SPNets: Differentiable Fluid Dynamics for Deep Neural Networks

Connor Schenck  
University of Washington, Nvidia Corporation  
schenckc@cs.washington.edu

Dieter Fox  
Nvidia Corporation, University of Washington  
dieterf@nvidia.com

Abstract: In this paper we introduce Smooth Particle Networks (SPNets), a framework for integrating fluid dynamics with deep networks. SPNets adds two new layers to the neural network toolbox: ConvSP and ConvSDF, which enable computing physical interactions with unordered particle sets. We use these layers in combination with standard neural network layers to directly implement fluid dynamics inside a deep network, where the parameters of the network are the fluid parameters themselves (e.g., viscosity, cohesion, etc.). Because SPNets are implemented as a neural network, the resulting fluid dynamics are fully differentiable. We then show how this can be successfully used to learn fluid parameters from data, perform liquid control tasks, and learn policies to manipulate liquids.

Keywords: Model Learning, Fluid Dynamics, Differentiable Physics

1 Introduction

From mixing dough to cleaning automobiles to pouring beer, liquids are an integral part of many everyday tasks. Humans have developed the skills to easily manipulate liquids in order to solve these tasks, however robots have yet to master them. While recent results in deep learning have shown a lot of progress in applying deep neural networks to challenging robotics tasks involving rigid objects [1–3], there has been relatively little work applying these techniques to liquids. One major obstacle to doing so is the highly unstructured nature of liquids, making it difficult to both interface the liquid state with a deep network and to learn about liquids completely from scratch.

In this paper we propose to combine the structure of analytical fluid dynamics models with the tools of deep neural networks to enable robots to interact with liquids. Specifically, we propose Smooth Particle Networks (SPNets), which adds two new layers, the ConvSP layer and the ConvSDF layer, to the deep learning toolbox. These layers allow networks to interface directly with unordered sets of particles. We then show how we can use these two new layers, along with standard layers, to directly implement fluid dynamics using Position Based Fluids (PBF) [4] inside the network, where the parameters of the network are the fluid parameters themselves (e.g., viscosity or cohesion). Because we implement fluid dynamics as a neural network, this allows us to compute full analytical gradients. We evaluate our fully differentiable fluid model in the form of a deep neural network on the tasks of learning fluid parameters from data, manipulating liquids, and learning a policy to manipulate liquids. In this paper we make the following contributions 1) a fluid dynamics model that can interface directly with neural networks and is fully differentiable, 2) a method for learning fluid parameters from data using this model, and 3) a method for using this model to manipulate liquid by specifying its target state rather than through auxiliary functions. In the following sections, we discuss related work, the PBF algorithm, SPNets, and our evaluations of our method.

2 Related Work

Liquid manipulation is an emerging area of robotics research. In recent years, there has been much research on robotic pouring [5–13]. There have also been several papers examining perception of liquids [14–19]. Some work has used simulators to either predict the effects of actions involving liquids [20], or to track and reason about real liquids [21]. However, all of these used either task specific models or coarse fluid dynamics, with the exception of [21], which used a liquid simulator, although it was not differentiable. Here we propose a fluid dynamics model that is fully differentiable and show how to use it solve several tasks.

One task we evaluate is learning fluid parameters (e.g., viscosity or cohesion) from data. Work by Elbrechter et al. [17] and Guevara et al. [22] focused on learning fluid parameters using actions to estimate differences between the model and the data. Other work has focused on learning fluid dynamics via hand-crafted features and regression forests [23], via latent-state physics models [24], or via conventional simulators combined with a deep net trained to solve the incompressibility constraints [25]. Both [24] and [25] use grid-based fluid representations, which allows them to use standard 3D convolutions to implement their deep learning models. In this paper, however, we use a particle-based fluid representation due to its greater efficiency for sparse fluids. In [23] the authors...
also use a particle-based representation, however they require hand-crafted features to allow their model to compute particle-particle interactions. Instead, we directly interface the particles with the model. While there have been several recent papers that develop methods for interfacing unordered point sets with deep networks [26–28], these methods focus on the task of object recognition, a task with significantly different computational properties than fluid dynamics. For that reason, we implement new layers for interfacing our model with unordered particle sets.

The standard method of solving the Navier-Stokes equations [29] for computing fluid dynamics using particles is Smoothed Particle Hydrodynamics (SPH) [30]. In this paper, however, we use Position Based Fluids (PBF) [4] which was developed as a counterpart to SPH. SPH computes fluid dynamics for compressible fluids (e.g., air); PBF computes fluid dynamics for incompressible fluids (e.g., water). Additionally, our model is differentiable with analytical gradients. There has been some work in robotics utilizing differentiable physics models [31] as well as differentiable rendering [32]. There has also been work on learning physics models using deep networks such as Interaction Networks [33, 34], which model the interactions between objects as relations, and thus are also fully differentiable. However, these works were primarily focused on simulating rigid body physics and physical forces such as gravity, magnetism, and springs. To the best of our knowledge, our model is the first fully differentiable particle-based fluid model.

