Data-Driven Synthesis of Smoke Flows with CNN-based Feature Descriptors

MENGYU CHU, Technical University of Munich
NILS THUEREY, Technical University of Munich

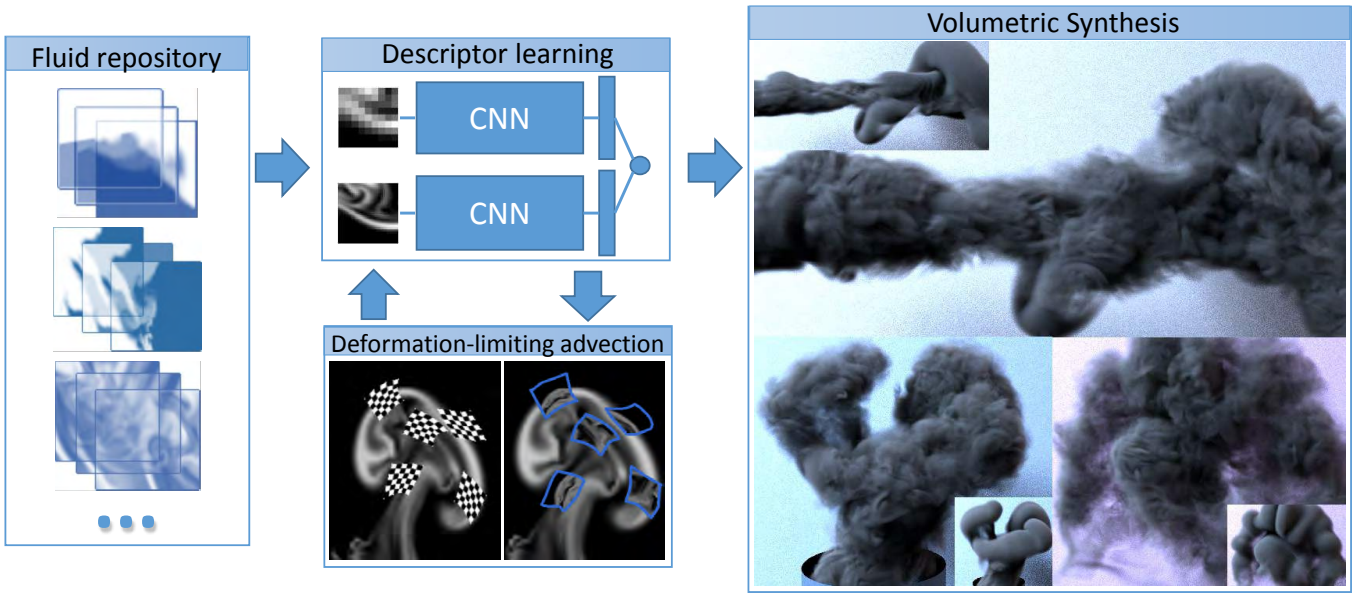


Fig. 1. We enable volumetric fluid synthesis with high resolutions and non-dissipative small scale details using CNNs and a fluid flow repository.

We present a novel data-driven algorithm to synthesize high resolution flow simulations with reusable repositories of space-time flow data. In our work, we employ a descriptor learning approach to encode the similarity between fluid regions with differences in resolution and numerical viscosity. We use convolutional neural networks to generate the descriptors from fluid data such as smoke density and flow velocity. At the same time, we present a deformation limiting patch advection method which allows us to robustly track deformable fluid regions. With the help of this patch advection, we generate stable space-time data sets from detailed fluids for our repositories. We can then use our learned descriptors to quickly localize a suitable data set when running a new simulation. This makes our approach very efficient, and resolution independent. We will demonstrate with several examples that our method yields volumes with very high effective resolutions, and non-dissipative small scale details that naturally integrate into the motions of the underlying flow.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; *Procedural animation*;

Additional Key Words and Phrases: fluid simulation, low-dimensional feature descriptors, convolutional neural networks

This work is supported by the *ERC Starting Grant* 637014.

ACM Reference format:
Mengyu Chu and Nils Thuerey. 2017. Data-Driven Synthesis of Smoke Flows with CNN-based Feature Descriptors. *ACM Trans. Graph.* 36, 4, Article 69 (July 2017), 14 pages.
DOI: http://dx.doi.org/10.1145/3072959.3073643

1 INTRODUCTION

Resolving the vast amount of detail of natural smoke clouds is a long standing challenge for fluid simulations in computer graphics. Representing this detail typically requires very fine spatial resolutions, which result in costly simulation runs, and in turn cause painfully long turn-around times. A variety of powerful methods have been developed to alleviate this fundamental problem: e.g., improving the accuracy of the advection step [Kim et al. 2005; Selle et al. 2008], post-processing animations with additional synthetic turbulence [Kim et al. 2008; Narain et al. 2008], and speeding up the pressure projection step [Ando et al. 2015; Lentine et al. 2010].

We take a different perspective to efficiently realize high resolution flows: we propose to use a large collection of pre-computed space-time regions, from which we synthesize new high-resolution volumes. In order to very efficiently find the best match from this repository, we propose to use novel, flow-aware feature descriptor. We will ensure that L_2 distances in this feature space will correspond to real matches of flow regions in terms of fluid density as well as

flow motion, so that we can very efficiently retrieve entries even for huge libraries of flow data sets.

Closely related to the goal of efficient high-resolution flows is the fight against numerical viscosity. This is a very tough problem, as the discretization errors arising in practical flow settings cannot be quantified with closed form expressions. On human scales the viscosity of water and air is close to zero, and the effects of numerical viscosity quickly lead to unnaturally viscous motions for practical resolutions. While we do not aim for a method that directly reduces numerical viscosity, we will show that it is possible to predict it's influence for typical smoke simulations in graphics. We do this in a data-driven fashion, i.e., we train a model to establish correspondences between simulations with differing amounts of numerical viscosity.

In our method, the calculation of descriptors and encoding the effects of discretization errors are handled by a convolutional neural network (CNN). These networks were shown to be powerful tools to predict the similarity of image pairs [Mobahi et al. 2009; Zagoruyko and Komodakis 2015], e.g., to compute optical flow or depth estimates. We leverage the regressive capabilities of these CNNs to train networks that learn to encode very small, yet expressive flow descriptors. These descriptors will encode correspondences in the face of numerical approximation errors, and at the same time allow for very efficient retrievals of suitable space-time data-sets from a repository.

We compute these correspondences and repository look-ups for localized regions in the flow. We will call these regions *patches* in the following, and we employ a deforming Lagrangian frame to track the patches over time. Compared with recording data from static or rigid regions, this has the advantage that small features are pre-computed and stored in the repository, and do not inadvertently dissipate. In this way, we also side-step the strict time step restrictions that fine spatial discretization usually impose. On the other hand, we have to make sure the regions do not become too ill-shaped over time. For this, we propose a new deformation-limiting advection scheme with an anticipation step. Motivated by the fractal nature of turbulence, and the pre-dominantly uni-directional energy transfer towards small scales in Richardson's energy cascade [Pope 2000], we match and track each patch independently. This results in a very efficient method, as it allows us to perform all patch-based computations in parallel.

To summarize, the contributions of our work are:

- a novel CNN-based approach for computing robust, low-dimensional feature descriptors for density and velocity,
- a deformation limiting patch advection with anticipation,
- an algorithm for efficient volumetric synthesis based on reusable space-time regions.

In combination, our contributions make it possible to very efficiently synthesize highly detailed flow volumes with the help of a re-usable space-time flow repository. At the same time, we will provide an evaluation of convolutional neural networks and similarity learning in the context of density-based flow data sets.

2 RELATED WORK

Fluid simulations for animation purposes typically leave out the viscosity term of the *Navier-Stokes* (NS) equations, and solve the inviscid *Euler* equations: $\mathrm{D}\mathbf{u}/\mathrm{D}t = -\nabla p + \mathbf{g}$, where $\mathbf{u}$, p and $\mathbf{g}$ denote velocity, pressure, and acceleration due to external forces, respectively. These simulations have a long history in Graphics [Kass and Miller 1990], but especially for larger resolutions, computing a pressure field to make the flow divergence-free can become a bottleneck. Different methods to speed up the necessary calculations have been proposed, e.g., multi-grid solvers [McAdams et al. 2010], coarse projections [Lentine et al. 2010], or lower-dimensional approximations [Ando et al. 2015]. Another crucial part of Eulerian solvers is computing the transport in the grid. Here, unconditionally stable methods are widely used, especially the semi-Lagrangian method [Stam 1999] and its more accurate versions [Kim et al. 2005; Selle et al. 2008]. As we focus on single-phase flows, we restrict the following discussion to corresponding works. For an overview of fluid simulations in computer graphics we recommend the book by R. Bridson [Bridson 2015].

While algorithms for reducing algorithmic complexity are vital for fast solvers, the choice of data-structures is likewise important. Among others, recent works have proposed ways to handle large-scale particle-based surfaces [Akinci et al. 2012], or highly efficient tree structures [Museth 2013].

Several works have also investigated viscosity for fluid animations, e.g., by proposing stable and accurate methods for discretization [Batty and Bridson 2008], efficient representations of viscous sheets [Batty et al. 2012], or illustrating the importance of viscosity for resolving shear flows near obstacles in the flow [Zhang et al. 2016]. While these are highly interesting extensions, most practical fluid solvers for computer animation omit solving for viscosity in order to reduce the computational work.

Vortex-based methods aim for better preserving the swirling motions of flows, which are typically highly important for detail and realism. Methods to amplify existing vorticity have been proposed [Fedkiw et al. 2001; Selle et al. 2005], while other works model boundary layer effects [Pfaff et al. 2009; Zhu et al. 2010] or anisotropic vortices [Pfaff et al. 2010]. However, the amount of representable details is still inherently limited by the chosen grid resolutions for these algorithms. Hybrid methods have likewise been proposed to couple Eulerian simulations with Lagrangian vorticity simulations [Golas et al. 2012], while other researchers have proposed extrapolations of flow motions [Zhang and Ma 2013]. Our approach instead off-loads the computational burden to produce small-scale details to a pre-computation stage.

