Physics-Informed Gaussian Splatting

Solving Partial Differential Equations with Gaussians

July 2024

Max Rensen

Delft University of Technology

Computer Graphics and Visualization Group

Supervisors:

Michael Weinmann
Benno Buschmann
Elmar Eisemann

Abstract

The prevalence of partial differential equations (PDEs) in modeling physics and the low speeds of numerical solvers demands more efficient solving methods. For this purpose, machine learning based methods have been proposed, but these are typically discrete, difficult to interpret and require large amounts of ground truth data to train. To address these issues, we propose a novel machine learning based solver that builds upon advancements in Gaussian based reconstruction. Our method represents the solution to a time-dependent target PDE purely in terms of Gaussians, making it completely continuous and meshless. These Gaussians are evolved by means of an autoregressive neural network that is applied to each Gaussian, integrating information from a local neighborhood of Gaussians. This enables the change of position, scale, rotation and value of the Gaussians to model the solution in a particle-like manner. Extensive experiments show the potential for Gaussians to model arbitrary physical phenomena. We also compare our approach to ground truth data and various state of the art methods, which demonstrates that the method performs well on short-term prediction, but does not match the state of the art for long-term prediction.

Contents

1 Introduction

Partial differential equations (PDEs) are at the core of modeling physical phenomena. Through PDEs, complex behaviors can be characterized with only a few relatively simple equations. However, solving these equations is non-trivial and finding exact solutions is often intractable, instead requiring complex numerical methods to find a high-quality approximate solution [40]. This numerical approximation necessitates computationally expensive solvers that must be applied separately to distinct problems described by the same PDE. To improve the efficiency of this approximation, researchers have applied various machine learning techniques to this problem [1, 19, 32]. Some of these methods learn the approximation using a large amount of ground truth data from a numerical solver [17, 50]. However, a lack of training data can hinder these types of approaches, so another type of machine learning method has been proposed that makes use of physics-informed neural networks (PINNs), where a physics-informed loss term used for training is derived directly from a target PDE, which requires no ground truth data to train the model [32, 42]. With these approaches, the majority of the computation is shifted to training the model on a target PDE, and the inference of this model is significantly faster than applying a numerical solver. In essence, the trained model will then contain a black-box solution to the PDE and it can be applied to unseen problems and rapidly provide an approximate solution. Typically, these approaches sacrifices some quality for the gain in performance when compared to numerical methods.

Recent advances in neural three-dimensional scene reconstruction have seen a lot of attention with neural radiance fields (NeRF) [9, 24] and 3D Gaussian splatting (3D-GS) [4, 14]. With NeRF an implicit representation of a 3D scene is learned through a model that can then produce an image of the scene from any viewpoint. On the other hand, 3D-GS provides an explicit representation of a 3D scene in terms of three-dimensional Gaussians, which can also be used to produce an image of the scene from any viewpoint. Both methods are trained by learning to closely represent a set of images of the scene from different viewpoints. 3D-GS provides much faster inference than NeRF, while still producing comparable reconstruction results, due to its explicit instead representation.

In this work, we combine insights from physics-informed machine learning and Gaussian splatting to form physics-informed Gaussian splatting, as illustrated in Figure 1. Namely, we wish to use the representational quality of Gaussians and the fast inference of physics-informed machine learning to model solutions to arbitrary time-dependent PDEs. Although some previous works explore dynamic Gaussian splatting [7, 21, 35, 49], also within the context of modeling physics [8, 12, 22, 52], none of these works explore the use of Gaussians for the purpose of solving PDEs. To that end, we perform extensive experiments on the Burgers' equation and Navier-Stokes equations to evaluate the representational and dynamic quality of Gaussians for this purpose and develop a method to dynamically update arbitrary sets of Gaussians to model solutions to a target PDE. We also compare our method to state of the art PINNs produces reasonable results for short-term prediction, but does not match the quality of state of the art approaches for long-term prediction. Our method is continuous, meshless and can potentially be integrated into 3D-GS methods. However, our approach is, in part,

limited by the initialization of the Gaussians, for which the number and arrangement of the Gaussians is highly relevant to the quality of the simulation and is also a significant bottleneck in terms of performance.

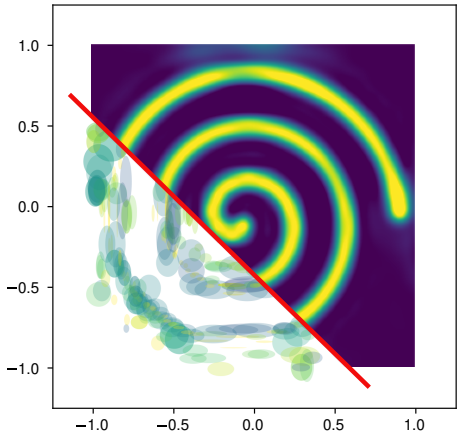


Figure 1: *Illustration of an Archimedean spiral modeled with Gaussians.*

With this work, we aim to answer the following research question:

What is the potential of a Gaussian representation for physics-informed machine learning, i.e., can a respective representation be dynamically updated to represent physical phenomena with high accuracy?

Specifically within the context of time-dependent phenomena pertaining to fluid dynamics, e.g. Burgers' equation and Navier-Stokes equations. To explore different aspects of the research question, we have devised the following sub-questions that we aim to answer throughout this work:

- *How can the initial Gaussians be distributed effectively?*

  The Gaussians ought to be able to represent any initial condition and be suitable for updating dynamically.

- *How does the number of Gaussians affect the performance and quality?*

  Densifying and pruning the Gaussians adaptively over time should aid in answering this.

- *How can information from neighboring Gaussians best be integrated?*

  A k-nearest neighbor approach might be insufficient when the Gaussians become anisotropic.

2 Background & Related Work

In the following sections, we begin with a brief introduction of partial differential equations (PDEs), which are at the core of our investigations. Then, we provide an overview of how these PDEs are typically solved using machine learning and describe prior works in this area. Lastly, we discuss recent work in static and dynamic scene reconstruction and Gaussian splatting [14], which forms the foundation of our method.

2.1 Partial Differential Equations

Physical phenomena are often characterized by partial differential equations (PDEs) [40]. These equations describe the change of one or multiple variables in a system over time and/or space.

An example of a simple space-dependent PDE is the Poisson equation:

$$\nabla^2 u(\mathbf{x}) = f(\mathbf{x})$$

Here $f(\mathbf{x})$ and $u(\mathbf{x})$ are two functions of space, where f is given and describes the environment, and we want to find the solution u describing the physical phenomena for any point in space $\mathbf{x}$.

An example of a time- and space-dependent PDE is the diffusion equation:

$$\dot{u}(t, \mathbf{x}) = \nabla^2 u(t, \mathbf{x})$$

Where $\dot{u}$ is the first-order time derivative of $u(t, \mathbf{x})$ and we again want to solve for u, this time for any point in time t and space $\mathbf{x}$.

Whereas such simpler examples can be solved analytically, more complex PDEs do not have a known solution and as such need to be solved numerically. Finding a numerical solution to a PDE usually requires a simplification of the problem by means of discretization. In our case only discretization of the temporal domain is of interest (the reason for this will be explained in Section 2.2), for which we employ the standard backward Euler method. This is more stable than the forward Euler method and less complex than higher order discretizations.

For simplicity we assume our target PDE will be in the following form:

$$\dot{u}(t, \mathbf{x}) = g(t, \mathbf{x})$$

Where $g(t, \mathbf{x})$ constitutes every term that does not involve the time derivative. The backward Euler method then gives the following:

$$u_{k+1} = u_k + h_t g_{k+1}$$

Where t_k is the k-th time step in our discretization, $h_t = t_{k+1} - t_k$, $u_k = u(t_k, \mathbf{x})$ and $g_k = g(t_k, \mathbf{x})$. The smaller the time step h_t, the more accurate this time discretization becomes, but the longer the optimization of the same time span will take. Substituting the relevant PDE into this expression then gives us an equation we can approximate. However, solving this also requires a priori knowledge about the domain

Ω of our problem and its boundary domain $\partial\Omega$, along with the boundary conditions and the initial conditions as follows:

$$\begin{aligned} u(0, \mathbf{x}) &= u_0(\mathbf{x}),\ \mathbf{x} \in \Omega && \text{(Initial condition)} \\ u(t, \mathbf{x}) &= b(t, \mathbf{x}),\ \mathbf{x} \in \partial\Omega && \text{(Dirichlet Boundary condition)} \\ \nabla u(t, \mathbf{x}) &= b(t, \mathbf{x}),\ \mathbf{x} \in \partial\Omega && \text{(Neumann Boundary condition)} \end{aligned}$$

The initial condition serves as the starting state of the phenomena and the boundary conditions describes what happens at the boundaries of a domain, such as the walls of a box.

The following equations are two well-known PDEs used in fluid dynamics, which are later to be approximated by our method.

Burgers' equation is often used as a simplified expression for the flow of a fluid as follows:

$$\dot{u} = \nu \nabla^2 u - u \frac{\partial u}{\partial x}$$

Where $u(t, \mathbf{x})$ is a scalar field describing the velocity of the fluid and ν is the kinematic viscosity.

Navier-Stokes equations are a more complete description of fluid flow as described by the following system of equations (vorticity form, assumes an incompressible fluid):

$$\begin{aligned} \nabla \cdot u &= 0 \\ \dot{w} &= \nu \nabla^2 w - u \cdot \nabla w + g \end{aligned}$$

Where $u(t, \mathbf{x})$ is a vector field describing the velocity, $w(t, \mathbf{x}) = \nabla \times u(t, \mathbf{x})$ is the vorticity, $g(t, \mathbf{x})$ are external forces and ν is again the kinematic viscosity.

The complexity of a fluid flow is often predicted with the Reynolds number Re, which is defined as follows:

$$\text{Re} = \frac{uL}{\nu} \tag{1}$$

Where u is the flow speed and L is the length of the system (i.e. the size of the domain). The Reynolds number is an indicator of the turbulence, so a higher Re indicates more expected turbulence.

Another equation that can be used for describing fluid-like behavior of periodic waves on a membrane is the **damped wave equation**. It has multiple definitions, but we use a simplified model, which is defined as follows:

$$\dot{u} = \nu \nabla^2 z - \gamma u$$
$$\dot{z} = u$$

Where $u(t, \mathbf{x})$ is a scalar field describing the velocity, $z(t, \mathbf{x})$ is the height field, ν is the stiffness constant an γ is the damping constant.

2.2 Physics-Informed Neural Networks

Traditionally, performing physical simulations described by complex partial differential equations has required the use of computationally expensive numerical solvers [40]. As with many computationally challenging problems, machine learning has been applied at an attempt to improve their efficiency [1, 19, 32]. As with many machine learning methods, the problem can be approached by training on large amounts of data [17, 50]. However, a lack of training data for some problems and difficulty in obtaining high-quality ground truth data has prompted the creation of a new approach that can learn from the target PDE, reducing or even eliminating the use of ground truth data. In the context of solving PDEs, these latter approaches are often called physics-informed machine learning, due to a (physics-informed) loss term derived from a corresponding target PDE [20, 32, 42]. This means the governing equation(s) and boundary conditions all induce a loss term that has to be minimized by some machine learning model. Many of these approaches thus make use of physics-informed neural networks (PINNs) to model the PDE solution.

The loss terms are simply defined as the residual of the PDE and boundary conditions, i.e. the difference between the right-hand side and left-hand side of the equations. Typically the mean squared error of the residuals is used to more strongly penalize higher deviations and to ensure the losses are non-negative. This gives us the following loss terms:

$$\mathcal{L}_{PDE} = MSE(\dot{u}(t, \mathbf{x}) - f(t, \mathbf{x})), \mathbf{x} \in \Omega \qquad \text{(PDE loss)}$$
$$\mathcal{L}_{BC} = MSE(u(t, \mathbf{x}) - b(t, \mathbf{x})), \mathbf{x} \in \partial\Omega \qquad \text{(Dirichlet BC loss)}$$
$$\mathcal{L}_{BC} = MSE(\nabla u(t, \mathbf{x}) - b(t, \mathbf{x})), \mathbf{x} \in \partial\Omega \qquad \text{(Neumann BC loss)}$$

Here $\mathbf{x}$ represents any point in the domain, so an exact solution would require an integral over the domain, which can only be computed exactly with analytical methods that are typically time consuming or intractable for complex problems. Instead, $\mathbf{x}$ is sampled randomly in continuous-space methods, or on a mesh in discrete-space methods, in order to approximate the integral describing the loss terms.

In these loss terms, typically only the derivatives over time require discretization, due to the convenient automatic differentiation present in neural networks, by definition. This automatic differentiation can be performed over any input to the network, such as spatial coordinates.

Initially, such physics-informed losses have been combined with small datasets containing ground truth data [31, 36]. Some more recent approaches avoid ground truth data altogether and successfuly learn to solve PDEs solely from the physics-informed loss [44, 46]. Training these models requires careful tuning of the weights of each PDE loss term involved. To overcome this, the loss function can be made adaptive based on the loss magnitude [51], or based on the loss statistics (such as the rate of improvement) [2]. Another method directs initial optimization towards earlier time steps by weighting with the inverse exponential of the loss sum at each time step [46].

A recent method embeds (some of) the PDE directly into the network architecture [23] by computing a finite difference approximation of the PDE for the input state, producing high-quality results. However, the method has only been trained on ground truth data from a numerical solver and compared to a baseline represented by a convolutional network, so it is unclear how it compares to more complex models.

There have been numerous different architectures proposed for solving PDEs, including but not limited to: convolutional networks [43, 44, 50], recurrent networks [37], transformers [56], neural operators [16, 18], Bayesian networks [53], evolutionary algorithms [1, 37] and Gaussian processes regression [10, 26].

2.3 Gaussian Splatting

Reconstructing a thee-dimensional scene from a collection of input images has seen a lot of attention in computer graphics and beyond since the introduction of neural radiance fields (NeRF) [4, 9, 14, 24]. This approach implicitly learns to represent a 3D scene from any viewpoint through a neural representation, taking as input a coordinate and view direction and estimating the color and density. A differentiable renderer then applies ray marching to produce an image for a target viewpoint. Given a collection of ground truth images of the desired scene from different viewpoints, the training proceeds by rendering the scene from different viewpoints through the differentiable renderer and comparing them directly to the expected ground truth images of those viewpoints, backpropagating the resulting reconstruction loss between the predictions and the reference images. This trained model can then be used, for example, to render the scene from unseen viewpoints.

Though some improvements have been made to the training and inference efficiency of NeRF [25, 54], real-time inference is still difficult to achieve. To that end, a recently repopularized method in computer graphics, 3D Gaussian splatting (3D-GS), instead makes use of a collection of unnormalized Gaussian distributions to represent a 3 dimensional scene [14, 58]. This approach directly learns the parameters of these Gaussians, along with a color and the opacity for each Gaussian, in order to represent a scene. Subsequently, much higher training and especially inference speeds are achieved without sacrificing quality. The collection of 3D Gaussians is rendered to a 2D image for a given viewpoint using a differentiable renderer. First, the mean μ and covariance matrix Σ of the 3D Gaussians are projected to screen space in the form of a 2D mean μ' and 2D covariance matrix Σ' as follows [58]:

$$\boldsymbol{\mu'} = PV\boldsymbol{\mu}$$
$$\Sigma' = JV\Sigma V^T J^T$$

Here V is the camera view matrix, P the projection matrix and J the Jacobian of the affine approximation of the projection matrix. Next, the resulting 2D Gaussians are rasterized [14]. This is achieved by alpha-blending the N sorted 2D Gaussians to get the color C for each pixel as follows:

$$C(\mathbf{x}) = \sum_{i=1}^{N} c_i \alpha_i(\mathbf{x}) \prod_{j=1}^{i-1}(1 - \alpha_j(\mathbf{x}))$$
$$\alpha_i(\mathbf{x}) = \sigma_i \exp(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu_i'})^T \Sigma_i'(\mathbf{x} - \boldsymbol{\mu_i'})) \tag{2}$$

Where $\mathbf{x}$ is the coordinate of the pixel and c_i, σ_i, $\boldsymbol{\mu_i'}$ and Σ_i' are the color, opacity, mean vector and covariance matrix for Gaussian i, respectively.