3 Position Based Fluids

In this paper, we implement fluid dynamics using Position Based Fluids (PBF) [4]. PBF is a Lagrangian approximation of the Navier-Stokes equations for incompressible fluids [29]. That is, PBF uses a large collection of particles to represent incompressible fluids such as water, where each particle can be thought of as an individual “packet” of fluid. We chose a particle-based representation for our fluids rather than a grid-based (Eulerian) representation as for sparse fluids, particles have better resolution for fewer computational resources. We briefly describe PBF here and direct the reader to [4] for details.

Figure 1 shows a general outline of the PBF algorithm for a single timestep. First, at each timestep, external forces are applied to the particles (lines 2–3), then particles are moved to solve the algorithm for a single timestep. First, at each particle


density at each particle is computed as

\[ \rho_i = \sum_{j \in P \setminus \{i\}} n_{ij} (\rho_k + \omega) W_{\omega}(d_{ij}, h) \]

where \( n_{ij} \) is the normalized vector from particle \( j \) to particle \( i \), \( \omega_k \) is the pressure at particle \( k \), \( W_{\omega} \) is a kernel function (i.e., monotonically decreasing continuous function), \( d_{ij} \) is the distance from \( i \) to \( j \), and \( h \) is the cutoff for \( W_{\omega} \) (that is, for all particles further than \( h \) apart, \( W_{\omega} \) is 0). The pressure at each particle is computed as

\[ \omega_k = \lambda_\omega \max (\rho_k - \rho_0, 0) \]

where \( \lambda_\omega \) is the pressure constant, \( \rho_k \) is the density of the fluid at particle \( k \), and \( \rho_0 \) is the rest density of the fluid. Density at each particle is computed as

\[ \rho_k = \sum_{j \in P} m_j W_{\rho}(d_{kj}, h) \]
where \( m_j \) is the mass of particle \( j \). For \( W_\omega \) we use \( \frac{m_j}{\rho_j} (1 - \frac{d_{ij}}{h})^\frac{3}{2} \) and for \( W_\nu \) we use \( \frac{m_j}{\rho_j} (1 - \frac{d_{ij}}{h})^2 \), the same as used in [4]. The details for computing \textsc{SolveCohesion}, \textsc{SolveSurfaceTension}, and \textsc{ApplyViscosity} are described in the appendix.

To compute the next set of particle locations \( P' \) and velocities \( V' \) from the current set \( P, V \), these functions are applied as described in the equation in figure 1. For the experiments in this paper, the constants are empirically determined and we set \( h \) to 0.1.

4 Smooth Particle Networks

In this paper, we wish to implement Position Based Fluids (PBF) with a deep neural network. Current networks lack the functionality to interface with unordered sets of particles, so we propose two new layers. The first is the ConvSP layer, which computes particle-particle pairwise interactions, and the second is the ConvSDF layer, which computes particle-static object interactions\(^1\). We combine these two layers with standard operators (e.g., elementwise addition) to reproduce the algorithm in figure 1 inside a deep network. The parameters are the \( \lambda_\omega \) values described in section 3. We implemented both forward and backward functions for our layers in PyTorch [35] with graphics processor support.

4.1 ConvSP

The ConvSP layer is designed to compute particle to particle interactions. To do this, we implement the layer as a smoothing kernel over the set of particles. That is, ConvSP computes the following

\[
\text{ConvSP}(X, Y) = \left\{ \sum_{j \in X} y_j W(d_{ij}, h) \mid i \in X \right\}
\]

where \( X \) is the set of particle locations and \( Y \) is a corresponding set of feature vectors\(^2\), \( y_j \) is the feature vector in \( Y \) associated with \( j \), \( W \) is a kernel function, \( d_{ij} \) is the distance between particles \( i \) and \( j \), and \( h \) is the cutoff radius (i.e., for all \( d_{ij} > h \), \( W(d_{ij}, h) = 0 \)). This function computes the smoothed values over \( Y \) for each particle using \( W \).

While this function is relatively simple, it is enough to enable the network to compute the solutions for pressure, cohesion, surface tension, and viscosity (lines 5–7 and 12 in figure 1). In the following paragraphs we will describe how to compute the pressure solution using the ConvSP layer. Computing the other 3 solutions is nearly identical.

To compute the pressure correction solution in equation (1) above, we must first compute the density \( \rho_k \) at each particle \( k \). Equation (3) describes how to compute the density. This equation closely matches the ConvSP equation from above. To compute the density at each particle, we can simply call \( \text{ConvSP}(P, M) \), where \( P \) is the set of particle locations and \( M \) is the corresponding set of particle masses. Next, to compute the pressure \( \omega_k \) at each particle \( k \) as described in equation (2), we can use an elementwise subtraction to compute \( \rho_k - \rho_\pi \), a rectified linear unit to compute the max, and finally an elementwise multiplication to multiply by \( \lambda_\omega \). This results in \( \Omega \), the set containing the pressure for every particle.

