A Extended background

A.1 Riemannian manifolds

A key definition in the development of our approach is the notion of a *chart*. Formally speaking, a chart on a smooth manifold $\mathcal{M}$ is a diffeomorphic mapping $\varphi : U \to \tilde{U}$ from an open set $U \subset \mathcal{M}$ to an open set $\tilde{U} \subseteq \mathbb{R}^d$, see Fig. 6 for an illustration.

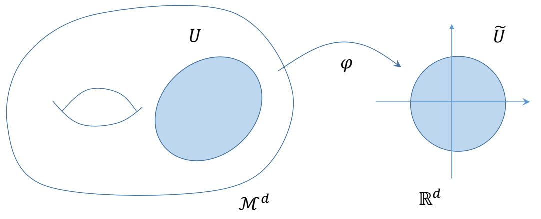


Figure 6: Coordinate chart φ applied on a smooth manifold $\mathcal{M}^d$ from an open set U to $\tilde{U}$.

As briefly introduced in § 2, to operate with Riemannian manifolds, it is common practice to exploit the Euclidean tangent spaces. We resort to mappings back and forth between $\mathcal{T}_{\boldsymbol{x}}\mathcal{M}$ and $\mathcal{M}$ using the exponential and logarithmic maps, namely, $\text{Exp}_{\boldsymbol{x}}(\boldsymbol{u}) : \mathcal{T}_{\boldsymbol{x}}\mathcal{M} \to \mathcal{M}$ and $\text{Log}_{\boldsymbol{x}}(\boldsymbol{y}) : \mathcal{M} \to \mathcal{T}_{\boldsymbol{x}}\mathcal{M}$. Another relevant operation is the parallel transport $\Gamma_{\boldsymbol{x}\to\boldsymbol{y}}(\boldsymbol{u}) : \mathcal{T}_{\boldsymbol{x}}\mathcal{M} \to \mathcal{T}_{\boldsymbol{y}}\mathcal{M}$, which moves manifold elements lying on different tangent spaces along geodesics on $\mathcal{M}$. Finally, as described in § C, we used the projection operator to project Euclidean vectors to the tangent space of the manifold. Formally, this orthogonal projection is defined as $\text{proju}_{\boldsymbol{x}}(\boldsymbol{v}) : \mathbb{R}^n \to \mathcal{T}_{\boldsymbol{x}}\mathcal{M}$, which is used to compute the Riemannian gradient as the orthogonal projection of the "Euclidean" gradient to the manifold tangent space. The specific operations for the unit hypersphere, used in this work, as provided in Table 1.

Operator	**Formula**
$d_{\mathcal{M}}(\boldsymbol{x},\boldsymbol{y})$	$\arccos(\boldsymbol{x}^\mathsf{T}\boldsymbol{y})$
$\text{Exp}_{\boldsymbol{x}}(\boldsymbol{u})$	$\boldsymbol{x}\cos(\lVert\boldsymbol{u}\rVert) + \overline{\boldsymbol{u}}\sin(\lVert\boldsymbol{u}\rVert)$ with $\overline{\boldsymbol{u}} = \frac{\boldsymbol{u}}{\lVert\boldsymbol{u}\rVert}$
$\text{Log}_{\boldsymbol{x}}(\boldsymbol{y})$	$d_{\mathcal{M}}(\boldsymbol{x},\boldsymbol{y})\,\frac{\boldsymbol{y}-\boldsymbol{x}^\mathsf{T}\boldsymbol{y}\,\boldsymbol{x}}{\lVert\boldsymbol{y}-\boldsymbol{x}^\mathsf{T}\boldsymbol{y}\,\boldsymbol{x}\rVert}$
$\Gamma_{\boldsymbol{x}\to\boldsymbol{y}}(\boldsymbol{u})$	$\left(-\boldsymbol{x}\sin(\lVert\boldsymbol{v}\rVert)\overline{\boldsymbol{v}}^\mathsf{T} + \overline{\boldsymbol{v}}\cos(\lVert\boldsymbol{v}\rVert)\overline{\boldsymbol{v}}^\mathsf{T}) + (\boldsymbol{I}_n - \overline{\boldsymbol{v}}\overline{\boldsymbol{v}}^\mathsf{T})\right)\boldsymbol{u}$, $\overline{\boldsymbol{v}} = \frac{\boldsymbol{v}}{\lVert\boldsymbol{v}\rVert}$, $\boldsymbol{v} = \text{Log}_{\boldsymbol{x}}(\boldsymbol{y})$
$\text{proju}_{\boldsymbol{x}}(\boldsymbol{v})$	$(\boldsymbol{I}_n - \boldsymbol{x}\boldsymbol{x}^\mathsf{T})\boldsymbol{v}$

Table 1: Principal operations on $\mathcal{S}^d$. For details, see [42]

A.2 Neural MODEs for Diffeomorphism Learning

A diffeomorphism can be constructed by solving an initial value problem (or integral) for Neural ODEs [30]. Futhermore, we can extend this idea to the manifold setting following [24, 25]. To do so, we start from the following theorem from Mathieu and Nickel [25]:

Theorem 2 (Vector flows). *Let $\mathcal{N}$ be a smooth complete manifold and $f_{\boldsymbol{\theta}}$ be a $\mathcal{C}^1$-bounded time-dependent vector field. Then there exists a global flow $\psi_{\boldsymbol{\theta}} : \mathcal{N} \times \mathbb{R} \to \mathcal{N}$ such that for each $t \in \mathbb{R}$, the map $\psi_{\boldsymbol{\theta}}(\cdot, t) : \mathcal{N} \to \mathcal{N}$ is a $\mathcal{C}^1$-diffeomorphism (i.e. $\mathcal{C}^1$ bijection with $\mathcal{C}^1$ inverse).*

