computed with $\psi_0(\mathbf{x} - \tilde{\alpha}\mathbf{u}_1)$.

Given the sequence end-point deformations $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N\}$ and a point in parameter space $\{\alpha_1, \ldots, \alpha_N\}$ a straight-forward approach is to apply each deformation sequentially:

$$\psi^1(\mathbf{x}, \boldsymbol{\alpha}) = \psi_0(\mathbf{x} - \alpha_1\mathbf{u}_1)$$
$$\psi^2(\mathbf{x}, \boldsymbol{\alpha}) = \psi^1(\mathbf{x} - \alpha_2\mathbf{u}_2)$$
$$\vdots$$
$$\psi^N(\mathbf{x}, \boldsymbol{\alpha}) = \psi^{N-1}(\mathbf{x} - \alpha_N\mathbf{u}_N). \qquad (1)$$

However, there are two disadvantages to this approach. The main problem of this approach is that the deformations $\mathbf{u}_i$ are only meaningful if applied with $\alpha_i = 1$ (see Fig. 4). This means, that if a previous deformation wasn't applied fully with a weight of 1, each subsequent deformation will lead to an accumulation of deformation errors. The second disadvantage of this simple method is the fact that many advection steps have to be applied in order to arrive at the final deformed SDF. This also affects performance as each advection step introduces additional computations. We present an alternative approach which aligns the deformation fields to the final position of the deformation sequence in order to combine them in a meaningful manner. To do this, we introduce the intermediate deformation fields:

$$\mathbf{u}_N^*(\mathbf{x}) = \mathbf{u}_N(\mathbf{x}),$$
$$\mathbf{u}_{N-1}^*(\mathbf{x}) = \mathbf{u}_{N-1}(\mathbf{x} - \mathbf{u}_N^*(\mathbf{x})),$$
$$\mathbf{u}_{N-2}^*(\mathbf{x}) = \mathbf{u}_{N-2}(\mathbf{x} - \mathbf{u}_N^*(\mathbf{x}) - \mathbf{u}_{N-1}^*(\mathbf{x})),$$
$$\vdots$$
$$\mathbf{u}_1^*(\mathbf{x}) = \mathbf{u}_1(\mathbf{x} - \mathbf{u}_N^*(\mathbf{x}) - \mathbf{u}_{N-1}^*(\mathbf{x}) \ldots - \mathbf{u}_2^*(\mathbf{x})). \qquad (2)$$

Setup · Solutions

$\mathbf{u}_1$ · $\mathbf{u}_2$ · Reference solution

Previous work for $\alpha_1 = 0.5, \alpha_2 = 1$ · $\alpha_1 = 0.5, \alpha_2 = 1$

$\alpha_1\mathbf{u}_1$ · $\alpha_2\mathbf{u}_2$ · Undesirable shape

Improved alignment for $\alpha_1 = 0.5, \alpha_2 = 1$

$\mathbf{v}$ · $\mathbf{V}$ · Aligned result (ours)

Figure 4: *Improved alignment: In row 1) the first two figures in red show two end-point deformations (red arrows). The first one ($\mathbf{u}_1$) moves a drop to the right, while $\mathbf{u}_2$ changes its size once it has reached its final position. Images with deformations show the source in dark blue, and the deformed surface in light blue. In this example, the two deformations should be combined such that the horizontal position and the drop size can be independently controlled by changing α_1 and α_2. E.g., on the top right, a correct solution for $\alpha_1 = 0.5, \alpha_2 = 1$ is shown. Row 2) shows how these deformations are applied in previous work: The second deformation acts on wrong parts of the surface, as the drop has not reached its left-most position for $\alpha_1 = 0.5$. The result is a partially deformed drop, shown in the middle row on the right. Note the deformed surface in the middle row center is on top of the source surface, and thus not visible; however, it is identical to the final result on the right. Row 3) shows our approach: In the bottom left we see the deformation field $\mathbf{v}$ from previous work. It is also the starting point of our improved alignment, but never applied directly. Our method then corrects the deformation by transforming it into $\mathbf{V}$, bottom center, which acts on the correct spatial locations. In this example, it means that the expanding velocities from $\mathbf{u}_2$ are shifted left to correctly expand the drop based on its initial position. Our method successfully computes the intended result, as shown in the bottom right image.*

In this way, we align all deformation fields to the final destination of the deformation sequence. The Eulerian representation we are using means that advection steps look backward to gather data, which in our context means that we start from the last deformation $\mathbf{u}_N$ to align previous deformations. Using the aligned deformation fields $\mathbf{u}^*$ we can include $\boldsymbol{\alpha}$ and assemble the weighted intermediate fields

$$\mathbf{v}(\mathbf{x}, \boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i \mathbf{u}_i^*(\mathbf{x}) \qquad (3)$$

and

$$\tilde{\mathbf{v}}(\mathbf{x}, \boldsymbol{\alpha}) = -\sum_{i=1}^{N} (1 - \alpha_i)\mathbf{u}_i^*(\mathbf{x}). \qquad (4)$$

The first deformation field $\mathbf{v}(\mathbf{x})$ represents the weighted sum of all aligned deformations, weighted with the correct amount of defor-

mation specified by the deformation parameters α_i. The second deformation $\tilde{\mathbf{v}}$, weighted by the inverse weights, is a correction term. Intuitively, this term represents the offset of the deformation field $\mathbf{v}$ from its destination. Therefore, we correct the position of $\mathbf{v}$ by this offset with the help of an additional forward-advection step calculated as:

$$\mathbf{V}(\mathbf{x} + \tilde{\mathbf{v}}(\mathbf{x}, \boldsymbol{\alpha}), \boldsymbol{\alpha}) = \mathbf{v}(\mathbf{x}, \boldsymbol{\alpha}), \qquad (5)$$

This gives us the final deformation field $\mathbf{V}(\mathbf{x}, \boldsymbol{\alpha})$. It is important to note that the deformation $\mathbf{v}$ for a position $\mathbf{x}$ is assembled at an initially unknown location $\mathbf{x}'$. It is then transported to $\mathbf{x}$ with the help of $\tilde{\mathbf{v}}$. This is illustrated in the following figure:
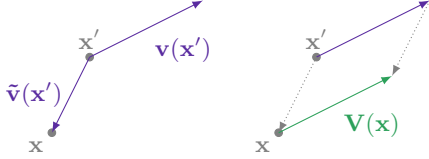
Thus, this step is not a regular advection step, as the deformation is being 'pushed' from $\mathbf{x} + \tilde{\mathbf{v}}(\mathbf{x}, \beta)$ to $\mathbf{x}$. In order to solve this advection equation we use an inverse semi-Lagrangian step. This step pushes values forward with linear interpolation. As multiple values can end up in a single location, we normalize their contribution. Afterwards, we perform several iterations of a flood fill step to make sure all cells in the target deformation grid receive a contribution (we simply extend and average deformation values from all initialized cells into uninitialized regions). The deformed SDF is then calculated by a regular advection equation applying the final, aligned deformation with

$$\psi(\mathbf{x}, \boldsymbol{\alpha}) = \psi_0(\mathbf{x} - \mathbf{V}(\mathbf{x}, \boldsymbol{\alpha})). \qquad (6)$$

Based on our correction step from Eq. (5) this method now respects the case when deformations are not applied fully. As each deformation $\mathbf{u}^*(\mathbf{x})$ is already aligned and does not depend on $\boldsymbol{\alpha}$, we can pre-compute them. To retrieve the final result it is now sufficient to sum up all deformations in $\mathbf{v}$ and $\tilde{\mathbf{v}}$, then apply one forward-advection step to compute $\mathbf{V}$, and finally deform the SDF by applying semi-Lagrange advection. While our method is identical with alignment from previous work [Thuerey 2017] for $\alpha_i = 1$, its importance for practical deformations with weights $\alpha_i \neq 1$ is illustrated in Fig. 4.

This algorithm for aligning deformations will be our starting point for generating initial deformed surfaces with end-point deformations. These deformed surfaces will then be adjusted and refined by our neural networks that we will explain in the following section.

4 Learning Deformations

In this section we will outline a deformation-aware machine learning approach to improve the approximation of the initial surface set ϕ_α. When adjusting the composition of deformations the algorithm explained so far will yield a smooth transition, but it will typically not capture the complex, non-linear behavior that occurs when fluid simulation parameters are changed. This is not surprising, due to the fact that the end-point deformations are optimized to yield the best possible final result for each $\alpha_i = 1$. We will show that our network can learn to adjust these deformation parameters α_i, and additionally that it can learn to generate new deformations to improve the quality of the generated liquid surfaces.

To train our machine learning model, we propose the following objective function, which measures the similiarity of a known reference surface ϕ_α and the corresponding, approximated result

$\psi(\mathbf{x}_i, \alpha)$. To this end, we introduce the numerical equivalent of the L_2 loss

$$L = \frac{1}{2} \sum_i \left(\psi(\mathbf{x}_i, \alpha) - \phi_\alpha(\mathbf{x}_i) \right)^2, \qquad (7)$$

which approximates the analytical loss $\|\psi(\mathbf{x}, \alpha) - \phi_\alpha(\mathbf{x})\|_2^2$. Here the integration over the domain was replaced by the summation over all cells with cell-centers at $\mathbf{x}_i$. Regardless of the approach we are using, we strive to match the surface of the advected solution $\psi(\mathbf{x}, \alpha) = 0$ to the surface of the reference solution $\phi_\alpha(\mathbf{x}) = 0$ as well as possible. This is reflected by a vanishing loss in Eq. (7).