With this differentiable rendering, training 3D-GS is strikingly similar to that of NeRF, only requiring a few adaptations to create an initial distribution of Gaussians and to densify the Gaussians in empty regions. Besides the increase in performance, another advantage of 3D-GS is its better interpretability. Where NeRF is essentially a black box, mapping a position and view direction to a color, 3D-GS stores all relevant information in the form of easily interpretable Gaussian distributions and their auxiliary variables.

2.3.1 Dynamic Gaussian Splatting

The popularity of 3D Gaussian splatting has sparked interest in using the approach for dynamic scenes. Various methods have been proposed to reconstruct dynamic scenes, or 3D videos, both for NeRF [6, 28] and 3D-GS [21, 35, 49]. In the latter case, the approach is often called 4D Gaussian splatting (4D-GS) for the additional temporal dimension

For NeRF, these methods either include a time component as an extra network parameter [6], or warp the ray position through a separate machine learning model when rendering [28].

For 4D-GS, the Gaussians are typically translated, scaled and rotated, either through an implicit time-dependent machine learning model [35, 49], or an explicit representation of the transformations [21]. Another approach instead uses 4D Gaussians to represent all dimensions (i.e. spatial and temporal) of the dynamic scene [7], requiring no updates of the Gaussians during inference, but instead a specialized rendering algorithm that takes slices in the temporal dimension.

More recently, the large 4D Gaussian reconstruction model (L4GM) [33] has been proposed, which generates a set of Gaussians per-frame for a given single-view video input, using a trained prior based on the earlier large multi-view Gaussian model (LGM) [38] for generating 3D Gaussians based on a static single-view image. This

means the Gaussians are generated for every frame and not evolved from frame to frame, though temporal and view consistency are achieved through self-attention.

2.3.2 Physics-Informed Gaussian Splatting

Making use of Gaussian splatting for physics-based simulations is underexplored in the literature. Unlike 3D-GS for regular dynamic scenes, the dynamics and extent of a physics-based simulation are unknown beforehand, as they are typically described by unknown initial conditions and PDEs that have no known exact solution (e.g. the Navier-Stokes equations). As such, solving them requires a different approach that integrates information from the current state of the simulation and from the governing PDE, instead of training the Gaussians on a single dynamic scene. The following works explore physics-based simulations within the context of 3D-GS, but none of the approaches aim to approximate solutions to PDEs.

PhysGaussian [52] enables interactive deformation of 3D-GS models by conserving mass and momentum through a material point method within a combined Lagrangian and Eulerian view, where the Gaussians are treated as discrete particle clouds. Their realistic and accurate deformations can range from plastic to static, but no effort is made to simulate any PDE besides the simple conservation principles.

DreamPhysics [12] and Physics3D [22] both use the material point method to generate high-quality simulations of Gaussians obtained from 3D-GS. They also both allow for inverse physics by obtaining physical parameters from a video diffusion prior. However, the material point method performs an approximate simulation that is typically not real-time and physically plausible, but not necessarily physically accurate.

Gaussian Splashing [8] integrates a position-based dynamics approach with 3D-GS models to combine fluids interacting with solids. Here, both the fluid and solid particles are represented by Gaussians, making their interaction seamless. However, like the material point method, the position-based dynamics method is a simulation based on constraints and approximations and is not necessarily physically accurate.

3 Methodology

In the following sections, we describe the methodology devised for solving time-dependent PDEs using Gaussians and machine learning. The method is heavily inspired by PINNs (Section 2.2), 3D-GS (Section 2.3.2) and related methods. We begin by describing how our method uses a mixture of Gaussians to represent solutions to PDEs, followed by how these Gaussians are initialized. Next, we provide a description of the machine learning model used to evolve the Gaussians dynamically, which is trained through a physics-informed loss (see Section 2.2). The experimental setup for training and evaluating the model is further explained in Section 5. Then we elaborate upon the refinement process, which is used to add or remove certain Gaussians. Lastly, we discuss how Gaussians can be used to represent the boundary conditions involved in solving PDEs. An overview of our method is shown in Figure 2.

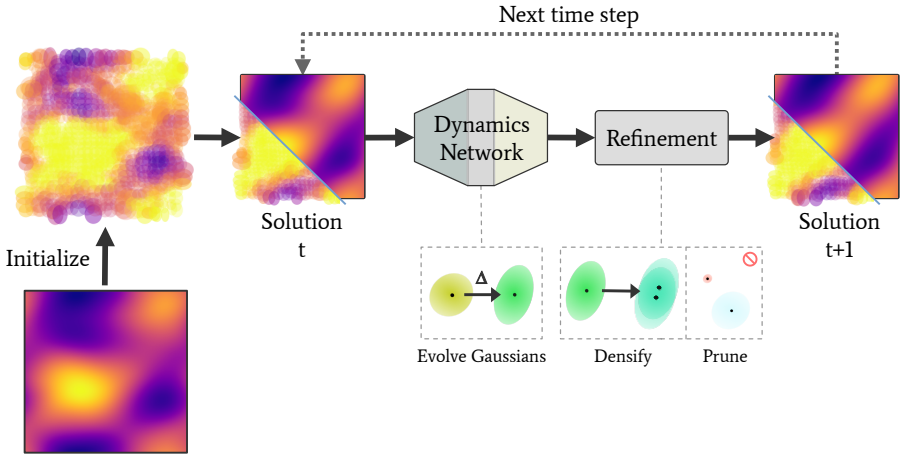


Figure 2: *Overview of our method used to dynamically update Gaussians to model solutions to a target PDE. First, the Gaussians are initialized according to some known initial condition. Then, the Gaussians are repeatedly evolved and refined in an iterative process to model the solution over an arbitrary number of time steps. Evolving the Gaussians is done through a dynamics network trained on the target PDE. The refinement step first prunes (removes) small or nearly invisible Gaussians and then densifies (splits) Gaussians in turbulent regions according to an appropriate metric.*

3.1 Gaussian Splatting for PDEs

With our approach, we wish to sample an n-dimensional solution to a target PDE from an m-dimensional domain. To that end, we adapt the 3D-GS algorithm [14] described in Section 2.3.2, which has originally been designed to render a 3D scene to a 2D image. Unlike 3D-GS, we do not perform alpha-blending and rasterization, instead we sample and combine all Gaussians for a given sample point. This is very similar to a mixture of Gaussian distributions, but without applying any normalization (see Figure

3). The normalization factor is discarded for simplicity; it can instead be learned as part of the Gaussian parameters (see below). By using this mixture of Gaussians, our solution is continuous, which means we are able to sample any point within the domain.

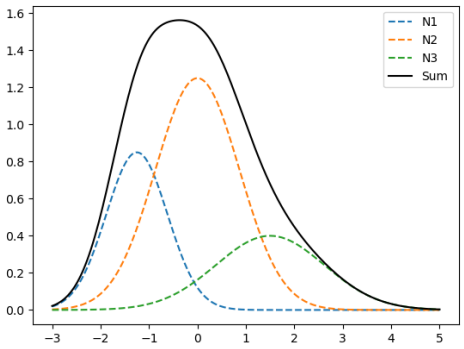


Figure 3: *Three unnormalized Gaussians (dashed lines) combined into a single mixture function (solid line). This mixture function is used to represent an approximate solution to the target PDE.*

The value $\mathbf{y}$ at a sample point $\mathbf{x}$ is computed by sampling from N Gaussians as follows:

$$\mathbf{y}(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i(\mathbf{x}) u_i \tag{3}$$
$$\alpha_i(\mathbf{x}) = \exp(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \Sigma_i (\mathbf{x} - \boldsymbol{\mu}_i))$$

Where for Gaussian i, u_i is the value, and α_i is a function of $\mathbf{x}$ as in Equation (2), but without the opacity and the projection to 2D. The value u_i is here assumed to be a scalar, but it can also be represented by a vector or even a tensor for more complex problems. For example, when solving the Navier-Stokes equation, we want to know the velocity at each point in our domain. Where the velocity is represented as an m-dimensional vector field, .i.e. $\mathbf{u} \in \mathbb{R}^m$. Since these variables are deeply intertwined, the same Gaussians can be used to for both elements of the vector.

The mean $\boldsymbol{\mu}_i$ is simply represented as an m-dimensional vector, but the covariance matrix Σ_i requires special attention, since it has to be symmetric and semi-positive definite (SPD) by definition. 3D-GS achieves this by constructing the covariance matrix from a diagonal matrix (scale) and a quaternion (rotation). However, since a quaternion is difficult to generalize to arbitrary dimensions, we decided to use an LDL decomposition instead, which gives us the following symmetric covariance matrix:

$$
\begin{aligned}
\Sigma &= LDL^T \\
&= \begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix} \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix} \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} && \text{(in 2D)} \\
&= \begin{pmatrix} s_1 & s_1 c^2 \\ s_1 c^2 & s_2 + s_1 c^2 \end{pmatrix} && \text{(in 2D)}
\end{aligned}
$$

This matrix is SPD when its eigenvalues are non-negative, which is the case when $s_1 s_2 \geq 0$, so we define $s_i = exp(s_i')$. Such a decomposition generalizes to any dimension, except 1D, where we only have a variance σ^2 and no covariance terms.

Our goal is to match $y(\mathbf{x})$ as closely as possible to the solution of our PDE within the domain. This essentially makes $\mathbf{y}(\mathbf{x})$ a finite decomposition of the solution function, in terms of Gaussians. However, the solution is typically unknown for complex non-linear PDEs, which makes an exact decomposition of the function impossible. As such, we require a further approximation, which is aided by our dynamic Gaussians.

Because $\mathbf{y}(\mathbf{x})$ approximates our solution function $\mathbf{u}(\mathbf{x})$, we can derive approximate spatial derivatives to be used in computing the residual:

$$
\begin{aligned}
\mathbf{u}(\mathbf{x}) &\approx \mathbf{y}(\mathbf{x}) \\
\frac{\partial}{\partial \mathbf{x}} \mathbf{u}(\mathbf{x}) \approx \frac{\partial}{\partial \mathbf{x}} \mathbf{y}(\mathbf{x}) &= -\sum_{i=1}^{N} v_i \alpha_i(\mathbf{x}) \Sigma_i \mathbf{x}
\end{aligned}
$$

This can be repeated to compute the Laplacian term and any further spatial derivatives. Furthermore, the derivative with respect to any of the Gaussian parameters (i.e. v_i, $\boldsymbol{\mu}_i$, σ_i and Σ_i) can be derived in a similar manner, which makes the Gaussian sampling completely differentiable.

3.2 Initializing Gaussians

In a time-dependent PDE, the initial conditions at time $t = 0$ are typically given in the form of a function (e.g. $u_0(\mathbf{x}),\ \mathbf{x} \in \Omega$, see Section 2.1). First, we need to obtain a set of Gaussians that represent these initial conditions and are then evolved from there to represent the solution to the target PDE.

To go from an initial function defined on a domain to a collection of Gaussians, we again draw inspiration from 3D Gaussian splatting [14]. Akin to Gaussian splatting, we directly update the Gaussian parameters through gradient descent, by comparing our current result to the desired initial condition. Unlike Gaussian splatting, we cannot directly use reconstruction loss for this training, because we typically want to represent a continuous function, instead of an image. Therefore, we use our sampling method (see Section 3.1) to sample points randomly from our domain and compare the sample output to the output of the initial function at the same coordinate. This random sampling can adhere to any distribution, as to emphasize important regions by sampling more often in those regions. However, a uniform distribution over the domain is most general, and no further distributions will be explored in this work.

Once a running average of the sample loss reaches a specified threshold or a maximum number of steps, the process is halted. Ideally, the resulting collection of Gaussians will cluster around regions with high information density – or a high local variance – since more turbulence is expected in these regions. Accordingly, since these regions require finer Gaussians during initialization to represent the high-frequency details, we expect this will occur naturally during training.

The similarity of our sampling with a mixture of Gaussians suggests that instead of gradient descent (GD), we could apply expectation maximization (EM) [3], which is often used for finding the maximum likelihood estimate for a mixture of Gaussians. However, the use of EM for Gaussian mixtures is typically limited to clustering, using at most a few dozen Gaussians. Although it should be possible to adapt EM to our use case, it is still an iterative optimization process and we do not expect much benefit compared to GD, so we have chosen to use GD for the initialization, which is well-established for a large number of Gaussians (i.e. 100K-500K [14]).

3.3 Dynamic Gaussians

There exist many different approaches for dynamically updating a collection of Gaussians, as described in Sections 2.3.1 and 2.3.2. However, these approaches are either direct simulations with knowledge of the solution function to the PDE, or are optimized for a single scenario. Instead, we require an approach that generalizes to different scenarios, without explicit a priori knowledge of the solution to the target PDE. To achieve this, we have chosen to use an autoregressive model that integrates information from the current state of the Gaussians and the PDE, and generates a prediction for the next state. To that end, we have opted to use a deep neural network that outputs the deltas (change) of the parameters for each Gaussian. This enables us to learn some implicit representation of the solution function to a PDE encoded in the network and generalize to multiple scenarios. An illustration of our method is shown in Figure 2.

Although completely continuous methods for dynamic Gaussians exist [21], these are designed for representing a single dynamic scene by fitting a curve to each Gaussian parameter, whereas we aim to to represent a multitude of scenarios based on arbitrary initial conditions. As such, an independent fitting of each Gaussian would not generalize well, nor enable the integration of neighborhood information. Furthermore, we expect the increased flexibility of a neural network compared to fitting a curve to be a requirement for complex physical phenomena. Another alternative would be to represent the PDE solution with Gaussians that extend into the temporal dimension [7], which would be based on the initial condition, similar to the 3D Fourier Neural Operator [17]. However, Gaussians do not extend into the time domain as well as Fourier decompositions, since Gaussians decay to zero and sine waves do not. As such, it would be difficult to imagine extending this to longer time sequences, especially since it is difficult to imagine adapting the Gaussians over time when they already extend into the temporal dimension. Instead, we require some form of discretization of the problem in the time domain, for which we use the backward Euler method, as described in Section 2.1. Nevertheless, it is possible to perform (linear) interpolation of the Gaussians between two time steps to gain an approximately continuous representation.

3.3.1 Model Architecture

An overview of the architecture of our network is shown in Figure 4. It is inspired by PointNet [29] in that it is divided into an encoder, an aggregation component and a decoder. This architecture enables an arbitrary number of Gaussians to aggregated and updated. The encoder in our network computes a latent vector for each Gaussian and its parameters, compared to points in PointNet. The aggregation component in our network aggregates latent vectors of the neighbors of each Gaussian separately, where PointNet aggregates all latent vectors into a single global latent vector. The decoder in our network is similar to the segmentation network of PointNet, where we compute the deltas for each Gaussian based on the local latent vector of that Gaussian, compared to computing the labels for each point based on the global latent vector in PointNet. A detailed description of the individual components is provided in the paragraphs below. An approach like this enables the use of an arbitrary number of Gaussians, but it does mean the model needs to be applied to each Gaussian separately. Furthermore, the separate application implies a lack of coordination between the Gaussians, which could be a limiting factor in terms of expressiveness. However, we expect that the local latent vector – which describes the state of the neighboring Gaussians – will enable the incorporation of the changes of its neighbors effectively. Furthermore, the principle of locality in classical physics ensures that any particle is directly influenced only by its surrounding particles, which means distant Gaussians are unlikely to affect one another on a small time scale and they need not be considered in the update.