Accordingly, we can compute the diffeomorphism on the manifold $\mathcal{N}$ by solving an IVP of Neural MODEs (2), where the integration is done over a manifold instead of an Euclidean space. During computation of the IVP, we can select the time interval $[t_s, t_e]$ as $[0, 1]$ without loss of generality. As a result, the diffeomorphism $\psi_{\boldsymbol{\theta}}$ and inverse diffeomorphism $\psi_{\boldsymbol{\theta}}^{-1}$ can be computed by integration forwards and backwards, respectively, as follows,

$$\boldsymbol{y} = \psi_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{x} + \int_0^1 f_{\boldsymbol{\theta}}(\boldsymbol{z}(\tau), \tau)d\tau, \quad \boldsymbol{x} = \psi_{\boldsymbol{\theta}}^{-1}(\boldsymbol{y}) = \boldsymbol{y} + \int_1^0 f_{\boldsymbol{\theta}}(\boldsymbol{z}(\tau), \tau)d\tau, \tag{9}$$

where $\boldsymbol{x} = \boldsymbol{z}(t_s)$ and $\boldsymbol{y} = \boldsymbol{z}(t_e)$ are two points on the manifold $\mathcal{N}$. Note that in this section we view the diffeomorphism $\psi_{\boldsymbol{\theta}}$ as a function on the Riemannian manifold $(\mathcal{N}, h)$. However, under this diffeomorphic mapping, we actually create a new Riemannian manifold $(\mathcal{M}, \hat{h})$ with $\hat{h}$ being a pullback metric $\hat{h}_{\boldsymbol{x}}(\boldsymbol{u}, \boldsymbol{v}) = h_{\psi_{\boldsymbol{\theta}}(\boldsymbol{x})}(D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}\,\boldsymbol{u}, D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}\,\boldsymbol{v})$. From this perspective, the diffeomorphism

$\psi_{\boldsymbol{\theta}}$ can be regarded as a mapping from a Riemannian manifold $(\mathcal{M}, \hat{h})$ to $(\mathcal{N}, h)$, that is $\psi_{\boldsymbol{\theta}} : \mathcal{M} \rightarrow \mathcal{N}$, with $(\mathcal{M}, \hat{h})$ and $(\mathcal{N}, h)$ being isometric Riemannian manifolds.

A.3 Integrators on manifolds

Finding a numerical solution for integrators on Riemannian manifolds is not straightforward and needs some additional considerations. The majority of existing ODE solvers are designed and carefully optimized for Euclidean spaces (e.g. Euler, Runge-Kutta, and adaptive solvers), which can be modified to compute the integral on Riemannian manifolds. In the following, we discuss two approaches for integration on Riemannian manifolds: *projection methods* and *integrators based on local coordinates* [36].

Projection methods: Assume we have a Neural MODE given by (2) on a $\mathcal{M}$, where $f_{\boldsymbol{\theta}}(\cdot, t) \in \mathcal{T}_{\boldsymbol{z}(t)}\mathcal{M}$ for all $\boldsymbol{z}(t) \in \mathcal{M}$. The standard projection methods compute a one-step integral with an arbitrary numerical integrator in Euclidean space and then projects the value onto the manifold $\mathcal{M}$, as depicted in Fig. 7. For example, an Euler solver is selected with step size dt which computes a single step from the initial point $\boldsymbol{z}_0$. The next state $\tilde{\boldsymbol{z}}_1$ is then computed as $\tilde{\boldsymbol{z}}_1 = \boldsymbol{z}_0 + f_{\boldsymbol{\theta}}(\boldsymbol{z}_0, 0)dt$. Finally, $\tilde{\boldsymbol{z}}_1$ is projected back onto the manifold by leveraging the exponential map.

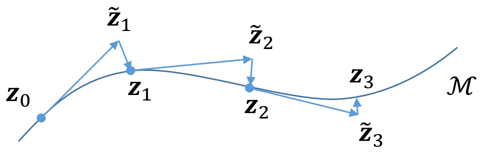


Figure 7: Standard projection method

Integrators on local coordinates: These methods consider a local representation of the manifold $\mathcal{M}$, which defines a coordinate map $\varphi : \mathcal{M}^d \supseteq U \rightarrow \tilde{U} \subseteq \mathbb{R}^d$, as described in § 2, with coordinates $\boldsymbol{w}(t) = \varphi(\boldsymbol{z}(t))$. Since this coordinate map is a diffeomorphism, we can compute its inverse φ^{-1}. The differential equation for local coordinates $\boldsymbol{w}(t)$ is then given by (3) in Euclidean space. This equation states that the differential of the coordinate map $D_{\varphi^{-1}(\boldsymbol{w}(t))}\varphi$ must be computed to project the vector fields on the manifold onto its coordinate codomain $\tilde{U}$. In this way, we can solve the equivalent ODE in (3) with classical ODE solvers in Euclidean space, as follows,

$$\boldsymbol{w}(t_e) = \boldsymbol{w}(t_s) + \int_{t_s}^{t_e} D_{\varphi^{-1}(\boldsymbol{w}(\tau))}\varphi \circ f_{\boldsymbol{\theta}}(\varphi^{-1}(\boldsymbol{w}(\tau)), \tau)d\tau. \tag{10}$$

As a result, the solution of the manifold ODE (2) and the equivalent ODE in (3) are connected via coordinate maps φ, so that any approximation $\boldsymbol{w}(t_n)$ provides an approximation for $\boldsymbol{z}(t_n)$. Theoretically there are infinite possible choices of local coordinates which can be selected before or during the integration process. However, for complex Riemmanian manifolds that are not globally diffeomorphic to the Euclidean space, it is impossible to find a single coordinate chart to cover the whole manifold. Therefore, different coordinate charts must be selected during the integration process. Combining the equivalent ODE and multiple choices of coordinates charts leads to the so-called approach *Dynamic Chart Method* [24]. We take advantage of this method to efficiently perform integration on manifolds.