While this loss function looks trivial at first, note that $\psi(\mathbf{x}_i, \alpha)$ from Eq. (6) encapsulates a series of highly non-linear deformation steps. In the following, a central challenge will be to compute reliable gradients in spite of these accumulated non-linearities.

4.1 Learning Optimal Deformation Parameters

We will first focus on the intermediate parameters $\boldsymbol{\alpha}$. We replace these simulation parameters in Eq. (6) with new parameters $\boldsymbol{\beta}(\boldsymbol{\alpha}) = (\beta_1(\boldsymbol{\alpha}), \dots, \beta_N(\boldsymbol{\alpha}))$ so that we can vary the extent to which a specific deformation is applied. We assume that there is an optimal function which gives us the best deformation parameters with respect to Eq. (7) for any given simulation parameter set $\boldsymbol{\alpha}$. We approximate this function with a feed-forward neural network. Thus, given a set of $\boldsymbol{\alpha}$ values an input, the network should learn to output corresponding weights $\boldsymbol{\beta}$ for the deformations, such that the initial surface matches the desired simulation state at α when the deformations are applied. This application of the weighted deformations includes our alignment step from Sec. 3.1.

The network is characterized by the number of layers in the neural network, by the activation function $h(x)$, the number of nodes in each each layer n_L and their respective weights θ_{ij}^l. We can then write the activation a_i^l of the i-th node in the l-th layer as a function of all the activations in the previous layer:

$$a_i^l = h \left(\sum_{j=1}^{n_{l-1}} \theta_{ij}^l \, a_j^{l-1} \right). \qquad (8)$$

We use the common simplified notation to include the bias in θ^l with a constant input for each node. Through composition, we can construct the parameter function $\beta(\alpha, \theta)$

$$\beta_i(\alpha, \theta) = h \left(\sum_{j=1}^{n_{L-1}} \theta_{ij}^L \, h \left(\sum_{k=1}^{n_{L-2}} \theta_{jk}^{L-1} \cdots \right) \right), \qquad (9)$$

which is a function of the input parameters α and all weights $\theta = (\theta_{11}^1, \dots, \theta_{n_l n_{l-1}}^L)$. With this function $\beta(\alpha, \theta)$, we replace all instances of α_i in Eq. (6) with $\beta_i(\alpha, \theta)$ to compute a differently weighted, and aligned final deformation field $\mathbf{V}$.

To train the neural network, we need to specify gradients of Eq. (7) with respect to the network weights θ_i. With the chain rule we obtain $\frac{\mathrm{d}}{\mathrm{d}\theta_{ij}^l} L = \frac{\mathrm{d}\beta}{\mathrm{d}\theta_{ij}^l} \frac{\mathrm{d}L}{\mathrm{d}\beta}$. Since the derivative of the network output β_i with respect to a specific network weight θ_{ij}^l is easily calculated with backpropagation [Bishop 2006], it is sufficient for us to specify the second term. The gradient of Eq. (7) with respect to the deformation parameter β_i is given by

$$\frac{\mathrm{d}}{\mathrm{d}\beta_i} L = \sum_j \frac{\mathrm{d}}{\mathrm{d}\beta_i} \psi(\mathbf{x}_j, \boldsymbol{\beta}) \left[\psi(\mathbf{x}_j, \boldsymbol{\beta}) - \phi_\alpha(\mathbf{x}_j) \right], \qquad (10)$$

where we have inserted Eq. (6). While the second term in the sum is easily computed, we need to calculate the first term by differentiating Eq. (6) with respect to β_i, which yields

$$\frac{\mathrm{d}}{\mathrm{d}\beta_i}\psi(\mathbf{x},\boldsymbol{\beta}) = -\frac{\mathrm{d}}{\mathrm{d}\beta_i}\mathbf{V}(\mathbf{x},\boldsymbol{\beta}) \cdot \nabla\psi_0(\mathbf{x} - \mathbf{V}(\mathbf{x},\boldsymbol{\beta})). \tag{11}$$

It is clear, that we need to calculate the change $\frac{\mathrm{d}}{\mathrm{d}\beta_i}\mathbf{V}(\mathbf{x},\beta)$ in the deformation field, when a small change is applied to the deformation parameter β_i. This is non-trivial as it is not possible to derive a closed form $\mathbf{V}(\mathbf{x},\beta)$ from Eq. (5). It is important to note that even for the case of small corrections $\tilde{\mathbf{v}}(\mathbf{x},\beta)$, Eq. (5) cannot be handled as another backward-advection step $\mathbf{V}(\mathbf{x},\boldsymbol{\alpha}) = \mathbf{v}(\mathbf{x} - \tilde{\mathbf{v}}(\mathbf{x},\boldsymbol{\alpha}),\boldsymbol{\alpha})$. While it might be tempting to assume that differentiating this advection equation will produce reasonable outcomes, it can lead to noticeable errors in the gradient. These in turn quickly lead to diverging results in the learning process, due to the non-linearity of the problem.

The correct way of deriving the change in $\mathbf{V}(\mathbf{x},\beta)$ is by taking the total derivative of $\mathbf{v}(\mathbf{x},\beta) = \mathbf{V}(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta)$ with respect to β_i:

$$\frac{\mathrm{d}}{\mathrm{d}\beta_i}\mathbf{v}(\mathbf{x},\beta)$$
$$= \frac{\partial}{\partial\beta_i}\mathbf{V}(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta) + \mathbf{J}_V(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta)\,\frac{\partial}{\partial\beta_i}\tilde{\mathbf{v}}(\mathbf{x},\beta), \tag{12}$$

where, $\mathbf{J}_V(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta)$ denotes the Jacobian of $\mathbf{V}$ with respect to $\mathbf{x}$, evaluated at $\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta)$. Rearranging Eq. (12) and inserting $\mathbf{v}$ and $\tilde{\mathbf{v}}$ yields

$$\frac{\partial}{\partial\beta_i}\mathbf{V}(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta) \tag{13}$$
$$= \frac{\mathrm{d}}{\mathrm{d}\beta_i}\mathbf{v}(\mathbf{x},\beta) - \mathbf{J}_V(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta)\,\frac{\partial}{\partial\beta_i}\tilde{\mathbf{v}}(\mathbf{x},\beta)$$
$$= \frac{\mathrm{d}}{\mathrm{d}\beta_i}\sum_{i=1}^{N}\beta_i\mathbf{u}_i^*(\mathbf{x}) + \mathbf{J}_V(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta)\,\frac{\partial}{\partial\beta_i}\sum_{i=1}^{N}(1-\beta_i)\,\mathbf{u}_i^*(\mathbf{x})$$
$$= [\mathbf{1} - \mathbf{J}_V(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta)]\,\mathbf{u}_i^*(\mathbf{x}). \tag{14}$$

We note that the Jacobian in the equation above has small entries due to the smooth nature of the deformations $\mathbf{V}$. Thus, compared to the unit matrix it is small in magnitude. (Note that this is not yet visible in Eq. (12).) We have verified in experiments that $\mathbf{J}_V$ does not improve the gradient significantly, and we thus set this Jacobian to zero, arriving at

$$\frac{\partial}{\partial\beta_i}\mathbf{V}(\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta),\beta) \approx \mathbf{u}_i^*(\mathbf{x}), \tag{15}$$

where the $\mathbf{u}^*$ are the deformation fields aligned for the target configuration from Eq. (2). We use Eq. (15) to estimate the change in the final deformation fields for changes of the i-th deformation parameter. We see that this equation has the same structure as Eq. (5). On the left-hand side, we have $\frac{\partial}{\partial\beta_i}\mathbf{V}$, evaluated at $\mathbf{x}+\tilde{\mathbf{v}}(\mathbf{x},\beta)$, whereas $\mathbf{u}_i^*$ on the right-hand side is evaluated at $\mathbf{x}$. To calculate $\frac{\mathrm{d}}{\mathrm{d}\beta_i}\mathbf{V}(\mathbf{x},\beta)$ then, we can use the same forward-advection algorithm, which is applied to the correction in Eq. (5).

With this, we have all the necessary tools to train the neural network, as we can calculate all the required terms for Eq. (10). Therefore, we can backpropagate these gradients into the neural network. In this way we can now optimize the deformation parameters to improve the results of the advection for any intermediate values of α_i. We will demonstrate the effects of this step in Sec. 7.
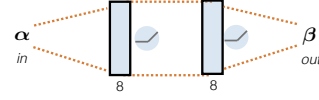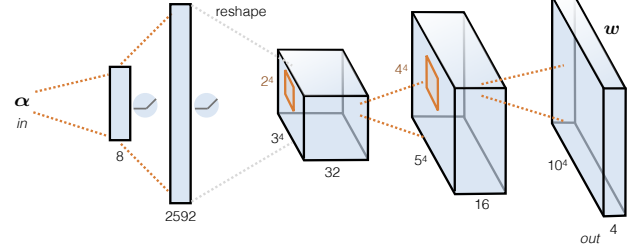
Figure 5: *Schematic of our two neural networks. The parameter network itself is very simple, its cost functions allows it to learn how to optimally apply multiple long-range, non-linear deformation fields. The generative network on the bottom instead learns to generate whole deformation fields to refine the final surface.*

4.2 Generating Deformations with Neural Networks

Our efforts so far have been centered around producing a good approximation of ϕ_α, with the given end-point deformations $\{\mathbf{u}_0,\ldots,\mathbf{u}_n\}$. The performance of this method is therefore inherently constrained by the amount of variation we can produce with the deformation inputs. Our approach so far only optimizes the weights β_i of the individual deformations $\mathbf{u}_i$ in order to produce $\mathbf{V}(\mathbf{x},\beta(\alpha))$.