Plugging these values into equation (1) is not as straightforward. It is not obvious how the term \( n_{ji}(\omega_i + \omega_j) \) could be represented by \( Y \) from the ConvSP equation. However, by unfolding the terms and distributing the sum we can represent equation (1) using ConvSP.

First, note that the vector \( n_{ji} \) is simply the difference in position between particles \( i \) and \( j \) divided by their distance. Thus we can replace \( n_{ji} \) as follows

\[
\delta p_i^\omega = \sum_{j \in P - \{i\}} \frac{p_k - p_j}{d_{ij}} (\omega_i + \omega_j) W_\omega(d_{ij}, h)
\]

where \( p_k \) is the location of particle \( k \). For simplicity, let us incorporate the denominator \( d_{ij} \) into \( W_\omega \) to get it out of the way. We define \( \overline{W_\omega}(d_{ij}, h) = \frac{1}{d_{ij}} W_\omega(d_{ij}, h) \).

Next we distribute the terms in the parentheses to get

\[
\delta p_i^\omega = \sum_{j \in P - \{i\}} (p_i \omega_i + p_j \omega_j - p_i \omega_i - p_j \omega_j) \overline{W_\omega}(d_{ij}, h).
\]

We can now rearrange the summation and distribute \( \overline{W_\omega} \) to yield

\[
\delta p_i^\omega = p_i \omega_i \sum \overline{W_\omega}(d_{ij}, h) + p_i \sum \omega_j \overline{W_\omega}(d_{ij}, h) - \omega_i \sum p_j \overline{W_\omega}(d_{ij}, h) - \sum p_j \omega_j \overline{W_\omega}(d_{ij}, h).
\]

\(^1\)The code for SPNets is available at https://github.com/cschemck/SmoothParticleNets

\(^2\)In general, these features can represent any arbitrary value, however for the purposes of this paper, we use them to represent physical properties of the particles, e.g., mass or density.
Here we omitted the summation term \( j \in P - \{i\} \) from our notation for clarity. We can compute this over all \( i \) using the ConvSP layer as follows

\[
\Delta P' = P \ast \Omega + \text{ConvSP}(P, \{1\}) + P \ast \text{ConvSP}(P, \Omega) - \Omega \ast \text{ConvSP}(P, P) - \text{ConvSP}(P, P \ast \Omega)
\]

where \( \ast \) represents elementwise multiplication and + and - are elementwise addition and subtraction respectively. \( \{1\} \) is a set containing all 1s.

4.2 ConvSDF

The second layer we add is the ConvSDF layer. This layer is designed specifically to compute interactions between the particles and static objects in the scene (line 9 in figure 1). We represent these static objects using signed distance functions (SDFs). The value \( \text{SDF}(p) \), where \( p \) is a point in space, is defined as the distance from \( p \) to the closest point on the object’s surface. If \( p \) is inside the object, then \( \text{SDF}(p) \) is negative.

We define \( K \) to be the set of offsets for a given convolutional kernel. For example, for a \( 1 \times 3 \) kernel in 2D, \( K = \{(0, -1), (0, 0), (0, 1)\} \). ConvSDF is defined as

\[
\text{ConvSDF}(X) = \left\{ \sum_{k \in K} w_k \min_{j \in \Omega} \text{SDF}(p_i + k \ast d) \right\} \in X
\]

where \( w_k \) is the weight associated with kernel cell \( k \), \( p_i \) is the location of particle \( i \), \( \text{SDF}_j \) is the \( j \)th SDF in the scene (one per rigid object), and \( d \) is the dilation of the kernel (i.e., how far apart the kernel cells are from each other). Intuitively, ConvSDF places a convolutional kernel around each particle, evaluates the SDFs at each kernel cell, and then convolves those values with the kernel. The result is a single value for each particle.

We can use ConvSDF to solve object collisions as follows. First, we construct \( \text{ConvSDF}_R \) which uses a size 1 kernel (that is, a convolutional kernel with exactly 1 cell). We set the weight for the single cell in that kernel to be 1. With a size 1 kernel and a weight \( w_k \) of 1, the summation, the kernel weight \( w_k \), and the term \( k \ast d \) fall out of the ConvSDF equation (above). The result is the SDF value at each particle location, i.e., the distance to the closest surface, where negative values indicate particles that have penetrated inside an object. We can compute that penetration \( R \) of the particles inside objects as

\[
R = \text{ReLU}(-\text{ConvSDF}_R(P))
\]