A.4 Adjoint Method

To incorporate Neural MODEs into deep learning frameworks, it is crucial to compute their gradients efficiently. According to Chen et al. [30], the adjoint sensitivity method allows us to treat the ODE solvers as a black box and compute the corresponding gradients by solving a second adjoint ODE backwards in time, instead of directly differentiating through ODE solvers (which is the naïve solution). This approach is significantly more memory efficient, especially for adaptive ODE solvers, which adjust the step size on the fly. Furthermore, the adjoint method can be generalized for ODEs on arbitrary manifolds Lou et al. [24].

Formally, let us consider a loss function $\mathcal{L} : \mathcal{M}^d \rightarrow \mathbb{R}$. To compute gradients of $\mathcal{L}$ with respect to any value of the state variable $\boldsymbol{z}(t)$ of the manifold ODE, we can define an *adjoint variable* $\boldsymbol{a}(t)^\mathsf{T} := D_{\boldsymbol{z}(t)}\mathcal{L}$, subject to,

$$\dot{\boldsymbol{a}}(t)^\mathsf{T} = -\boldsymbol{a}(t)^\mathsf{T} D_{\boldsymbol{z}(t)} f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t). \tag{11}$$

If the loss $\mathcal{L}$ merely depends on the end value $\boldsymbol{z}(t_e)$, to get the gradient of $\mathcal{L}$ w.r.t. the starting value $\boldsymbol{z}(t_s)$, we solve the end value problem (EVP) of the adjoint ODE (11). Since this adjoint ODE contains two variables: $\boldsymbol{z}(t)$ and $\boldsymbol{a}(t)^\mathsf{T}$, solving the EVP of it requires knowledge about $\boldsymbol{z}(t)$ over the entire time span, which leads to solving a combination of the adjoint ODE (11) and the Neural MODE (2) backwards. Furthermore, the end values are needed for both variables $\boldsymbol{z}$ and $\boldsymbol{a}$ to solve ODEs backwards from end time t_e to starting time t_s. The end value $\boldsymbol{z}(t_e)$ needs to be computed first through the forward integration of the Neural MODE, while $\boldsymbol{a}(t_e)^\mathsf{T} = D_{\boldsymbol{z}(t_e)}\mathcal{L}$ can be obtained via classical backpropagation, since the loss $\mathcal{L}$ is only a function of $\boldsymbol{z}(t_e)$.

Finally, to compute the differential of the diffeomorphism $\psi_{\boldsymbol{\theta}}$ with respect to the starting value $\boldsymbol{z}(t_s)$ (which will help us compute the pullback operator in § 3.3), we consider a special loss $\mathcal{L}_i = q_i(\boldsymbol{z}(t_e))$, where q_i is the i-th component of a function $q : \mathcal{N}^d \to \mathbb{R}^d$. We introduce the function q to help us construct a valid loss $\mathcal{L}_i = q_i \circ \psi_{\boldsymbol{\theta}}$, which maps from $\mathcal{M}^d$ to $\mathbb{R}$. Furthermore, we define $\boldsymbol{l} = (\mathcal{L}_1, \dots, \mathcal{L}_d)^\mathsf{T} = q(\boldsymbol{z}(t_e)) = (q \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{z}(t_s))$. Given the previous definition, we can define $\tilde{\boldsymbol{A}}(t) = D_{\boldsymbol{z}(t)}\boldsymbol{l}$ as an adjoint variable as follows,

$$\tilde{\boldsymbol{A}}(t) = D_{\boldsymbol{z}(t)}\boldsymbol{l} = D_{\boldsymbol{z}(t)}(q \circ \psi_{\boldsymbol{\theta}}) = D_{\boldsymbol{z}(t_e)}q \circ D_{\boldsymbol{z}(t)}\psi_{\boldsymbol{\theta}} = D_{\boldsymbol{z}(t_e)}q \circ \boldsymbol{A}(t), \tag{12}$$

where $\boldsymbol{A}(t)$ is the derivative of the diffeomorphism with respect to $\boldsymbol{z}(t)$. Consequently, we build the corresponding adjoint ODE,

$$\begin{aligned} \dot{\tilde{\boldsymbol{A}}}(t) &= -\tilde{\boldsymbol{A}}(t) \circ D_{\boldsymbol{z}(t)} f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t), \\ D_{\boldsymbol{z}(t_e)}q \circ \dot{\boldsymbol{A}}(t) &= -D_{\boldsymbol{z}(t_e)}q \circ \boldsymbol{A}(t) \circ D_{\boldsymbol{z}(t)} f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t), \\ \dot{\boldsymbol{A}}(t) &= -\boldsymbol{A}(t) \circ D_{\boldsymbol{z}(t)} f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t). \end{aligned} \tag{13}$$