So-called *up-res* techniques, which increase the apparent resolution of a flow field, are another popular class of algorithms to simulate detailed flows at moderate computational cost. Different variations have been proposed for grid-based simulations [Kim et al. 2008; Narain et al. 2008], for mesh-based buoyancy effects [Pfaff et al. 2012] or for particle-based simulation [Mercier et al. 2015]. While these works represent highly useful algorithms to generate detailed animations, all small-scale features of a flow still have to be resolved and advected at simulation time. This causes significant amounts of computational work. The complexity of the advection step with

one of the semi-Lagrangian methods is linear in the number of unknowns. Thus, decreasing the cell size from Δx to $\Delta x/2$ results in eight times more spatial degrees of freedom. In addition, we typically have to reduce the time step size accordingly to prevent time integration errors from dominating the solution. This means there we face a roughly 16 times higher workload to compute the motion of advected quantities, such as the smoke densities. In contrast, our method fully decouples the fine spatial scales with the help of CNN-based descriptors. Thus our approach works with pre-computed space-time regions of *real* flow data, instead of resorting to procedural models. As such, we side-step the steep increase in complexity resulting from very fine spatial discretization. Instead, our method purely works with low-resolution fluid data and low-dimension descriptors during simulation time. The only high resolution operation is the density synthesis, which can be performed on a per frame basis at render time, in a trivially parallel fashion.

Our method shares similarities with previous approaches for synthesizing textures for animated flows. While early works focused on the problem of tracking texture data on liquid surfaces [Bargteil et al. 2006; Kwatra et al. 2007], a variety of interesting algorithms to synthesize two-dimensional fluid textures has been developed over the years [Kwatra et al. 2005; Narain et al. 2007], with various additions and improvements [Jamriška et al. 2015]. These texture synthesis approaches were extended to work with flow velocities by Ma et al. [Ma et al. 2009]. Notably, this is the only work performing the synthesis in three dimensions, as the cost for synthesizing 3D volumes is typically significant even for moderate sizes. Unlike texture synthesis, we do not directly generate a high-resolution output, but rather focus on efficiently and accurately retrieving appropriate data sets from a large repository of pre-computed data. In addition, our machine learning approach gives us the freedom to encode both similarity and physical properties in the look-up.

The *stamping* approach used in movie productions similarly reuses smaller regions of previous simulations [Wrenninge and Zafar 2011]. However, this approach typically rigidly moves the stamps, and does not align their content with the simulation. We found that a controllable deformation is a crucial component to make the Lagrangian representation of the patches work. A similar idea was explored previously for animating two-dimensional flows with frequency controlled textures [Yu et al. 2010]. While their algorithm uses a measure of the amount of deformation to blend in undeformed content, we explicitly compute a new deformed state for our patches that takes into account both the flow transport and undesirable deformations. This leads to increased life time of the patch regions, and reduces blending operations correspondingly.

Machine learning with neural networks has been highly successful for a variety of challenging computer vision problems. In particular, convolutional neural networks (CNNs) are particularly popular [Krizhevsky et al. 2012; Simonyan and Zisserman 2014], and several papers from this active area of research have targeted image similarity, e.g., to compute depth maps [Zagoruyko and Komodakis 2015], or one-shot image classification [Koch et al. 2015]. The first networks using a shared branch structure (so-called *Siamese*) were proposed by Bromley et al. [Bromley et al. 1993], while learning descriptors with L_2 distances was employed for, e.g., descriptors of human faces [Chopra et al. 2005]. In this context, we will employ a variant of the popular hinge loss function [Cao et al. 2006; Mobahi et al. 2009], which represents the best convex approximation of a binary loss. While similarity of image pairs has been studied before, our aim is to work with density and velocity functions of fluid simulations. We will demonstrate that it is beneficial to pay special attention to the velocity representation for learning.

Very few works so far exist that combine machine learning algorithms with animating fluids. An exception is the regression-forest-based approach for smoothed particle hydrodynamics [Ladicky et al. 2015]. This algorithm computes accelerations for a Lagrangian NS solver based on carefully designed input features, with the goal to enable faster liquid simulations. Our approach, on the other hand, aims for automatically extracting the best possible set of discriminative features. We demonstrate that neural networks can perform this task very well, and that we can in turn use the descriptors to establish correspondences between coarse and fine flow regions. Two other works share our goal to employ neural networks for single-phase flows: one focusing on learning the pressure projection step with a CNN [Tompson et al. 2016], and another one learning local updates to remove divergence [Yang et al. 2016]. Especially the former of the two uses a deep convolutional network structure that is similar to ours. However, their approach focuses on encoding the full pressure field of a flow simulation with a neural network. In contrast, our networks learn robust descriptors for smaller regions of the flow. As such, our method can be easily used with arbitrary resolutions. Both methods above work with the full output resolution, and do not target the fine spatial resolutions we aim for with our method.

3 FLOW SIMILARITY

Given two flow simulations of the same effect, one being a coarse approximation, the other one being a more accurate version, e.g., based on a finer spatial discretization, and a spatial region Ω, our goal is to compute a similarity score s for the two inputs. We consider functions F_c and F_f of the coarse and the fine flow, respectively, where F could be a scalar value such as smoke density, or alternatively could also include the velocity, i.e., $F \in \mathbb{R}^3 \rightarrow \mathbb{R}^4$. We will revisit which flow variables to include in F in Section 4.5, but for now we can assume without loss of generality that F is a scalar function. In order to compute similarity, we need to extract enough information from a region of the flow such that s can infer similarity from it. We will sample the flow functions in a regular grid within Ω, assuming that F is sufficiently smooth to be represented by point samples. All point samples from this grid are then combined into an input vector $\mathbf{x}_c$ and $\mathbf{x}_f$ for coarse and fine simulations, respectively.

Given these inputs, we aim for computing $s(\mathbf{x}_c, \mathbf{x}_f)$ for Ω such that s approaches zero if the pair is actually one that corresponds to the same phenomenon being represented on the coarse and fine scales. For increasing dissimilarity of the flows, s should increase. This dissimilarity can, e.g., result from considering different regions Ω in the fine and coarse simulations, or when the two are offset in time.

A first guess would be use an L_2 distance to compute s as $\int_\Omega \|\mathbf{x}_f - \mathbf{x}_c\|^2 d\mathbf{x}$. This turns out to be a sub-optimal choice, as even small translations can quickly lead to undesirably large distance values.

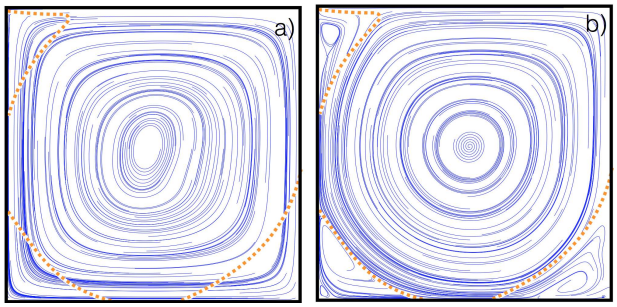


Fig. 2. Stream lines of a lid driven cavity simulation without (left) and with viscosity (right) is shown. The orange lines indicate the correct position of the center vortex [Erturk et al. 2005]. In this case, the graphics approach commonly used to leave out viscosity leads to a very different vortex shape.

Worse, in the presence of numerical viscosity, different resolutions for F_c and F_f can quickly lead to significantly different velocity and density content even when they should represent the same fluid flow. Fig. 2 illustrates how strongly viscosity can influence the outcome of a simulation. Numerical viscosity is typically comprised of errors throughout all parts of the solver that influence velocity (although the advection step is arguably the largest contributor). For practical algorithms, no closed form solution exists to detect or quantify these errors. Instead of manually trying to find heuristics or approximations of how these numerical errors might propagate and influence solutions, we transfer this task to a machine learning algorithm.

In addition, our goal is not only to measure the distance between two inputs, but rather, given a new coarse input, we want to find the best match from a large collection of pre-computed data sets. Thus we map the correspondence problem to a feature space, such that distances computed with a simple distance metric, e.g. Euclidean distance, corresponds to the desired similarity s. We will use a non-linear neural network to learn discriminative feature descriptors $\mathbf{d}(\mathbf{x}) \in \mathbb{R}^m$, with m as small as possible. Given a coarse flow region F_c we can then retrieve the best match from a set of fine regions F_f by minimizing $||\mathbf{d}(\mathbf{x}_f) - \mathbf{d}(\mathbf{x}_c)||^2$.

In the following, we will first describe our approach for learning flow descriptors with CNNs. Afterwards we will explain our deformation-aware patch tracking, which represents an important component in order to achieve invariance with respect to large-scale motions. Finally we explain how to synthesize high-resolution volumes for rendering.

4 LEARNING FLOW SIMILARITY

As our approach focuses on computing distances between flow inputs with convolutional neural networks, we will briefly review the most important details below. On a high level, we can view neural networks as a general methodology to approximate arbitrary functions f, where we consider typical regression problems of the form $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$, i.e., given an input $\mathbf{x}$ we want to approximate the result $\mathbf{y}$ as closely as possible given a representation f based on the parameters $\mathbf{w}$. Thus, for our machine learning problems, the components of $\mathbf{w}$ are the degrees of freedom we wish to compute.

In the following, we choose neural networks (NNs) to represent f, for which a thorough overview can be found, e.g., in the book by C. Bishop [Bishop 2007].

NNs are represented by networks of nodes, which, in contrast to our understanding of nature, are usually modeled with continuous functions. A key ingredient are so-called activation functions g, that introduce non-linearity and allow NNs to approximate arbitrary functions. For a layer of nodes L in the network, the output of the i'th node $y_{i,L}$ is computed with

$$y_{i,L} = g\Big(\sum_{j=0}^{n_{L-1}} w_{ij,L-1}\, y_{j,L-1} \Big) . \tag{1}$$

Here, n_L = denotes number of nodes in layer L, and, without loss of generality we assume $y_{0,L} = 1$. This constant input for each node models a bias term, which is crucial to shift the output of the activation function. We employ this commonly used formulation to merge all degrees of freedom in $\mathbf{w}$. Thus, in the following, we will not distinguish between regular weights and biases. We can rewrite Eq. (1) using a weight matrix W as $\mathbf{y}_L = \mathbf{g}(W_{L-1}\mathbf{y}_{L-1})$. In this equation, $\mathbf{g}$ is applied component-wisely to the input vector. Note that without $\mathbf{g}$ we could fold a whole network with its multiple layers into a single matrix W_0, i.e., $\mathbf{y} = W_0\mathbf{x}$, which could only be used to compute linearized approximations. Thus, in practice $\mathbf{g}$ is crucial to approximate generic, non-linear functions. Typical choices for $\mathbf{g}$ are hyperbolic tangent or sigmoid functions. For the learning process, the network is typically given a loss function $l(\mathbf{y}, f(\mathbf{x}, \mathbf{w}))$, that calculates the quality of the generated output with respect to $\mathbf{y}$. The loss function needs to be differentiable such that it's gradient can be back-propagated into the network, in order to update the weights.