where \( \text{ReLU} \) is a rectified linear unit. \( R \) now contains the minimum distance each particle would need to move to be placed outside an object, or 0 if the particle is already outside the object. Next, to determine which direction to “push” penetrating particles so they no longer penetrate, we need to find the direction to the surface of the object. Without loss of generality, we describe how to do this in 3D, but this method is applicable to any dimensionality. We construct \( \text{ConvSDF}_X \), which uses a \( 3 \times 1 \times 1 \) kernel, i.e., 3 kernel cells all placed in a line along the X-axis. We set the kernel cell weights \( w_k \) to -1 for the cell towards the negative X-axis, +1 for the cell towards the positive X-axis, and 0 for the center cell. We construct \( \text{ConvSDF}_Y \) and \( \text{ConvSDF}_Z \) similarly for the Y and Z axes. By convolving each of these 3 layers, we use local differencing in each of the X, Y, and Z dimensions to compute the normal of the surface of the object \( n_{SDF} \), i.e., the direction to “push” the particle in.

We can then update the particle positions \( P' \) as follows

\[
P' = P' + R \ast n_{SDF}.
\]

That is, we multiply the distance each particle is penetrating an object \( R \) by the direction to move it in \( (n_{SDF}) \) and add that to the particle positions.

4.3 Smooth Particle Networks (SPNets)

Using the ConvSP and ConvSDF layers described in the previous sections and standard network layers, we design SPNets to exactly replicate the PBF algorithm in figure 1. That is, at each timestep, the network takes in the current particle positions \( P \) and velocities \( V \) and computes the fluid dynamics by applying the algorithm line-by-line, resulting in new positions \( P' \) and velocities \( V' \). We show an SPNet layout diagram in the appendix. By repeatedly applying the network to the new positions and velocities at each timestep, we can simulate the flow of liquid over time. We utilize elementwise layers, rectified linear layers (ReLU), and our two particle layers ConvSP and ConvSDF to compute each line in figure 1. Since elementwise and ReLU layers are differentiable, and because we implement analytical gradients for ConvSP and ConvSDF, we can use backpropagation through the whole network to compute the gradients. Additionally, our layers are implemented with graphics processor support, which means that a forward pass through our network takes approximately \( \frac{1}{15} \) of a second for about 9,000 particles running on an Nvidia Titan Xp graphics card.

5 Evaluation & Results

To demonstrate the utility of SPNets, we evaluated it on three types of tasks, described in the following sections. First, we show how our model can learn fluid parameters from data. Next, we show how we can use the analytical gradients of our model to do liquid control. Finally, we show how
we can use SPNets to train a reinforcement learning policy to solve a liquid manipulation task using policy gradients.

5.1 Learning Fluid Parameters

We evaluate SPNets on the task of learning, or estimating, some of the $\lambda$ fluid parameters from data. This experiment illustrates how one can perform system identification on liquids using gradient-based optimization. Here we frame this system identification problem as a learning problem so that we can apply learning techniques to discover the parameters. We use Nvidia FleX [36] to generate ground truth data, and then use backpropagation and gradient descent to iteratively update the parameter values until convergence. FleX is a commercially available physics simulation engine which implements fluid dynamics using Position Based Fluids (PBF).

Given sequences $P = \{P_t\}$ and $V = \{V_t\}$ of particle positions and velocities over time generated by FleX, at each iteration we do the following. First we randomly sample $B$ particle positions $P$ and velocities $V$ from $P, V$ to make a training batch $P_B, V_B$. Next, SPNet is used to roll out the dynamics $T$ timesteps forward in time to generate $\tilde{P}_{B+T}, \tilde{V}_{B+T}$, the predicted particle positions and velocities after $T$ timesteps. We then compute the loss $L(\tilde{P}_{B+T}, \tilde{V}_{B+T}, P_{B+T}, V_{B+T})$ between the predicted positions and velocities and the ground truth positions and velocities. Since our model is differentiable, we can use backpropagation to compute the gradient of the loss with respect to each of the fluid parameters. We then take a gradient step to update the parameters. This process is repeated until the parameters converge.

We used the ladle scene shown in Figure 2a to test our method. Here, the liquid rests in a rectangular container as a ladle scoops some liquid from the container and then pours it back into the container. We generated 9 sequences, one for each combination of the cohesion parameter $\lambda_c \in \{0.05, 0.1, 0.15\}$ and viscosity parameter $\lambda_v \in \{30, 60, 90\}$ (we fixed all other fluid parameters). Each sequence lasted exactly 620 frames. We set our batch size to 8, $T$ to 2, and use Adam [37] with default parameter values and a learning rate of $1e^{-2}$ to update the fluid parameters at each iteration. We evaluate using 2 different loss functions. The first is an L1 loss between the predicted parameter values and the ground truth parameter values after each iteration of training for both the L1 loss and the projection loss. The color of the lines indicates the ground truth $\lambda_c$ and $\lambda_v$ values.