Moreover, we have initial and final conditions defined as $\boldsymbol{A}(t_s) = D_{\boldsymbol{z}(t_s)}\psi_{\boldsymbol{\theta}}$ and $\boldsymbol{A}(t_e) = D_{\boldsymbol{z}(t_e)}\boldsymbol{z}(t_e) = \boldsymbol{I}$. As a result, to compute $\boldsymbol{A}(t_s)$, we can construct an "augmented" ODE composed of the Neural MODE (2) and the adjoint ODE $\dot{\boldsymbol{A}}(t) = -\boldsymbol{A}(t)D_{\boldsymbol{z}(t)}f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t)$. After computing the IVP of the Neural MODE to obtain $\boldsymbol{z}(t_e)$, we can then compute the differential of the diffeomorphism $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}$ by solving the EVP of the augmented ODE, as shown in Algorithm 1.

Algorithm 1: Differential of diffeomorphism constructed by a Neural MODE

```
Input: start time $t_s$, start end $t_e$, final states $z(t_e)$ and $A(t_e) = I$
1     $s_0 = [z(t_e), I]$
2     % augmented ODE using (2) and adjoint ODE with variable $A$
3     def augmented_dynamics($[z(t), A(t)], t$):
4         return $[f_\theta(z(t), t), -A(t) D_{z(t)} f_\theta(z(t), t)]$
5     end
6     $[z(t_s), D_{z(t_s)} l]$ = ManifoldODESolve($s_0$, augmented_dynamics, $t_e, t_s$)
7 return $z(t_s)$, $D_{z(t_s)} l$
```

To emphasize once more, solving this augmented ODE system backwards requires knowing the end value $\boldsymbol{z}(t_e)$, which can only be obtained by solving the first ODE (Neural MODE) forwards. In other words, this process is essentially similar to the classical neural network backpropagation in the sense that we must first perform forward pass through the network and then back-propagate the error to obtain the gradients.

B Stability Analysis of RSDS

B.1 Lyapunov stability on Riemannian Manifolds

Before we prove the Lyapunov stability of RSDS (4), we first introduce the following theorem

Theorem 3 (Stability of geodesic vector fields)**.** *Let $\mathcal{N}$ be a Riemannian manifold with logarithmic map $\mathrm{Log}_{\boldsymbol{y}} : \mathcal{N} \to \mathcal{T}\boldsymbol{y}\mathcal{N}$ at $\boldsymbol{y} \in \mathcal{N}$. A dynamical system on this manifold can be formulated as,*

$$\dot{\boldsymbol{y}} = k(\boldsymbol{y})\, \mathrm{Log}_{\boldsymbol{y}}(\boldsymbol{y}^*), \tag{14}$$

where $k(\boldsymbol{y}) > 0 \; \forall \boldsymbol{y} \neq \boldsymbol{y}^*$ and $\boldsymbol{y}^*$ is the origin (equilibrium) point on $\mathcal{N}$. The velocity $\dot{\boldsymbol{y}}$ of such a dynamical system points along the geodesic curves converging to $\boldsymbol{y}^*$. Moreover, let a Lyapunov function on such a manifold be formulated as $V(\boldsymbol{y}) := \langle F, F \rangle_{\boldsymbol{y}^*}$, where $F = \mathrm{Log}_{\boldsymbol{y}^*}(\boldsymbol{y})$ defines a vector field composed of the initial velocities of all geodesics departing from the origin $\boldsymbol{y}^*$. This Lyapunov function V satisfies,

$$V(\boldsymbol{y}^*) = 0, \quad \dot{V}(\boldsymbol{y}^*) = 0, \quad V(\boldsymbol{y}) > 0, \; \forall \, \boldsymbol{y} \neq \boldsymbol{y}^*, \quad \dot{V}(\boldsymbol{y}) < 0, \; \forall \, \boldsymbol{y} \neq \boldsymbol{y}^*.$$

As a result, the above dynamical system is globally asymptotically stable on the Riemannian manifold $\mathcal{N}$ with a single equilibrium point $\boldsymbol{y}^*$.

The full proof of Theorem 3 can be found in [23]. Note that the dynamical system defined by $\dot{\boldsymbol{y}} = g_{\boldsymbol{\gamma}}(\boldsymbol{y}) = k_{\boldsymbol{\gamma}}(\boldsymbol{y}) g_n(\boldsymbol{y})$ as in § 3.1, corresponds to the one defined in Theorem 3 by substituting $k(\boldsymbol{y})$ with $\frac{k_{\boldsymbol{\gamma}}(\boldsymbol{y})}{\|\mathrm{Log}_{\boldsymbol{y}}(\boldsymbol{y}^*)\|_2}$. Therefore, it is easy to observe that our canonical vector field on $\mathcal{N}$ is also globally asymptotically stable at the attractor point $\boldsymbol{y}^*$. To analyze the stability properties of the dynamical system (4) on the manifold $\mathcal{M}$, we can construct a new Lyapunov function $\tilde{V}(\boldsymbol{x}) := V(\psi_{\boldsymbol{\theta}}(\boldsymbol{x}))$ by applying the diffeomorphism $\psi_{\boldsymbol{\theta}}$. Since this diffeomorphism $\psi_{\boldsymbol{\theta}}$ is a one-to-one and onto mapping, there exists also a single equilibrium point on $\mathcal{M}$, i.e $\boldsymbol{x}^* = \psi_{\boldsymbol{\theta}}^{-1}(\boldsymbol{y}^*)$. Then, we can prove that $\boldsymbol{x}^* \in \mathcal{M}$ is globally asymptotically stable.