A key component driving many of the recent deep-learning successes are so called *convolutional layers*. These layers can be used to the exploit spatial structure of the input data, and essentially learn filter kernels for convolutions. In the neural network area, these filters are often called *feature maps*, as they effectively detect consistent features of the inputs that are important to infer the desired output. This concept is especially useful for data such as sound (1D), or images (2D), and directly extends to volumetric data sets such as those from fluid simulations. The feature maps typically have only a small set of non-zero weights, e.g., 5×5 or 3×3 weights. As the inputs consist of vector quantities, e.g., RGB values for images, neural networks typically employ stacks of feature maps. Hence, we can think of the feature maps as higher-order tensors. For volumetric inputs with three spatial dimensions storing a vector-valued function, we use fourth-order tensors to represent a single feature map for a convolutional layer. Applying this feature map yields a scalar 3D function. Most practical networks learn multiple feature maps at once per layer, and thus generate a vectorial value for the next layer, whose dimension equals the number of feature maps of the previous layer.

These convolutional layers are commonly combined with *pooling* layers which reduce the size of a layer by applying a function, such as the maximum, over a small spatial region. Effectively, this down-samples the spatially arranged output of a layer, in order to decrease the dimensionality of the problem for the next layer

[Krizhevsky et al. 2012]. Note that convolutional neural networks in principle cannot do more than networks consisting only of fully connected layers. However, the latter usually have a significantly larger number of degrees of freedom. This makes them considerably more difficult to train, while convolutional networks lead to very reasonable network sizes and training times, making them very attractive in practice. Smaller networks also have greatly reduced requirements for the amounts of input data, and can be beneficial for regularization [Bishop 2007].

Next we will explain the challenges in our setting when choosing a suitable loss function for the neural networks, and outline the details of our network architectures afterwards.

4.1 Loss Functions

When computing our flow similarity metric, a first learning approach could be formulated as $s = f_s(\mathbf{x}_1, \mathbf{x}_2, \mathbf{w})$, where f_s is again the learned function represented by $\mathbf{w}$, $\mathbf{x}_1$ and $\mathbf{x}_2$ represent a pair of input features extracted from two simulations, and s is the output indicating how similar the input pair is. As the inputs for our regression problems stem from a chaotic process, i.e. turbulent flow, the inputs look "noisy" from a regression standpoint, and the training dataset is usually not linearly separable. It is crucial that the learning process not only encodes a notion such as Euclidean distance of the two inputs, but also learns to use the whole space represented by its network structure to compute the similarity. At the same time, the network needs to reliably detect dissimilar pairs. Hence, the availability of negative pairs is crucial for establishing robust similarity between true positives.

The basic problem of learning similarity could be stated as follows: given a pair of inputs, generate a label $y \in \{1, -1\}$, i.e., the pair is similar (1), or not (−1). While a naive L_2 loss to learn exactly these labels is clearly insufficient, a slightly improved loss function could be formulated as $l_n = -y f(\mathbf{x}_1, \mathbf{x}_2, \mathbf{w})$. In this case, the network would be rewarded to push apart positive and negative pairs, but due to the lack of any limit, the learned values would diverge to plus and minus infinity [Chapelle et al. 2006]. Instead, it is crucial to have a loss function that does not unnecessarily constrain the regressor, and at the same time gives it the freedom to push correctly classified pairs apart as much as necessary. The established loss function in this setting is the so called *hinge* loss, which can be computed with:

$$l_h(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} \max(0, 1 - f_s(\mathbf{x}_1, \mathbf{x}_2, \mathbf{w})) , & y = 1 \\ \max(0, 1 + f_s(\mathbf{x}_1, \mathbf{x}_2, \mathbf{w})) , & y = -1 \end{cases} \tag{2}$$

This loss function typically leads to significant improvements over the naive loss functions outlined above. When using a NN representation in conjunction with the loss function of Eq. (2), a feature descriptor can be extracted by using the outputs of the last fully connected layer as a descriptor [Zagoruyko and Komodakis 2015].

While this approach works, we will demonstrate that it is even better to embed the L_2 distance of the descriptors directly into the hinge loss [Mobahi et al. 2009]. As we later on base our repository lookups on these distances, it is important to guide the NN towards encoding discriminative distances based on the feature descriptors themselves, instead of only optimizing for a final similarity score s. In order to do this, we can re-formulate the learning problem to generate the descriptor itself, using the descriptor distance as the "dissimilarity".

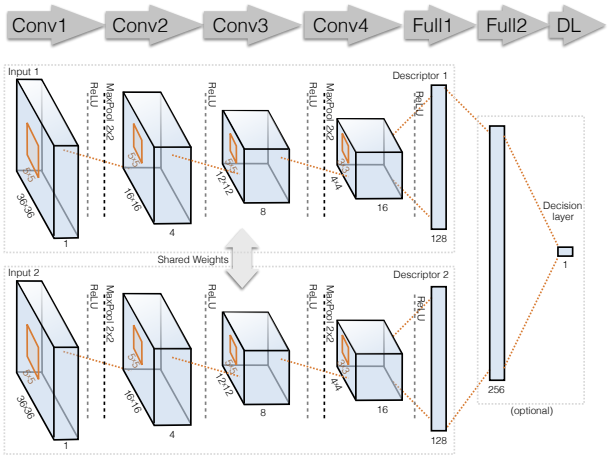


Fig. 3. Our CNN architecture with two convolution stacks with shared weights, followed by a feature and an optional decision layer.

In the following we will denote the outputs of a specific *descriptor* layer of our network with $\mathbf{d}_w(\mathbf{x})$, where $\mathbf{x}$ is the input for which to compute the descriptor. Based on these descriptors we change the regression problem to $f_e(\mathbf{x}_1, \mathbf{x}_2, \mathbf{w}) = \beta - \alpha ||\mathbf{d}_w(\mathbf{x}_1) - \mathbf{d}_w(\mathbf{x}_2)||, \alpha > 0$. Here we have introduced the parameters α and β to fine tune the onset and steepness of the function. Using f_e instead of f_s in Eq. (2) yields

$$l_e(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} \max(0, \alpha_p + ||\mathbf{d}_w(\mathbf{x}_1) - \mathbf{d}_w(\mathbf{x}_2)||) , & y = 1 \\ \max(0, \alpha_n - ||\mathbf{d}_w(\mathbf{x}_1) - \mathbf{d}_w(\mathbf{x}_2)||) , & y = -1 , \end{cases} \tag{3}$$

where we have replaced the α, β parameters by $\alpha_{p,n}$ which can be used to fine tune the margins individually for positive and negative pairs, as we will discuss below. Note that we normalize the descriptors $\mathbf{d}_w$ when extracting them for Eq. (3). This significantly improves convergence, and supports learning distributions of components, rather than absolute values in the descriptor. We will demonstrate that this loss function clearly outperforms the other alternatives, after describing the details of our neural networks to be used in conjunction with this loss function.

4.2 CNN Architecture

Our neural networks consist of a typical stack of convolution layers which translate spatially arranged data into an increasing number of feature signals with lower spatial resolution. A visual summary of the network is given in Fig. 3. As our network compares two inputs, it initially has two branches which contain a duplicated stack of convolutional layers with shared weights. Each branch acts separately on one input vector to reduce its dimensionality. The outputs of the last convolutional layer of each branch (*Conv3* in our case) are concatenated into a serial vector containing the two feature descriptors (layer *Full1*). For networks that regress a single similarity score with loss function like Eq. (2), we add another fully connected layer (*Full2*) and an output layer with a single output node that computes the final similarity score. However, these two layers are optional. When training with the hinge embedding loss, Eq. (3), we omit these two layers.

We will use the following abbreviations to specify the network structure: convolutional layer (CL), max pooling layer (MP), and fully connected layer (FC). We start with inputs of size 36x36 in 2D. The input from a low-resolution simulation is linearly up-scaled to this resolution (typically we use a four times lower resolution for the coarse simulation). One convolutional branch of our network yields 2^2 points with 32 features each, i.e., 128 values in total. These 128 outputs are concatenated into the final feature descriptor layer $\mathbf{d}_w$ with normalization. For three dimensional inputs, we extend the spatial dimension in each layer correspondingly. Hence, the first layer has a resolution of 5x5x5x4 in 3D, and the final spatial resolution is 2^3 with 32 features. Thus in 3D, our feature descriptor D_w has dimension 256.

4.3 Data Generation

For machine learning approaches it is crucial to have good training data sets. Specifically, in our setting the challenge is to create a controlled environment for simulations with differing discretizations, and resulting in different numerical viscosities. Without special care, the coarse and fine versions will deviate over time, and due to their non-linearity, initial small differences will quickly lead to completely different flows. In the following, we consider flow similarity within a chosen time horizon t_r. Note that this parameter depends on the setup under consideration, e.g., for very smooth and slow motions, larger values are suitable, while violent and fast motions mean flows diverge faster. We will discuss the implications of choosing t_r in more detail below.

We use randomized initial condition to create our training data. Given an initial condition, we set up two parallel simulations, one with a coarse resolution of r_c cells per axis, and we typically use a four times higher resolution $r_f = 4r_c$ for the fine version. While it would be possible to simply run a large number of simulations for a time t_r, we found that it is preferable to instead synchronize the simulation in intervals of length t_r. Here, we give priority to the high resolution, assuming that it's lower numerical viscosity is closer to the true solution. We thus re-initialize the coarse simulation in intervals of length t_r with a low-pass filtered version of the fine simulation.

This synchronization leads to a variety of interesting and diverse flow configurations over time, which we would otherwise have to recreate manually with different initial conditions. For our data generation, we found buoyant flows to be problematic in rectangular domains due to their rising motion. Using tall domains would of course work, but typically wastes a significant amount of space. Instead, we compute a center of mass for the smoke densities during each time step. We then add a correction vector during the semi-Lagrange advection for all quantities to relocate the center of mass to the grid center. During the data generation runs, we seed patches throughout the volume, and track them with the same algorithm we use for synthesis later on. Thus, we also employ our deformation limiting there, which we describe in detail in Section 5.1. For each patch region, we record the full coarse and fine velocity / density functions within each deforming patch region for each time step. Currently, we track the patches with the fine simulation, and use the same spatial region in the coarse simulation.

The recorded pairs of spatial regions for one time step give us the set of positive pairs for training. Note that coarse and fine data in these regions may have diverged up to the duration t_r. To create negative pairs, we assign a random fine data set to each coarse input. Two features from a negative pair are either recorded by different patches, or recorded by the same patch, but in different time steps. Therefore, for any coarse feature example $\mathbf{x}_1$ in our training and evaluation dataset, there is only one fine feature example $\mathbf{x}_2$ marked as relevant.