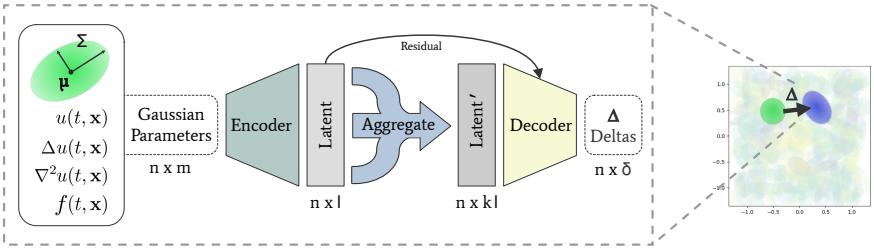
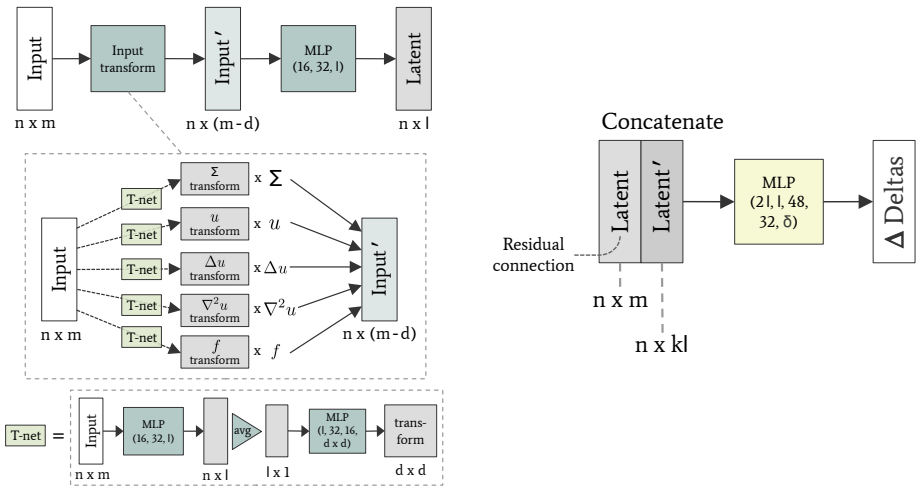


Figure 4: *Overview of the network architecture used in our method. Gaussian parameters are input into the network, then aggregated into a local latent vector, and lastly the network ouputs the change in Gaussian parameters* Δ*. Here* n *is the number of Gaussians,* m *the input size,* l *the latent size and* δ *the output size.*

Input Besides the Gaussian parameters, i.e. the mean $\boldsymbol{\mu}$, covariance Σ and value $\mathbf{u}$, additional information is computed to guide the model. Firstly, we compute the spatial derivatives – namely the first-order spatial derivatives and the Laplacian – of our solution at the location of the means of each Gaussian by accumulating the spatial derivatives of all Gaussians at those points. These spatial derivatives, along with the parameters describing the problem are then used to compute the right-hand side of the target PDE $f(t, \mathbf{x})$ at the same points [23]. All these input variables are combined to give the model a better understanding of the current state of the solution.



(a) *Encoder* **(b)** *Decoder*

Figure 5: *Architecture for the encoder and decoder components of the network. The encoder encodes the Gaussian parameters into latent vectors and the decoder decodes local latent vectors into the change in Gaussian parameters Δ. See Figure 4 for the role of these components in the network.*

Encoder & Decoder The encoder and decoder components are shown in Figures 5a and 5b, respectively. Besides some minor changes, mostly in the width and depth of the network, these components remain largely similar to the encoder and segmentation decoder networks from PointNet [29]. However, we have removed the batch normalization from each layer and replaced ReLU with the hyperbolic tangent for the activation, since we found these changes to work better on our problem spaces. The reason for the poor performance of ReLU can likely be explained by the large deltas produced by the network with this activation, which makes the model diverge rapidly and often during training.

For the encoder, more input parameters are considered, as mentioned above, which are all transformed by the input transform. Each input parameter has its own transformation matrix, which is constructed from an average pooling over all Gaussian parameters and the same transformations are used for each Gaussian. The goal of this input transform is to enable the model to learn invariances (e.g. different scales) of the problem. We do not consider a local neighborhood of Gaussians here, since we intend for the model to learn global invariances. Once all the input has been transformed, it is processed by an MLP to compute a latent vector for each Gaussian. Unlike PointNet, this latent vector is not transformed again, since the latent size L in our case is much smaller (64 in our case and 1024 in PointNet). It is important to note that we do not encode the mean $\boldsymbol{\mu}$ in the latent vector, since the absolute position in space is not relevant for the PDEs we are considering, so we discard it after transforming the input to prevent overfitting the model on the data. Instead, we wish to encode information about the position of a Gaussian relative to its neighbors, which will be described in

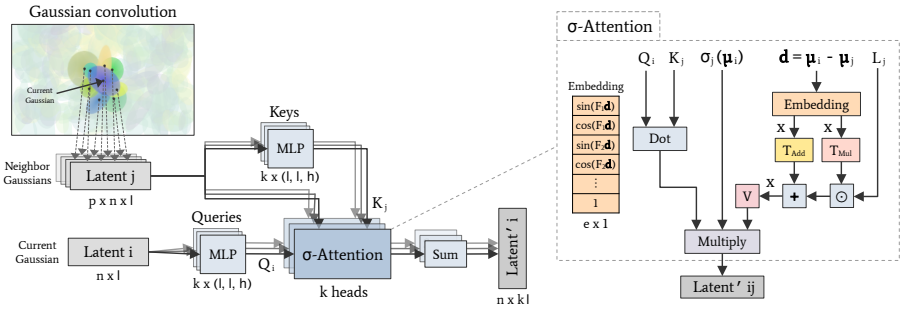


Figure 6: *Overview and architecture for the neighbor aggregation component of the network. Here, for each Gaussian, the overlapping neighbor Gaussians are aggregated into a local latent vector. This aggregation makes use of an attention-like [41] component to factor in the relevance of each neighbor. See Figure 4 for the role of this component in the network.*

the neighbor aggregation below.

For the decoder, we keep the skip connection from the individual Gaussian latent vectors, which encode the Gaussian parameters, since the deltas will be directly applied to (some of) these parameters. This means that for every Gaussian we combine its latent vector with the local feature vector describing its neighbors. This combined latent vector is then processed by a deep network to compute the output deltas. These output deltas represent the change in Gaussian parameters at each time step.

Neighbor Aggregation Unlike the encoder and decoder components, the neighbor aggregation significantly diverges from PointNet, which uses a simple max pooling to obtain a global latent vector. Instead, we aim to find a local latent vector for each Gaussian that describes its neighboring Gaussians. To that end, we perform a kind of convolution on the Gaussians, akin to DeltaConv [48], but with an additional component inspired by attention [41], which enables the network to attend to neighbors differently; we denote the component σ-attention. An illustration of the neighbor aggregation is shown in Figure 6.

We begin by convolving the Gaussians, which entails finding all p overlapping neighboring Gaussians for each Gaussian. Here, p is distinct for each Gaussian, which enables the model to generalize to arbitrary configurations. However, configurations where p falls outside the typical range seen during training are likely to be challenging, since the model might not be able to effectively aggregate the number neighbors and decode the resulting latent vector.

For every Gaussian, we apply k σ-attention heads on each of its neighbors, where k should be larger for more complex problems. This attention mechanism does not attend the neighbors with each other, only with the current Gaussian.

Before performing the σ-attention, each latent vector is processed by two shallow MLPs for each attention head: one for computing the keys $\mathbf{K}_j$ and one for the queries $\mathbf{Q}_i$, both using the latent vectors $\mathbf{L}_j$ from the encoder. This means for each Gaussian,

we obtain k key vectors and k query vectors of length h. The query vectors are used to 'query' the neighbors of a Gaussian through the corresponding key vectors, in order to determine the relevance of each neighbor, which will then be multiplied with its latent vector to compute the contribution of the neighbor to the local latent vector. As with regular attention, the dot product of these vectors $Q_i \cdot K_j$ then provides the relevance of neighbor Gaussian j for Gaussian i.

Next, the unnormalized neighbor latent vector $\hat{\mathbf{L}}_{ij}$ is computed by multiplying the $L \times L$ value matrix V with the the positionally embedded latent vector $\hat{\mathbf{L}}_{D_{ij}}$. This positional embedding is similar to the positional embedding used in transformers, but instead of embedding the absolute position, it embeds the distance to the current Gaussian. We thus denote it a distance embedding.

This embedding is formed by first multiplying (element-wise) the latent vector $\mathbf{L}_j$ with a learned embedding vector, and then adding a different learned embedding vector, in order to model a linear transformation. Both embedded latent vectors are created by first embedding the distance $\mathbf{d}_{ij} = \mu_i - \mu_j$, i.e. the vector from the mean of the neighbor to the mean of the current Gaussian. Then we multiply two trainable matrices of size $L \times E$ – one for the additive distance transform T_{Add} and one for the multiplicative distance transform T_{Mul} – with the embedding vector of size $E \times 1$.

The embeddings are shared across attention heads, but are evaluated between the Gaussians and each of its neighbors. We chose to use both an additive and a multiplicative embedding to allow the direction of the neighbor to affect the sign of certain latent features, which is impossible to achieve with only an additive embedding. The additive embedding is kept to still enable the distance to be added as a feature to the latent vector.

For the embedding vector, we use a Fourier feature mapping [30] that maps the low-dimensional distance vector to a high-dimensional embedding vector $\mathbf{D}(\mathbf{d}_{ij})$ as follows:

$$\mathbf{D}(\mathbf{d}_{ij}) = [sin(F_1\mathbf{d}_{ij}),\ cos(F_1\mathbf{d}_{ij}),\ \ldots,\ sin(F_E\mathbf{d}_{ij}),\ cos(F_E\mathbf{d}_{ij}), 1]^T$$

Where $F_i \sim \mathcal{N}(0, 10)$ is randomly initialized and not trainable, i.e. it remains static once initialized. We found these parameters to give a nice spread around the origin.

Next, the embedded latent vector is computed as follows:

$$\hat{\mathbf{L}}_{D_{ij}} = T_{Add}\mathbf{D}(\mathbf{d}_{ij}) + T_{Mul}\mathbf{D}(\mathbf{d}_{ij}) \odot \mathbf{L}_j$$

Where T_{Add} is the additive distance transform, T_{Mul} is the multiplicative distance transform, $\mathbf{L}_j$ is the latent vector of Gaussian j and $\odot$ is the element-wise product.

With that, we compute the attended latent vector as follows:

$$\hat{\mathbf{L}}_{ij} = (\mathbf{Q}_i \cdot \mathbf{K}_j) \cdot \alpha_j(\boldsymbol{\mu}_i) \cdot V\hat{\mathbf{L}}_{D_{ij}}$$

Where $\mathbf{Q}_i$ is the query vector of Gaussian i, $\mathbf{K}_j$ is the key vector of Gaussian j, α_j is the density of Gaussian j at the mean $\boldsymbol{\mu}_i$ of Gaussian i and V is the value matrix.

Lastly, the local feature vector is computed by simply summing over the latent vectors of all its neighbors and normalizing:

$$\mathbf{L}'_i = \frac{1}{\sum_j \alpha_j(\boldsymbol{\mu}_i)} \sum_j \hat{\mathbf{L}}_{ij}$$

Regular attention applies softmax over the weights of the individual latent vectors to normalize them relative to each other. However, we chose to not impose such strong constraints between the neighbors, as to enable them to contribute independent features to the local latent vector. Instead, we normalize by the total density, to enable processing any number of Gaussians, both distant and proximate.

Naturally, these computations have to be repeated separately for each of the k attention heads, after which the k feature vectors are simply be concatenated to arrive at the final latent vector describing the local neighborhood.

3.4 Pruning & Densification

The method defined thus far does not change the number of Gaussians after initialization (see Section 3.2), even though that number might not be sufficient for the entire simulation. As such, underdefined regions should be densified by adding Gaussians to those regions. Furthermore, redundant or unnecessary Gaussians should be pruned for efficiency, since the model will still need to update these Gaussians.

In 3D Gaussian Splatting [14], Gaussians are added and removed in a refinement step, where densification adds new Gaussians and pruning removes Gaussians. These steps are necessary for scene reconstruction because it would otherwise be difficult to determine the required number of Gaussians for a given scene beforehand and result in regions that are under- or overreconstructed. Furthermore, it addresses issues with local minima in Gaussian mixtures [5].

The refinement step in 3D-GS is applied periodically during training, which makes it unsuitable for our approach, since we do not perform any optimization steps after training, but we still need to prune and densify during inference.

Densification in 3D-GS is based on the gradients of the mean with respect to the loss. We can estimate the loss and as such the gradients of the mean, but this is an expensive process and the gradient of the mean is not necessarily a suitable metric for our method, since we are minimizing different types of loss terms. As such, we need to find a metric that works well in general (or at least for a specific PDE) and that can be computed per Gaussian efficiently. Experiments to find such a suitable metric are described in Section 5.3.

The splitting procedure is also slightly different from 3D-GS, as illustrated in Figure 7. Instead of moving the Gaussian in the direction of the gradient of the mean, we find the largest principal component of the Gaussian and create two identical copies of the original Gaussian, removing the original. The value of the two Gaussians is halved and they are displaced in the positive and negative direction of the principal component, with a magnitude of a single standard deviation. The largest principal component

is the of the eigenvalue and eigenvector corresponding to the maximum eigenvalue. We found no benefit in cloning the Gaussians.

Pruning in 3D-GS splatting is performed by removing Gaussians either with an α below a given threshold ϵ or with a scale that is 'too large'. This approach is applicable to our approach, except that we have combined the alpha with the color (value u in our case), so instead we check if the norm of the value $||u||_2$ is below a given threshold ϵ. Furthermore, although we can also remove Gaussians that are 'too large' we have not observed these to occur often during inference, and they are likely to be of importance to the solution rather than nuisance.

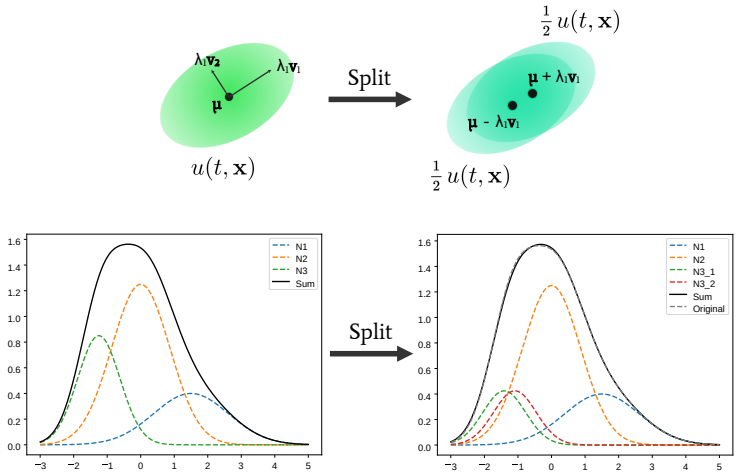


Figure 7: *Example of the splitting procedure used in our densification step, based on moving two copies with halved value in the positive and negative direction of the largest principal component of the original Gaussian. The procedure (in 2D) is illustrated on the top and an example (in 1D) is demonstrated on the bottom.*

3.5 Boundary Conditions

So far, we have ignored the boundary conditions involved in solving PDEs, as was described in Sections 2.1 and 2.2. Since the means of the Gaussians have not been included in the network for computing the latent vectors, there is no way for the model to minimize the boundary loss term $\mathcal{L}_{BC}$. Ideally, we want our model to generalize to arbitrary domains for the same PDE, which means including the mean would be insufficient, as the boundary conditions might change. As such, we need a more general method for providing information about the boundaries to the network. One possibility would be to add another variable to the input for each Gaussian, which is the vector from the mean to the nearest boundary. However, we have decided to go for a simpler and more robust approach that defines the boundary in terms of Gaussians, as shown in Figure 8. This means we do add another input variable for each Gaussian, but it

is simply a flag that indicates whether the Gaussian is part of the boundary or not. In this manner, the neighbor aggregation will include boundary Gaussians, which will provide information about the proximity of arbitrarily complex boundaries, as long as they are sufficiently well represented. Consequently, we need to initialize the boundary Gaussians in a suitable manner. However, since the boundary conditions are always known beforehand, we chose to define them manually. The automatic generation of these boundary Gaussians from the description of the initial condition is an interesting direction for future work. Another thing to note is that the boundary conditions themselves can be problem-specific, meaning that they are subject to change. To address this, we add another variable to the input of boundary Gaussians (and set it to zero for non-boundary Gaussians), which indicates the value of the boundary condition at that point, along with a flag that indicates whether it is a Dirichlet or Neumann boundary condition. However, to retain focus we limit the evaluations to exclusively use constant Dirichlet boundary conditions in this work (i.e. they do not change within the experiments).