To verify this condition, we first prove that the dynamics under the coordinate change with diffeomorphism $\psi_{\boldsymbol{\theta}}$ indeed corresponds to (4). Given $\boldsymbol{y} = \psi_{\boldsymbol{\theta}}(\boldsymbol{x})$ and $\dot{\boldsymbol{y}} = g_{\boldsymbol{\gamma}}(\boldsymbol{y})$, it follows that,

$$\dot{\boldsymbol{y}} = D_{\boldsymbol{x}} \psi_{\boldsymbol{\theta}}(\boldsymbol{x}) \circ \dot{\boldsymbol{x}}, \quad \text{then,} \quad \dot{\boldsymbol{x}} = (D_{\boldsymbol{x}} \psi_{\boldsymbol{\theta}}(\boldsymbol{x}))^{-1} \dot{\boldsymbol{y}} = \left((D_{\boldsymbol{x}} \psi_{\boldsymbol{\theta}}(\boldsymbol{x}))^{-1} \circ g_{\boldsymbol{\gamma}} \circ \psi_{\boldsymbol{\theta}} \right)(\boldsymbol{x}).$$

Note that the pullback operator $D_{\boldsymbol{y}} \psi^*$ indeed corresponds to $D_{\boldsymbol{x}}(\psi_{\boldsymbol{\theta}}(\boldsymbol{x}))^{-1}$ as discussed in § 3.3. Then, we compute the time derivative of the Lyapunov function $\tilde{V}(\boldsymbol{x}(t))$ as follows,

$$\begin{aligned} \frac{d\tilde{V}(\boldsymbol{x}(t))}{dt} &= \frac{d(\tilde{V} \circ (\psi_{\boldsymbol{\theta}}^{-1} \circ \psi_{\boldsymbol{\theta}}) \circ \boldsymbol{x})(t)}{dt} = \frac{d(\tilde{V} \circ \psi_{\boldsymbol{\theta}}^{-1}) \circ (\psi_{\boldsymbol{\theta}} \circ \boldsymbol{x})(t)}{dt} \\ &= \frac{d(V \circ \boldsymbol{y})(t)}{dt} = \frac{dV(\boldsymbol{y}(t))}{dt}, \end{aligned} \tag{15}$$

where we first introduced the identity mapping $\boldsymbol{I} = \psi_{\boldsymbol{\theta}}^{-1} \circ \psi_{\boldsymbol{\theta}}$ and then applied some regrouping. This implies that the new Lyapunov function $\tilde{V}$ satisfies all Lyapunov conditions defined as,

$$\begin{aligned} &\tilde{V}(\boldsymbol{x}^*) = V(\psi_{\boldsymbol{\theta}}(\boldsymbol{x}^*)) = V(\boldsymbol{y}^*) = 0, \quad \dot{\tilde{V}}(\boldsymbol{x}^*) = \dot{V}(\psi_{\boldsymbol{\theta}}(\boldsymbol{x}^*)) = \dot{V}(\boldsymbol{y}^*) = 0, \\ &\tilde{V}(\boldsymbol{x}) = V(\psi_{\boldsymbol{\theta}}(\boldsymbol{x})) = V(\boldsymbol{y}) > 0, \; \forall \, \boldsymbol{x} \neq \boldsymbol{x}^*, \quad \dot{\tilde{V}}(\boldsymbol{x}) = \dot{V}(\psi_{\boldsymbol{\theta}}(\boldsymbol{x})) = \dot{V}(\boldsymbol{y}) < 0, \; \forall \, \boldsymbol{x} \neq \boldsymbol{x}^*. \end{aligned}$$

We have therefore proved that our RSDS provides globally asymptotically stable vector fields on Riemannian manifolds. From the above derivation, we also showed that a diffeomorphism can be used to describe a *change of coordinates* for Riemannian manifolds, as briefly mentioned in § 2. Thus, we can regard $\boldsymbol{x}$ and $\boldsymbol{y}$ as two coordinate representations for the same underlying dynamical system on $\mathcal{M}$. Hence, the original dynamical system with coordinates $\boldsymbol{x} \in \mathcal{M}$ is characterized by the same stability properties as the canonical dynamical system with coordinates $\boldsymbol{y}$ on $\mathcal{N}$.

B.2 Quasi-global Asymptotic Stability

For Riemannian manifolds of particular topologies, it is not possible to guarantee *global* stability. To begin with, the Poincaré-Hopf theorem establishes that for any vector field v on $\mathcal{M}$, the sum of the Poincaré indices over all the isolated zeroes is equal to the Euler characteristic $\chi(\mathcal{M})$. Global convergence to a single equilibrium point means a single zero with index 1. However, for compact manifold such as the 2-sphere, which has Euler characteristic 2, there must exist at least another zero. This means that global asymptotically stability is not achievable for such a manifold.

To address this problem, we assume the following: Let us consider again the case of the Sphere manifold. For a geodesic vector field defined on the Sphere, the remaining isolated zero corresponds to the cut locus of the attractor point $\mathrm{Cut}(\boldsymbol{x}^*)$, where the vector field converges to. Our assumption is that a geodesic vector field converges to a single attractor $\boldsymbol{x}^*$ for all points on $\mathcal{M}$ except the attractor's cut locus $\mathrm{Cut}(\boldsymbol{x}^*)$. Such an assumption implies that theoretically we may only guarantee "*quasi-global* asymptotic stability". Although our assumption might be restrictive when designing geodesic vector fields on compact Riemannian manifolds (e.g., the Sphere), there exist several non-compact manifolds without cut locus (e.g., the manifold of symmetric positive definite matrices) [43], for which the aforementioned assumption may not be necessary.