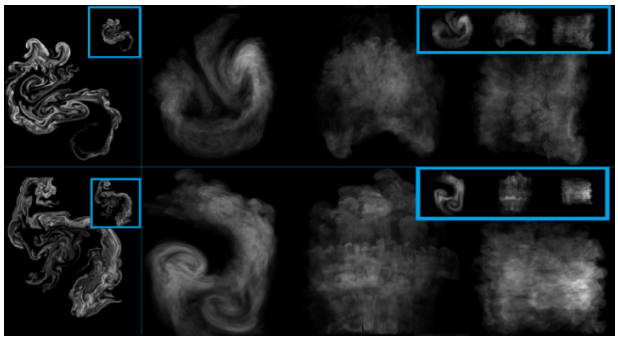


Fig. 4. Examples of our data generation for training, in both 2D (left) and 3D (front, left and top views). The coarse simulation (with blue outline) is synchronized with the high resolution data in intervals t_r.

In this way, we have created several combined simulations in both 2D and 3D, with $t_r = 20$ and $t_r = 40$, to generate training datasets as well as evaluation datasets. These t_r are selected so that the resulting training data have a maximum discriminative capabilities. For smaller intervals, the network presumably only sees very similar inputs, and hence cannot generate expressive descriptors. When the interval becomes too large, inputs can become too dissimilar for the NN. In general, t_r is negatively correlated to the time step, kinetic energy and resolution difference. We currently select the t_r manually through comparisons, and an automatic calculation of t_r remains an interesting future direction.

Several images of our data generation in 2D and 3D can be found in Fig. 4. The detailed simulations have a 4 times higher resolution. For training, we generated 18,449 positive pairs for 2D and 16,033 pairs in total for 3D. In 2D, every training batch contains 1:1 positive and negative pairs. The latter ones are randomly generated from all positive ones while training. While a ratio of 1:1 was sufficient in 2D, training with this ratio turned out to be slow in 3D. We found that increasing the number of negative pairs improves the learning efficiency, while having negligible influence on the converged state of the CNNs. Thus, we use a ratio of 1:7 for 3D training runs. For our evaluations below, the datasets with $t_r = 20$ and $t_r = 40$ give very similar results. Since the latter one shows a clearer differences between the methods, in Section 4.4, we will focus our evaluations on the dataset with $t_r = 40$, which has 5440 and 5449 positive pairs in 2D and 3D respectively.

4.4 Evaluation

In order to evaluate and compare the success of the different approaches it would be straight forward to compute descriptors with

a chosen method for a coarse input i, and then find the best match from a large collection of fine pairs. If the best match is the one originally corresponding to i, we can count this as a success, and failure otherwise. In this way, we can easily compute a percentage of successfully retrieved pairs. However, this metric would not represent our application setting well. Our goal is to employ the descriptors for patches in new simulations, that don't have a perfectly corresponding one in the repository. For these we want to robustly retrieve the closest available match. Thus, rather than counting the perfect matches, we want to evaluate how reliably our networks have learned to encode the similarity of the flow in the descriptor space. To quantify this reliability, we will in the following measure the true positive rate, which is typically called *recall*, over the cut-off rank k.

The recall over a cut-off rank is commonly employed in the information retrieval field to evaluate ranked retrieval results [Manning et al. 2008]. Recall stands for the percentage of correctly retrieved data sets over all given related ones. The rank in this case indicates the number of nearest neighbors that are retrieved from the repository for a given input. In particular, for our evaluation dataset with N pairs, with a given cut-off k, we evaluate the recall for all N coarse features, and thus kN pairs are retrieved in total per evaluation. In these retrieved pairs, if r pairs are correctly labeled as related, the recall at cut-off k would be r/N. In such a case, a perfect method, would yield 100% for the recall at $k = 1$, and then be constant for larger k. In practice, methods will slowly approach 100% for increasing k, and even the worst methods will achieve a recall of 100% for $k = N$. Thus, especially a first range of small k values is interesting to evaluate how reliably a method has managed to group similar data in the Euclidean space of the descriptor values.

We first compare two CNN-based descriptors created with the two loss functions explained above, and the popular, hand-crafted HOG descriptor in 2D. The latter is a commonly employed, and very successful feature descriptor based on histograms of gradients of a local region. As can be seen in Fig. 5, the HOG descriptor fares worst in this setting. Beyond a rank of 6, it's recall is clearly below that of the regular hinge loss CNN descriptor. The hinge embedded loss function yields the best descriptor, which in this case is consistently more than 10% better from rank 10 on. The high recall rates show that our CNN successfully learns to extract discriminative features, and can do so with a higher accuracy than conventional descriptors.

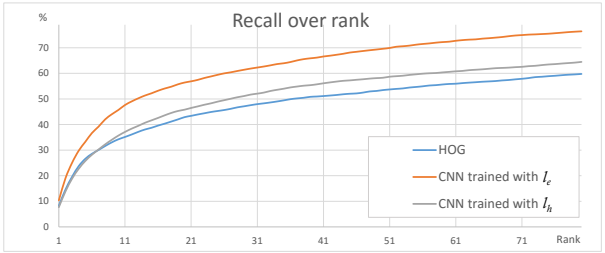
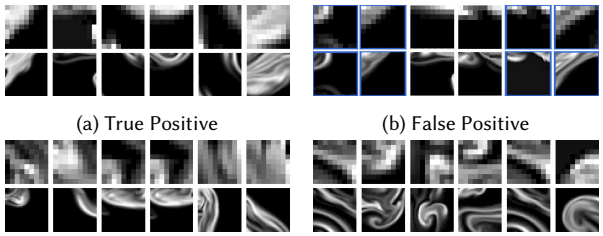


Fig. 5. Recall over rank for HOG (blue), CNN trained with l_h on similarity output node (gray), and CNN trained with l_e on descriptors directly (orange).

We also investigate the influence of the threshold α_p and α_n in the loss function l_e in Eq. (3). As the parametrization $[0.0, 0.7]$ has a slightly higher accuracy among others, we will use these parameters in the following. We found that it is not necessary to have any margin on the positive side of the loss function l_e, but on the negative side, a relatively large margin gives our CNN the ability to better learn the dissimilarities of pairs.

Fig. 6 shows some of the top ranking true and false correspondences made by our CNN for the smoke density pairs. Correct positive and negative pairs are shown on the left. False negative pairs are related ones, for which the CNN descriptors still have a large difference, while the false positive ones are mistakenly matched pairs which were not related. In practice, the false negative pair have no effects on the synthesized results, in contrast to the false positives. However, we notice that these false positives typically contain visually very similar data. As such, these data sets will be unlikely to introduce visual artifacts in the final volume. Some of these false positives actually stem from the same tracked patch region during data generation, and were only classified wrongly in terms of their matched time distance. These pairs are marked with blue borders in Fig. 6(b).



(a) True Positive (b) False Positive

(c) True Negative (d) False Negative

Fig. 6. Top-ranking density pairs matched by our CNN.

4.5 Descriptors for Flow Motions

Up to now we have only considered smoke densities for establishing correspondences, however, in the fluid simulation context we also have velocity information. The velocities strongly determine the smoke motion over time, and as such they are likewise important for making correspondences between the data of a new simulation and the data-sets in the repository.

To arrive at a method that also takes the flow motion into account, we use two networks: one is trained specifically for density descriptors, while we train the second one specifically for the motion. This yields two descriptors, $\mathbf{d}_{den}$ and $\mathbf{d}_{mot}$, which we concatenate into a single descriptor vector for our repository lookups with

$$\mathbf{d}(\mathbf{x}) = \frac{1}{\sqrt{1 + w_m^2}} \begin{bmatrix} \mathbf{d}_{den}(\mathbf{x}) \\ w_m \mathbf{d}_{mot}(\mathbf{x}) \end{bmatrix}. \tag{4}$$

Note that the separate calculation of density and motion descriptor mean that we can easily re-scale the two halves to put more emphasis on one or the other. Especially when synthesizing new simulations results, we put more emphasis on the density content with $w_m = 0.6$.

For the motion descriptor, we use the vorticity as input, i.e., $\omega = \nabla \times \mathbf{u}$. During the synthesis step, motion descriptors generated from vorticity offer significantly better look-ups than the ones trained

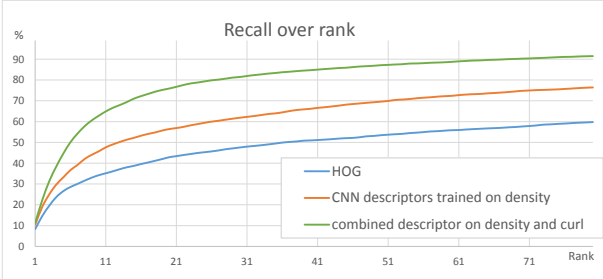


Fig. 7. Using curl descriptors as well as density descriptors improves matching performance.

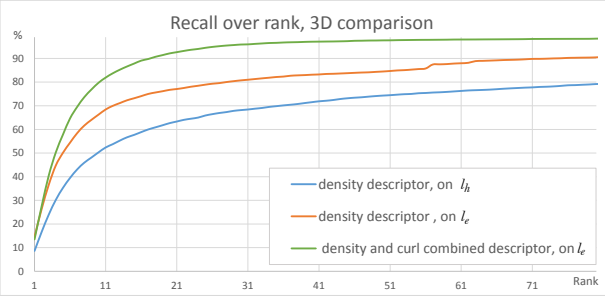


Fig. 8. In 3D, using curl descriptors as well as density descriptors improves matching performance.

with **u**, as the vorticity better reflects local changes in the flow field. Due to our scale separation with patch motion and content, our goal is to represent local, relative motions with our descriptors, instead of, e.g., large scale translations or stretching effects. Using a combined density and curl descriptor improves the recall rate even further. A comparison with our 2D data set is shown in Fig. 7, e.g., at rank 11, the recall improves by ca. 35%, and we see similar gains in three dimensions (shown in Fig. 8). Note that these two figures use a weight of $w_m = 1.0$ for the curl descriptor.

Due to the aforementioned improvements in matching accuracy, this approach represents our final method. In the following we will use a double network, one trained for densities and a second one trained for the curl to compute our descriptors.