To allow for more variation, we propose to generate an additional space-time deformation field $\mathbf{w}(\boldsymbol{\alpha})$, that changes with the simulation parameters $\boldsymbol{\alpha}$. Once again, we model this function with a neural network, effectively giving the network more expressive capabilities to directly influence the final deformed surface. Therefore, we will now explain how to set up and train this generative neural network architecture. Typically, generative approaches take low-dimensional input data and create high-dimensional outputs. The problem with such an approach lies in the vast number of degrees of freedom it can lead to, especially for our four dimensional setting.

To counter this issue, we choose a network structure with additional four-dimensional deconvolution layers. The deconvolution layers can be seen as the transpose of a regular convolution layer, and they can drastically decrease the number of network weights. The resulting structure of our neural network closely resembles an inverted regular convolutional network, and is illustrated in Fig. 5. We apply the trained deformation with an additional advection step:

$$\tilde{\psi}(\mathbf{x}) = \psi_0\left(\mathbf{x} - \mathbf{V}(\mathbf{x},\beta(\alpha,\theta))\right), \tag{16}$$
$$\psi(\mathbf{x}) = \tilde{\psi}\left(\mathbf{x} - \mathbf{w}(\mathbf{x},\alpha,\Theta)\right). \tag{17}$$

This way, the network only has to learn to refine the deformed surface $\tilde{\psi}$ to accommodate the nonlinear behavior of ϕ_α.

The velocity field $\mathbf{w}(\mathbf{x},\alpha,\Theta)$ is characterized by the network structure, the network weights $\Theta = (\Theta_{11}^1,\ldots,\Theta_{n_l n_{l-1}}^L)$ and the activation function $H(x)$. As input, we supply the network with the simulation parameters $\alpha = (\alpha_i,\ldots,\alpha_N)$ as a vector. The output of the network are four-component vectors, with the resolution $R_x \times R_y \times R_z \times R_t$. Note that in general the SDF resolution

and the deformation resolution do not need to be identical. Given a fixed SDF resolution, we can use a smaller resolution for the deformation, which reduces the number of weights and computations required for training significantly. This means that in practice each four-dimensional vector of the deformation acts on a region of the SDF, for which we assume the deformation to be constant. Therefore, we write the deformation field as

$$\mathbf{w}(\mathbf{x}, \alpha, \Theta) = \sum_j X_j(\mathbf{x}) \, \mathbf{w}_j(\alpha, \Theta), \qquad (18)$$

where $X_j(\mathbf{x})$ is the indicator function of the j-th region on which the four-dimensional deformation vector $\mathbf{w}_j(\alpha)$ acts. This vector is the j-th output of the neural network.

To train the network weights Θ, we need to calculate the gradient of the loss-function Eq. (7) with respect to the network weights. Just like in the previous section, it is sufficient to specify the gradient with respect to the network outputs $\mathbf{w}_i(\alpha, \Theta)$, for the update of Θ we can rely on backpropagation. Deriving Eq. (7) yields
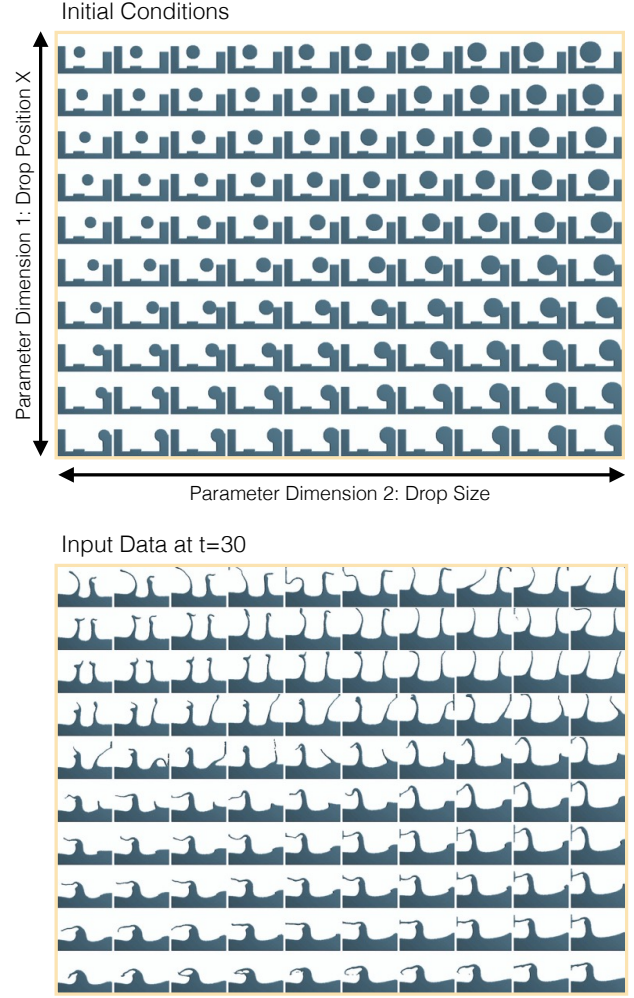
$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}_i} L$$
$$= \sum_j \frac{\mathrm{d}}{\mathrm{d}\mathbf{w}_i} \psi(\mathbf{x}) \, (\psi(\mathbf{x}) - \phi_\alpha(\mathbf{x}))$$
$$= \sum_j \frac{\mathrm{d}}{\mathrm{d}\mathbf{w}_i} \tilde{\psi} \, (\mathbf{x} - \mathbf{w}(\mathbf{x}, \alpha, \Theta)) \, (\psi(\mathbf{x}) - \phi_\alpha(\mathbf{x}))$$
$$= - \sum_j X_i(\mathbf{x}_j) \, \nabla \tilde{\psi}(\mathbf{x}_j - \mathbf{w}(\mathbf{x}_j, \alpha, \Theta)) \, (\psi(\mathbf{x}_j, \alpha) - \phi_\alpha(\mathbf{x}_j)) \, .$$

$$(19)$$

Thus, we can calculate the derivative by summation over the region that is affected by the network output $\mathbf{w}_i$. The gradient term is first calculated by evaluating a finite difference stencil on $\tilde{\psi}(\mathbf{x}_j)$ and then advecting it with the corresponding deformation vector $\mathbf{w}(\mathbf{x}_j, \alpha, \Theta)$. The other terms in Eq. (19) are readily available. Alg. 1 summarizes how the training of the network weights Θ is performed.

Data: training samples from ϕ_α
Result: trained network weights θ
for *each training sample* $\{\tilde{\alpha}, \tilde{\phi}\}$ **do**
    evaluate neural network to compute $\beta(\tilde{\alpha})$;
    load reference SDF $\tilde{\phi}$;
    load initial SDF ψ_0;
    calculate $\mathbf{V}(\mathbf{x}_i, \beta(\tilde{\alpha}))$;
    advect ψ_0 with $\mathbf{V}$ and store as $\tilde{\psi}$;
    calculate $\nabla \tilde{\psi}$;
    evaluate neural network to compute $\mathbf{w}_i(\tilde{\alpha})$;
    assemble $\mathbf{w}(\mathbf{x}_i)$ from $\mathbf{w}_i(\tilde{\alpha}, \Theta)$ according to Eq. (18);
    advect $\tilde{\psi}$ with $\mathbf{w}$;
    advect $\nabla \tilde{\psi}$ with $\mathbf{w}$;
    for *each* $\mathbf{w}_i$ **do**
        calculate the gradient $\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}_i} L$ according to Eq. (19);
    end
    backpropagation with $\frac{\mathrm{d}}{\mathrm{d}\mathbf{w}_i} L$ to adjust Θ;
end

Algorithm 1: Training the network weights

Working in conjunction, our two machine learning steps capture significantly more complex behavior of the fluid space-time surface ϕ_α over the whole parameter domain. We first adjust the application of the initial deformations with $\boldsymbol{\beta}(\boldsymbol{\alpha})$, and then let our network

Initial Conditions

Parameter Dimension 1: Drop Position X

Parameter Dimension 2: Drop Size

Input Data at t=30

Figure 6: *The top image illustrates the initial conditions of our two dimensional parameter space setup. It consists of a set of two-dimensional liquid simulations, which vary the position of the liquid drop along x as α_1, and its size as α_2. The bottom half shows the data used for training at $t = 30$. Note the significant amount of variance in positions of small scale features such as the thin sheets. Both images show only a subset of the whole data.*

generate an additional, input-dependent deformation field. We will demonstrate in Sec. 7 that learning a correction to the previously advected SDF $\tilde{\psi}$ performs significantly better than only learning the deformation from ψ_0 in a single step.

4.3 Training Details

To train both networks we apply the gradient descent method in combination with backpropagation [Bishop 2006]. More specifically, we use the ADAM optimizer, which is a standard for training neural networks. The outputs of the parameter network are the deformation parameters $\beta_i(\boldsymbol{\alpha})$. The network structure that we use to model this is a simple network with two fully connected layers with 8 nodes each.