We use the ladle scene shown in Figure 2a to test our method. Here, the liquid rests in a rectangular container as a ladle scoops some liquid from the container and then pours it back into the container. We generated 9 sequences, one for each combination of the cohesion parameter $\lambda_c \in \{0.05, 0.1, 0.15\}$ and viscosity parameter $\lambda_v \in \{30, 60, 90\}$ (we fixed all other fluid parameters). Each sequence lasted exactly 620 frames. We set our batch size to 8, $T$ to 2, and use Adam [37] with default parameter values and a learning rate of $1e^{-2}$ to update the fluid parameters at each iteration. We evaluate using 2 different loss functions. The first is an L1 loss between the predicted and ground truth parameter values and velocities. This is possible because we know which particle in FleX corresponds to which particle in the SPNet prediction. In real world settings, however, such a data association is not known, so we evaluate a second loss function that eschews the need for it. We use the projection loss, which simulates a camera observing the scene and generating binary pixel labels as the observation (similar to the heatmaps generated in [15]). We compute the projection loss between the predicted positions and velocities projected onto a virtual camera image, adding a small Gaussian around the projected pixel-positions of each particle, and then passing the entire image through a sigmoid function to flatten the pixel values. The loss is then the L1 difference between the projected image of the predicted particles and the ground truth particles. Projecting the particles as a Gaussian allows us to compute smooth gradients backwards through the projection. For the ladle scene, the camera is placed horizontally from the ladle, looking at it from the side.

Figures 2b and 2c show the difference between the ground truth and estimated values for the cohesion $\lambda_c$ and viscosity $\lambda_v$ parameters when training the model on each of the 9 sequences. In all 9 cases and for both losses, the network converges to the ground truth parameter values after only
a couple hundred iterations. While the L1 loss tended to converge slightly faster (which is to be expected with a more dense loss signal), the projection loss was equally able to converge to the correct parameter values, indicating that the gradients computed by our model through the camera projection are able to correctly capture the changes in the liquid due to its parameters. Note that for the projection loss the camera position is important to provide the silhouette information necessary to infer the liquid parameters.

5.2 Liquid Control

To test the efficacy of the gradients produced by our models, we evaluate SPNets on 3 liquid control problems. The goal in each is to find the set of controls $\mathcal{U} = \{u_t\}$ that minimize the cost

$$L = \sum_t l(P_t, V_t, u_t)$$  \hspace{1cm} (4)$$

where $l$ is the cost function, $\mathcal{U}$ is the set of particle positions at time $t$, and $V_t$ is the set of particle velocities at time $t$. $P_t$ and $V_t$ are defined by the dynamics as follows

$$P_t, V_t = SPN(P_{t-1}, V_{t-1}, OP(u_t))$$  \hspace{1cm} (5)$$

where $SPN$ is the fluid dynamics computed by SPNets, and OP transforms the control $u_t$ to the poses of the rigid objects in the scene at time $t$. The initial positions $P_0$ and velocities $V_0$ of the particles, the loss function $l$, and the control function OP are fixed for each specific control task.

To optimize the controls $\mathcal{U}$, we utilize Model Predictive Control (MPC) [38]. MPC optimizes the controls for a short, finite time horizon, and then re-optimizes at every timestep. Specifically, given the current particle positions $P_t$ and velocities $V_t$ and the set of controls $\mathcal{U}$ computed at the previous timestep, MPC first computes the future positions $P_{t+1},...,P_{t+T}$ and velocities $V_{t+1},...,V_{t+T}$ by repeatedly applying the SPNet for some fixed horizon $T$. Then, MPC sums the loss over this horizon as described in equation (4) and computes the gradient of the loss $L$ with respect to each control $\Delta u_t$ via our differentiable model. Finally, the updated controls $\mathcal{U}'$ are computed as follows

$$\mathcal{U}' = \left\{ u_i - s \frac{\Delta u_i}{\Delta t} \mid i \in [t, t + T] \right\}$$

where $s$ is a fixed step size. The first control $u'_t \in \mathcal{U}'$ is executed, the next particle positions $P_{t+1}$ and velocities $V_{t+1}$ are computed, and this process is repeated to update all controls again. We set $T$ to 10 and use velocity controls on our 3 test scenes.

The Plate Scene: Figure 3a shows the plate scene. It consists of a plate surrounded by 8 bowls. A small amount of liquid is placed on the center of the plate, and the plate must be tilted such that the liquid falls into a given target bowl. The controls for this task are the rotation of the plate about the x (left-right) and z (forward-backward) axes. We set the loss function for this scene to be the L2 (i.e., euclidean) distance between the positions of the particles and a point in the direction of the target bowl. We ran 8 evaluations on this scene, once with each bowl as the target.