4.6 Direct Density Synthesis

Seeing the generative capabilities of modern neural networks, we found it interesting to explore how far these networks could be pushed in the context of high-resolution flows. Instead of aiming for the calculation of a low-dimensional descriptor, the NN can also be given the task to directly regress a high resolution patch of smoke densities. An established network structure for this goal is a stack of convolutional layers to reduce the input region to a smaller set of feature response functions, which then drive the generation of the output with stack of convolution-transpose layers [Krizhevsky et al. 2012].

We ran an extensive series of tests, and the best results we could achieve for a direct density synthesis are shown in Fig. 9. In this

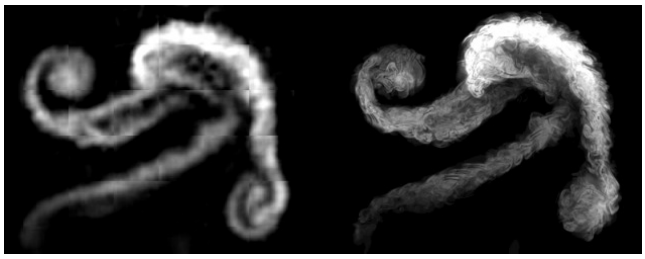


Fig. 9. Directly trying to synthesize densities with CNNs yields blurred results that lack structures (left). Our algorithm computes highly detailed flows for the same input (right).

case, the network receives a region of 16x16 density values, and produces outputs of four times higher resolution (64x64) with the help of two convolutional layers, a fully connected layer, and four deconvolutional layers. While we could ensure convergence of the networks without overfitting, and relatively good temporal stability, the synthesized densities lack any detailed structures. This lack of detail arises despite the fact that this network has a significantly larger number of weights than our descriptor network, and had more training data at its disposal.

There is clearly no proof that it is impossible to synthesize detailed smoke densities with generative neural networks, but we believe that our tests illustrate that the chaotic details of turbulent smoke flows represent an extremely challenging task for NNs. Especially when trying to avoid overfitting with a sufficiently large number of inputs, the turbulent motions seem like noise to the networks. As a consequence, they learn an *averaged* behavior that smoothes out detailed structures. These results also motivate our approach: we side-step these problems by learning to encode the distance between flow regions, and supplying best matched details in our flow repository at render time, instead of learning and generating details directly.

5 MOTION AND SYNTHESIS

For each patch, we track its motion with a local grid. We call these grids *cages* in the following, to distinguish them from the Cartesian grids of the fluid simulations, and we will denote the number of subdivision per spatial axis with n_{cage}, with a resulting cell size Δx_{cage}. Below, we describe our approach to limit excessive deformation of these cages.

5.1 Deformation-limiting Motion

Even simple flows exhibit large amounts of rotational motion, that quickly lead to very undesirable stretching and inversion artifacts. An example can be seen on the left side of Fig. 10. Thus, when advecting the Lagrangian patch regions through a velocity field, we are facing the challenge to avoid overly strong distortions while making the patch follow the pre-scribed motion. Inspired by previous work on *as-rigid-as-possible* deformations [Igarashi et al. 2005; Sorkine et al. 2004], we express our Lagrangian cages in terms of differential coordinates, and minimize an energy functional in the least-squares sense to retrieve a configuration that limits deformation while adhering to the flow motion.

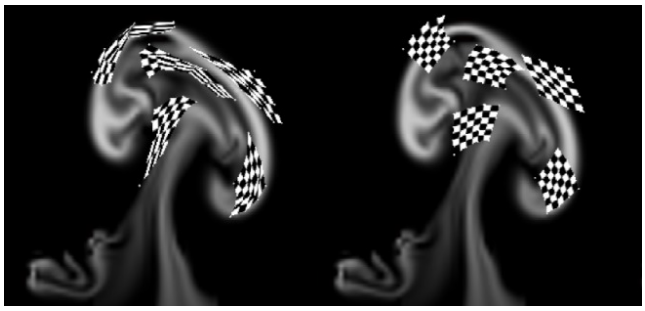


Fig. 10. Naive patch advection (left) quickly leads to distorted regions. Our deformation limiting advection (right) can keep stable regions while following the fluid motion.

For the differential coordinates, we span an imaginary tetrahedron between an arbitrary vertex $\mathbf{v}_3$, and its three neighboring vertices $\mathbf{v}_{0,1,2}$, as shown in Fig. 11. For the undeformed state of a cell, the position of $\mathbf{v}_3$ can be expressed with rotations of the tetrahedron edges as

$$\mathbf{v}_3 = \mathbf{v}_c + (R_{\mathbf{e}_{1,0}}\mathbf{e}_{p,2} + R_{\mathbf{e}_{2,1}}\mathbf{e}_{p,0} + R_{\mathbf{e}_{0,2}}\mathbf{e}_{p,1})/3\sqrt{2}\,. \tag{5}$$

Here $\mathbf{e}_{i,j}$ denotes the edge between points i and j, and $R_\mathbf{v}$ is the the 3x3 rotation matrix that rotates by 90 degrees around axis $\mathbf{v}$. $\mathbf{v}_c$ is the geometric center of the triangle spanned the three neighbors, i.e., $\mathbf{v}_c = (\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2)/3$.

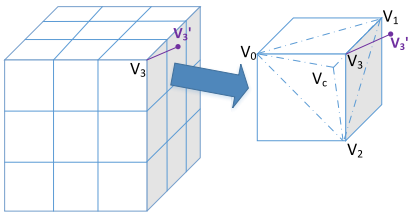


Fig. 11. An example of patch cage with $n = 3$. The deformation error is accumulated from every cell, as shown on the right. The distance between a target position v_3 and the advected position v_3' yields the deformation error. v_3 is expressed in terms of $v_{0,1,2}$ and their center point v_c.

We can rewrite Eq. (5) as $\mathbf{v}_3 = A\mathbf{v}_0 + B\mathbf{v}_1 + C\mathbf{v}_2$, where

$$\begin{aligned} A &= I/3 + (R_{\mathbf{e}_{0,2}} + R_{\mathbf{e}_{1,0}} - 2R_{\mathbf{e}_{2,1}})/2\sqrt{2} \\ B &= I/3 + (R_{\mathbf{e}_{1,0}} + R_{\mathbf{e}_{2,1}} - 2R_{\mathbf{e}_{0,2}})/2\sqrt{2} \\ C &= I/3 + (R_{\mathbf{e}_{2,1}} + R_{\mathbf{e}_{0,2}} - 2R_{\mathbf{e}_{1,0}})/2\sqrt{2}\,. \end{aligned}$$

For a new position of $\mathbf{v}_3'$, e.g. later during a simulation run, we can measure the squared error with

$$E_{\{\mathbf{v}_3'\}} = \|A\mathbf{v}_0' + B\mathbf{v}_1' + C\mathbf{v}_2' - \mathbf{v}_3'\|^2. \tag{6}$$

Correspondingly, we can compute an overall deformation error for the whole cage with $m = (n+1)^3$ new positions $\mathbf{v}'$ with

$$E_{defo}(\mathbf{v}') = \frac{1}{n^3}\sum_{i=0}^{n^3}\sum_{j=0}^{8} E_{\{\mathbf{v}_{ij}'\}}/(\frac{1}{n})^2 = \frac{\mathbf{v}'^T G\mathbf{v}'}{n}\,. \tag{7}$$

For a whole patch cage with n^3 cells, we accumulate the deformation energy for the eight corners in each cell. The energy for a single vertex is given by $E_{\{\mathbf{v}_{ij}'\}}$ above, where i and j are the index of the cell and its corner respectively. The right side of Eq. (7) is a shortened notation, where G is a $3m \times 3m$ matrix containing the accumulated contributions for all points of a cage. Since every vertex has at most 6 connected neighbors, every row vector in G has at most 19 entries, corresponding to the x, y and z positions of its neighbors, and a diagonal entry for itself. Minimizing this quadratic form directly will lead to a trival solution of zero, so it is necessary to solve this problem with suitable constrains. In practice we want the solution to respect the advected positions. For this we add an additional advection error $\|\mathbf{v} - \mathbf{v}'\|^2$ that pulls the vertices towards the advected positions, i.e., $\mathbf{v}'$. Thus, the total energy we minimize is:

$$E(\mathbf{v}) = \lambda_0 \frac{\mathbf{v}^T G\mathbf{v}}{n} + \frac{1}{m}\sum \|\mathbf{v} - \mathbf{v}'\|^2\,, \tag{8}$$

where $\mathbf{v}'$ is the advected position, $m = (n+1)^3$ is the number of vertices in the grid, and λ_0 controls the balance between advection and deformation. Note that in the original formulation, $R_\mathbf{v}$, and thus also G, are expressed in terms of $\mathbf{v}$, making the whole problem non-linear. Under the assumption that the advected coordinates do not deform too strongly within a single step, which we found to be a valid assumption even for the large CFL numbers used in graphics, we linearize the optimization problem by computing G' using $\mathbf{v}'$ in Eq. (5). The full minimization problem is now given by

$$\frac{\partial E(\mathbf{v})}{\partial \mathbf{v}} = 0 \;\approx\; 2[(\lambda G' + I)\mathbf{v} - \mathbf{v}'],\quad \lambda = \frac{m}{n}\lambda_0\,, \tag{9}$$

where we have introduced a scaling factor $\frac{m}{n}$, that makes the system independent of the chosen cage resolution. We compute the final position of a patch cage by solving the linear system $\mathbf{v} = (\lambda G' + I)^{-1}\mathbf{v}'$. Note that this system of equations is relatively small, with $3m$ unknowns per patch. Hence, it can be solved very efficiently with a few steps of a conjugate gradient solver, independently for all patches.

As our goal is to track large scale motions with our patches, we have to respect the different spatial scales merged in the flow field. To reduce the effects of small scale perturbations in the flow, we advect the patches with a low-pass filtered version of the velocity, where the filter is chosen according to the cage cell size Δx_{cage}.

Anticipation. To prevent abrupt changes of densities, we fade patches in and out over a time interval t_f for rendering (see below). For our examples, we use a t_f of 40 time steps. Unfortunately, this temporal fading means that when patches are fully visible, they are typically already strongly deformed due to the swirling flow motions. We can circumvent this problem by letting the patches *anticipate* the flow motion. I.e., for a new patch at time t, we backtrack its motion and deformation to the previous time $t - t_f$. This leads to completely undeformed patch configurations exactly at the point in time when they are fully visible.

