A Additional results

A.1 How does the choice of learnable function class and design of encodings impact EulerFlow?

EulerFlow at its core is an optimization loop over a simple, feedforward ReLU-based multi-layer perception inherited from Neural Scene Flow Prior (Li et al., 2021b). How does this choice of learnable function class impact the performance of EulerFlow? To better understand these design choices we examine the choice of non-linearity and time feature encoding.

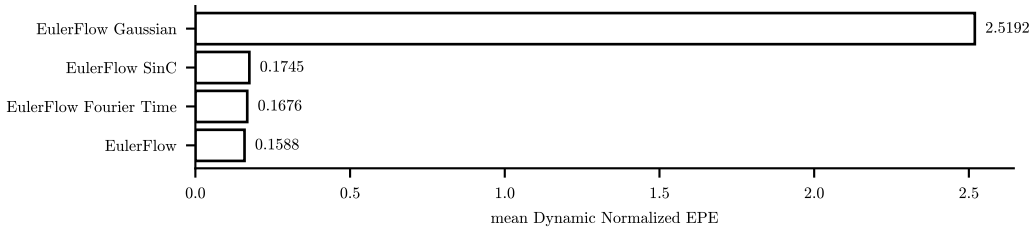

Figure 15: Mean Dynamic Normalized EPE of EulerFlow on the Argoverse 2 val split for less-smooth configurations of its learnable function class. These results indicate that the smoothness of the ReLU non-linearity proposed by Li et al. transfers well to EulerFlow.

One of Li et al.'s core theoretical contributions demonstrates that NSFP's ReLU MLP is a good prior for scene flow because it represents a smooth learnable function class, and scene flow is often locally smooth with respect to input position. However, unlike NSFP, EulerFlow is fitting flow over a full ODE; while it seems reasonable to assume that this ODE is typically also locally smooth, cases like adjacent cars moving rapidly in opposite directions may benefit from the ability to model higher frequency, less locally smooth functions. To test this hypothesis, we ablate EulerFlow by replacing its normalized time with higher frequency sinusoidal time embeddings (mirroring Wang et al.'s proposed time embedding for NTP), as well as try other popular non-linearities like SinC (Ramasinghe et al., 2024) and Gaussian (Chng et al., 2022) from the coordinate network literature. Figure 15 features negative results on these ablations across the board; Gaussians were unable to converge due the extremely high frequency representation triggering early stopping, while the use of SinC and higher frequency time embeddings both resulted in worse overall performance, indicating that Li et al.'s smooth function prior does indeed seem appropriate for EulerFlow's neural prior.

A.2 EulerFlow with Monocular Depth Estimates

While EulerFlow only consumes point clouds, we can leverage RGB-based video monocular depth estimators to fit scene flow. In Figure 16, we use DepthCrafter (Hu et al., 2024) to generate a point cloud from the raw RGB of the tabletop video from Figure 14, Row 4.

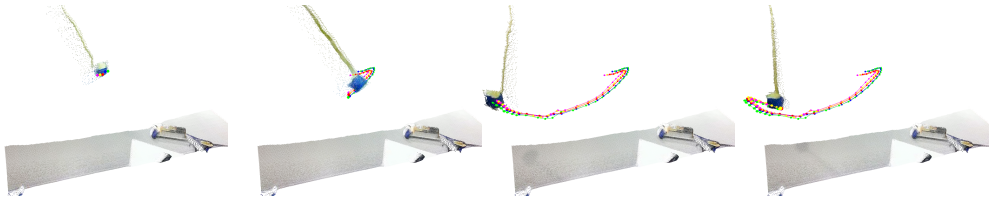

Figure 16: Visualizations of EulerFlow's emergent 3D point tracking behavior on monocular depth estimates from DepthCrafter (Hu et al., 2024). Interactive visualizations available at `eulerflow.github.io`.

A.3 How Does EulerFlow Fail?