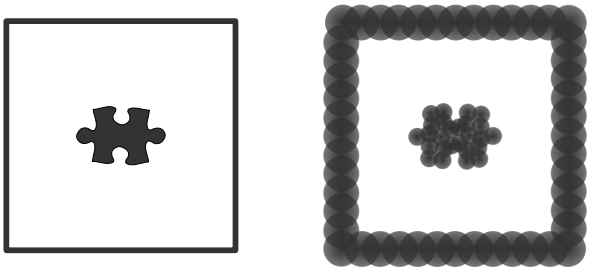


Figure 8: *Defining PDE boundary conditions (left) by means of additional Gaussians (right). In this example, the black box is the outer boundary, beyond which all points ought to adhere to the boundary conditions. The same holds within the puzzle piece in the center.*

4 Implementation

For the implementation we use PyTorch [27] and CUDA.

4.1 Efficient Gaussian Sampling

Our sampling approach described in Section 3.1 has to accumulate information from every Gaussian. This is clearly inefficient when dealing with a large number of Gaussians. To address this, we use an optimization similar to the fast 3D Gaussian splatting rasterization algorithm by Kerbl et al. [14].

First, the domain is tiled into equal squares based on the axis-aligned bounding box of the sample points. In 3D-GS, the domain is a 2D image on which the Gaussians are rasterized, but for our sampling approach, we generalize this optimization to any dimension by tiling hypercubes instead of squares. Next, each sample point is assigned to its respective tile. All Gaussians are then culled according to their 99th percentile. This ensures that Gaussians that contribute very little to a point – at most 1% of their density – are not considered in the accumulation. Typically, this means we sacrifice a small amount of accuracy for a large performance boost. Lastly, we apply the aforementioned sampling in parallel (on the GPU) for every tile.

Besides the difference in accumulation of Gaussians, our implementation differs from the 3D Gaussian splatting rasterization in that it deals with a continuous space, where arbitrary points are sampled in the same dimension as the Gaussians reside in. On the other hand, 3D-GS projects 3D Gaussians onto an arbitrary 2D plane with equally spaced points representing pixels.

4.2 Efficient Neighbor Aggregation

The aggregation of neighbors is split into three steps, as described in Section 3.3.1: a preprocessing step, a forward pass and a backward pass. First, in the preprocessing step, the neighbors of each Gaussian are found by checking for overlap. Then, in the forward pass, the σ-attention is applied to the neighbors of each Gaussian. Lastly, the backward pass simply computes the gradients of the output from forward pass with respect to the inputs. The preprocessing step only has to be applied once each time step, which is then used for the forward and backward passes of each of the k attention heads.

All Gaussians are evolved separately in our model, so we apply each of the steps of our aggregation in parallel (on the GPU) for every Gaussian. Furthermore, the k forward and backward passes for each attention head is also performed in parallel.

The forward and backward pass are difficult to optimize any further, since they are highly sequential. Instead, it makes sense to optimize the preprocessing step. A naive implementation simply checks all Gaussians to find out whether they overlap. We slightly simplify this overlapping computation, by assuming the Gaussians are spheres with a radius according to their largest principal component. Further optimization requires the use of a bounding volume that enables us to skip neighbors in different volumes. However, we found that even with 10k Gaussians, the preprocessing step consumes less than 10% of the computation time. As such, we decided to not optimize the implementation any further and focus our efforts elsewhere.

5 Experiments & Evaluation

In the following sections, we discuss various experiments to evaluate the applicability of Gaussians for modelling physical phenomena. This includes the representational quality of Gaussians in static and dynamic scenarios.

5.1 Static Representation

The representational quality of Gaussians has been demonstrated for various 2D [55] and 3D [14] scenarios. However, these scenes do not include turbulent examples that are commonplace in modelling physics. Here we describe such scenarios and validate the ability of Gaussians to represent them.

Typically, initial conditions used for modelling PDEs are smooth and relatively simple, such as the first two examples shown in FIgure 9. However, the solution to the PDE may become very turbulent over time, e.g. for the Navier-Stokes equations when using high Reynolds numbers (see Equation 1). In order to verify the representational quality of Gaussians for modelling PDEs, we will test both smooth and turbulent examples. To achieve this, we use the initialization of Gaussians described in Section 3.2 for various functions and images. For illustrative purposes, we only focus on one and two dimensions for our experiments and evaluation, but our method is by no means limited to these scenarios and is expected to generalize to arbitrary dimensions.

The three example functions and two turbulent example images used for this experiment are shown in Figure 9, alongside the sampled results and corresponding Gaussians. These solutions were each trained for 50k iterations, the final loss and total number of Gaussians for each example are shown in Table 1.

Example	Gaussians	Loss
Gaussian	120	9.6e-8
Sinusoid	389	2.8e-6
Abstract	11918	3.6e-2
Φ_{Flow} [39]	401	1.9e-4
Turbulence [47]	5534	1.5e-3

Table 1: *Loss and total number of Gaussians for the initialization examples shown in Figure 9.*

These results indicate that Gaussians are excellent for representing smooth structures, evident from the low losses on the 'Gaussian' and 'Sinusoid' examples using relatively few Gaussians. Furthermore, the more turbulent examples 'Φ_{Flow}' [39] and 'Turbulence' that are more representative of physical phenomena can also be reconstructed well through Gaussians. However, the 'Abstract' example shows the difficulty in representing sharp edges, as evidenced by the high loss, the needle-like Gaussians and the large total number of Gaussians used. Many of the corresponding Gaussians near the sharp discontinuities of this example are exceedingly anisotropic, i.e. the ratio between the largest and smallest principal components is large, which results in a bleeding effect into the smooth part of the 'Abstract' example. Since such sharp discontinu-

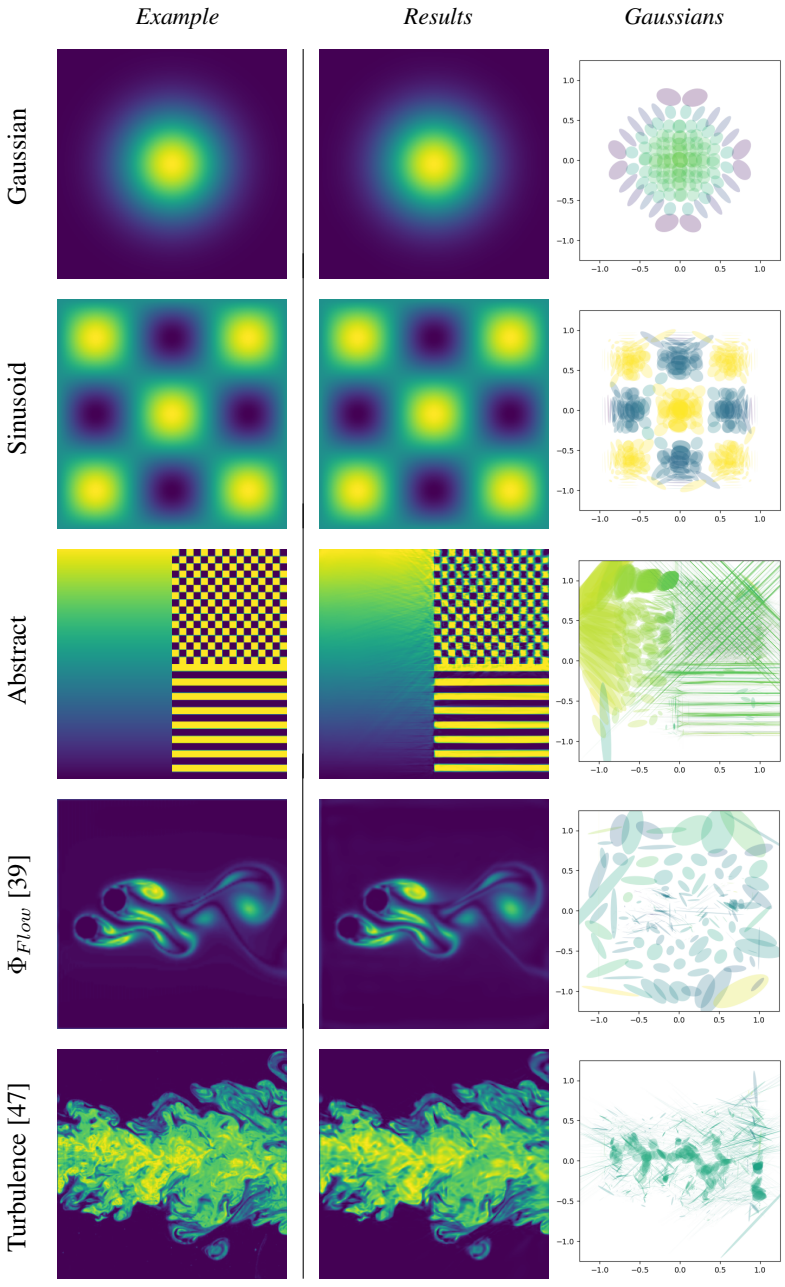


Figure 9: *Various functions and images used for validating the representational capacity of Gaussians for modelling physical phenomena, alongside the sampled results and corresponding Gaussians after training.*

ities are uncommon in physical phenomena, we do not further optimize our approach to address the anisotropic Gaussians, but regularization based on the aforementioned ratio could serve as a remedy in these cases [13]. As can be seen, especially in the 'Turbulence' example, the Gaussians indeed center around the regions with high information density, which will be beneficial for evolving the dynamics, as more change tends to happen around these regions.

In short, Gaussians appear suitable for representing (static) solutions to complex PDEs.

5.2 Dynamic Representation

Here we describe two types of optimization techniques for modelling time-dependent PDEs through dynamic Gaussians, such that we can validate the use of Gaussians for this purpose.

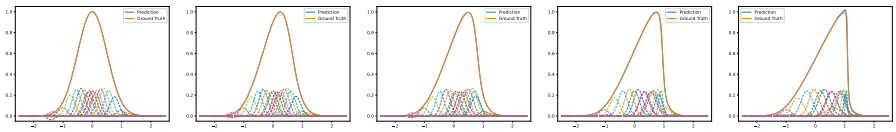
5.2.1 Explicit Optimization

In the following experiments, we perform an explicit optimization of a collection of Gaussians. The Gaussians are initialized as described in Section 3.2. Once the initial condition is sufficiently represented by the Gaussians, the solution to the desired PDE is explicitly optimized when the problem is discretized in time. This means we have a collection of Gaussians at each time step and use the Gaussians of the current time step as a starting point for inference of the representation of the next time step. The optimization is done through gradient descent with the physics-informed loss $\mathcal{L}_{PDE}$ (see Section 2.2) and our differentiable Gaussian sampling (see Section 3.1). In order to obtain a spatially continuous approximation of the solution to the PDE, we again sample random points within the domain and compute $\mathcal{L}_{PDE}$ at those points, and then minimizing this loss by optimizing the Gaussian parameters. For simplicity, no further loss-terms are used in this experiment (i.e. $\mathcal{L}_{BC}$ is not included here). For simplicity, we do not perform any pruning or densification during the optimization, such as described in Section 5.3.

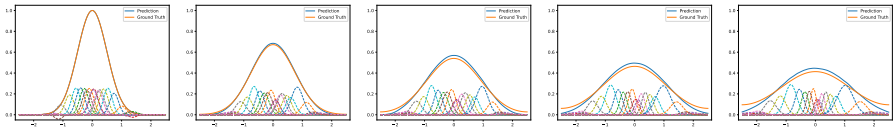
After optimization, we wish for the resulting Gaussians to smoothly represent the solution to the desired PDE function continuously in space and discretely in time for the specified initial condition. With this approach, a change in initial condition requires the Gaussians to again be optimized from the ground up, which makes it unsuitable as an effective surrogate model for PDEs. Nevertheless, it should suffice as to provide insights into the applicability of Gaussians for solving time-dependent PDEs.

We have performed this experiment on three 1D time-dependent problems: Burgers' equation, the diffusion equation and the wave equation. The results of this experiment are shown in Figure 10. In each problem, the standard normal distribution is used as the initial condition and a time step of $h_t = 0.05$ is used, though the results are displayed at different time steps (denoted Δt). Since each problem starts with the same number of Gaussians and we wish to compare the resulting solutions, we use 20 Gaussians for all three problems. The ground truths have been computed using a numerical solver.

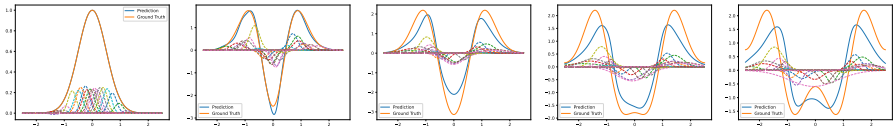
From these experiments, it can be seen that the Gaussians can represent the PDE solutions reasonably well for a variety of different equations. Nevertheless, the results are not perfect reconstructions of the ground truth, especially for the wave equation, which can likely be improved with a smaller time step or a different optimization method.



(a) *Burgers' equation,* $\Delta t = 0.25$.



(b) *Diffusion equation,* $\Delta t = 0.15$.



(c) *Wave equation,* $\Delta t = 0.1$

Figure 10: *Results for the explicit optimization experiments compared to the ground truth for three different PDEs. The ground truth is shown as a solid orange line and the prediction as a solid blue line, while the individual Gaussians making up the prediction are shown as dashed lines of various colors.*

5.2.2 Implicit Optimization

Since the aforementioned explicit optimization of Gaussians is insufficient as a general surrogate model for PDEs, a different approach is required. For this purpose, we propose a machine learning model that will dynamically update the parameters of each Gaussian (as described in Section 3.3), aiming to generalize to arbitrary initial conditions. To enable this, the model is trained on the desired PDE with various initial conditions that are representative of the expected initial conditions. This training happens in a similar manner to the explicit optimization, i.e. by sampling random points within the domain and optimizing the model on the physics-informed loss $\mathcal{L}_{PDE}$ at these points. However, we include additional loss terms: one for the boundary condition(s) $\mathcal{L}_{BC}$ (see Section 2.2) and another loss term for the similarity of the Gaussians $\mathcal{L}_{SIM}$. The similarity loss term forces the deltas output by the model to be small, such that the Gaussians do not change more than necessary to model the solution, which also serves to stabilize the training. This similarity loss is as follows:

$$\mathcal{L}_{SIM} = \frac{1}{N}\sum_{i}^{N}(\alpha_\mu||\delta\mu_i||_2^2 + \alpha_\Sigma||\delta\Sigma_i||_2^2 + \alpha_u||\delta u_i||_2^2)$$

Where $\delta\mu_i$, $\delta\Sigma_i$ and δu_i are the change in mean μ, covariance matrix Σ and value u of Gaussian i, respectively and α_μ, α_Σ and α_u are the corresponding weighting factors. We have found, through trial and error in our experiments, that $\alpha_\mu = 3$, $\alpha_\Sigma = 3$ and $\alpha_u = 1$ work well.