The deformation network requires more degrees of freedom, as we can expect the function $\mathbf{w}(\boldsymbol{\alpha})$ to be much more complex. As such, we use a combination of fully connected layers and deconvolution
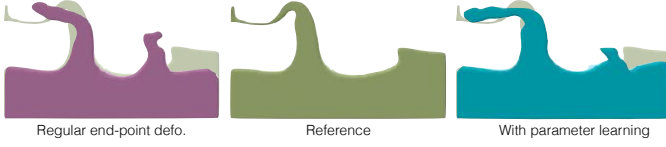
Figure 7: *An example of our parameter learning approach. From left to right: direct application of deformation weights, reference only, and parameters learned by our NN. Linear and NN versions show the reference in light yellow in the background. Our NN adjusts the deformation parameters to better represent the left sheet.*

layers to model the deformation field $\mathbf{w}(\boldsymbol{\alpha})$. Fig. 5 depicts a sketch of this network structure. We choose the first two layers to be fully-connected, with 16 and 2592 nodes respectively, and ReLU non-linearities.

The output of the second fully connected layer represents a four-dimensional vector array with dimensions 3^4 containing 32 features. The resolution of the deformation field is then upscaled with multiple four-dimensional deconvolution layers depending on the final resolution of the deformation field. To facilitate training, we choose it such that the final SDF resolution is a multiple of the deformation resolution. We also noticed that regular SDFs can lead to overly large error values far away from the surface. Thus, we apply the $\tanh$ function to the SDF values, in order to put more emphasis on the surface region. This slightly improved our training results.

As often the case with PDEs, special care is required for boundary conditions. In our case, this means the sides of our domain. Here, we use the common practice of Neumann boundary conditions for values outside of the computational domain. Hence, the SDF values extend to the outside with constant values. The problem with this approach is that this leads to vanishing gradients $\nabla\psi_0(\mathbf{x}) = \mathbf{0}$ in Eq. (19), which in turn often leads to artificial minima or maxima in the loss function. While this could be alleviated with strong regularization, we opted against this approach, and instead perform an on-the-fly distance extrapolation. Here we use the knowledge that the outside of our domain will always contain values outside of the liquid volume. Thus, we increase the retrieved SDF values by the distance of the retrieval point to the domain boundary. In this way, $\nabla\psi_0(\mathbf{x})$ can still be evaluated correctly at the boundaries.

For our implementation we use the *TensorFlow* framework. We integrated our custom deformation-aware loss function (7) and a four-dimensional deconvolution, which is currently, to the best of our knowledge, not supported by any machine-learning library.

Finally, the training is performed separately for both networks. Since the end-point deformations are applied first, we typically start by training the parameter network with 1000 training steps at a learning rate of 0.001. Once this is completed, we switch to the deformation network, and train it for another 9000 iterations, with the same learning rate.

4.4 Evaluation

In order to evaluate our method, we will first use a two-dimensional setup parameter space with two dimensional surfaces. For this purpose we use the SDFs extracted from two-dimensional simulations of a drop falling into a basin. As simulation parameters we choose α_1 to be the size of the drop, and α_2 to be its initial x-position. From this simulation we extract a single frame at $t = 30$, which gives us a two-dimensional parameter-space $\boldsymbol{\alpha} = (\alpha_1, \alpha_2)$, where each instance of $\boldsymbol{\alpha}$ has a corresponding two-dimensional SDF. This setup is illustrated in Fig. 6.

(a) *Parameter Learning*   **(b)** *Deformation learning*
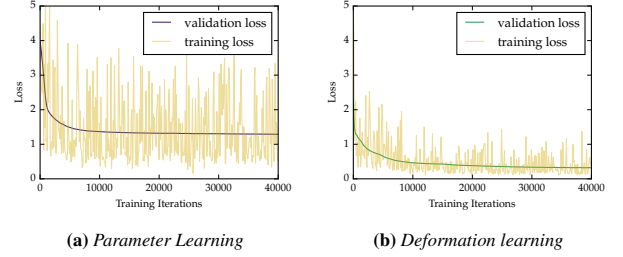
Figure 8: *Loss during training both for parameter learning and deformation learning. In yellow we show the loss for the current sample, while the dark line displays the loss evaluated on the validation set.*
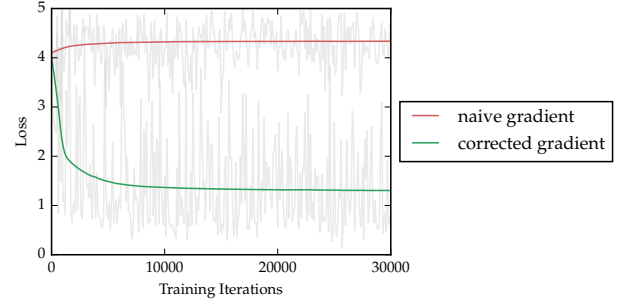
Figure 9: *Training with different gradient approximations: validation loss with a simplified advection (red), and the correct gradient (green). The simplified version does not converge.*

We span the parameter domain with two deformations $\mathbf{u}_1$ and $\mathbf{u}_2$. In order to train the networks described in section 4, we sample the parameter domain with a regular 44×49 grid, which gives us 2156 training samples.

To train the network we divide these samples into 2056 training samples and 100 validation samples. The samples are randomized and used individually in each iteration. Fig. 8 shows the validation loss and the training loss over the iterations both for parameter learning and for deformation learning. We observe that in both cases the learning process reduces the loss, and finally converges to a stable solution. This value is lower in the case of deformation training, which can be easily explained with the increased expressive capabilities of the deformation network. We verified that the solution converged by continuing training for another 36000 steps, during which the solution changed only minimally. In Fig. 7 we see the actual results of our parameter training algorithm. While the improvement is subtle, we can see that both splashes have an improved shape after adjusting the deformation parameters with the neural network.

For the gradient of Eq. (10) we mentioned above that it might seem attractive to approximate the effect of the forward-advection with a regular backward-advection step, as this simplifies the calculations significantly. However, due to the strong non-linearity of our setting, this is does not yield a usable algorithm, as the network will not converge with this simplification, as shown in Fig. 9.

The effect of our deformation learning approach is illustrated in Fig. 10. This figure compares our full method (on the right) with several other algorithms. As a baseline, the result from previous work is shown on the far left. As the surfaces of this example have larger distances and more complex correspondences than those originally demonstrated with previous work [Thuerey 2017], the

Previous work [Thuerey'16]   PCA reconstruction   NN, End-point defo. only   Reference only   With NN deformation
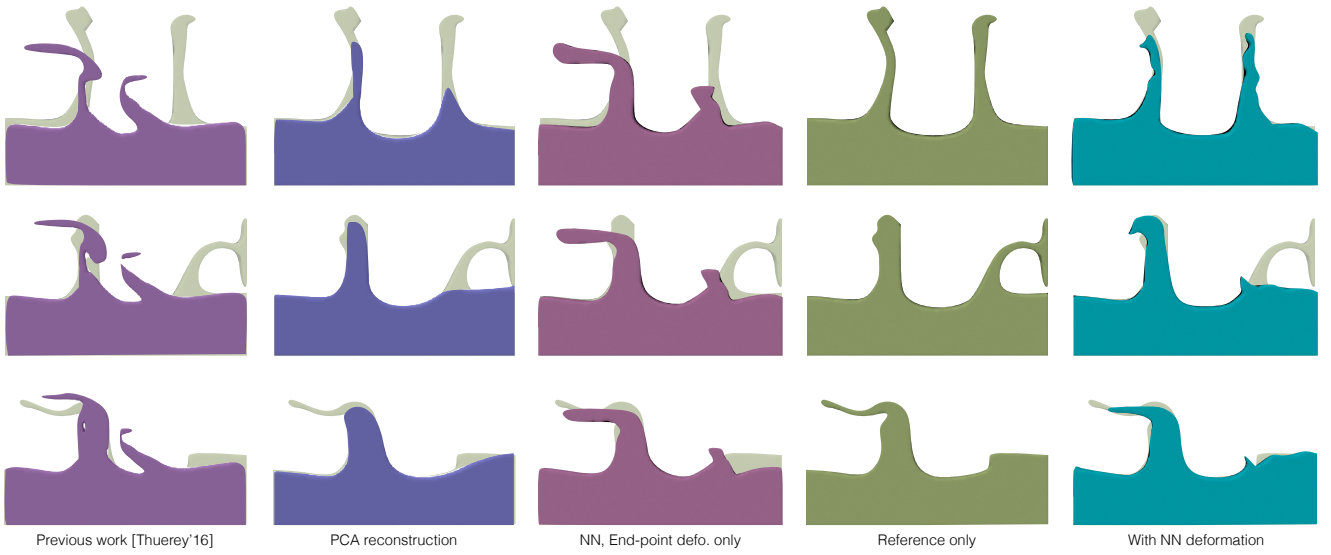
Figure 10: *Different example surfaces from the 2D parameter space of Fig. 6. From left to right: surfaces from previous work, surfaces reconstructed with PCA, adjusted deformations using a NN for only parameter learning, the reference surfaces, and on the far right the output of our full method with a deformation from the NN. Note that none of the other methods is able to reconstruct both arms of liquid in the first row, as well as the left sheet in the bottom row. The reference surfaces are shown in light yellow in the background for each version.*

closest-distance correspondences lead to an unsatisfactory results. A different, but popular approach for non-linear dimensionality reduction, which can be considered as an alternative to our method, is to construct a reduced basis with PCA. Using the mean surface with four eigenvectors yields a similar reduction to our method in terms of memory footprint. We additionally re-project the different reference surfaces into the reduced basis to improve the reconstruction quality of the PCA version. The result is a very smooth surface that fails to capture any details of the behavior of the parameter space, as can be seen in the second column of Fig. 10.