C Final RSDS framework

We here prove the final RSDS equation (5) is equivalent to the original diffeormorphism-based learning framework (4) and give more details about network structures for each component. First, let us summarize our framework as discussed in § 3. Given a manifold $\mathcal{M}$, we can evaluate the velocity $\dot{\boldsymbol{x}} \in \mathcal{T}_{\boldsymbol{x}}\mathcal{M}$ at location $\boldsymbol{x} \in \mathcal{M}$ using (4) and the canonical dynamics defined in § 3.1. As a result, we obtain the whole RSDS framework formulated as

$$\dot{\boldsymbol{x}} = (D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ g_{\boldsymbol{\gamma}} \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x}) = D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}}(\dot{\boldsymbol{y}}) \quad \text{with} \quad g_{\boldsymbol{\gamma}}(\boldsymbol{y}) = \begin{cases} k_{\boldsymbol{\gamma}}(\boldsymbol{y})\frac{\text{Log}_{\boldsymbol{y}}(\boldsymbol{y}^*)}{\|\text{Log}_{\boldsymbol{y}}(\boldsymbol{y}^*)\|_2},\ \forall\ \boldsymbol{y} \neq \boldsymbol{y}^* \\ \boldsymbol{0},\ \boldsymbol{y} = \boldsymbol{y}^* \end{cases},$$

where

- $\psi_{\boldsymbol{\theta}} : \mathcal{M} \to \mathcal{N}, \boldsymbol{x} \mapsto \boldsymbol{y}$ is a learnable diffeomorphism parameterized by the Neural MODE,
- $g_{\boldsymbol{\gamma}} : \mathcal{N} \to \mathcal{T}_{\boldsymbol{y}}\mathcal{N}$ is a parameterized geodesic vector field with the equilibrium point $\boldsymbol{y}^*$,
- $D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} : \mathcal{T}\boldsymbol{y}\mathcal{N} \to \mathcal{T}\boldsymbol{x}\mathcal{M}$ is the pullback operator,
- $k_{\gamma}(\boldsymbol{y})$ is a positive scaling factor $\forall\ \boldsymbol{y} \in \mathcal{N}$.

This learning framework is mainly parametrized by $\boldsymbol{\theta}$ (corresp. the diffeomorphism) and $\boldsymbol{\gamma}$ (corresp. the scaling factor of the canonical dynamics). These two parameters sets provide large capacity to learn very complex vector fields. However, the smoothness of the learned vector fields can not be easily guaranteed since it is affected by the aforementioned parameters, which are separately constructed with two independent neural networks. Additionally, since the diffeomorphism is responsible for both the direction and magnitude of the vector field, the diffeomorphism network $\psi_{\boldsymbol{\theta}}$ needs to find a compromise, which potentially weakens the capacity to learn the direction of the vector field. To address these two issues, we introduce the following improvements. We reparameterize the direction and magnitude of the learned vector fields separately, as follows

$$\begin{aligned} \dot{\boldsymbol{x}} &= (D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ k_{\boldsymbol{\gamma}}g_n \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x}) = D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}}\ k_{\boldsymbol{\gamma}}(\boldsymbol{y})\ g_n(\boldsymbol{y}), \\ &= k_{\boldsymbol{\gamma}}(\boldsymbol{y})\ D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}}\ g_n(\boldsymbol{y}) = \tilde{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})\ (D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ g_n \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x}), \end{aligned} \tag{16}$$

where $g_n : \mathcal{N} \to \mathcal{T}\boldsymbol{y}\mathcal{N}$ denotes the normalized geodesics vector field as described in § 3.1, and $\tilde{k}_{\boldsymbol{\gamma}}(\boldsymbol{x}) = k_{\boldsymbol{\gamma}}(\psi^{-1}_{\boldsymbol{\theta}}(\boldsymbol{y}))$ is an intermediate variable only for the purpose of the derivation. We then redefine a new scaling factor $\hat{k}_{\boldsymbol{\gamma}}(\boldsymbol{x}) := \tilde{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})\|(D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ g_n \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x})\|_2$, which leads to

$$\dot{\boldsymbol{x}} = \tilde{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})(D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ g_n \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x}) = \hat{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})\frac{(D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ g_n \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x})}{\|(D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} \circ g_n \circ \psi_{\boldsymbol{\theta}})(\boldsymbol{x})\|_2}, \tag{17}$$

which is the proposed RSDS (5), as defined in § 3. Since there are no fundamental changes on the learning framework with the above modifications, the stability properties remains unchanged as long as the newly defined scaling factor $\hat{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})$ is strictly positive.