Figure 4a shows the results of each of the evaluations on the plate scene. In every case, the optimization produced a trajectory where the plate would “dip” in the direction of the target bowl, wait until the liquid had gained sufficient momentum, and then return upright, which allowed the liquid to travel further off the edge of the plate. Note that simply “dipping” the plate would result in the liquid falling short of the bowl. For all bowls except one, this resulted in 100% of the liquid being placed into the correct bowl. For the one bowl, when it was set as the target, all but a small number of the liquid particles were placed in the bowl. Those particles landed on the lip of the bowl, eventually rolling off onto the ground. Nonetheless, it is clear that our method is able to effectively solve this task in the vast majority of cases.

The Pouring Scene: We also evaluated our method on the pouring scene, shown in Figure 3b. The goal of this task is to pour liquid from the cup into the bowl. The control is the rotation of the cup about the z (forward-backward) axis, starting from vertical. Note that there is no limit on the rotation; the cup may rotate freely clockwise or counter-clockwise. Since the cup needs to perform
Figure 4: Results from the liquid control task. From left to right: The plate scene. The numbers in each bowl indicate the percent of particles that were successfully placed in that bowl when that bowl was the target. The pouring scene. The x axis is the targeted pour amount and the y axis is the amount of liquid that was actually poured where the red marks indicate each of the 11 pours. The catching scene. Shown is the percent of liquid caught by the target cup where the rows indicate the initial direction of movement of the source.

We ran 11 evaluations of this scene, varying the desired amount of poured liquid between 75g and 275g. In every case, all liquid either remained in the cup or was poured into the bowl; no liquid was spilled on the ground. For that reason, in figure 4b we show how close each evaluation was to the given desired pour amount. In every case, the amount poured was within 11g of the desired amount, and the average difference across all 11 runs between actual and desired was 5g. Note that the initial rotation of the cup happens implicitly; our loss function only specifies a desired target for the liquid, not any explicit motion. This shows that physical reasoning about fluids using our model enables solving fine-grained reasoning tasks like this.

The catching scene: The final scene we evaluated on was the catching scene, shown in Figure 3c. The scene consisted of two cups, a source cup in the air filled with liquid and a target cup on the ground. The source cup moved arbitrarily while dumping the liquid in a stream. The goal of this scene is to shift the target cup along the ground to catch the stream of liquid and prevent it from hitting the ground. The control is the x (left-right) position of the cup. In order to ensure smooth gradients, we set the loss to be the x distance between each particle and a point on the lip of the cup closest to the bowl. Once a desired amount of liquid has left the cup, we switch to the second part, which is a standard regularization loss, i.e., the loss is the rotation of the cup squared, which encourages it to return upright.

We ran 8 evaluations of our model, varying the movement of the source cup. In every case, the source cup would initially move left/right, then after a fixed amount of time, would switch directions. Half the evaluations started with left movement, the other half right. We vary the switch time between 3.3s, 4.4s, 5.6s, and 6.7s. The first column of the table in Figure 4c shows the percentage of liquid caught in the cup. In all cases, the vast majority of the liquid was caught, with only a small amount dropped due largely to the time it took the target cup to initially move under the stream. It is clear from these results and the liquid control results on the previous two scenes that our model can enable fine-grained reasoning about fluid dynamics.

5.3 Learning a Liquid Control Policy via Reinforcement Learning

Finally, we evaluate our model on the task of learning a policy in a reinforcement learning setting. That is, the control $u_t$ at timestep $t$ is computed as

$$u_t = \pi(o_t, \theta)$$

where $o_t$ is the observation at time $t$, $\theta$ are the policy parameters, and $\pi$ is a function mapping the observation (and policy parameters) to controls. Since we have access to the full state, we compute the observation $o_t$ as a function of the particle positions $P_t$ and velocities $V_t$. The goal is to learn the parameters $\theta^*$ that best optimize a given loss function $l$.

To do this, we can use a technique very similar to the MPC technique which we described in the previous section. The main difference is that because the controls $u_t$ are a function of the policy, we optimize instead the policy parameters $\theta$. We rollout the policy for a fixed number of timesteps, compute the gradient of the policy parameters with respect to the loss, and then update the parameters. This is possible because our model is fully differentiable, so we can use backpropagation to compute the gradients backwards through the rollout.
We test our methodology on the catching scene. To train our policy, we use the data generated by the 8 control sequences from the previous section using MPC on the catching Scene. At each iteration of training, we randomly sample a different timestep \( t \) for each of the 8 sequences, then rollout the policy starting from the particle positions \( P_t \) and velocities \( V_t \). We initialize the target cup X position by adding Gaussian noise to the X position of the target cup at time \( t \) in the training sequences. The observation is computed by projecting the particles onto a virtual camera image as described in section 5.1. The camera is positioned so that both cups are in its field of view. Its X position is set to be the same as the X position of the target cup, that is, the camera moves with the target cup.