As we discuss in Section 6.1, EulerFlow does not understand projective geometry — its optimization losses use Chamfer Distance which directly associates points, sometimes resulting in moving shadows

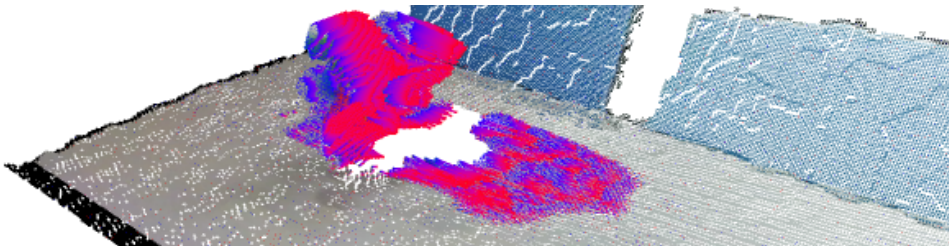

Figure 17: Visualizations of one of the failure modes of EulerFlow where flow is predicted on the edges of the moving "shadow" in the point cloud. Interactive visualizations available at `eulerflow.github.io`.

on background objects. To demonstrate this, we select a particularly egregious example in Figure 17, featuring a frame from the jack being thrown across the table. Due to the moving shadow cast by the jack onto the table, EulerFlow incorrectly assigns flow to the table surface nearby the jack, particularly on the leading edge, even though the table surface is stationary.

B FAQ

B.1 What datasets did you pretrain on?

EulerFlow is not pretrained on any datasets. It is a test-time optimization method (akin to NeRFs), and as we show with our tabletop data, this means it runs out-of-the-box on arbitrary point cloud data.

B.2 Why didn't you use a Neural ODE or a Liquid Neural Network?

Neural ODEs (Chen et al., 2018) take variable size and number of steps in latent space to do inference; imagine a ResNet that can use an ODE solver to dynamically scale the impact of the residual block, as well as decide the number of residual blocks. They are not a function class specially designed to fit derivative estimates well. Similar to Neural ODEs, Liquid Neural Networks (Hasani et al., 2021) focus on the same class of problems and are similarly not applicable.

B.3 Why didn't you do experiments on FlyingThings3D / <simulated dataset>?

Most popular synthetic datasets do not contain long observation sequences (Mayer et al., 2016; Butler et al., 2012), but instead include standalone frame pairs. Our method leverages the long sequence of observations to refine our neural estimate of the true ODE. Indeed, on two frames, EulerFlow collapses to NSFP.

More importantly, these datasets are also not representative of real world environments. To quote Chodosh et al.: "[FlyingThings3D has] unrealistic rates of dynamic motion, unrealistic correspondences, and unrealistic sampling patterns. As a result, progress on these benchmarks is misleading and may cause researchers to focus on the wrong problems." Khatri et al. also make this point by highlighting the importance of meaningfully breaking down the object distribution during evaluation identify performance on rare safety-critical categories. FlyingThings3D does not have meaningful semantics; it's not obvious what things even matter or how to appropriately break down the scene.

Instead, we want to turn our attention to the sort of workloads that *do* clearly matter — describing motion in domains like manipulation or autonomous vehicles, where it seems clear that scene flow, if solved, will serve as powerful primitive for downstream systems. This is why we performed qualitative experiments on the tabletop data we collected ourselves; to our knowledge, no real-world dynamic datasets of this nature exist with ground truth annotations, but we want to emphasize that EulerFlow works in such domains, and consequently EulerFlow and other Scene Flow via ODE-based methods can be used as a primitive in these real world domains.

C EULERFLOW IMPLEMENTATION DETAILS

Our neural prior θ is a straightforward extension to NSFP's coordinate network prior[6]; however, instead of taking a 3D space vector (positions $X, Y, Z \in \mathbb{R}$) as input, we encode a 5D space-time-direction vector: positions $X, Y, Z, \in \mathbb{R}$, sequence normalized time $t \in [-1, 1]$ (i.e. the point cloud time scaled to this range), and direction $d \in \{\texttt{BWD} = -1, \texttt{FWD} = 1\}$. This simple encoding scheme enables description of arbitrary regions of the ODE, allowing for the ODE to be queried at frequencies different from the sensor frame rate. Euler integration enables simple implementation of multi-step forward, backward, and cyclic consistency losses without extra bells and whistles. For efficiency, we use Euler integration with Δt set as the time between observations for our ODE solver, enabling support for arbitrary sensor frame rates, and set the cycle consistency balancing term $\alpha = 0.01$ and optimization window $W = 3$ for all experiments.