Initialization. When seeding a new, undeformed patch at a given position, we found that having axis-aligned cages is not the best option. Inspired by classic image feature descriptors, we initialize

the orientations of our cages according to the gradient of the density field. Specifically, we calculate gradient histograms in the cage region Ω, and use the top ranked directions as main directions. The solid angle bins of 3D gradient histograms can be defined using meridians and parallels [Scovanner et al. 2007]. We evenly divide azimuth θ_i and polar angle ϕ_j with step sizes $\Delta\theta = \Delta\phi = \pi/n_b$, resulting in $2n_b$ and n_b subdivisions for the azimuth and the polar angle, respectively. 3D vectors are then specified as (r, θ, ϕ), where r denotes the radius. The unit sphere is divided into a set of bins $\{\mathbf{b}_{ij}\}, 0 \le i < 2n_b, 0 \le j < n_b$, where $\mathbf{b}_{ij} = (1, \theta_i - \Delta\theta/2, \phi_j - \Delta\phi/2)$ denotes the normalized central direction of the bin in the spherical coordinate system. Based on these bins, histograms are calculated as:

$$h_{ij} = \frac{1}{A_{ij}} \sum_{o\in\Omega} w\left(\left|\frac{\phi_o-\phi_j}{\Delta\phi} + \frac{1}{2}\right|\right) w\left(\left|\frac{\theta_o-\theta_i}{\Delta\theta} + \frac{1}{2}\right|\right) r_o G\left((\mathbf{x}-\mathbf{o})^2\right) \quad (10)$$

where $\mathbf{o}$ is the position of a sample point in region Ω with density gradient (r_o, θ_o, ϕ_o), $w(d) = \max(0, 1-d)$, A_{ij} represents the solid angle of $\mathbf{b}_{ij}$, and G denotes a Gaussian kernel.

At the position $\mathbf{x}$ of a new patch, we compute the gradient histogram for the smoke density d_s as outlined above. The histogram has a subdivision of $n_b = 16$, and the region Ω is defined as a 9^3 grid around $\mathbf{x}$. We choose the main direction of the patch as $\mathbf{b}_k$, where k denotes the histogram bin with $k = \arg\max_{i,j}(h_{ij})$. For the secondary direction, we recompute the histogram with gradient vectors in the tangent plane. Thus, instead of ∇d_s we use $(\nabla d_s - (\nabla d_s \cdot \mathbf{b}_x)\mathbf{b}_x)$. The initial orientation of the patch is then defined in terms of $(\mathbf{b}_x, \mathbf{b}_y, \mathbf{b}_x \times \mathbf{b}_y)$. This "gradient-aware" initialization narrows down the potential descriptor space, which simplifies the learning problem, and leads to more robust descriptors.

Discussion. While as-rigid-as-possible deformations have widely been used for geometric processing task, our results show that they are also highly useful to track fluid regions while limiting deformation. In contrast to typical mesh deformation tasks, we do not have handles as constraints, but instead an additional penalty term that keeps the deformed configuration close to the one pre-scribed by the flow motion. The direct comparison between a regular advection, and our cage deformations is shown in Fig. 10. It is obvious that a direct advection is unusable in practice, while our deformation limiting successfully lets the cages follow the flow, while preventing undesirable deformations.

The anticipation step above induces a certain storage overhead, but we found that it greatly reduces the overall deformation, and hence increase the quality of the synthesized densities. We found the induced memory and storage overheads to be negligible in practice.

5.2 Synthesis and Rendering

In the following, we will outline our two-pass synthesis algorithm, as well as the steps necessary for generating the final volumes for rendering. A pseudo-code summary of the synthesis step is given in Algorithm 1.

In the forward pass for flow synthesis, new patches are seeded and advected step by step, and are finally faded out. To seed new patches, a random seeding grid with spacing $s_p/2$ is used, $s_p = n\Delta x_{\text{cage}}$ being the size of a patch. In addition, we make use of a patch weighting grid w_s. It uses the native resolution of the simulation, and acts as a threshold to avoid new patches being seeded too closely to existed ones. w_s accumulates the spatial weights of patches, i.e., spherical kernels centered at the centroids of each patch cage with a radius of $s_p/2$. A linear falloff is applied for the last two-thirds of its radius, ramping from one to zero. At simulation time, we typically accumulate the patch weights in w_s without applying the patch deformations. This is in contrast to render time, where we deform the patch kernels. The high-resolution weight grid with deformed patch weights for rendering will be denoted w_r to distinguish it from the low-resolution version w_s. As w_s is only used for thresholding the creation of new particles, we found that using un-deformed kernels gives very good results with reduced runtimes.

New patches will not be introduced at a sampling position $\mathbf{x_n}$ unless $w_s(\mathbf{x}_n) < 0.5$. In practice, this means the distance to the closest patch is larger than $s_p/3$. For each newly assigned patch, we compute its initial gradient-aligned frame of reference with Eq. (10), calculate the CNN inputs at this location, and let both CNNs generate the feature descriptors. Based on the descriptors, we look-up the closest matches from our repository with a pre-computed

ALGORITHM 1: Pseudo-code for our algorithm.

```
Flow quantities:
density grid d, velocity grid u, weight grid w_s, patch cages c_j, scalar weights w_j.

Pre-computation:
for t = 0 to t_end,e do
    Run exemplar simulation, update d_e, u_e
    PatchAdvection(u_e)
    SeedNewPatches(w_s)
    Compute CNN descriptors d from d_e, u_e
    Save d, and high-resolution d_e in patch regions to disk
    Accumulate patch spacial weights in w_s
end

Runtime synthesis:
G = load descriptors from repository
// forward pass:
for t = 0 to t_end,t do
    Run source simulation, update d_s, u_s
    PatchAdvection(u_s, c)
    SeedNewPatches(w_s)
    Compute CNN descriptors d from d_s, u_s
    for j = 1 to v_patches do
        if Patch j unassigned then
            Find closest descriptor to d_j in G
        end
        else
            Update and check quality of d_j
            LimitDeformation(c_j)
            Update patch fading weights w_j (fade out)
            Store j, c_j, w_j for frame t
        end
    end
    Store patches at time t , d_s, u_s
    Accumulate patch spacial weights in w_s
end
// backward pass:
for t = t_end,t to 0 do
    Load d_s, u_s , patches at time t
    for j = 1 to v_patches do
        if Anticipation active for Patch j then
            PatchAdvection(−u_s, c)
            Store j, c_j, w_j for frame t
            Update w_j (fade in)
        end
    end
end
```

kd-tree. For successfully matched patches, our deformation limiting advection is performed over their lifetime. The maximal lifetimes are determined by the data set lengths of matched repository patches, which are typically around 100 frames. We remove ill-suited patches, whose re-evaluated descriptor distance is too large for the current flow configuration, or whose deformation energy in Eq. (7) exceeds a threshold. In practice, we found a threshold of $0.15 s_p^2$ to work well for all our examples. After the forward pass, matched patches anticipate the motion of the target simulation in a backward pass. Here we move backwards through time, and advect all newly created patches backward over the course of the fade in interval. Finally, for each frame we store the coarse simulation densities, as well as patch vertex positions $\mathbf{v}$, together with their repository IDs and scalar, temporal fading weights. This data is all that is necessary for the rendering stage.

During synthesis the deformation limiting patch advection effectively yields the large motions which conform to the input flow, while small scale motion is automatically retrieved from the repository frame by frame when we render an image. Note that we only work with the low-dimensional feature descriptors when synthesizing a new simulation result, none of the high-resolution data is required at this stage.

At render time we synthesize the final high-resolution volume. To prepare this volume, we also need to consider spatial transitions between the patches amongst each other, and transitions from the patch data to the original simulation. For this, we again accumulate the deformed spherical patch kernels into a new grid w_r with the final rendering resolution, to spatially weight the density data.

We then map the accumulate the weighted high-resolution content of a patch data set in the high resolution rendering volume. As our repository contains densities that are normalized to [0..1], and our descriptor is invariant to scaling and offset transformations, we map the repository content to the min-max range of the densities in the target region. To blend the contributions of overlapping patches, we normalize the accumulated high resolution density by w_r. Additionally, we use a blurred version of the original coarse densities as a mask. We noticed that patches can sometimes drift outside of the main volume while moving with the flow. The density mask effectively prevents patches from contributing densities away from the original volume.

6 RESULTS AND DISCUSSION

We will now demonstrate that our approach can efficiently synthesize high-resolution volumes for a variety of different flows. All runtimes in the following are given averages per frame. Typically, we run a single time step per frame of animation.

As a first example we have simulated a simple rising plume example, shown in Fig. 12. Note that gravity acts along the x-axis in our setup. In this case, the resolution of the original simulation was $108 \times 60 \times 60$, and on average, 388 patches were active over the course of the simulation. Our approach is able to synthesize a large amount of clearly-defined detail to the input flow that does not dissipate over time. Details of this simulation setup, as well as for the other examples, can be found in Table 1.

A second example is shown in Fig. 15. Here we add an additional obstacle, which diverts the flow. For the patch motion, we simply extend flow velocities into the obstacles, and add a slight correction along the gradient of the obstacle's signed distance function if a patch vertex inadvertently ends up inside the obstacle. Our deformation limiting advection smoothly guides the patches around the obstacle region.

A different flow configuration with colliding jets of smoke is shown in Fig. 16. For this setup, on average 486 patches were active. Note that our patches contain 72^3 cells in this case. Thus, the effective resolution for this simulation was ca $560 \times 360 \times 360$ cells.

Table 1. Details of our animation setups and repository data generation.

Scenes	Fig.	Base res.	base patch res.	Avg. patches	Time
PlumeX	Fig. 12	$108 \times 60 \times 60$	15^3	388	5.3s
Obstacle	Fig. 15	$76 \times 64 \times 64$	12^3	362	3.9s
Jets	Fig. 16	$90 \times 60 \times 60$	12^3	486	4.0s
Comp. L_2	Fig. 14	$50 \times 80 \times 50$	9^3	682	2.23s
Comp. Wlt	Fig. 17	$50 \times 80 \times 50$	9^3	647	2.42s

	Res.	Patch res.	Patch no.	Patch storage
Repo.	440^3	72^3	14894	5.1GB densities, 30 MB descriptors

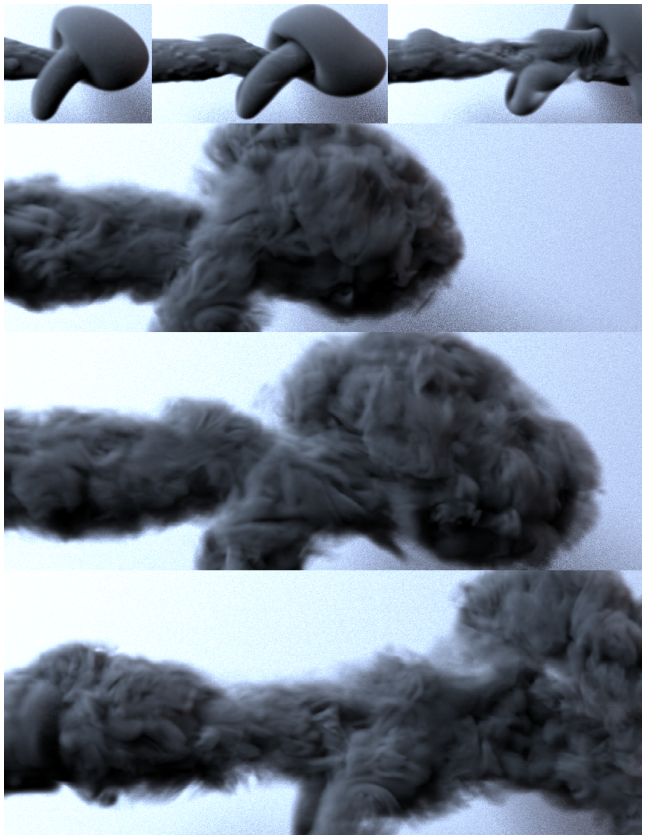


Fig. 12. We apply our method to a horizontal plume simulation. The top three images show the input, the bottom three the high-resolution densities generated with our method. On average, 388 patches were active per timestep.