The middle column of this figure shows the surfaces obtained with the learned deformation weights with our parameter NN (Fig. 5 top), but without an additional NN deformation network. As this case is based on end-point deformations, it cannot adapt to larger changes of surface structure in the middle of the domain. In contrast, using our full pipeline with an NN deformation yields surfaces that adapt to the varying behavior in the interior of the parameter space, as shown on the right side of Fig. 10. However, it is also apparent that the deformations generated by the NN do not capture every detail of the references. Our reduced representation in this case is regularized by the varying reference surfaces in small neighborhoods of $\boldsymbol{\alpha}$, and it learns an averaged behavior from the inputs.

Discussion It is worth noting that using the end-point deformations only, we would have to store many additional deformations in order to accurately resolve the complex behavior for intermediate simulation parameters. This problem even gets worse when the dimensionality of the simulation parameters is increased, as we would require a dense web of deformations covering the whole parameter domain in this case. We circumvent this problem with our generative approach, which is based on the assumption that an approximate flow behavior is already present in the initial surface ψ_0 and the deformed version $\tilde{\psi}$. Based on this approximation, the network only has to learn how to deform the intermediate surfaces so that they match the references throughout the whole parameter domain. We have demonstrated that in two-dimensions, our method is able learn a very compact representation of the liquid surface $\Gamma(\boldsymbol{\alpha})$

that allows the generation of very good approximations. For this reason, we see our method as a specialized technique to perform non-linear-dimensionality reduction on the space of liquid surfaces $\Gamma(\boldsymbol{\alpha})$.

5 Long-range Deformations

Our neural network can successfully refine the deformations based on an initial set of pre-computed deformations along the sides of the parameter space. Therefore, any improvement for these initial deformations typically also translates into improvements of quality for the output of our neural network. For this reason, we will propose an additional step that targets long-range correspondences, before discussing more complex four-dimensional results.

When computing the end-point deformations we have two closely related goals: to accurately match the target, and to establish correct correspondences between source and target. The algorithm we use (optical flow), and many others such as the iterative closest point methods, work by matching closest distances. For inputs with significant differences, a deterioration of both desired qualities goes hand in hand: parts of the target shape can get lost, and the likelihood of reconstructing a target feature from an arbitrary different source feature increases. We make the following observation: high similarity between inputs improves matching quality and correspondences, and this similarity is not proportional to distances in simulation parameter space. I.e., two inputs A and B with a distance d in $\boldsymbol{\alpha}$ can be more similar than A and a third point C with a distance less than d. This is a result of the high non-linearity of the input simulations. This effect is visible in Fig. 3, where the first image is closer to the third one on the right, while it is closer to the second one in parameter space.

Therefore, given the problem to compute a deformation from source surface at simulation parameter $\boldsymbol{\alpha}_a$ to a surface at $\boldsymbol{\alpha}_b$, we subdivide this interval by inserting additional data points. We can, e.g., use linear interpolation on the simulation parameters, and generate additional four-dimensional surfaces as data points along the line. In the following, we will denote the subdivisions of the in-
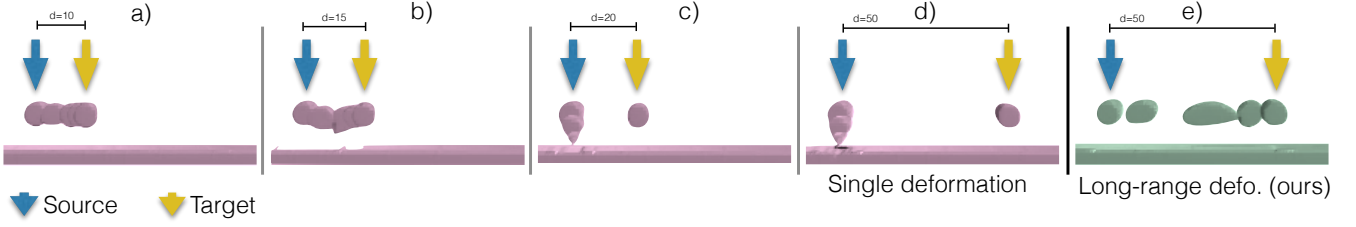
Figure 11: *This figure illustrates our approach for long-range deformations. We match a sphere above a basin at different horizontal positions. Source position is indicated by the blue arrow and target position by the yellow arrow. For growing distances d, a single deformation step quickly merges the drop into the basin (c,d). The combined deformation of our approach (e) recovers the whole tangential motion along the surface of the basin.*

terval $[\boldsymbol{\alpha}_a, \boldsymbol{\alpha}_b]$ with $a_0, a_1, a_2, ...a_{l-1}, a_l$, where $a_0 = a, a_l = b$, and deformations for a section of this interval with $\mathbf{u}_{a_i,a_j}$ (with $i, j \in [0..l]$). We typically use a fixed spacing for the additional parameter samples a_i, but our method is not restricted to a regular sampling of the [a,b]-interval. To shorten the notation, we will drop α in the following section for subscripts, i.e., instead of $\phi\alpha_i$, we will use ϕ_i. Thus, the deformations with double indices cover a partial distance of a single dimension of our parameter space, in contrast to the final endpoint deformations, e.g., $\mathbf{u}_j$, which cover the full extent of a single dimension j of the parameter space. In order to reach the target state ϕ_b we want to minimize the error between the final surface ψ (computed by deforming ϕ_{α_a}) and the target, i.e. $e(\psi, \phi_{\alpha_b})$, where $e(\cdot, \cdot)$ computes the error between the two surfaces. In the following we use the indicator function from *FlOF* for e [Thuerey 2017], but an L_2 norm of the SDF values with $\tanh$, as for our neural networks, is likewise applicable.

In contrast to previous work [Raveendran et al. 2014; Thuerey 2017], we compute a single end-point deformation deformation to retrieve $\tilde{\phi}_a$ with a merged sequence of carefully selected deformations $\mathbf{u}_{l_0,l_1}, \mathbf{u}_{l_1,l_2}, ..., \mathbf{u}_{l_{e-1},l_e}$ that minimizes the accumulated error. Each step in the deformation sequence potentially introduces an error that is propagated towards the target. This error is not propagated additively when applying several deformations in a row. However, the additive accumulation of errors represents a practical lower bound. Consider an example deformation from interval position i to k via an intermediate point j. For a surface Φ deformed by two deformations as $\Phi = \phi_i(\mathbf{x} - \mathbf{u}_{j,k}(\mathbf{x}) - \mathbf{u}_{i,j}(\mathbf{x} - \mathbf{u}_{j,k}(\mathbf{x})))$ the error bound is given by

$$e(\Phi, \phi_k) > e(\phi_i(\mathbf{x} - \mathbf{u}_{i,j}(\mathbf{x})), \phi_j) + \\ e(\phi_j(\mathbf{x} - \mathbf{u}_{j,k}(\mathbf{x})), \phi_k). \quad (20)$$

The reason is that $\mathbf{u}_{i,j}$ will typically introduce imperfections, and $\phi_i(\mathbf{x} - \mathbf{u}_{i,j}(\mathbf{x}))$ will be a worse starting point for $\mathbf{u}_{j,k}$ than ϕ_j. Thus, we minimize the accumulated errors along a sequence of discrete pairs $S = \{(i_0, j_0), (i_1, j_1), ..., (i_{n_S}, j_{n_S})\}$, each with an associated error value, and with $i_0 = a, j_{n_S} = b$. We calculate the total error for the sequence E_S with the error metric e as

$$E_S = \sum_{(i_k, j_k) \in S} e(\phi_{i_k}(\mathbf{x} - \mathbf{u}_{i_k, j_k}(\mathbf{x})), \phi_{j_k}). \quad (21)$$

Minimizing E_S is a classic dynamic programming problem. We compute a dense directed graph G, where the associated cost for each edge is the deformation error of the corresponding deformation. Thus, given a sequence of surfaces $\phi_0, \phi_1, ..., \phi_l$ we compute deformations from every surface ϕ_i towards the next n_j surfaces, adding the edge $d_{i,j}$ with the cost from Eq. (21) to G. Theoretically, we could compute all possible deformations, but we found that $n_j \approx l/4$ works well in practice. The optimal sequence S is then computed by using Dijkstra's algorithm on G, and we compute the final deformation $\mathbf{u}_{a,b}$ by concatenating the deformations of S with our alignment from Sec. 3.1 to yield a single end-point deformation for one dimension of the parameter space.

Examples Fig. 11 shows an example deformation computed with this approach. In this example, the goal is to move a droplet with a distance of ca. 15 cells from the surface of a basin horizontally. We match a source surface (with the drop on the left) with a target where the drop is translated by various distances to the right. For targets with a distance larger than 15, a single regular deformation merges the drop into the basin, and re-creates the target droplet from the basin, instead of moving the droplet horizontally. While a single deformation fails at preserving the correspondences of the drop for distances of 20 and above (Fig. 11c,d), our algorithm can recover the whole translation of the droplet across the domain (Fig. 11e). While our method successfully establishes correspondences, the motion along the way is typically not linear, as can be seen in Fig. 11.

The error values from Eq. (21) for this test case are shown in Fig. 13. For this simple case, the error values change smoothly in the graph, and large distances consistently get worse. A diagonal high-error line at a distance of ca. 18 cells is clearly visible in the graph. This is the critical point where the source drop has a similar distance to the target drop as well as the basin. This typically causes the drop to break up when applying the deformation, leading to larger error measurements. Our algorithm detects the increased error, and for this case makes one step towards this error barrier. Afterwards, it stays just below it to reach the target state. The initial position of the droplet is $x = 25$, and our algorithm finds the solution to combine deformations $\mathbf{u}_{25,39}, \mathbf{u}_{39,49}, \mathbf{u}_{49,59}, \mathbf{u}_{67,75}$ to reach the target at position $x = 75$.