To ensure the smoothness of the magnitude part and provide good extrapolation in regions beyond the demonstrations, we leverage RBF networks to parametrize $\hat{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})$. Specifically, we define the scaling factor $\hat{k}_{\boldsymbol{\gamma}}$ as $\hat{k}_{\boldsymbol{\gamma}}(\boldsymbol{x}) = e^{\kappa_{\boldsymbol{\gamma}}(\boldsymbol{x})+\epsilon}$, with $\kappa_{\boldsymbol{\gamma}} : \mathbb{R}^n \supset \mathcal{M} \to \mathbb{R}$, and ϵ being a very small constant to ensure positive definiteness numerically. Note that the RBF network needs to be adapted to the Riemannian setting as $\boldsymbol{x} \in \mathcal{M}$,. To do so, we replace the usual L_2-norm by the Riemannian distance,

$$\kappa_{\boldsymbol{\gamma}}(\boldsymbol{x}) = \sum_i w_i \cdot \phi\left(\frac{d_{\boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{c}_i)}{\sigma_i}\right), \tag{18}$$

where w_i are linear weights, $\boldsymbol{c}_i$ are the RBF centers (obtained via k-means on manifolds), σ_i is similar to the standard deviation for Gaussian distributions, ϕ represents the RBFs, e.g. $\phi(r) = e^{-r^2}$, and $d_{\boldsymbol{x}}$ is the Riemannian distance induced by Riemannian metrics as defined in § 2 and Table 1.

Regarding the parameterization of the diffeomorphism $\psi_{\boldsymbol{\theta}}$, we use a simple fully connected neural network (FCNN). While a classical FCNN can be considered as a general function approximator in Euclidean space, dynamics model $f_{\boldsymbol{\theta}}$ represented as a neural MODE requires to account for constraint that the output of network must lie on the tangent space at the input. Due to this, we add an additional projection operator proju to the last layer of the network to project its output onto the tangent space at the input location. More formally, let us assume that states $\boldsymbol{z}(t)$ lie on the manifold $\mathcal{M}^d$ which is embedded in an ambient space $\mathbb{R}^n$. Given a $\mathcal{C}^1$ network $\eta_{\boldsymbol{\theta}}: \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$, the vector field on the manifold $f_{\boldsymbol{\theta}}: \mathbb{R}^n \times \mathbb{R} \to \mathcal{TM}$ can be obtained as,

$$f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t) = \mathrm{proju}_{\boldsymbol{z}(t)} \circ \eta_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t). \tag{19}$$

Note that the state variable $\boldsymbol{z}$ lies on a Riemannian manifold, but we do not explicitly consider this in the input layer of the network, as we are instead accounting for the ambient space in which the corresponding manifold is embedded.

In summary, the final RSDS learning framework provides several advantages. On the one hand, a separate magnitude parameterization $\hat{k}_{\boldsymbol{\gamma}}(\boldsymbol{x})$ allows us to design the network such as the Riemannian RBF network (18) performs better extrapolation in regions where no training data are available and guarantees smoothness. Since $\hat{k}_{\boldsymbol{\gamma}}$ must be positive definite on the entire manifold, the Riemannian RBF network can provide a constant magnitude value $\hat{k}_{\boldsymbol{\gamma}} = 1$ with $\kappa_{\boldsymbol{\gamma}} = 0$ when $\boldsymbol{x}$ is far away from demonstrations. Note that we tried to use FCNN for this parameterization but it outputs very small values $\hat{k}_{\boldsymbol{\gamma}} \to 0$ for datapoints beyond the demonstrations region, despite we use an exponential function to guarantee positive definiteness. The reason for this is that FCNN can not provide predictable values for inputs which are too dissimilar from the training data, and it turns out to output $\kappa_{\boldsymbol{\gamma}} \to -\infty$ for unseen inputs. On the other hand, the parameterization of the diffeomorphism $\psi_{\boldsymbol{\theta}}$ can completely focus on learning the vector field directions, improving the reproduction accuracy.

C.1 A short discussion on Riemannian approaches for robot motion generation

Note that Riemannian geometry has been also leveraged to design robot motion policies that build on the geometry of classical mechanical systems, as proposed in recent works on Riemannian Motion Policies [44], and more recently in a generalization called Geometric Fabrics [45]. In this context, RSDS may be seen as learning a control policy represented by a first-order dynamical system, which may be used as a motion policy into the RMPs framework, as RMPs provide a geometric robot motion structure that combines several motion polices with associated Riemannian metrics. In other words, we may fuse RSDS skills with additional motion policies, like obstacle avoidance behaviors, under the RMPs framework [44]. Note that RSDS represents a learned first-order dynamic motion policy that does not explicitly consider a specific type of underlying mechanical system. In contrast, RMPs build on Riemannian geometric control to design robot motion policies whose geometry is characterized by, e.g., the associated inertia matrix of the robot dynamics.

It is worth highlighting the difference on the geometric aspects regarding RMPs and RSDS. Broadly speaking, the Riemannian structure of RMPs arises from the Riemannian metric associated to the kinetic energy of the robot dynamics [44], while the Riemannian structure of RSDS comes from the fact that the state variable $\boldsymbol{x}$ of the robot in operational space lies on a Riemannian manifold (due to the orientation representation). Notice that these two different ways in which Riemannian geometry is leveraged in robotics are complementary to each other. On a related note, Riemannian geometry has been recently leveraged to formulate robot motion generation mechanisms that build on geodesics on learned Riemannian manifolds [46, 47]. This approach differs from the aforementioned methods as the Riemannian geometry is not intrinsically given by the system dynamics or geometric constraints of the system state. Instead, the geometric aspect arises from the assumption that trajectories of a robot motion skill define a nonlinear smooth surface that can be interpreted as a data-driven Riemannian manifold.