The whole simulation took only 4.0s/frame, which is very efficient given the high effective resolution.

For the three examples above, we use w_s accumulated from deformed patch kernels. By considering the deformation, w_s better represents the coverage of the patches. However, since we limit deformations, we found that using undeformed kernels generates equivalent visual results with reduced calculation times. Besides, there is still significant room left for accelerating our implementation further. E.g., we run the fluid solve on the CPU, and we only use GPUs for the CNN descriptor calculations.

Evaluations: The CNN descriptors not only increase the recall rate, but also improve the visual quality by retrieving patches that better adhere to the input flow. This can be seen in Fig. 13, where our result is shown on the right, while the left hand side uses our full pipeline with a simplified distance calculation, i.e., without the use of a CNN. Instead, we use an L_2 distance of down-sampled versions of d_s and the curl of $\mathbf{u}_s$. These values are normalized and used as descriptors directly. Specifically, we down-sampled d_s to a resolution of 7^3, and the curl to 5^3, resulting in a combined descriptor with $(343 + 375)$ entries. This is similar to the size of our CNN-based descriptors. Since CNN descriptors have a good understanding of correlation between different fluid resolutions, they offer results with small-scale vortices and vivid structures that fit the target well, while the simple descriptors sometimes offer plain and noisy structures (the blue regions in Fig. 13). Additionally, the simple descriptors can introduce un-plausible motions, which becomes apparent in the regions marked in purple in Fig. 13. There, we know from theory that the baroclinic contribution to vorticity should be along the cross product of density and pressure gradient. Thus, the vertical structures caused by the simple descriptors are not plausible for the buoyancy driven input simulation.

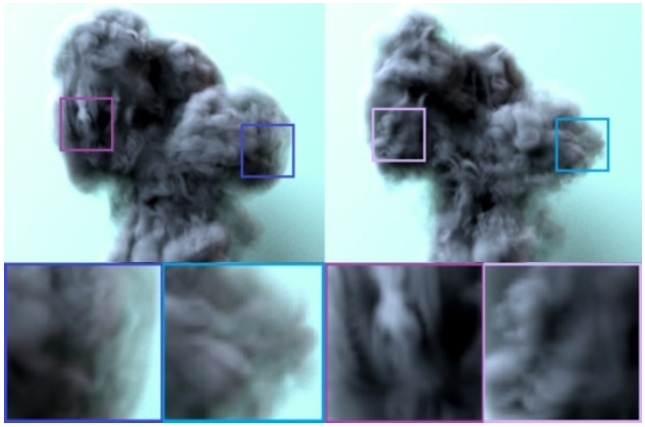


Fig. 13. Using our algorithm with simple descriptors (left) can result in overly regular structures (blue rectangles) and sub-optimal matches (purple rectangles). Chances for such patch assignments are reduced with the CNN-based descriptors (right).

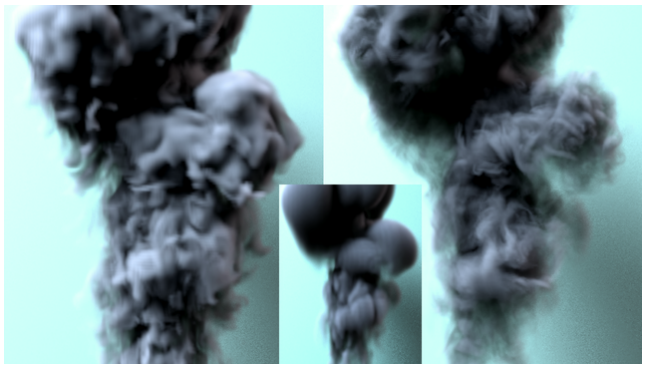


Fig. 14. Based on a simulation with $50 \times 80 \times 50$ cells (center), the wavelet turbulence method (left) generates volumes with $150 \times 240 \times 150$ cells with 2.75s/frame. Our method (right) yields effective resolutions of $400 \times 640 \times 400$ with 2.23s/frame.

Based on the setup from Fig. 13, we further compare our approach to the wavelet turbulence method [Kim et al. 2008] as a representative from the class of up-res methods. In order to make the approaches comparable, we consider their performance given a limited and equivalent computational budget. With this budget the wavelet turbulence method produces results with a 3 times higher resolution, consuming 2.75s/frame. This is close to the 2.23s/frame our method requires. However, our method effectively yields an 8 times higher resolution (see Table 1). For our simulation, this setup used the w_s field with undeformed kernel evaluations. Apart from the difference in detail, the wavelet turbulence version exhibits a noticeable deviation from the input flow in the lower part of the volume. Here numerical diffusion accumulates to cause significant drift over time, while our method continues to closely conform to the input.

Finally, we compare our method to a regular simulation with doubled resolution. As expected, this version results in a different large scale motion, and in order to compare the outputs, we applied our CNN based synthesis method to a down-sampled version of the high resolution simulation. While the regular high resolution scene spends 2.5s/frame, our method takes only 2.42s, but offers fine details, as shown in Fig. 17.

Limitations and Discussion: One limitation of our approach is that we cannot guarantee fully divergence-free motions on small scales. For larger scales, our outputs conform to the original, divergence-free motion. The small scale motions contained in repository patches are likewise recorded from fully divergence-free flows, but as our patches deform slightly, the resulting motions are not guaranteed to be divergence-free. Additionally, spatial blending can introduce regions with divergent motions. Our algorithm shares this behavior with other synthesis approaches, e.g., texture synthesis. However, as we do not need to compute an advection step based on these motions, our method avoids accmuluating mass losses (or gains) over time.

There are many avenues for smaller improvements of our neural network approach, e.g., applying techniques such as batch normalization, or specialized techniques for constructing the training set. However, we believe that our current approach already demonstrates that deforming Lagrangian patch regions are an excellent basis for CNNs in conjunction with fluid flows. It effectively makes our learning approach invariant to large scale motions. Removing

Fig. 15. Our method applied to the flow around a cylindrical obstacle. The resolution of the underlying simulation is only $76 \times 64 \times 64$.

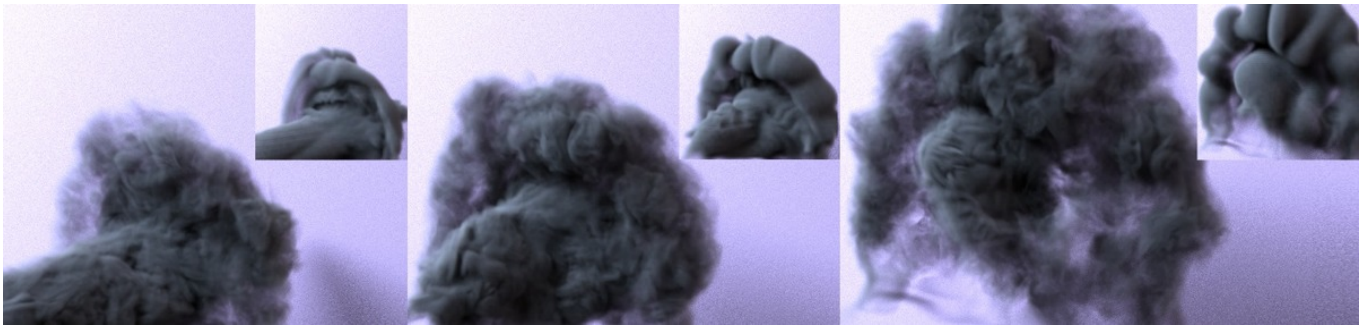


Fig. 16. Two colliding jets of smoke simulated with our approach. The whole simulation including descriptor calculations, look-up, and patch advection took only 4.0 seconds per frame on average.

these invariants for machine learning problems is an important topic, as mentioned e.g. by Ladicky et al. [2015]. Apart from motion invariance, we arrive at an algorithm that can easily applied to any source resolution. For other CNN-based approaches this is typically very difficult to achieve, as networks are specifically trained for a fixed input and output size [Tompson et al. 2016].

Due to its data-driven nature, our method requires more hard-disk space than procedural methods. As shown in Table 1, the density data of the patches in our 3D repository takes up ca. 5.1GB of disk space. Fortunately, we only load the descriptors at simulation time: ca. 15MB for density descriptors, and another 15MB for curl descriptors. At render time, we have to load ca. 400 data sets per frame, i.e. ca. 137MB in total.

Another consequence of our machine-learning approach is that our network is specifically adapted to a set of algorithmic choices. Thus, for a different solver, it will be advantageous to re-train the network with a suitable data set and adapted interval t_r. It will be an interesting area of future work whether a sufficiently deep CNN can learn to compute descriptors for larger classes of solvers.

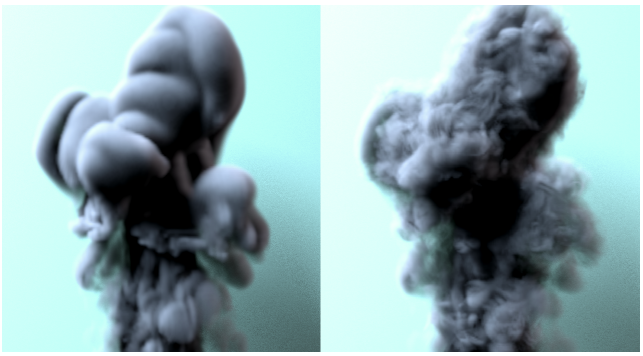


Fig. 17. In order to compare a full simulation of $100 \times 160 \times 100$ (left) with our approach, we downsample the full simulation to $50 \times 80 \times 50$, and then apply our algorithm (right). The latter spends 2.42s/frame, while the full simulation requires 2.51s/frame.