A second example using the dataset of Fig. 6 can be seen in Fig. 12. Here the source is shown on the left, and the target surface on the right. The single deformation, in line with previous work [Thuerey 2017], mostly restores the target shape, but erroneously merges both "arms" into the single wave of the target. In contrast, our method reconstructs the wave using only the left arm of the source shape, and correctly merges the right arm into the surface of the basin. The full sequence can be seen in the accompanying video.

Note that while this approach takes into account the in-between data to improve correspondences, the resulting deformations still can not re-construct any of the details in-between the end-points. We leave it to our neural network step to correct this behavior during deformation refinement.
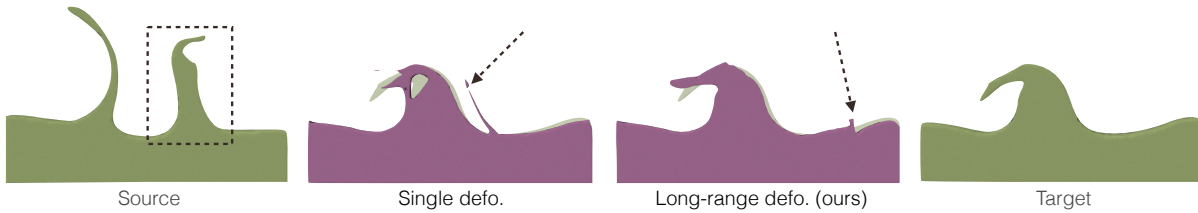
Figure 12: *A more complex surface illustrating the improvements of our long-range deformation approach. The source surface (left image) is matched with the target (far right) using a single deformation from previous work (middle left), and our approach for long-range deformations (middle right). The latter preserves more of the target, and correctly establishes correspondences between only the left arm of the source and the target sheet. The right arm of the source (dashed blue box) is correctly merged into the bulk volume in our method (dashed arrow), while the single deformation erroneously merges it into the left arm.*
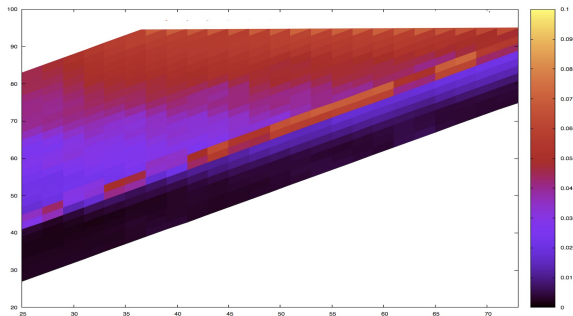
Figure 13: *This plot shows the error values for the different intermediate steps of Fig. 11 from Eq. (21). The x-axis denotes source, the y-axis target position of the droplet.*

6 Mobile Device Implementation

The significant amount of pre-calculations to compute our reduced model pays off especially for interactive settings. To demonstrate the high performance our representation achieves, we have implemented a proof-of-concept application for mobile devices. Our application is divided into two parts: a main loop used for the surface generation and rendering, and a second loop which handles the user interaction, and calculates a final deformation for the main loop to display. Both loops are working with separated threads and are using double-buffering to provide the results.

The performance intensive tasks of our pipeline are computed with *RenderScript*, which is a framework which distributes tasks onto the CPU and GPU. The neural network instead will be computed on the CPU using the *TensorFlow* C++ library in conjunction with our custom 4D deconvolution. Here, we load a graph and a pre-trained set of weights computed during a pre-processing stage. Thus, our application only handles the evaluation of the network, but not the training. In order to guarantee temporal coherence, we temporally blend the SDFs when restarting the simulation or switching to a new deformation.

As we have a full implicit surface of the liquid, we implemented a three-state particle system, to model splashes, bubbles and foam floating on the surface [Ihmsen et al. 2012]. We approximate the fluid velocity using the gradients of the SDF. For bubbles and floating particles, we can again make use of the implicit representation to project them onto the liquid surface. The scene is rendered with a custom raytracer, that handles reflection, refraction, and shading with an environment map. We use depth-based attenuation to visualize the liquid volume, and we shade the surface based on curvature to highlight detail. Both depth and curvature can be easily

retrieved from the implicit surfaces that our algorithm generates.

Our second thread becomes active whenever the user interacts with the application. User input typically yields a selection of a point in our parameter space $\boldsymbol{\alpha}$. Based on this value, we first let the neural network calculate the adjusted parameters $\boldsymbol{\beta}$. With these parameters, we calculate the aligned end-point deformations, and combine the resulting field with the deformation calculated by the neural network. While we typically perform two advection steps during our off-line implementation, we can use our alignment in this real-time setting to combine end-point deformation and neural network deformation into a single field. In this way, we only have to calculate a single advection step to extract the final, three-dimensional SDF surfaces from slices of the 4D buffers. Once the calculations are completed, the main loop swaps the active deformation buffer with the buffer of the newly generated one, and starts displaying the new configuration.
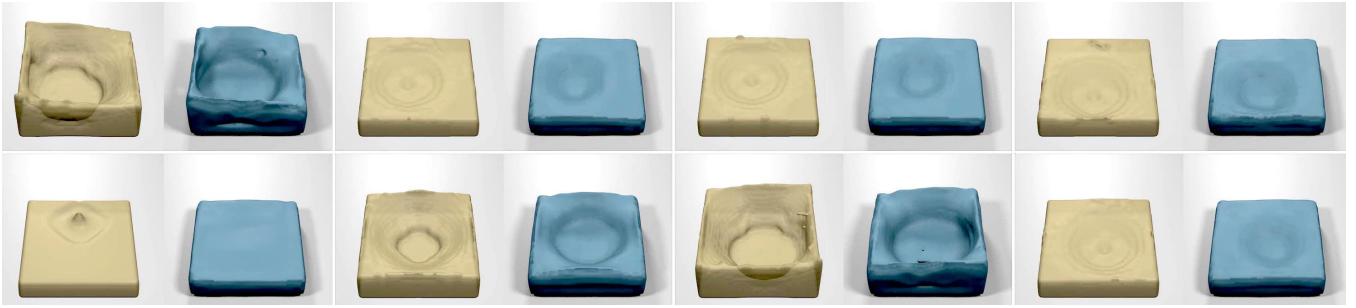
7 Results

Below we will demonstrate results of our algorithm for full, three-dimensional simulations. In the following, we will show that our network approach is effective at learning good approximations of the four-dimensional liquid surfaces and that we can render them very efficiently in real-time scenarios.

Liquid Drop falling into a Basin As our first 4D test case, we chose a drop of liquid falling into a basin. As our simulation parameters we chose the x and y coordinates of the initial drop position, as well as the size of the drop. Due to non-linearity of splashes and wall interactions and due to the three-dimensional parameter space, this set of inputs has a wide variety of sheets and splashes. Examples are shown on the left of Fig. 1, and in our video. To generate the training data, we sample the parameter space on a regular grid, and run simulations, each with a spatial resolution of 100^3 to generate a total of 1764 reference SDFs.

As a first test we want to demonstrate the effectiveness of the neural network deformation by showcasing it on its own. For this purpose, we conduct a test, were ψ_0 is simply a basin filled with liquid at rest. We retrieve the result directly by evaluating $\psi(\mathbf{x}) = \psi_0 \left(\mathbf{x} - \mathbf{w}(\mathbf{x}, \alpha, \Theta) \right)$. This means we do not employ any end-point deformations for this test. As ψ_0 contains only a planar surface that is not moving, the neural network has to learn to represent all motions of the liquid. For this test we only consider the latter two-thirds in time, i.e. after the drop has merged with the basin. Fig. 14 shows several results from the space of solutions, after training for 10000 iterations. For all pairs, our network correctly approximates the large scale motion of the splashes. Considering the lack of features in ψ_0, this is a very good result, as there are

Figure 14: *Deformation learning with a flat basin as starting point. The reference is depicted in yellow, the deformed result in blue. Although the initial SDF ψ_0 is just a flat, planar surface, the network is able to capture a significant amount of the motion.*
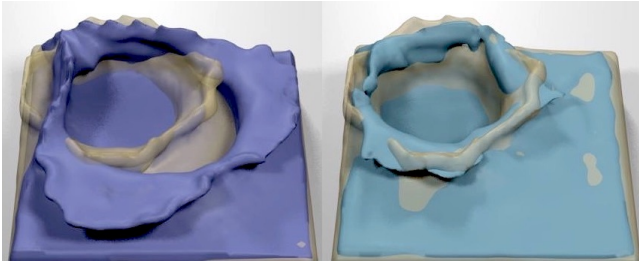
Figure 15: *In yellow the reference, in purple on the left the deformed surface after applying the end-point deformations. In blue on the right, our final surface with the network deformation. Our result is significantly closer to the reference surface.*

Table 1: *Performance details of our mobile version measured on a Samsung S7 device.*

	Drop	Stairs
SDF resolution	40^4	50^4
Deformation resolution	20^4	25^4
Network evaluation	74 ms	531 ms
Deformation generation	440 ms	1357 ms
SDF generation	11 ms	30 ms
Rendering	65 ms	80 ms
Framerate	12 fps	9 fps

no small scale details the network can shift in space and time to represent the targets.

We now explain the setup for our full pipeline with this drop setup. We start with the biggest possible drop falling into the upper right corner of the basin. We then span the parameter space with the three deformations $\mathbf{u}_1$, $\mathbf{u}_2$ and $\mathbf{u}_3$. The first two parameters are the x- and y-coordinates of the drop. The third parameter stands for the size of the drop.