EulerFlow's definition of TruncatedChamfer is symmetric[7], i.e. TruncatedChamfer(A, B) = TruncatedChamfer(B, A). If this symmetric TruncatedChamfer is naively implemented via a performant differentiable CUDA accelerated K=1NN computation[8] from A to B and from B to A, on an NVIDIA V100, EulerFlow spends roughly 80% of its compute time performing these KNN checks. To accelerate this, we precompute exact GPU accelerated KD-Trees (Grandits et al., 2021) for the input point clouds $\{P_0, \ldots, P_N\}$, and when possible query those trees instead of computing K=1NN. In practice, we found these queries are almost instant, and reduce the time spent computing K=1NNs to about 40% of the total wall-clock time of EulerFlow.

Additionally, while Equation 3 is phrased as independent Euler integration steps for each timestep, we are able to share integration across the losses; we perform two integrations from t to $t + W$ and t to $t - W$, and use the intermediary locations along this trajectory as inputs to intermediary losses.

C.1 FORMULATING THE ODE

Given a (possibly moving) particle in some canonical frame (i.e. time 0), we define a function $L(x_0, y_0, z_0, t)$ that can describe its location at an arbitrary future time t, i.e. a Lagrangian description of motion (Figure 4).

$$L(x_0, y_0, z_0, t) = x_t, y_t, z_t \tag{4}$$

For notational clarity to access x_t, y_t, z_t individually, we can define

$$L_x(x_0, y_0, z_0, t) = x_t \tag{5}$$
$$L_y(x_0, y_0, z_0, t) = y_t \tag{6}$$
$$L_z(x_0, y_0, z_0, t) = z_t \tag{7}$$

Similarly, we can define $F(x_t, y_t, z_t, t)$ to describe the instantaneous velocity of a point x_t, y_t, z_t at some arbitrary time t, i.e. a Eulerian description of motion (Figure 4).

$$\frac{dL(x_0, y_0, z_0, t)}{dt} = \frac{dL}{dt} = \left(\frac{dL_x}{dt}, \frac{dL_y}{dt}, \frac{dL_z}{dt}\right) = F(x_t, y_t, z_t, t) \tag{8}$$

F is defined in terms of the total derivative of L with respect to t, as x_0, y_0, z_0 are initial conditions that do not vary with time (i.e. $\frac{dL}{dt} = \frac{\partial L}{\partial t} + \frac{\partial L}{\partial x_0}\frac{dx_0}{dt} + \frac{\partial L}{\partial y_0}\frac{dy_0}{dt} + \frac{\partial L}{\partial z_0}\frac{dz_0}{dt} = \frac{\partial L}{\partial t}$, as $\frac{dx_0}{dt} = \frac{dy_0}{dt} = \frac{dz_0}{dt} = 0$). We can exactly define L recursively in terms of the initial conditions and F, i.e.

[6] Hyperparameters (e.g. filter width of 128) of NSFP's prior are kept fixed, except for depth (Section 5.2.3).

[7] This is in keeping with NSFP, but in opposition to other methods like FastNSF (Li et al., 2023). We found that a symmetric definition provided non-trivial performance improvements to EulerFlow.

[8] We used PyTorch3D (Ravi et al., 2020), which has custom CUDA operations with CUDA templated support for single neighbor differentiable KNN.

$$L(x_0, y_0, z_0, t) = (x_0, y_0, z_0) + \int_0^t F(L_x(x_0, y_0, z_0, \tau), L_y(x_0, y_0, z_0, \tau), L_z(x_0, y_0, z_0, \tau), \tau) d\tau \tag{9}$$

or, more compactly,

$$L(x_0, y_0, z_0, t) = (x_0, y_0, z_0) + \int_0^t F(x_\tau, y_\tau, z_\tau, \tau) d\tau \tag{10}$$

Our function L can thus be defined as a multi-dimensional ODE in terms of F with initial conditions x_0, y_0, z_0.