7 CONCLUSIONS

We have presented a novel CNN-based method to realize a practical data-driven workflow for smoke simulations. Our approach is the first one to use a flow repository of space-time datasets to synthesize high-resolution results with only a few seconds of runtime per frame. At the same time, our work represents a first demonstration of the usefulness of descriptor learning in the context of fluids flows, and we have shown that it lets us establish correspondences between different simulations in the presence of numerical viscosity. As our approach is a data-driven one, it can be used for any choice of Navier-Stokes solver, as long as enough input data is available. Additionally, the localized descriptors make our approach independent of the simulation resolution.

We believe the direction of data-driven flow synthesis is a very promising area for computer graphics applications. Art-directable solvers that are at the same time fast and stable are in high demand. In the future, we believe it will be very interesting to extend our ideas towards stylization of flows, and towards synthesizing not only an advected quantity such as smoke density, but the flow velocity itself.

ACKNOWLEDGMENTS

The authors would like to thank Marie-Lena Eckert for helping with generating the video, and Daniel Hook for his work on the direct density synthesis with CNNs.

REFERENCES

Gizem Akinci, Markus Ihmsen, Nadir Akinci, and Matthias Teschner. 2012. Parallel Surface Reconstruction for Particle-Based Fluids. *Computer Graphics Forum* 31, 6 (2012), 1797–1809.

Ryoichi Ando, Nils Thürey, and Chris Wojtan. 2015. A Dimension-reduced Pressure Solver for Liquid Simulations. *Computer Graphics Forum* 34, 2 (2015), 10.

Adam Bargteil, Tolga Goktekin, James O'brien, and John Strain. 2006. A semi-Lagrangian contouring method for fluid simulation. *ACM Transactions on Graphics* 25, 1 (2006), 19–38.

Christopher Batty and Robert Bridson. 2008. Accurate viscous free surfaces for buckling, coiling, and rotating liquids. In *Proc. Symposium on Computer Animation*. ACM/Eurographics, 219–228.

Christopher Batty, Andres Uribe, Basile Audoly, and Eitan Grinspun. 2012. Discrete viscous sheets. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 113.

C Bishop. 2007. *Pattern Recognition and Machine Learning*. Springer.

Robert Bridson. 2015. *Fluid Simulation for Computer Graphics*. CRC Press.

Jane Bromley, James W Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. 1993. Signature verification using a "Siamese" time delay neural network. *Int. J. of Pattern Recognition and Artificial Intelligence* 7, 04 (1993), 669–688.

Yunbo Cao, Jun Xu, Tie-Yan Liu, Hang Li, Yalou Huang, and Hsiao-Wuen Hon. 2006. Adapting ranking SVM to document retrieval. In *Proc. Research and Development in Information Retrieval*. ACM, 186–193.

O Chapelle, B Schölkopf, and A Zien. 2006. *Semi-supervised learning, ser. Adaptive computation and machine learning*. Cambridge, MA: The MIT Press.

Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *Conference on Computer Vision and Pattern Recognition*, Vol. 1. IEEE, 539–546.

Ercan Erturk, Thomas C Corke, and Cihan Gökçöl. 2005. Numerical solutions of 2-D steady incompressible driven cavity flow at high Reynolds numbers. *International Journal for Numerical Methods in Fluids* 48, 7 (2005), 747–774.

Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. 2001. Visual Simulation of Smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. ACM, New York, NY, USA, 15–22. DOI: https://doi.org/10.1145/383259.383260

Abhinav Golas, Rahul Narain, Jason Sewall, Pavel Krajcevski, Pradeep Dubey, and Ming Lin. 2012. Large-scale Fluid Simulation Using Velocity-vorticity Domain Decomposition. *ACM Trans. Graph.* 31, 6, Article 148 (Nov. 2012), 9 pages.

Takeo Igarashi, Tomer Moscovich, and John F. Hughes. 2005. As-rigid-as-possible Shape Manipulation. *ACM Trans. Graph.* 24, 3 (July 2005), 1134–1141. DOI: https://doi.org/10.1145/1073204.1073323

Ondrej Jamriška, Jakub Fišer, Paul Asente, Jingwan Lu, Eli Shechtman, and Daniel Sỳkora. 2015. LazyFluids: appearance transfer for fluid animations. *ACM Transactions on Graphics* 34, 4 (2015), 92.

M. Kass and G. Miller. 1990. Rapid, Stable Fluid Dynamics for Computer Graphics. *ACM Transactions on Graphics* 24, 4 (1990), 49–55.

Byungmoon Kim, Yingjie Liu, Ignacio Llamas, and Jarek Rossignac. 2005. FlowFixer: Using BFECC for Fluid Simulation. In *Proc. Conference on Natural Phenomena*. Eurographics, 51–56.

Theodore Kim, Nils Thürey, Doug James, and Markus Gross. 2008. Wavelet Turbulence for Fluid Simulation. *ACM Transactions on Graphics* 27 (3) (2008), 50:1–6.

Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. 2015. Siamese Neural Networks for One-Shot Image Recognition. In *Conference on Computer Vision and Pattern Recognition*, Vol. 37. IEEE.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. NIPS, 1097–1105.

Vivek Kwatra, David Adalsteinsson, Theodore Kim, Nipun Kwatra, Mark Carlson, and Ming C Lin. 2007. Texturing fluids. *Visualization and Computer Graphics* 13, 5 (2007), 939–952.

Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. 2005. Texture optimization for example-based synthesis. *ACM Transactions on Graphics* 24, 3 (2005), 795–802.

Lubor Ladicky, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. 2015. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 199.

Michael Lentine, Wen Zheng, and Ronald Fedkiw. 2010. A novel algorithm for incompressible flow using only a coarse grid projection. *ACM Transactions on Graphics* 29, 4 (2010), 114.

Chongyang Ma, Li-Yi Wei, Baining Guo, and Kun Zhou. 2009. Motion field texture synthesis. *ACM Transactions on Graphics (TOG)* 28, 5 (2009), 110.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

A. McAdams, E. Sifakis, and J. Teran. 2010. A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids. In *Proceedings of Symposium on Computer Animation*. ACM/Eurographics, 65–74.

Olivier Mercier, Cynthia Beauchemin, Nils Thuerey, Theodore Kim, and Derek Nowrouzezahrai. 2015. Surface Turbulence for Particle-based Liquid Simulations. *ACM Trans. Graph.* 34, 6, Article 202 (Oct. 2015), 10 pages. DOI: https://doi.org/10.1145/2816795.2818115

Hossein Mobahi, Ronan Collobert, and Jason Weston. 2009. Deep learning from temporal coherence in video. In *International Conference on Machine Learning*. ACM, 737–744.

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics* 32, 3 (2013), 27.

Rahul Narain, Vivek Kwatra, Huai-Ping Lee, Theodore Kim, Mark Carlson, and Ming C Lin. 2007. Feature-guided dynamic texture synthesis on continuous flows. In *Proc. Rendering Techniques*. Eurographics, 361–370.

Rahul Narain, Jason Sewall, Mark Carlson, and Ming C. Lin. 2008. Fast Animation of Turbulence Using Energy Transport and Procedural Synthesis. *ACM Transactions on Graphics* 27, 5 (2008), article 166.

T. Pfaff, N. Thuerey, J. Cohen, S. Tariq, and M. Gross. 2010. Scalable fluid simulation using anisotropic turbulence particles. In *ACM Transactions on Graphics*, Vol. 29. ACM, 174.

Tobias Pfaff, Nils Thuerey, and Markus Gross. 2012. Lagrangian Vortex Sheets for Animating Fluids. *ACM Transactions on Graphics* 31, 4 (2012), 112.

Tobias Pfaff, Nils Thuerey, Andrew Selle, and Markus Gross. 2009. Synthetic Turbulence Using Artificial Boundary Layers. *ACM Trans. Graph.* 28, 5, Article 121 (Dec. 2009), 10 pages. DOI: https://doi.org/10.1145/1618452.1618467

Stephen B. Pope. 2000. *Turbulent Flows*. Cambridge University Press.

Paul Scovanner, Saad Ali, and Mubarak Shah. 2007. A 3-dimensional Sift Descriptor and Its Application to Action Recognition. In *Proceedings of the 15th ACM International Conference on Multimedia (MM '07)*. 357–360.

Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An Unconditionally Stable MacCormack Method. *J. Sci. Comput.* 35, 2-3 (June 2008), 350–371.

Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. 2005. A vortex particle method for smoke, water and explosions. In *ACM Transactions on Graphics (TOG)*, Vol. 24. ACM, 910–914.

Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. 2004. Laplacian Surface Editing. In *Proc. Symposium on Geometry Processing*. ACM/Eurographics, 175–184.

Jos Stam. 1999. Stable Fluids. In *Proc. SIGGRAPH*. ACM, 121–128.

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. 2016. Accelerating Eulerian Fluid Simulation With Convolutional Networks. *arXiv preprint arXiv:1607.03597* (2016).

Magnus Wrenninge and Nafees Bin Zafar. 2011. Production Volume Rendering. ACM SIGGRAPH Course notes. (July 2011).

Cheng Yang, Xubo Yang, and Xiangyun Xiao. 2016. Data-driven projection method in fluid simulation. *Computer Animation and Virtual Worlds* 27, 3-4 (2016), 415–424.

Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. 2010. Lagrangian Texture Advection: Preserving both Spectrum and Velocity Field. *IEEE Transactions on Visualization and Computer Graphics* 17(11) (2010), 1612–1623.

Sergey Zagoruyko and Nikos Komodakis. 2015. Learning to compare image patches via convolutional neural networks. In *Proc. Computer Vision and Pattern Recognition*. IEEE, 4353–4361.

Xinxin Zhang, Minchen Li, and Robert Bridson. 2016. Resolving Fluid Boundary Layers with Particle Strength Exchange and Weak Adaptivity. *ACM Transactions on Graphics* 35, 4 (2016), 76:1–76:8.

Yubo Zhang and Kwan-Liu Ma. 2013. Spatio-temporal Extrapolation for Fluid Animation. *ACM Trans. Graph.* 32, 6, Article 183 (Nov. 2013), 8 pages.

Bo Zhu, Xubo Yang, and Ye Fan. 2010. Creating and preserving vortical details in sph fluid. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 2207–2214.