The still frame of Fig. 15 illustrates the gain in quality we can achieve with our neural network deformation. Also note the increase in small scale detail we achieve in comparison to the flat surface test above. Several other examples from the whole space of solutions can be found in the video. This direct comparison illustrates that our network successfully learns to accurately correct the complex space-time surface of our inputs.

We run the trained model for this setup in our mobile application to demonstrate the high performance of our approach. In this application the user can interact by tapping on the screen to select the drop position. Fig. 16 depicts a capture of our android application. In Table 1 we have listed the runtimes and framerates of our mobile application. The deformation calculation step is the most time consuming one; e.g. 440ms using deformations of size 20^4. Here, our implementation does not yet use the GPU for computing the deconvolutions. However, this calculation of deformations is hidden from the user with our double buffering setup. Once the deformation is calculated, rendering is the most expensive task with 65ms on average.

Note that the original simulation for this setup (to generate a single input) took 530 seconds on average with a parallel implementation using all cores of a desktop workstation. Assuming a best-case slowdown of only 4x for the mobile device, it would require more

than 30 minutes to run the original simulation. Reducing the spatial and temporal resolution would result in faster runtimes, but also in a correspondingly reduced quality. Thus, our reduced model generates detailed outputs highly efficiently, with only very moderate computational resources.

In addition, popular approaches such as particle based liquid simulations would require additional time and implementation complexity to generate a renderable surface from the particle cloud. Our choice to work with implicit surface data directly yields a 3D dataset that can be visualized with ray-marching.

Our approach very efficiently represents a region of complex behavior with a strongly reduced representation. This becomes obvious when comparing our method with a direct interpolation of SDF data-sets. Our algorithms requires a single full-resolution SDF, three half resolution deformations, and the neural network (which is negligible in practice). While a single 40^4 SDF requires ca. 2.5 mio. scalar values, all deformations and network weights require ca. 2 mio. scalars. Thus our representation encodes the full behavior with less storage than two full SDFs. To illustrate this point, we show the result of a direct SDF interpolation in Fig. 17. Here we sample the paramter space with 8 SDFs in total (at all corners of the 3D parameter space). Hence, this version requires more than 4x the storage our approach requires. Despite the additional memory, direct SDF interpolations lead to very obvious, and undesirable artifacts. The result shown in the lower row of Fig. 17 neither represents the initial drop, nor the resulting splash. Rather, it leads to strong ghosting artifacts, and an overall loss of detail.

Liquid flowing over Stairs Our second test setup illustrates a different parameter space that captures a variety of obstacle boundary conditions with $\boldsymbol{\alpha}$. Our first two simulation parameters are the heights of two stair-like steps, while the third parameter is control-
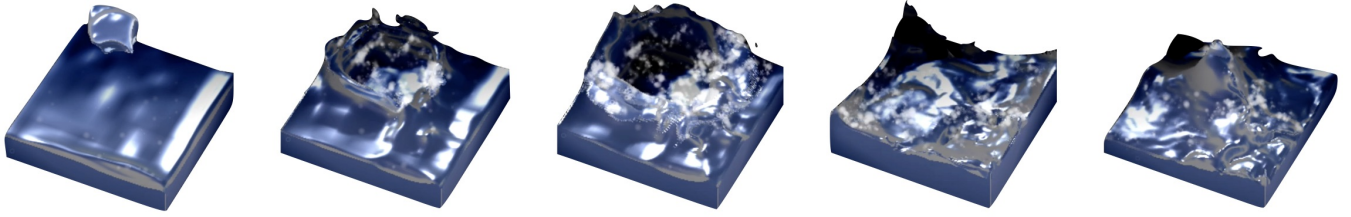
Figure 16: *Several screens from the drop test case in our mobile application with the drop setup.*

Our approach

- -

Direct SDF Interpolation
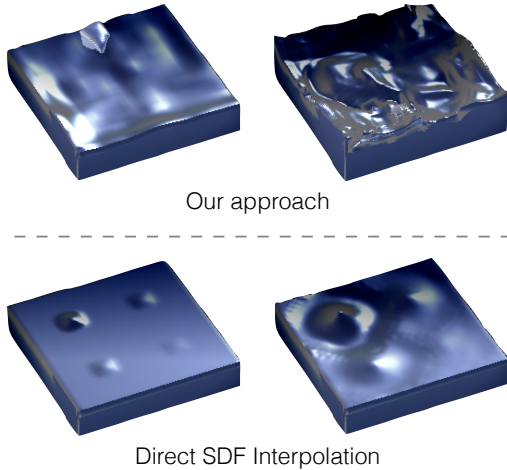
Figure 17: *Two frames generated with our approach (top) and with a direct SDF interpolation using a similar amount of overall memory (bottom). The latter looses the inital drop shape, and leads to strong ghosting artifacts with four repeated surface waves.*

ling the position of a middle divider obstacle. The liquid flows in a U-shaped manner around the divider, down the stairs. For this setup, we used a slightly higher overall resolution for both space-time SDFs, as well as for the output of the network, details can be found in Table 1.

Fig. 18 depicts still frames captured from our mobile application for this setup. With this setup the user can adjust stair heights and wall width dynamically while deformations are computed in the background. While this setup has more temporal coherence in its motion than the drop setup, the changing obstacle boundary conditions lead to strongly differing streams of liquid down the stairs. E.g., changing the position of the divider changes the flow from a narrow, fast stream to a slow, broad front.

Please note that the mobile application is part of our submission materials, and we encourage readers to try out our method. Unfortunately, we only support Android at the moment.

Discussion and Limitations Our formulation is centered around our loss function which can evaluate network outputs according to the resulting difference to a reference it produces. Using this loss function we were able to combine prescribed deformations such that they generate optimal results across the whole parameter range. Our generative neural network can then capture additional non-linear interactions that the end-point deformations missed.

While we have demonstrated the effectiveness of this approach, it is apparent that the deformations generated by our algorithm can

Table 2: *Overview of our 2D and 4D simulation and machine learning setups. Timings were measured on a Xeon E5-1630 with 3.7GHz.* Res, SDF *and* Defo *denote resolutions for simulation, training, and the NN deformation, respectively;* Sim *and* Train *denote simulation and training runtimes.*

Setup	Res.	SDF	Defo.	Sim.	Train
2D setup, Fig. 6	100^2	100^2	25^2	-	186s
Drop setup, Fig. 16	$100^3 \times 100$	40^4	10^4	8.8m	22m
Stairs setup, Fig. 18	$110^3 \times 110$	50^4	15^4	9.7m	186m

introduce spatial and temporal discontinuities. This is noticeable when deforming the flat surface in Fig. 14. We believe it is important to show these artifacts, as this is how the unmodified output of our neural network behaves. However, as we use an Eulerian representation for the deformations, it would be relatively straight forward to smoothen the deformation, or, even better, to introduce a regularization term in the loss function. However, we believe that in its current form we have demonstrated that the trained networks have learned to deform the surfaces towards the desired targets, as demonstrated, e.g., in Fig. 15.

Currently, our training steps also require a large amount of input data. While we believe this is necessary to some extent to specifiy the target space, we have experimented with using only a smaller, randomly chosen subset of samples for training. Our tests indicate that it is possible to achieve similarly good results with data-set reduced by a factor of two to four in comparison to our full, densely sampled training sets.

Right now our pipeline is not physics-aware, as it purely considers space-time surfaces, and neglects potential constraints from the governing equations of the physical systems (e.g., conservation of mass). While previous work shares this limitation [Raveendran et al. 2014; Thuerey 2017], this property also makes these methods more general, as no assumptions need to be made about the inputs. However, it is clearly an interesting topic for future work to incorporate physics constraints.

Additionally, our approach is only applicable in settings where inputs can be grouped around a reasonably representative initial surface ψ_0. We have demonstrated that we can achieve a significant amount of variance with our examples above. For future extensions, we believe it will be very interesting to automate the process of finding good sub-spaces where our method is applicable from even larger spaces of liquid behavior.

8 Conclusions

We have presented a novel method to pre-compute a complex liquid space with deformation-aware neural networks. Due to the cho-
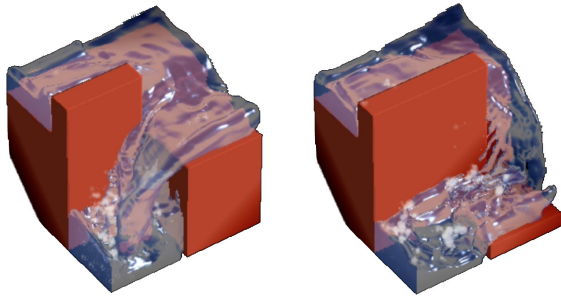
Figure 18: *These screens illustrate our stairs setup running in our mobile application. The front corner is lowered from the first to the second image, and the divider pushed forward.*

sen representation, our approach allows for real-time interactions with complex liquid effects. To the best of our knowledge, we are also the first to introduce a machine-learning approach that actively learns to deform surfaces.

We believe that it is exciting to see how such generative neural networks can be successfully applied to problems in the context of fluid simulations. In the future, it will be interesting to simulate other types of phenomena with our synthesis pipeline, and we plan to investigate how our deformation-aware networks perform for optical flow-like problems outside of the area of fluid simulations.

References

BAILER, C., VARANASI, K., AND STRICKER, D. 2016. Cnn-based patch matching for optical flow with thresholded hinge loss. *arXiv preprint: 1607.08064*.

BARBIČ, J., AND JAMES, D. L. 2005. Real-time subspace integration for st. venant-kirchhoff deformable models. *ACM transactions on graphics (TOG) 24*, 3, 982–990.