Since the observation is effectively binary pixel labels, we use a relatively simple model to learn the policy. We use a convolutional neural network with 1 convolutional layer (10 \( 3 \times 3 \) kernels with stride of 2) followed by a rectified linear layer, followed by a linear layer with 100 hidden units, followed by another rectified linear layer, and finally a linear layer with 1 hidden unit. We feed the output through a hyperbolic tangent function to ensure it stays within a fixed range. We trained the policy for 1200 iterations using the Adam [37] optimizer with a learning rate of \( 2 \times 10^{-5} \). The input to the network is a \( 160 \times 120 \) image and the output is the control.

The middle column of the table in figure 3c shows the percent of liquid caught by the target cup when using the policy to generate the controls for the same 8 sequences as were used in section 5.2 on the catching scene. In all 8 cases, the vast majority of the liquid was caught by the target cup. However, these were the same 8 sequences that the policy was trained on. To test the generalization ability of the policy, we modified the sequences as follows. For all the training sequences, the source cup rotated counter-clockwise (CCW) when pouring. To test the policy, we had the source cup follow the same X movement, but rotated clockwise (CW) instead, i.e., in training the liquid always poured from the left side of the source, but for testing it poured out of the right side. The percent of liquid caught by the target cup when using the policy for the CW case is shown in the third column of the table in figure 3c. Once again the policy is able to catch the vast majority of the liquid. The main point of failure is when the source cup initially moves to the left. In this case, as the source cup rotates, the liquid initially only appears in the upper-left of the image. It’s not until the liquid has traveled downward several centimeters that the policy begins to move the target cup under the stream, causing it to fail to catch the initial liquid. This behavior makes sense, given that during training the policy only ever encountered the source cup rotating CCW, resulting in liquid never appearing in the upper-left of the image. Nonetheless, these results show that, at least in this case, our method enables us to train robust policies for solving liquid control tasks.

6 Conclusion & Future Work

In this paper we presented SPNets, a method for computing differentiable fluid dynamics and their interactions with rigid objects inside a deep network. To do this, we developed the ConvSP and ConvSDF layers, which allow the model to compute particle-particle and particle-rigid object interactions. We then showed how these layers can be combined with standard neural network layers to compute fluid dynamics. Our evaluation in Section 5 showed how a fully differentiable fluid model can be used to 1) learn, or identify, fluid parameters from data, 2) control liquids to accomplish a task, and 3) learn a policy to control liquids. This is the power of model-based methods: they are widely applicable to a variety of tasks. Combining this with the adaptability of deep networks, we can enable robots to robustly reason about and manipulate liquids. Importantly, SPNets make it possible to specify liquid identification and control tasks in terms of the desired state of the liquid; the resulting controls follow from the physical interaction between the liquid and the controllable objects. This is in contrast to prior approaches to pouring liquids, for instance, where the relationships between controls and liquid states have to be specified via manually designed functions.

We believe that by combining model-based methods with deep networks for liquids, SPNets provides a powerful new tool to the roboticist’s toolbox for enabling robots to handle liquids. A next step for future work is to combine SPNets with real perceptual data. For instance, by connecting a convolutional neural network to an SPNet, the image features could be directly integrated into the fluid dynamics model to enable a robot to track the state of real liquids using closed-loop feedback, similar to [21] however without the need for a thermographic camera. Additionally, one could add a set of parameters to SPNets to facilitate learning a residual model between the analytical fluid model and real observed fluids, or even to learn the dynamics of different types of substances such as sand or flour. SPNets can also be used to perform more complex manipulation tasks, such as mixing multiple liquid ingredients in a bowl, online identification and prediction of liquid behavior, or using spoons to move liquids, fluids, or granular media between containers.


References


A Position Based Fluids Continued

In section 3 of the main paper we gave a brief overview of the Position Based Fluids (PBF) algorithm. The key steps of the PBF algorithm are moving the particles (lines 2–3 in figure 1 in the main paper), iteratively solving the constraints imposed by the incompressibility of the fluid (lines 4–10), and updating the velocities (lines 11–12). Solving the constraints entails iteratively moving each particle to better satisfy each constraint until the constraints are satisfied. In PBF, this is approximated by repeating the inner loop (lines 4–10) a fixed number of times.

In the main paper we described some of the details of computing the various constraint solutions. Here we describe computing the solutions to the SolveCohesion (line 6), SolveSurfaceTension (line 7), and ApplyViscosity (line 12) functions.

The cohesion correction $\delta p_i^c$ for each particle $i$ is computed as

$$\delta p_i^c = \sum_{j \in P - \{i\}} \lambda c n_{ij} W_c(d_{ij}, h)$$

where $\lambda c$ is the cohesion constant and $W_c$ is a kernel function. For $W_c$ we use

$$W_c(d, h) = \frac{1}{d_0^3} \left( \frac{d}{h} \right)^3 + \frac{d_0}{d_0^2 + d_0 + 1} \left( \frac{d}{h} \right)^2 - 1$$

where $d_0$ is the fluid rest distance as a fraction of $h$. For this paper we fix $d_0$ to 0.5.