The total loss is then defined in the following manner:

$$\mathcal{L} = \alpha_{SIM}\mathcal{L}_{SIM} + \alpha_{PDE}\mathcal{L}_{PDE} + \alpha_{BC}\mathcal{L}_{BC}$$

Where α_{SIM}, α_{PDE} and α_{BC} are the weighting factors of the loss terms, where we have found $\alpha_{SIM} = 0.1$, $\alpha_{PDE} = 1$ and $\alpha_{BC} = 1$ to work well.

The similarity loss is averaged over all Gaussians, and the PDE and boundary losses are averaged over their own set of random points, which are re-sampled every epoch of training. An example of these two sets of randomly sampled points within a square domain are shown in Figure 11. To ensure that Gaussians are not susceptible to skipping over the boundary to avoid inducing a loss, we sample the boundary points not only on the boundary, but also further outside the domain where strictly neither the PDE nor the boundary conditions apply. Since we only make use of constant Dirichlet boundary conditions, this implementation is sufficient. However, for more complex boundary conditions, an additional loss could ensure that Gaussians do not move outside of the domain.

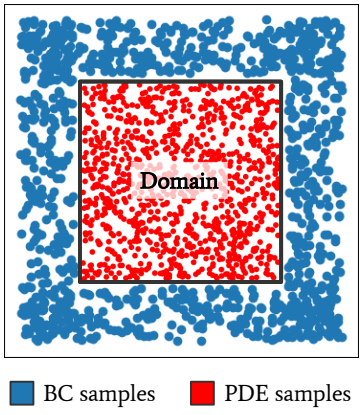


Figure 11: *Two sets of random samples used for computing the physics-informed loss $\mathcal{L}_{BC}$ (red, inside the domain) and the boundary condition loss $\mathcal{L}_{PDE}$ (blue, outside the domain and on the boundary).*

Every epoch of training, we train the model for a predefined number of steps $k = T/h_t$, depending on the desired time resolution T and time steps h_t. The loss

is computed on each of these steps separately and, although possible, we chose not to propagate the gradient to previous time steps, to save on memory and training time. Moreover, due to the vanishing gradient effect, the benefit of propagating the gradient to previous time steps is expected to be marginal compared to the increased computation time.

In order to ensure that the loss of earlier time steps in the epoch is not sacrificed for later time steps, we perform an accumulative (negative) exponential weighting of the loss based on previous time steps [46].

We only train the model on the 2D case for these experiments and we use a unit domain (x and y range between -1 and 1). Furthermore, no boundary conditions, nor pruning or densification will be used in these experiments. These will instead be explored in Sections 5.4 and 5.3, respectively.

Since we chose not to train on any ground truth data for our model, we have to find suitable initial conditions that enable the model to exploit the Gaussians effectively. This initialization does not use any pre-defined initial condition, so we instead need to find initial values for the Gaussian parameters. To that end, we experimented with various initialization methods for the training, namely 'Uniform', 'Uniform Noise' and 'Random', see Figure 12. All random sampling in this methods is from a standard normal distribution. For the 'Uniform' method, we distribute the Gaussians uniformly in the domain with equal variance and no covariance, then assign their values according to a normal distribution with a randomly sampled mean and variance. This variance is expected to be at least twice as large as the variance of the individual Gaussians. The 'Uniform Noise' method is based on the 'Uniform' method, but it always uses the standard normal distribution for assigning the values. Then, a small amount of randomly sampled noise is added to the parameters of each Gaussian. Lastly, the 'Random' method assigns all Gaussian parameters entirely randomly. For the last two methods, we apply the hyperbolic tangent to the means of the Gaussians, in order to prevent them being initialized outside the domain. All three methods have a random (square) number of Gaussians chosen uniformly (between 225 and 1600) alongside the other randomization. The randomization is applied every epoch.

Burgers' equation We train one models on each of the three different initialization methods for 3k epochs on the Burgers' equation with $\nu = \frac{1}{10\pi}$ and $h_t = 1.0$. Then, the first evaluation of the trained models is performed on the 'Uniform' method the standard normal distribution with 30×30 Gaussians. A second evaluation is performed on the Gaussians obtained from the 'Gaussian' initialization described in Section 3.2. Lastly, another evaluation is performed on an instance from the 'Uniform Noise' method, but without perturbing the values u (the same instance is used for each model). The final losses on the evaluations are shown in Table 2. The resulting solutions and corresponding Gaussians for the uniform evaluation are shown in Figure 13.

It is clear from these results that the 'Random' method performs the worst and does not learn the dynamics well. On the uniform evaluation, the 'Uniform' method performs the best of all three methods. However, on the noisy and initial evaluation, the 'Uniform Noise' method significantly outperforms the 'Uniform' method. Since

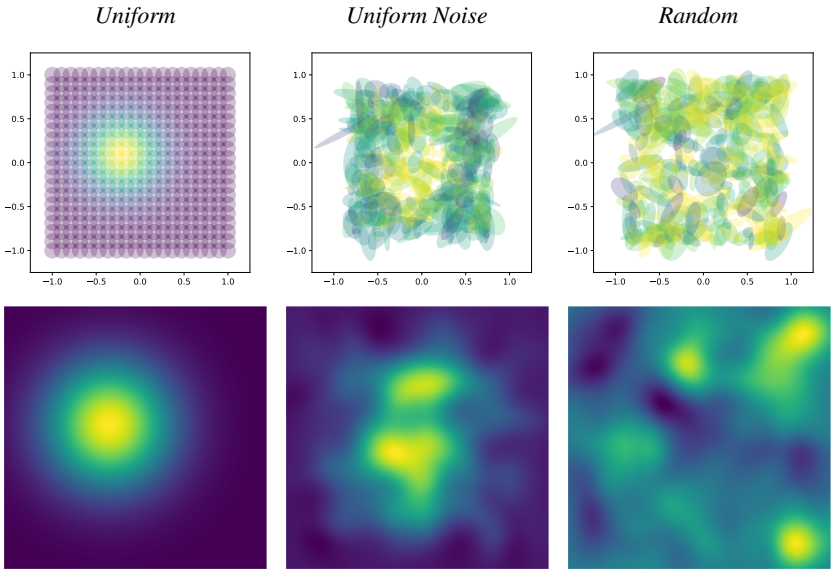
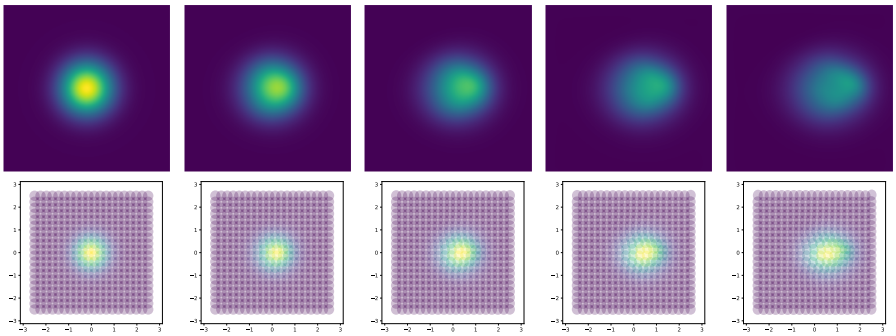


Figure 12: *Three different initialization methods we experimented with. Each is initialized with* 400 *Gaussians in the unit domain.*

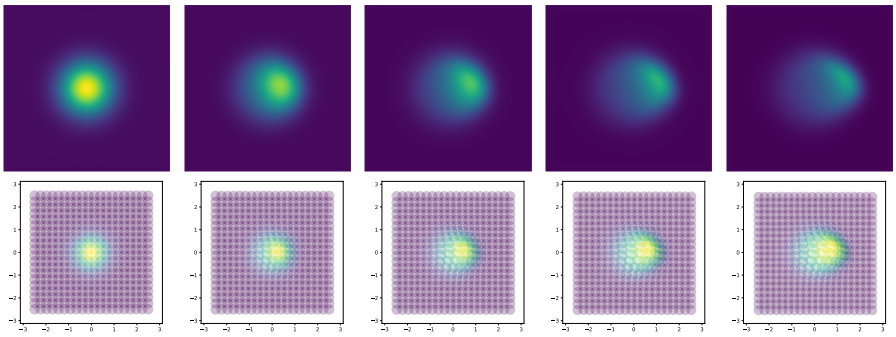
Method	Evaluation 1	Evaluation 2	Evaluation 3
'Uniform'	**48.879**	91.417	219.15
'Uniform Noise'	75.458	**73.221**	**45.336**
'Random'	117.39	99.033	51.915

Table 2: *Losses on the evaluation of the three different initialization methods shown in Figure 12.*

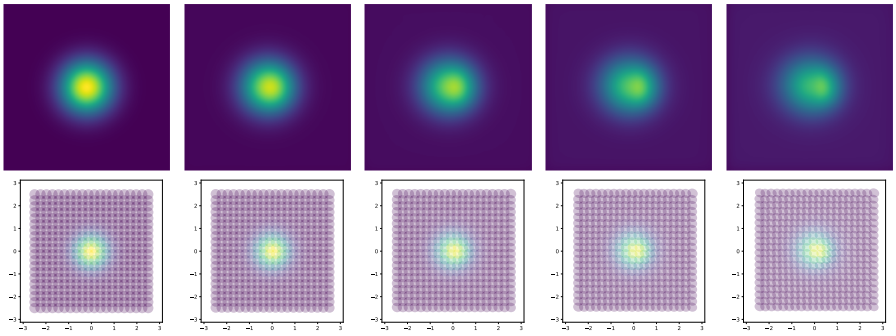
we do not expect a uniform distribution of Gaussians in practice and want our method to generalize and be able to integrate the addition or removal of Gaussians (see Section 5.3), we have chosen to train the Burgers' equation model using the 'Uniform Noise' method.



(a) *Uniform*



(b) *Uniform Noise*



(c) *Random*

Figure 13: *Results of the three different initialization methods shown in Figure 12 on the uniform evaluation.*

Navier-Stokes equations To enable the comparison of our method to other methods and to ground truth data, we train our model on the Φ_{Flow} [39] dataset generated by Li et al. [17]. This dataset contains nearly ground truth data generated from a numerical solver. We use this data to generate initial Gaussians with our initialization method described in Section 3.2. However, the domain for this problem is defined on the surface of a torus as shown in Figure 14, which means there are no boundaries and no boundary conditions. Furthermore, the data only expresses the vorticity $w(\mathbf{x}) = \nabla \times u(\mathbf{x})$, but the underlying PDE is defined on the velocity $u(\mathbf{x})$ (see Section 2.1), which has to be converted when initializing and evaluating.

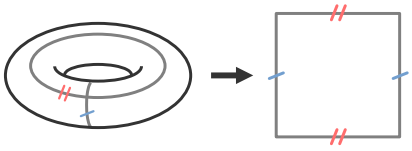


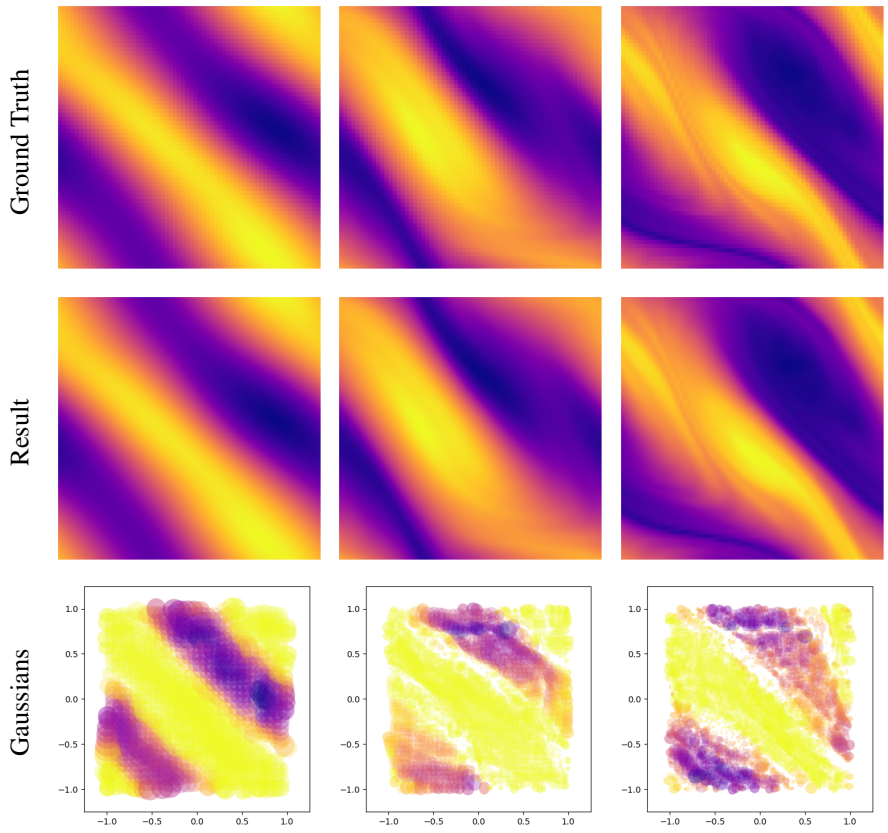
Figure 14: *Depiction of the surface of a torus unwrapped onto a 2D image.*

To enable the Gaussians to reside on the surface of a torus, we ensure that they are splatted beyond the image boundaries, i.e. across the opposite boundaries. Moreover, when the center of any Gaussian exceeds an image boundary, it is teleported to the opposite boundary. Training the model remains unchanged, as rendering beyond the boundaries remains entirely differentiable.

Due to the complexity of training our model on the Navier-Stokes equations using exclusively the aforementioned losses, we have decided to add an additional reconstruction loss term that makes use of the ground truth data. This loss term is as follows:

$$\mathcal{L}_{Recon} = MSE(\tilde{w}(t, \mathbf{x}) - w(t, \mathbf{x})), \qquad \mathbf{x} \in \Omega$$

Here $\tilde{w}(t, \mathbf{x})$ and $w(t, \mathbf{x})$ are the ground truth data and the prediction of our model, respectively, at time t and position $\mathbf{x}$. In a like manner to the $\mathcal{L}_{PDE}$ and $\mathcal{L}_{BC}$ (see Section 2.2), we approximate this reconstruction loss by averaging over many uniform random samples within the domain. Since the ground truth data only has a limited resolution of 64×64, but we wish our solution to be continuous, we compute the loss by finding the nearest ground truth pixel to the sampled point. Other interpolation methods (e.g. linear or cubic) are also possible, but any method enforces some (likely incorrect) structure on the data, so we chose the simplest method. An example of the initialization for three different values of ν are shown in Figure 15. These figures demonstrate that the initialized state can interpolate both smooth ($\nu = 10^{-3}$ and $\nu = 10^{-4}$) and sharp ($\nu = 10^{-5}$) features to form a continuous representation of the low resolution ground truth data.



(a) $\nu = 10^{-3}$ **(b)** $\nu = 10^{-4}$ **(c)** $\nu = 10^{-5}$

Figure 15: *Example of initial conditions used for the evaluation of the Navier-Stokes model along with the results after initialization for various values of ν. Note that the ground truth examples are 64×64 pixels, so the intermediate values have to be interpolated during initialization.*

5.3 Pruning & Densification

Here we test various criteria for densification, along with the effect of pruning. We perform the experiments on each of our PDEs of interest separately, in order to find the best criterion specific to each PDE. The process for pruning and densification is described in Section 3.4.

Burgers' equation For the Burgers' equation, we propose three distinct criteria and some of their combinations, which are each computed at the Gaussian means, and the top 2nd-percentile is then split. These criteria are the change of sample value between two time steps $\dot{u}$, the Laplacian $\nabla^2 u$ and the inverse total density σ_{Sum}. We perform the evaluations on a model trained with the aforementioned initialization for 5k epochs. The resulting physics-informed losses for these criteria on various different evaluations are shown in Table 3. From these results we find that the $\sigma_{Sum} \cdot \dot{u}$ criterion performs best for the Burgers' equation. An example of this densification is shown in Figure 16.