BELL, S., AND BALA, K. 2015. Learning visual similarity for product design with convolutional neural networks. *ACM Transactions on Graphics (TOG) 34*, 4, 98.

BISHOP, C. M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

CHENTANEZ, N., AND MÜLLER, M. 2011. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics (TOG) 30*, 4, 82.

CHENTANEZ, N., MÜLLER, M., MACKLIN, M., AND KIM, T.-Y. 2015. Fast grid-free surface tracking. *ACM Transactions on Graphics (TOG) 34*, 4, 148.

CHU, M., AND THUEREY, N. 2017. Data-driven synthesis of smoke flows with CNN-based feature descriptors. *ACM Transaction on Graphics (ToG) 36(4)*.

DE WITT, T., LESSIG, C., AND FIUME, E. 2012. Fluid simulation using laplacian eigenfunctions. *ACM Transactions on Graphics (TOG) 31*, 1, 10.

DOSOVITSKIY, A., FISCHERY, P., ILG, E., HAZIRBAS, C., GOLKOV, V., VAN DER SMAGT, P., CREMERS, D., BROX, T., ET AL. 2015. Flownet: Learning optical flow with convolutional networks. In *International Conference on Computer Vision (ICCV)*, IEEE, 2758–2766.

ENRIGHT, D., NGUYEN, D., GIBOU, F., AND FEDKIW, R. 2003. Using the Particle Level Set Method and a Second Order Accurate Pressure Boundary Condition for Free-Surface Flows. *Proc. of the 4th ASME-JSME Joint Fluids Engineering Conference*.

GIRSHICK, R., DONAHUE, J., DARRELL, T., AND MALIK, J. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. Conference on Computer Vision and Pattern Recognition*, IEEE, 580–587.

GOODFELLOW, I. J., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. 2014. Generative adversarial nets. *stat 1050*, 10.

GUPTA, M., AND NARASIMHAN, S. G. 2007. Legendre fluids: a unified framework for analytic reduced space modeling and rendering of participating media. In *Proc. Symposium on Computer Animation*, ACM/Eurographics, 17–25.

IHMSEN, M., AKINCI, N., AKINCI, G., AND TESCHNER, M. 2012. Unified spray, foam and air bubbles for particle-based fluids. *The Visual Computer 28*, 6-8, 669–677.

IHMSEN, M., ORTHMANN, J., SOLENTHALER, B., KOLB, A., AND TESCHNER, M. 2014. SPH fluids in computer graphics. In *State of the Art Reports*, Eurographics, 21–42.

IIZUKA, S., SIMO-SERRA, E., AND ISHIKAWA, H. 2016. Let there be color!: joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification. *ACM Transactions on Graphics (TOG) 35*, 4, 110.

ILG, E., MAYER, N., SAIKIA, T., KEUPER, M., DOSOVITSKIY, A., AND BROX, T. 2016. Flownet 2.0: Evolution of optical flow estimation with deep networks. *arXiv preprint: 1612.01925*.

JAMES, D. L., AND FATAHALIAN, K. 2003. Precomputing interactive dynamic deformable scenes. *Proc. SIGGRAPH 22*, 3.

JAMES, D. L., BARBIČ, J., AND PAI, D. K. 2006. Precomputed acoustic transfer: output-sensitive, accurate sound generation for geometrically complex vibration sources. *ACM Transactions on Graphics (TOG) 25*, 3, 987–995.

JONES, A. D., SEN, P., AND KIM, T. 2016. Compressing fluid subspaces. In *Proc. Symposium on Computer Animation*, ACM/Eurographics, 77–84.

KASS, M., AND MILLER, G. 1990. Rapid, Stable Fluid Dynamics for Computer Graphics. *ACM Trans. Graph. 24*, 4, 49–55.

KIM, T., AND DELANEY, J. 2013. Subspace fluid re-simulation. *ACM Trans. Graph. 32*, 4 (July), 62:1–62:9.

KIM, T., AND JAMES, D. L. 2012. Physics-based character skinning using multido-main subspace deformations. *IEEE transactions on visualization and computer graphics 18*, 8, 1228–1240.

KIM, D., KOH, W., NARAIN, R., FATAHALIAN, K., TREUILLE, A., AND O'BRIEN, J. F. 2013. Near-exhaustive precomputation of secondary cloth effects. *ACM Transactions on Graphics (TOG) 32*, 4, 87.

KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, NIPS, 1097–1105.

LADICKY, L., JEONG, S., SOLENTHALER, B., POLLEFEYS, M., AND GROSS, M. 2015. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG) 34*, 6, 199.

MACKLIN, M., AND MÜLLER, M. 2013. Position based fluids. *ACM Transactions on Graphics (TOG) 32*, 4, 104.

MACKLIN, M., MÜLLER, M., CHENTANEZ, N., AND KIM, T.-Y. 2014. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG) 33*, 4, 153.

MANTEAUX, P.-L., SUN, W.-L., FAURE, F., CANI, M.-P., AND O'BRIEN, J. F. 2015. Interactive detailed cutting of thin sheets. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, ACM, 125–132.

MCNAMARA, A., TREUILLE, A., POPOVIĆ, Z., AND STAM, J. 2004. Fluid Control Using the Adjoint Method. *ACM Trans. Graph. 23*, 3, 449–456.

PENTLAND, A., AND WILLIAMS, J. 1989. Good vibrations: Modal dynamics for graphics and animation. *Proc. SIGGRAPH 23*, 3.

PRANTL, L., BONEV, B., AND THUEREY, N., 2017. Neural Liquid Drop. *https://play.google.com/store/apps/details?id=fluidsim.de.interactivedrop*.

RANJAN, A., AND BLACK, M. J. 2016. Optical flow estimation using a spatial pyramid network. *CoRR abs/1611.00850*.

RAVEENDRAN, K., THUEREY, N., WOJTAN, C., AND TURK, G. 2014. Blending Liquids. *ACM Trans. Graph. 33 (4)* (August), 10.

RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. 1988. Learning representations by back-propagating errors. *Cognitive modeling 5*, 3, 1.

SANGKLOY, P., BURNELL, N., HAM, C., AND HAYS, J. 2016. The sketchy database: learning to retrieve badly drawn bunnies. *ACM Transactions on Graphics (TOG) 35*, 4, 119.

SOLENTHALER, B., AND PAJAROLA, R. 2009. Predictive-corrective incompressible sph. *ACM transactions on graphics (TOG) 28*, 3, 40.

SOLOMON, J., DE GOES, F., PEYRÉ, G., CUTURI, M., BUTSCHER, A., NGUYEN, A., DU, T., AND GUIBAS, L. 2015. Convolutional wasserstein distances: Efficient optimal transportation on geometric domains. *ACM Transactions on Graphics (TOG) 34*, 4, 66.

STAM, J. 1999. Stable Fluids. In *Proc. ACM SIGGRAPH*, ACM, 121–128.

STANTON, M., HUMBERSTON, B., KASE, B., O'BRIEN, J., FATAHALIAN, K., AND TREUILLE, A. 2014. Self-refining games using player analytics. *ACM Trans. Graph. 33 (4)*, 9.

STAVA, O., BENEŠ, B., BRISBIN, M., AND KŘIVÁNEK, J. 2008. Interactive terrain modeling using hydraulic erosion. In *Proc. Symposium on Computer Animation*, ACM/Eurographics, 201–210.

TENG, Y., MEYER, M., DEROSE, T., AND KIM, T. 2015. Subspace condensation: full space adaptivity for subspace deformations. *ACM Transactions on Graphics 34*, 4, 76.

TENG, Y., LEVIN, D. I., AND KIM, T. 2016. Eulerian solid-fluid coupling. *ACM Transactions on Graphics 35*, 6, 200.

THUEREY, N. 2017. Interpolations of Smoke and Liquid Simulations. *Transactions on Graphics 36(1)* (July), 15.

TOMPSON, J., SCHLACHTER, K., SPRECHMANN, P., AND PERLIN, K. 2016. Accelerating eulerian fluid simulation with convolutional networks. *arXiv preprint: 1607.03597*.

TREUILLE, A., LEWIS, A., AND POPOVIĆ, Z. 2006. Model reduction for real-time fluids. *ACM Trans. Graph. 25*, 3 (July), 826–834.

UM, K., HU, X., AND THUEREY, N. 2017. Splash modeling with neural networks. *arXiv preprint*.

WANG, H., MILLER, G., AND TURK, G. 2007. Solving general shallow wave equations on surfaces. In *Proc. Symposium on Computer Animation*, ACM/Eurographics, 229–238.

WEI, L., HUANG, Q., CEYLAN, D., VOUGA, E., AND LI, H. 2015. Dense human body correspondences using convolutional networks. *arXiv preprint: 1511.05904*.

WERBOS, P. J. 1974. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University. Department of Applied Mathematics.

WICKE, M., STANTON, M., AND TREUILLE, A. 2009. Modular bases for fluid dynamics. *ACM Trans. on Graphics 28*, 3 (Aug.), 39.

XU, H., AND BARBIČ, J. 2016. Pose-space subspace dynamics. *ACM Transactions on Graphics (TOG) 35*, 4, 35.

YANG, C., YANG, X., AND XIAO, X. 2016. Data-driven projection method in fluid simulation. *Computer Animation and Virtual Worlds 27*, 3-4, 415–424.

ZHU, Y., AND BRIDSON, R. 2005. Animating Sand as a Fluid. *ACM Trans. Graph. 24*, 3, 965–972.