The surface tension correction $\delta p_i^s$ for each particle $i$ is computed using the following 2 equations

$$\delta p_i^s = \sum_{j \in P - \{i\}} \frac{\lambda s}{\rho_0} (n_{ij} - n_i) I(d_{ij} \leq h)$$

where $\lambda s$ is the surface tension constant, $n_k$ is the normal of the fluid surface at particle $k$, and $I$ is the indicator function. The normal $n_k$ is computed as

$$n_k = \sum_{j \in P - \{i\}} n_{ij} W_c(d_{ij}, h)$$

where $W_c$ is the same kernel function used for the cohesion constraint.

Finally, the viscosity update $\delta v_i$ for each particle $i$ computed by ApplyViscosity is

$$\delta v_i = \sum_{j \in P - \{i\}} \frac{\lambda v}{\rho_0} (v_j - v_i) W_v(d_{ij}, h)$$

where $\lambda v$ is the viscosity constant, $v_k$ is the velocity of particle $k$, and $W_v$ is the same kernel function used to compute the density.

B SPNet Diagram

As described in section 3, Smooth Particle Networks (SPNets) implements the Position Based Fluids (PBF) algorithm, which is shown in the main paper in figure 1. Figure 5 shows the layout of SPNets as a network diagram. The network takes as input the current particle positions $P$ and velocities $V$, and computes the fluid dynamics for a single timestep resulting in the new positions $P'$ and velocities $V'$. For clarity, the functions SolvePressure, SolveCohesion, SolveSurfaceTension, and SolveObjectCollisions are collapsed into individual boxes in figure 5a. The full layout for SolvePressure and SolveObjectCollisions are shown in figures 5c and 5d respectively.

The first operation the network performs is to apply the external forces to the particles, line 2 of the PBF algorithm (shown in figure 1 in the main paper) and the lavender box in the upper-left of figure 5a here. Next the network updates the particle positions according to their velocities, line 3 of PBF and the element-wise multiplication and addition immediately to the right of ApplyForces. After this, the network iteratively solves the fluid constraints (lines 5–9), shown by the SolveConstraints boxes in figure 5a. Here we show 3 constraint solve iterations, however in principle the network could have any number. Each constraint solve partially updates the particle positions to better satisfy the given constraints.

We consider 3 constraints in this paper: pressure (line 5), cohesion (line 6), and surface tension (line 7). Each is shown as an individual box in figure 5a. Figure 5c shows the full network layout for the pressure constraint. This exactly computes the solutions to equations 1–3 from the main paper as derived in section 4.1. Note the column under the leftmost ConvSP layer in figure 5a; it computes the pressure set $\Omega$. This is then used to compute the result of the other 4 ConvSP layers. The final step of each constraint solve iteration is to solve the object collisions. The expansion of this box is shown in figure 5d. The ConvSDF layer on the left computes the particle penetration $R$ into the SDFs, and the 3 on the right compute the normal $n_{SDF}$ of the SDFs. Note that in this diagram
Figure 5: The layout for SPNet. The upper-left shows the overall layout of the network. The functions \textsc{SolvePressure}, \textsc{SolveCohesion}, \textsc{SolveSurfaceTension}, and \textsc{SolveObjectCollisions} are collapsed for readability. The lower-right shows the expansion of the \textsc{SolveObjectCollisions} function, with the line in the top of the box being the input to the \textsc{SolveObjectCollisions} in the upper-left diagram and the line out of the bottom of the box being the line out of the box. The lower-left shows the expansion of the \textsc{SolvePressure} function. For clarity, the input line (which represents the particle positions) is colored green and the line representing the particle pressures is colored pink.
we show the layout for particles in 3D (there are 3 ConvSDF layers on the right of figure 5d, one to compute the normal direction in each dimension), however this can applied to particles in any dimensionality.

After finishing the constraint solve iterations, the network computes the adjusted particle velocities based on how the positions were adjusted (line 11 of the PBF algorithm), shown in figure 5a as the element-wise subtraction and multiplication above the APPLY VISCOSITY box. Finally, the network computes the viscosity, shown in the tan box in the bottom-right of figure 5a. Viscosity only affects the particles velocities, so the output positions of the particles are the same as computed by the constraint solver.

There are several parameters and constants in this network. In the APPLY FORCES box in the upper-left of figure 5a, Gravity is set to be $-9.8 \frac{m}{s^2}$ and $\Delta t$ is set to be $\frac{1}{60}$. The rest density $\rho_0$, shown in the APPLY VISCOSITY box in the lower-right of figure 5a and in the SOLVE PRESSURE box in figure 5c, is set empirically based on the rest density of water. The fluid parameters $\lambda_w$ and $\lambda_v$ are shown in figure 5c and the lower-right of figure 5a respectively.