Criterion	Evaluation 1	Evaluation 2	Evaluation 3	Evaluation 4
No split	75.458	73.221	45.336	141.87
$\dot{u}$	72.750	69.695	**44.578**	143.07
$\nabla^2 u$	67.726	70.605	46.157	148.42
$\dot{u} \cdot \nabla^2 u$	71.801	69.794	45.842	144.91
σ_{Sum}	75.248	73.036	45.094	142.27
$\sigma_{Sum} \cdot \dot{u}$	72.000	**68.284**	44.627	**140.61**
$\sigma_{Sum} \cdot \nabla^2 u$	**66.949**	71.806	46.092	143.75

Table 3: *Losses for the densification criteria on various evaluations for Burgers' equation.*

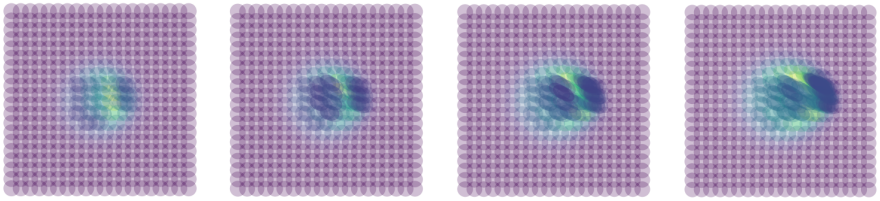


Figure 16: *Example of the densification of Gaussians for Burgers' equation based on the splitting criterion that multiplies the time derivative of the solution with the density* $\sigma_{Sum} \cdot \dot{u}$.

Navier-Stokes equations For the Navier-Stokes equations, we use the same criteria as for the Burgers' equation, but we consider additional criteria, since our Navier-Stokes model expresses the solution as a vector field instead of a scalar field, on which additional operators can be applied. These criteria are the curl $\nabla \times u$ and the divergence $\nabla \cdot u$. Additionally, we evaluate the criteria on u as well as on $w = \nabla \times u$ when possible. We perform the evaluations on a model trained with the aforementioned initialization

for 5k epochs on the $\nu = 10^{-3}$ data. The resulting physics-informed losses for these criteria averaged over different samples from the $\nu = 10^{-4}$ dataset are shown in Table 4. From these results we find that the σ_{Sum} criterion performs best on average for the evaluated Navier-Stokes equations. In fact, it is the only criterion that performs better than the baseline without splitting.

Criterion	$\mathcal{L}_{PDE}$
No split	749.66
$\dot{u}$	1080.6
$\nabla^2 u$	1273.0
$\nabla \times u$	803.70
$\nabla \cdot u$	795.86
$\dot{w}$	739.22
$\nabla^2 w$	794.27
σ_{Sum}	**723.38**
$\sigma_{Sum} \cdot \dot{u}$	791.15
$\sigma_{Sum} \cdot \nabla^2 u$	1038.2
$\sigma_{Sum} \cdot \nabla \times u$	787.43
$\sigma_{Sum} \cdot \nabla \cdot u$	764.09
$\sigma_{Sum} \cdot \dot{w}$	736.81
$\sigma_{Sum} \cdot \nabla^2 w$	763.64

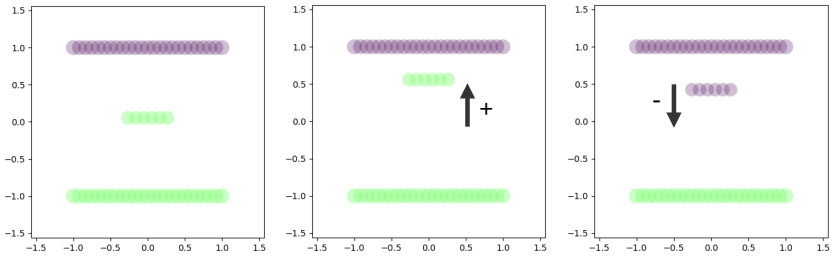
Table 4: *Losses for various densification criteria averaged over different samples from the $\nu = 10^{-4}$ dataset for the Navier-Stokes equations.*

It is important to note that for neither set of equations, the decision of the best criterion is unanimous between the evaluations, even when using the same parameters, which means different problems might benefit from different criteria. Additional criteria not discussed here could also be advantageous, which is further discussed in Section 7.3.

5.4 Boundary Conditions

Here we verify whether the boundary Gaussians described in Section 3.5 have the desired effect of providing boundary information to the dynamic Gaussians model. To that end, our first experiment is a contrived example that does not make use of a physics-informed loss. Instead, the goal is to repel/bounce the Gaussians away from the (horizontal) boundaries. The setup of this experiment is shown in Figure 17a and the desired outcome in Figures 17b and 17c.

To train this example, we create a new loss term that penalizes the difference between the (1D) value of the Gaussian and its change in y, i.e. the Gaussians ought to move in the direction of their value. Furthermore, the top boundary condition (at $y = 1$) is equal to -1 and the bottom boundary condition (at $y = -1$) is equal to 1, such that the Gaussians change direction – they are repelled – when they reach the boundaries. We still use the same conservation loss $\mathcal{L}_{CON}$ as before, to reduce unnecessary change in movement, except in the y-direction, and change in value, except at the boundaries. This means the Gaussians should solely move in the y-direction, indefinitely bouncing between the top and bottom boundaries.



(a) *Initial Gaussians* **(b)** *Gaussians before reaching the top boundary* **(c)** *Gaussians after reaching the top boundary*

Figure 17: *Experimental setup and expectations for testing the boundary Gaussians. We expect the Gaussians to bounce between the top and bottom boundaries, without movement in the x-direction, or any change in parameters besides the value and y-position. The Gaussians shown in the center are the only non-boundary Gaussians.*

The results for the aforementioned experiment are shown in Figure 18. Although the Gaussians do not exactly align with the expectations (since this would require careful tuning of the losses and hyperparameters), we can see the Gaussians are repelled by the boundaries as desired. This means the model can effectively integrate information from neighboring boundary Gaussians.

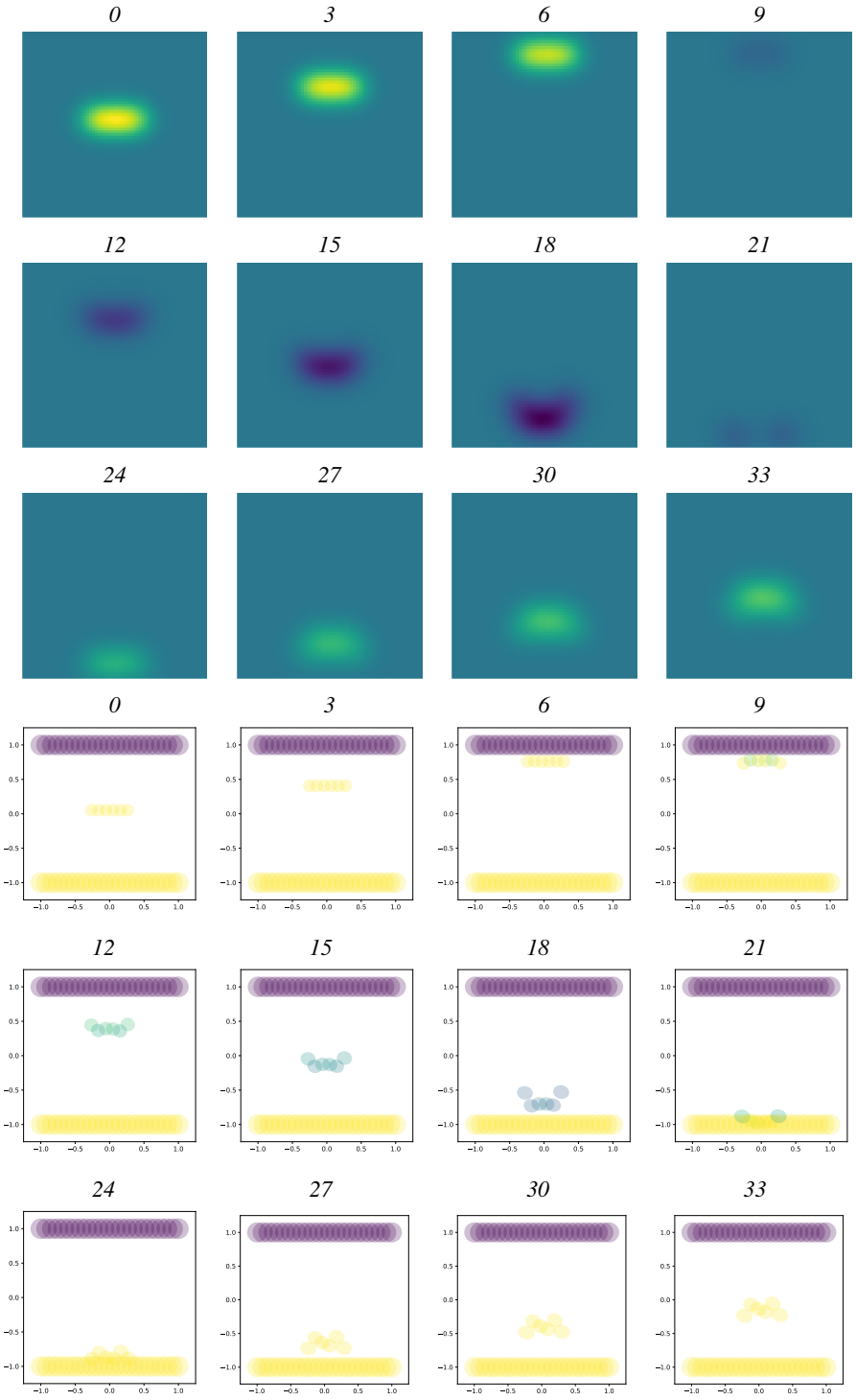


Figure 18: *Results for the boundary condition experiment illustrated in Figure 17. The sampled results are shown on the top and the corresponding Gaussians on the bottom. Frame numbers are shown above each figure.*

5.5 Burgers' Equation Evaluation

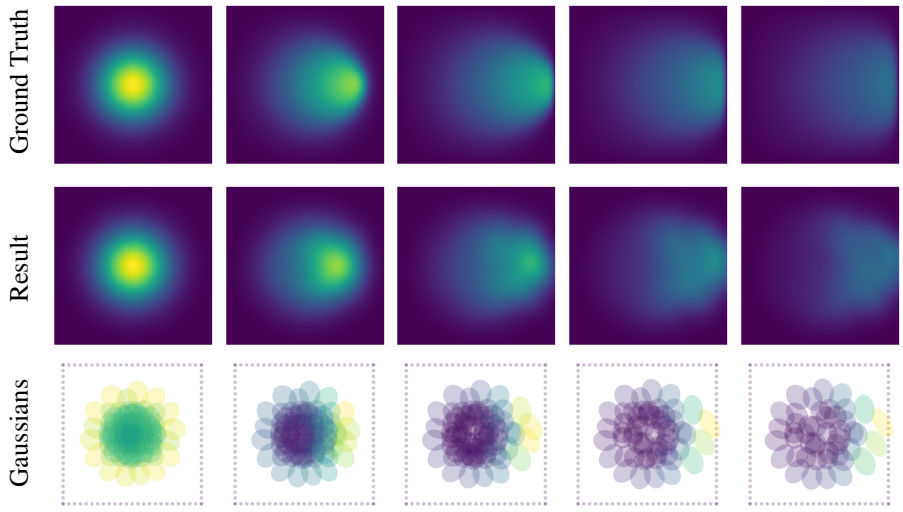
We evaluate our model trained on Burgers' equation by means of a comparison to numerical data. This numerical data is generated with *py-pde* [57], which should produce data close to ground truth solutions when using a fine discretization. The model has not been trained on any ground truth data, only on the physics-informed loss term and auxiliary loss terms, and the same model is used for all evaluations. The metric used for the comparison is the relative L^2-norm, normalized by the L^2-norm of the ground truth data. In addition, we list the total physics-informed loss $\mathcal{L}_{PDE}$ for the evaluations. We use a latent vector size of $L = 64$, an embedding size of $E = 25$ and $k = 2$ attention heads.

We devise various example problems on which the evaluation is performed. Each problem includes boundaries at the edge of the 5×5 domain and the initial Gaussians are generated through the initialization discussed in Sections 3.2 and 5.1. First, evaluation 1 is a simple 2D Gaussian distribution. Second, evaluation 2 contains two 2D Gaussian distributions, a narrow one to the left, and a large one to the right. The results of these evaluations are shown in Table 5. Predictions from our model for all three problems, alongside the ground truth data, are shown for the various time steps in Figure 19. For a runtime comparison with the numerical solver, the *py-pde* [57] solver takes around 30 seconds to solve 10 time steps on an Nvidia Tesla V100 GPU, whilst our method takes less than 2 seconds on the same hardware.

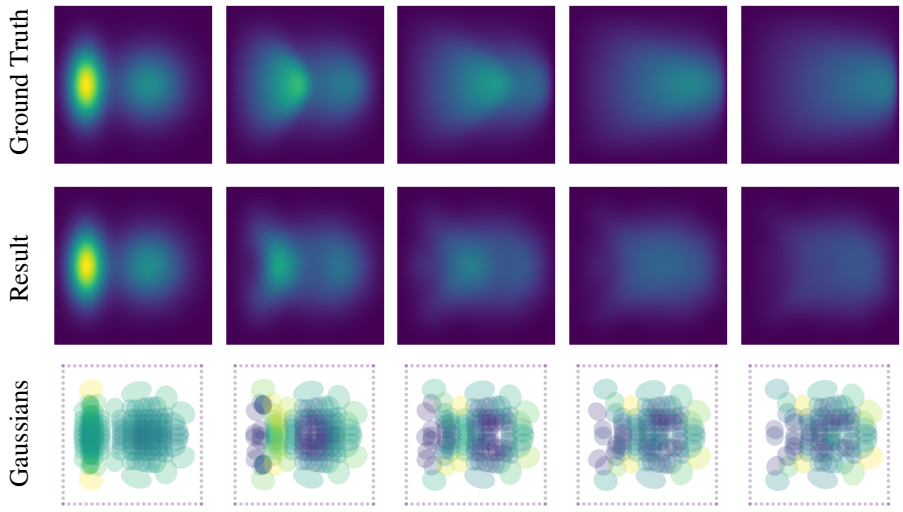
	Evaluation 1	Evaluation 2
L^2-norm	0.1616	0.2611
$\mathcal{L}_{PDE}$	45.551	59.342

Table 5: *Evaluation of our model trained on Burgers' equation in terms of the L^2-norm relative to the ground truth data and in terms of the total physics-informed loss.*

From these results, we find that the simple case (evaluation 1) can be solved with high quality, but the more difficult case (evaluation 2) is not approximated as well by our method, especially in the early time steps. Besides the increased difficulty, this can also be attributed to the similarity of the two evaluations to the training data (see Section 5.2.2), where the evaluation 1 is similar to the training data, but an example similar to evaluation 2 is unlikely to have appeared often during training.