C.2 Arbitrary start and end times from the Eulerian formulation

In the above derivation, L requires that a moving point be defined in terms of a canonical frame defined at time 0, as is common in the deformation in reconstruction literature. However, the Eulerian formulation has no such requirement, allowing us to select arbitrary start and end times across different point queries. To showcase this, we can query F to extract the trajectory of a particle at t across the range $[t, t']$ starting at x_t, y_t, z_t simply by changing the range of the integral in Equation 10, i.e.

$$E(x_t, y_t, z_t, t, t') = (x_t, y_t, z_t) + \int_t^{t'} F(x_\tau, y_\tau, z_\tau, \tau) d\tau \tag{11}$$

While E and L appear similar on their face, E is strictly more flexible than L. In principle you could choose to redefine L to use t as the time for your canonical frame, but this is a *global* choice; you cannot do this on a per-query basis. However, with E's Eulerian framing, we can extract a different point's trajectory from the entirely different range $t^\dagger$ to $t^\ddagger$ (i.e. $E(x_{t^\dagger}, y_{t^\dagger}, z_{t^\dagger}, t^\dagger, t^\ddagger)$) without concern for a canonical frame definition. It need not even be the case that $t < t'$; indeed, this extraction works even if $t > t'$, i.e. extracting the backwards trajectory through time.

C.3 Euler Integration to approximately solve the ODE

If F is of arbitrary form and we want to compute the concrete values of L, we cannot exactly compute the continuous integral from 0 to t; we must approximate this with finite differences. Thus, we split the time range 0 to t into k steps, where each step is of size $\frac{t}{k}$. Thus, we can again define L via recursion, but this time explicitly.

$$L(x_0, y_0, z_0, 0) = (x_0, y_0, z_0) \tag{12}$$

$$L(x_0, y_0, z_0, \tau + \frac{t}{k}) \approx L(x_0, y_0, z_0, \tau) + \frac{t}{k} \cdot F(x_\tau, y_\tau, z_\tau, \tau), \tag{13}$$

or directly without recursion,

$$L(x_0, y_0, z_0, t) \approx (x_0, y_0, z_0) + \sum_{n=1}^{k} \frac{t}{k} \cdot F(x_{n\frac{t}{k}}, y_{n\frac{t}{k}}, z_{n\frac{t}{k}}, n\frac{t}{k}) \tag{14}$$

This finite difference solving approach is Euler integration.

C.4 Estimating the flow field with EulerFlow's neural prior

For a given scene, we do not have access to L or F directly; these are are the *true* functions that uniquely characterize the underlying motion of the scene that we are trying to estimate. For EulerFlow, we represent our estimate of the scene's flow field F with a neural prior, θ, i.e.

$$F(x, y, z, t) \approx \theta(x, y, z, t) \tag{15}$$

and thus

$$L(x_0, y_0, z_0, t) \approx (x_0, y_0, z_0) + \sum_{n=1}^{k} \frac{t}{k} \cdot \theta(x_{n\frac{t}{k}}, y_{n\frac{t}{k}}, z_{n\frac{t}{k}}, n\frac{t}{k}) \tag{16}$$

and, using the arbitrary start and end definition from Appendix C.2, with k steps from the range t to t' and $\delta = \frac{t'-t}{k}$

$$E(x_t, y_t, z_t, t, t') \approx E_\theta(x_t, y_t, z_t, t, t') = (x_t, y_t, z_t) + \sum_{n=1}^{k} \delta \cdot \theta(x_{n\delta+t}, y_{n\delta+t}, z_{n\delta+t}, n\delta + t) \tag{17}$$

This formulation makes EulerFlow highly flexible, enabling optimization of θ's estimate of F with objectives that take either an Eulerian view (directly on θ via Equation 15) or a Lagrangian view (on point rollouts for arbitrary start and end ranges via Equation 17).