(a) *Evaluation 1*



(b) *Evaluation 2*

Figure 19: *Evaluation results of our model train on Burgers' equation using physics-informed loss without ground truth data.*

5.6 Navier-Stokes Equations Evaluation

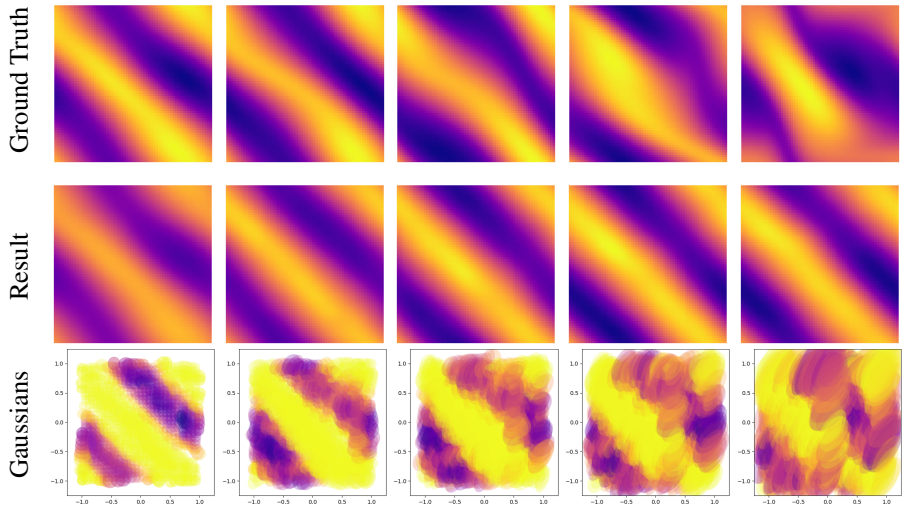
For the evaluation of our model on the Navier-Stokes equations for incompressible fluids, we compare our predictions to data produced using Φ_{Flow} [39] by Li et al. [17]. Again, this comparison uses the relative L^2-norm with the numerical data – which should be close to ground truth data. This same data has been used by various state of the art methods, such as LE-PDE [50], FNO-2D [17] and FNO-3D [17] (here, the third dimension is the temporal dimension, not an additional spatial dimension). Moreover, we also compare with evaluations of TF-Net [45], U-Net [34] and ResNet [11]. These additional evaluations have been performed by Li et al. [17]. All of the aforementioned methods can integrate multiple past time steps, which our method is currently not capable, as will be further discussed in Section 7.2. For the evaluations, these methods integrate the first 10 time steps of the ground truth as input, and predict the appropriate number of proceeding time steps. At an attempt to make the comparison between our method as fair as possible, we decided to start our prediction from the tenth time step of the simulation (i.e. $T = 9$). The evaluations have been performed on various different kinematic viscocities, namely $\nu = 10^{-3}$, $\nu = 10^{-4}$ and $\nu = 10^{-5}$, giving respective Reynolds numbers of approximately $Re = 10^3$, $Re = 10^4$ and $Re = 10^5$ (see Section 2.1). Unlike with the Burgers' equation model, here we train a separate model for each evaluation dataset, to keep in line with the other methods. We use a latent vector size of $L = 64$, an embedding size of $E = 33$ and $k = 4$ attention heads.

The results are shown in Table 6 and example predictions of our model for various time steps and values of ν are shown in Figure 20, alongside the ground truth data for the same time steps. The runtime is computed on the $nu = 10^{-3}$ model run for 40 time steps, where 'full' includes updating and rendering each time step and 'evo' only includes the rendering of the final time step. For a runtime comparison with a numerical solver, the Φ_{Flow} solver takes in the order of several minutes to produce the longest data sequence ($T = 50$) on high-end hardware [50]. Depending on the complexity of the initial condition, the initialization of our method – which is not included in Table 6, since it can be precomputed – takes approximately 10 seconds on an Nvidia Tesla V100 GPU. As such, the initialization would form a significant bottleneck in real-world use, which is further discussed in Sections 7.2 and 7.3.

These results demonstrate that our model cannot learn the long-term dependencies of the complex partial differential equations well in its current state. We also perform an additional evaluation of a model trained only on the first 5 time steps of the $\nu = 10^-3$ dataset, to investigate the performance on short-term prediction, of which an example is shown in Figure 21. These results, with an L^2-norm of 0.4129, demonstrate that the model can learn the dynamics much better on a shorter time span. As such, we expect more fine-tuning of the model might enable it to produce higher quality results in the long-term as well, which will be discussed in Section 7.

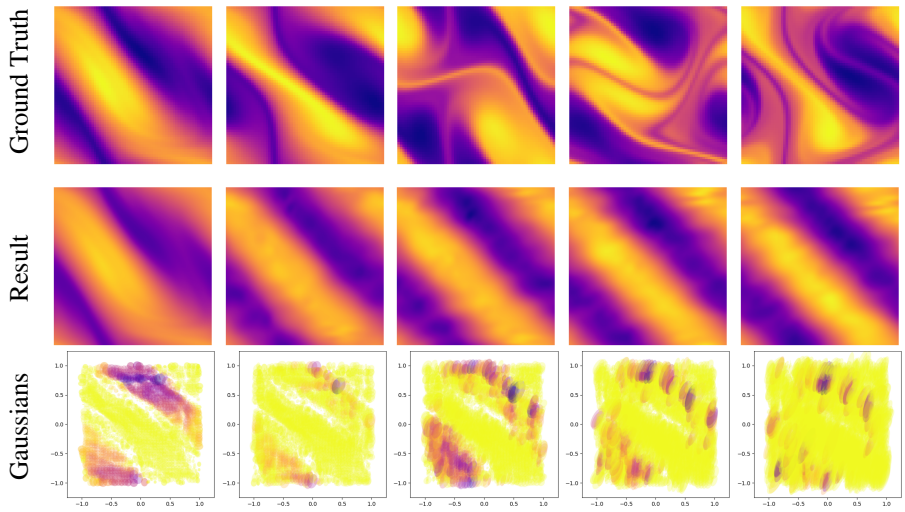
Method	Runtime full (ms)	Runtime evo (ms)	$\nu = 10^{-3}$ $T = 50$ $N = 1000$	$\nu = 10^{-4}$ $T = 30$ $N = 1000$	$\nu = 10^{-5}$ $T = 20$ $N = 1000$
FNO-3D [17]	24	24	0.0086	0.1918	0.1893
FNO-2D [18]	140	140	0.0128	0.1559	0.1556
U-Net [34]	813	813	0.0245	0.2051	0.1982
TF-Net [45]	428	428	0.0225	0.2253	0.2268
ResNet [11]	317	317	0.0701	0.2871	0.2753
LE-PDE [50]	48	15	0.0146	0.1936	0.1862
Ours	10003	9913	0.5385	0.6037	0.4452

Table 6: *Evaluation of our method on the Navier-Stokes equations using the relative L^2-norm for various numerical datasets produced with PhiFlow [39]. These results are compared with evaluations of various different methods on the same dataset [17, 50].*

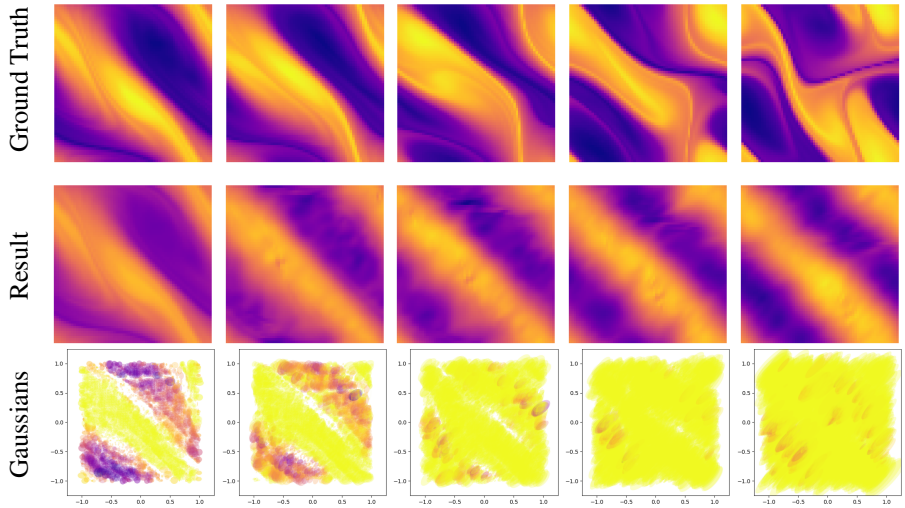


(a) $\nu = 10^{-3}$

Figure 20: *Continued on the next page...*



(a) $\nu = 10^{-4}$



(b) $\nu = 10^{-5}$

Figure 20: *(Continued) Long-term evaluation results of our model trained on the Navier-Stokes equations using physics-informed loss and ground truth data for different values of* ν.

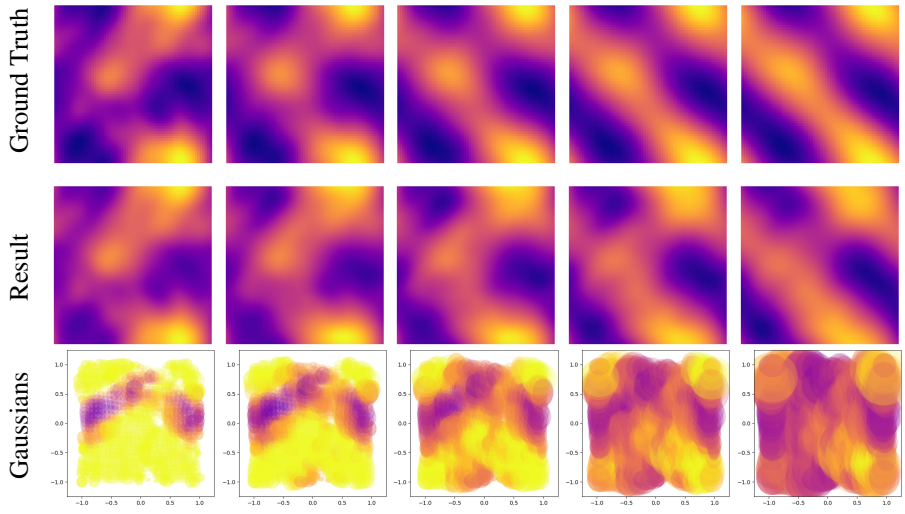


Figure 21: *Short-term evaluation results of our model trained on the Navier-Stokes equations using physics-informed loss and ground truth data for* $\nu = 10^{-3}$.

5.7 Ablations

Although we have constructed most of our method step by step, the end result might contain components that have little to no added benefit, or even reduce the performance of the model. In order to verify this, we ablate different components of our architecture individually. For these ablations, we use the Burgers' model trained as before and compute the physics-informed loss $\mathcal{L}_{PDE}$ and the relative L^2-norm as before on the same evaluations. We train each of the ablated models for 5k epochs. The results for the ablations of various components is shown in Table 7.

These ablations show us that the current architecture does not contain any redundant components. The model performs the worst without the residual connection from the encoder, i.e. the per-Gaussian latent vector, to the decoder, which makes sense, since it communicates to the decoder what the current Gaussian parameters contain. The neighbor aggregation makes a surprisingly small difference for the first evaluation, likely because without it, the model learns to simply move the Gaussians to the right based on their current value. However, for the more difficult second evaluation, the absence of the neighbor aggregation significantly reduces the performance of the model. Lastly, the input transformation does not appear to play a substantial role in the architecture, but it ablating it still negatively affects the performance of the model.

Ablation	Evaluation 1 ($\mathcal{L}_{PDE}$)	(L^2-norm)	Evaluation 2 ($\mathcal{L}_{PDE}$)	(L^2-norm)
No Ablation	45.551	0.1616	59.342	0.2611
Input Transformation	68.176	0.1762	65.272	0.2917
Residual connection	264.871	1.386	555.931	1.731
Neighbor Aggregation	40.261	0.2263	243.27	0.7089

Table 7: *Ablations of various components of the architecture and their corresponding PDE loss and L^2-norm for the Burgers' equation evaluation.*

6 Conclusion

In this work, we have proposed a new method for approximating partial differential equations (PDEs), making use of physics-informed machine learning and Gaussian splatting. We represent the solution as a set of Gaussians, which are updated by an autoregressive machine learning model over discrete time steps. The Gaussians are combined into a single mixture function at every time step, representing the solution to the target PDE, and they can be interpolated between the time steps. This representation is continuous, meshless and interpretable, making it distinct from most contemporary methods with similar objectives. By aggregating information from a local neighborhood of Gaussians, the Gaussian parameters are updated. The model can be trained with and without ground truth data and learn the dynamics of arbitrary PDEs.

Extensive experiments and evaluations of various aspects of our method have demonstrated its potential for solving PDEs. Although a comparison to other physics-informed machine learning methods on the Navier-Stokes equations has revealed that our method does not currently match state of the art methods when it comes to long-term prediction, it does perform better on short-term prediction. Additional fine-tuning and improvements are likely required to address this and other short-comings. Nevertheless, we believe the advantages of our approach, alongside its applicability to Gaussian splatting techniques, makes this work a valuable first step in the direction of solving PDEs with Gaussians.

7 Discussion

The results and experiments shown in previous sections demonstrate the potential for Gaussians to represent physical phenomena in the form of solutions to partial differential equations. We have demonstrated that our method can learn purely from the physics-informed loss, but also from ground truth data. A comparison of our model with data from numerical solvers (see Sections 5.5 and 5.6), which is close to ground truth data, reveals that our method can reproduce these results convincingly on small time scales, but diverges on larger time scales due to the accumulating error and uncertainty. Furthermore, we have compared our results with state of the art methods (see Section 5.6), particularly LE-PDE [50], FNO-2D [17] and FNO-3D [17]. From this, we have found that our method does not match the quality of the other methods on long-term prediction, in part due to the unfair comparison, but it does again demonstrate a reasonable quality can be achieved in short-term prediction by our method.

In the following, we discuss the advantages and limitations of our approach and the use of Gaussians for the purpose of solving PDEs in general. Additionally, we provide possible improvements to our method that can form the basis for future work.

7.1 Advantages

There are various advantages to the use of Gaussians for solving PDEs, which we elaborate upon here. Firstly, a mixture of Gaussians is completely continuous, which enables sampling the solution at any spatial resolution. Furthermore, by interpolating the deltas, the solution can also be sampled at different temporal resolutions. This continuity is in contrast to many other PINNs that solve on a discretized grid [15, 23, 43, 32]. Nevertheless, other continuous methods do exist [17, 44], and even discretized methods can be made pseudo-continuous through interpolation. Next, the use of Gaussians as an intermediate representation makes our method more interpretable than most other PINNs that are completely black-box. Namely, the Gaussians can be interpreted as particle-like (especially in the case of fluid dynamics). Another advantage is that our approach does not require a mesh (such as a grid), only knowledge about the domain and its boundary. However, other methods that work on arbitrary meshes also exist [18]. Lastly, our method can potentially be integrated with Gaussian splatting techniques to provide physically accurate simulations of 3D scenes (see Section 7.3 for more details).

7.2 Limitations

Although Gaussians have many advantages, there are also clear limitations to our approach that should not be overlooked. Firstly, the quality of the results in part depend on the number of Gaussians that are available. Too many Gaussians and it becomes challenging to update all of them simultaneously, too few Gaussians and the solution cannot be sufficiently expressed. Even though our refinement step densifies and prunes Gaussians, the quality of the simulation is still highly dependent on the initial number and arrangement of the Gaussians. Next, training the model is more complex than with other methods, due to the complex interactions between the Gaussians that the model

needs to be able to anticipate, alongside modeling the solution to the PDE. Another limitation is the inability of the model to integrate multiple past time steps, as opposed to other methods [17, 50]. Although the current solution should only depend on the state of the previous time step, these additional states could provide valuable information on the trajectory of the solution. Lastly, the initialization of our approach is time consuming and does not take into account the optimality of the initialized Gaussians with regards to the dynamic Gaussians, though a solution is proposed in Section 7.3.

Presumably, these limitations and a lack of more rigorous fine-tuning of the model results in our method performing subpar when compared to the current state of the art.

7.3 Future Work

Since this work is the first to explore the use of Gaussians for solving PDEs, there are many unexplored areas for future work. We hope this work can serve as a basis for further exploration in this direction. Furthermore, the proposed aggregation/convolution of Gaussians could potentially serve a similar purpose for Gaussians as DeltaConv [48] does for point clouds, e.g. segmentation or classification.

Additional experiments with different PDEs, boundary conditions, dimensions (i.e. 3D) and domain shapes can further demonstrate the benefits and limits of the method. Furthermore, careful fine-tuning is likely to expose many incremental improvements to the architecture and training procedure. An analysis of individual Gaussians could reveal to what extent they are particle-like.

A selection of more considerable improvements and/or additions are as follows:

Initializing Gaussians Firstly, initializing the Gaussians is currently a significant bottleneck in terms of inference speed and likely also inference quality. A conceivable method for improving the initialization involves a learned prior, similar to LGM [38], which generates 3D Gaussians from an image or text directly through a machine learning model instead of optimizing the Gaussians through gradient descent as with 3D-GS [14]. To generate the initial Gaussians based on the target initial condition, the model can be trained through the same procedure as our current initialization (see Section 3.2). This still requires an efficient method for encoding the initial condition of the PDE to input into the model. However, simply discretizing the initial condition into an image (or a volume in 3D) should suffice, since the generated Gaussians are still sampled at arbitrary positions when computing the loss during training, which mean they learn to estimate the underlying initial condition from the discretized input. This learned prior can then be trained jointly with the dynamics network. Not only can such a learned prior based on LGM be used to generate the initial Gaussians for a time-dependent PDE, but it could potentially also be used to provide a continuous solution for time-independent PDEs. Where, instead of training the model on the initial condition, it is trained to produce the solution $u(\mathbf{x})$ given the right-hand side $f(\mathbf{x})$ (see Section 2.1).

Boundary Gaussians The current boundary Gaussians, though effective, require manual placement of the Gaussians. Ideally, these boundary Gaussians could be generated automatically just as with the initial Gaussians. For simple boundary conditions, the Gaussians can be placed at regular intervals, as we do now. However, more complex irregular boundaries require a different approach.

Densification Another improvement could be made in terms of the refinement, especially the densification. We have experimented with various densification methods, but our current approach only splits some of the Gaussians once per time step. This severely limits the number of additional Gaussians introduced in underdefined regions. A learning-based method might provide a better densification, but training such a method to add (and possibly remove) Gaussians at the 'right' position is challenging. Our preliminary experiments on such a learning-based method have proved unfruitful.

Conservation Since the Gaussians provide a particle-like interpretation of the solution, we expect it should be possible to integrate conservation terms into the approach – namely, conservation of mass, momentum and energy – to improve the results. However, when attempting to conserve energy by ensuring the total volume under the Gaussians remains constant throughout the simulation, the quality of the results declined.

Gaussian splatting Lastly, we expect our method to integrate well into existing Gaussian-based methods, specifically Gaussian splatting. For example, given a static scene generated by 3D-GS [14], one could imagine interacting with the fluids in the scene, similar to PhysGaussian [52] and Gaussian Splashing [8], but with more accurate physics. Another idea is to perform inverse physics on a dynamic Gaussian splatting scene, in which the PDE is known, but its parameters – such as the kinematic viscocity ν – are not. These parameters can then be derived by modeling the scene with a guess of the parameters and iteratively updating the parameters based on the error between the predicted dynamics and the ground truth, similar to DreamPhysics [12] and Physics3D [22].

References

[1] Lucie P. Aarts and Peter van der Veer. "Neural Network Method for Solving Partial Differential Equations". In: *Neural Processing Letters* 14.3 (Dec. 2001), pp. 261–271. ISSN: 1573-773X. DOI: 10.1023/A:1012784129883.

[2] Rafael Bischof and Michael Kraus. "Multi-Objective Loss Balancing for Physics-Informed Deep Learning". In: (2021). DOI: 10.13140/RG.2.2.20057.24169. arXiv: 2110.09813 [cs].

[3] Nizar Bouguila and Wentao Fan, eds. *Mixture Models and Applications*. Unsupervised and Semi-Supervised Learning. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-23875-9 978-3-030-23876-6. DOI: 10.1007/978-3-030-23876-6.

[4] Yuanhao Cai et al. *Radiative Gaussian Splatting for Efficient X-ray Novel View Synthesis*. Mar. 2024. DOI: 10.48550/arXiv.2403.04116. arXiv: 2403.04116 [cs, eess].

[5] David Charatan et al. *pixelSplat: 3D Gaussian Splats from Image Pairs for Scalable Generalizable 3D Reconstruction*. Dec. 2023. DOI: 10.48550/arXiv.2312.12337. arXiv: 2312.12337 [cs].

[6] Yilun Du et al. *Neural Radiance Flow for 4D View Synthesis and Video Processing*. Sept. 2021. DOI: 10.48550/arXiv.2012.09790. arXiv: 2012.09790 [cs].

[7] Yuanxing Duan et al. *4D Gaussian Splatting: Towards Efficient Novel View Synthesis for Dynamic Scenes*. Feb. 2024. DOI: 10.48550/arXiv.2402.03307. arXiv: 2402.03307 [cs].

[8] Yutao Feng et al. *Gaussian Splashing: Dynamic Fluid Synthesis with Gaussian Splatting*. Jan. 2024. DOI: 10.48550/arXiv.2401.15318. arXiv: 2401.15318 [cs].

[9] Kyle Gao et al. *NeRF: Neural Radiance Field in 3D Vision, A Comprehensive Review*. Nov. 2023. DOI: 10.48550/arXiv.2210.00379. arXiv: 2210.00379 [cs].

[10] Giulio Giacomuzzo et al. "Embedding the Physics in Black-box Inverse Dynamics Identification: A Comparison Between Gaussian Processes and Neural Networks". In: *IFAC-PapersOnLine*. 22nd IFAC World Congress 56.2 (Jan. 2023), pp. 1584–1590. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2023.10.1858.

[11] Kaiming He et al. *Deep Residual Learning for Image Recognition*. Dec. 2015. DOI: 10.48550/arXiv.1512.03385. arXiv: 1512.03385 [cs].

[12] Tianyu Huang et al. *DreamPhysics: Learning Physical Properties of Dynamic 3D Gaussians with Video Diffusion Priors*. June 2024. DOI: 10.48550/arXiv.2406.01476. arXiv: 2406.01476 [cs].

[13] Junha Hyung et al. *Effective Rank Analysis and Regularization for Enhanced 3D Gaussian Splatting*. June 2024. arXiv: 2406.11672 [cs].

[14] Bernhard Kerbl et al. *3D Gaussian Splatting for Real-Time Radiance Field Rendering*. Aug. 2023. DOI: 10.48550/arXiv.2308.04079. arXiv: 2308.04079 [cs].

[15] Dmitrii Kochkov et al. "Machine Learning Accelerated Computational Fluid Dynamics". In: *Proceedings of the National Academy of Sciences* 118.21 (May 2021), e2101784118. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.2101784118. arXiv: 2102.01010 [physics].

[16] Nikola Kovachki et al. *Neural Operator: Learning Maps Between Function Spaces*. Apr. 2023. DOI: 10.48550/arXiv.2108.08481. arXiv: 2108.08481 [cs, math].

[17] Zongyi Li et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. May 2021. DOI: 10.48550/arXiv.2010.08895. arXiv: 2010.08895 [cs, math].

[18] Zongyi Li et al. *Fourier Neural Operator with Learned Deformations for PDEs on General Geometries*. July 2022. DOI: 10.48550/arXiv.2207.05209. arXiv: 2207.05209 [cs, math].

[19] Zongyi Li et al. *Neural Operator: Graph Kernel Network for Partial Differential Equations*. Mar. 2020. DOI: 10.48550/arXiv.2003.03485. arXiv: 2003.03485 [cs, math, stat].

[20] Zongyi Li et al. *Physics-Informed Neural Operator for Learning Partial Differential Equations*. July 2023. DOI: 10.48550/arXiv.2111.03794. arXiv: 2111.03794 [cs, math].

[21] Youtian Lin et al. *Gaussian-Flow: 4D Reconstruction with Dynamic 3D Gaussian Particle*. Dec. 2023. DOI: 10.48550/arXiv.2312.03431. arXiv: 2312.03431 [cs].

[22] Fangfu Liu et al. *Physics3D: Learning Physical Properties of 3D Gaussians via Video Diffusion*. June 2024. DOI: 10.48550/arXiv.2406.04338. arXiv: 2406.04338 [cs].

[23] Xin-Yang Liu et al. "Multi-Resolution Partial Differential Equations Preserved Learning Framework for Spatiotemporal Dynamics". In: *Communications Physics* 7.1 (Jan. 2024), pp. 1–19. ISSN: 2399-3650. DOI: 10.1038/s42005-024-01521-z.

[24] Ben Mildenhall et al. *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*. Aug. 2020. DOI: 10.48550/arXiv.2003.08934. arXiv: 2003.08934 [cs].

[25] Thomas Müller et al. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–15. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3528223.3530127. arXiv: 2201.05989 [cs].

[26] Guofei Pang and George Em Karniadakis. "Physics-Informed Learning Machines for Partial Differential Equations: Gaussian Processes Versus Neural Networks". In: *Emerging Frontiers in Nonlinear Science*. Ed. by Panayotis G. Kevrekidis, Jesús Cuevas-Maraver, and Avadh Saxena. Nonlinear Systems and Complexity. Cham: Springer International Publishing, 2020, pp. 323–343. ISBN: 978-3-030-44992-6. DOI: 10.1007/978-3-030-44992-6_14.

[27] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703.

[28] Albert Pumarola et al. "D-NeRF: Neural Radiance Fields for Dynamic Scenes". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2021, pp. 10313–10322. DOI: 10.1109/CVPR46437.2021.01018.

[29] Charles R. Qi et al. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. Apr. 2017. DOI: 10.48550/arXiv.1612.00593. arXiv: 1612.00593 [cs].

[30] Ali Rahimi and Benjamin Recht. "Random Features for Large-Scale Kernel Machines". In: *Advances in Neural Information Processing Systems*. Vol. 20. Curran Associates, Inc., 2007.

[31] Maziar Raissi and George Em Karniadakis. "Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations". In: *Journal of Computational Physics* 357 (Mar. 2018), pp. 125–141. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2017.11.039.

[32] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. Nov. 2017. DOI: 10.48550/arXiv.1711.10561. arXiv: 1711.10561 [cs, math, stat].

[33] Jiawei Ren et al. *L4GM: Large 4D Gaussian Reconstruction Model*. June 2024. arXiv: 2406.10324 [cs].

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. May 2015. DOI: 10.48550/arXiv.1505.04597. arXiv: 1505.04597 [cs].

[35] Richard Shaw et al. *SWAGS: Sampling Windows Adaptively for Dynamic 3D Gaussian Splatting*. Dec. 2023. DOI: 10.48550/arXiv.2312.13308. arXiv: 2312.13308 [cs].

[36] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A Deep Learning Algorithm for Solving Partial Differential Equations". In: *Journal of Computational Physics* 375 (Dec. 2018), pp. 1339–1364. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.08.029.

[37] Christopher Smith, John Doherty, and Yaochu Jin. "Multi-Objective Evolutionary Recurrent Neural Network Ensemble for Prediction of Computational Fluid Dynamic Simulations". In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. July 2014, pp. 2609–2616. DOI: 10.1109/CEC.2014.6900552.

[38] Jiaxiang Tang et al. *LGM: Large Multi-View Gaussian Model for High-Resolution 3D Content Creation*. Feb. 2024. DOI: 10.48550/arXiv.2402.05054. arXiv: 2402.05054 [cs].

[39] *Tum-Pbs/PhiFlow*. https://github.com/tum-pbs/PhiFlow. May 2024.

[40] J.J.I.M. van Kan, A. Segal, and Fred Vermolen. *Numerical Methods in Scientific Computing: 2nd Edition*. TU Delft OPEN Publishing, 2023. DOI: 10.59490/t.2023.009.

[41] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs].

[42] Nils Wandel, Michael Weinmann, and Reinhard Klein. *Learning Incompressible Fluid Dynamics from Scratch – Towards Fast, Differentiable Fluid Models That Generalize*. Mar. 2021. DOI: 10.48550/arXiv.2006.08762. arXiv: 2006.08762 [cs, stat].

[43] Nils Wandel, Michael Weinmann, and Reinhard Klein. "Teaching the Incompressible Navier-Stokes Equations to Fast Neural Surrogate Models in 3D". In: *Physics of Fluids* 33.4 (Apr. 2021), p. 047117. ISSN: 1070-6631, 1089-7666. DOI: 10.1063/5.0047428. arXiv: 2012.11893 [physics].

[44] Nils Wandel et al. *Spline-PINN: Approaching PDEs without Data Using Fast, Physics-Informed Hermite-Spline CNNs*. Mar. 2022. DOI: 10.48550/arXiv.2109.07143. arXiv: 2109.07143 [physics].

[45] Rui Wang et al. *Towards Physics-informed Deep Learning for Turbulent Flow Prediction*. June 2020. DOI: 10.48550/arXiv.1911.08655. arXiv: 1911.08655 [physics, stat].

[46] Sifan Wang, Shyam Sankaran, and Paris Perdikaris. *Respecting Causality Is All You Need for Training Physics-Informed Neural Networks*. Mar. 2022. DOI: 10.48550/arXiv.2203.07404. arXiv: 2203.07404 [nlin, physics:physics, stat].

[47] J. Westerweel et al. "Momentum and Scalar Transport at the Turbulent/Non-Turbulent Interface of a Jet". In: *Journal of Fluid Mechanics* 631 (July 2009), pp. 199–230. ISSN: 1469-7645, 0022-1120. DOI: 10.1017/S0022112009006600.

[48] Ruben Wiersma et al. "DeltaConv: Anisotropic Operators for Geometric Deep Learning on Point Clouds". In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–10. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3528223.3530166. arXiv: 2111.08799 [cs].

[49] Guanjun Wu et al. *4D Gaussian Splatting for Real-Time Dynamic Scene Rendering*. Oct. 2023. DOI: 10.48550/arXiv.2310.08528. arXiv: 2310.08528 [cs].

[50] Tailin Wu, Takashi Maruyama, and Jure Leskovec. *Learning to Accelerate Partial Differential Equations via Latent Global Evolution*. Oct. 2022. DOI: 10.48550/arXiv.2206.07681. arXiv: 2206.07681 [physics].

[51] Zixue Xiang et al. “Self-Adaptive Loss Balanced Physics-informed Neural Networks”. In: *Neurocomputing* 496 (July 2022), pp. 11–34. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2022.05.015.

[52] Tianyi Xie et al. *PhysGaussian: Physics-Integrated 3D Gaussians for Generative Dynamics*. Nov. 2023. DOI: 10.48550/arXiv.2311.12198. arXiv: 2311.12198 [cs].

[53] Liu Yang, Xuhui Meng, and George Em Karniadakis. “B-PINNs: Bayesian Physics-Informed Neural Networks for Forward and Inverse PDE Problems with Noisy Data”. In: *Journal of Computational Physics* 425 (Jan. 2021), p. 109913. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.109913.

[54] Alex Yu et al. *pixelNeRF: Neural Radiance Fields from One or Few Images*. May 2021. DOI: 10.48550/arXiv.2012.02190. arXiv: 2012.02190 [cs].

[55] Xinjie Zhang et al. *GaussianImage: 1000 FPS Image Representation and Compression by 2D Gaussian Splatting*. Mar. 2024. arXiv: 2403.08551 [cs, eess].

[56] Zhiyuan Zhao, Xueying Ding, and B. Aditya Prakash. *PINNsFormer: A Transformer-Based Framework For Physics-Informed Neural Networks*. Oct. 2023. DOI: 10.48550/arXiv.2307.11833. arXiv: 2307.11833 [cs].

[57] David Zwicker. “py-pde: A Python package for solving partial differential equations”. In: *Journal of Open Source Software* 5.48 (2020), p. 2158. DOI: 10.21105/joss.02158. URL: https://doi.org/10.21105/joss.02158.

[58] M. Zwicker et al. “EWA Volume Splatting”. In: *Proceedings Visualization, 2001. VIS ’01.* Oct. 2001, pp. 29–538. DOI: 10.1109/VISUAL.2001.964490.