D Computation of the Pullback Operator

D.1 Pullback operator via constrained optimization

Here we provide more insights about the loss of rank of the differential $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}$ and derive a pullback operator via constrained optimization using the $\mathcal{S}^d$ manifold as a study case. We consider a 2-

Sphere manifold embedded in $\mathbb{R}^3$, whose tangent space has the same dimensionality as the manifold, namely, $\dim(\mathcal{T}_{\boldsymbol{x}}\mathcal{M}) = 2$. This implies that the operator $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})$ should be a mapping between 2-dimensional spaces. However, since we embed the 2-Sphere and its tangent spaces in $\mathbb{R}^3$, the matrix representation for $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})$ is a 3×3 matrix. This matrix is a sort of overparameterization for a mapping between 2-dimensional spaces, and therefore it is rank-deficient and can not be directly inverted. The rank deficiency arises from the geometric constraints of the Riemannian manifold, which impose linear dependencies in the vector space spanned by the matrix columns of $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})$.

Given the relationship between $\dot{\boldsymbol{x}}$ and $\dot{\boldsymbol{y}}$ with $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})\dot{\boldsymbol{x}} = \dot{\boldsymbol{y}}$ and the geometric constraints associated to $\mathcal{S}^d$ manifolds, as discussed in § 3.3, we can compute a solution for $\dot{\boldsymbol{x}}$ via constrained optimization. For our study case on $\mathcal{S}^2$, we augment the matrix $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}$ to include the associated geometric constraint, and then we compute the pseudo-inverse to obtain the final solution, as follows

$$\begin{aligned}
&\begin{bmatrix} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) \\ \boldsymbol{x}^{\mathsf{T}} \end{bmatrix} \dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\boldsymbol{y}} \\ 0 \end{bmatrix}, \quad \begin{bmatrix} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) \\ \boldsymbol{x}^{\mathsf{T}} \end{bmatrix}^{\mathsf{T}} \begin{bmatrix} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) \\ \boldsymbol{x}^{\mathsf{T}} \end{bmatrix} \dot{\boldsymbol{x}} = \begin{bmatrix} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) \\ \boldsymbol{x}^{\mathsf{T}} \end{bmatrix}^{\mathsf{T}} \begin{bmatrix} \dot{\boldsymbol{y}} \\ 0 \end{bmatrix}, \\
&\left[D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{x}\boldsymbol{x}^{\mathsf{T}} \right] \dot{\boldsymbol{x}} = D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} \dot{\boldsymbol{y}}, \\
&\dot{\boldsymbol{x}} = \left[D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{x}\boldsymbol{x}^{\mathsf{T}} \right]^{-1} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} \dot{\boldsymbol{y}}, \\
&D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} = \left[D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{x}\boldsymbol{x}^{\mathsf{T}} \right]^{-1} D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}}.
\end{aligned} \tag{20}$$

This pullback operator $D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}}$ is only specific to $\mathcal{S}^d$, since the geometric constraint considered here is only valid for this type of manifold. This means that if we need to learn stable vector fields on different Riemannian manifolds, we will need to manually define the corresponding constraints. These constraints can be highly non-linear for other Riemannian manifolds (e.g., matrix manifolds such as the space of symmetric positive-definite matrices), which may not accept a close-form solution as our previous study case. Therefore, although this approach is a potential solution, it is neither general nor scalable for computing the pullback operator.

D.2 Pullback operator via modified adjoint method

Since we can solve an EVP of the Neural MODE (2) from t_e to t_s to obtain $\psi^{-1}_{\boldsymbol{\theta}}$, we can consider the same adjoint method in App. A.4 to compute the pullback operator $D_{\boldsymbol{y}}\psi^{\star}_{\boldsymbol{\theta}} = D_{\boldsymbol{y}}(\psi^{-1}_{\boldsymbol{\theta}})$. We design a new loss function $\mathcal{L}_i = q_i(\boldsymbol{z}(t_s))$, where q_i is the i-th component of an arbitrary function $q : \mathcal{M}^d \to \mathbb{R}^d$. Then, we define a new adjoint variable $\boldsymbol{A}^*(t) = D_{\boldsymbol{z}(t)}(\psi^{-1}_{\boldsymbol{\theta}})$, similarly to computation of differential $D_{\boldsymbol{x}}\psi_{\boldsymbol{\theta}}$ in App. A.4. As a result, we obtain an adjoint ODE $\dot{\boldsymbol{A}}^*(t) = -\boldsymbol{A}^*(t) D_{\boldsymbol{z}(t)} f_{\boldsymbol{\theta}}(\boldsymbol{z}(t), t)$ with starting value $\boldsymbol{A}(t_s) = D_{\boldsymbol{z}(t_s)}\boldsymbol{z}(t_s) = \boldsymbol{I}$ and end value $\boldsymbol{A}^*(t_e) = D_{\boldsymbol{z}(t_e)}(\psi^{-1}_{\boldsymbol{\theta}}) = D_{\boldsymbol{y}}(\psi^{-1}_{\boldsymbol{\theta}})$. Then, solving the IVP of augmented ODEs composed of the Neural MODE (2) and adjoint ODE with adjoint variable $\boldsymbol{A}^*(t)$ allows us to compute the pullback operator $D_{\boldsymbol{y}}(\psi^{-1}_{\boldsymbol{\theta}})$, as described in Algorithm 2.

Algorithm 2: Diffeomorphism and differential of inverse diffeomorphism

```
Input: start time $t_s$, start end $t_e$, initial states $z(t_s) = x$ and $A^*(t_s) = I$
1     $s_0 = [x, I]$
2     % augmented ODE using (2) and adjoint ODE with variable $A^*$
3     def augmented_dynamics($[z(t), A^*(t)], t$):
4         return $[f_\theta(z(t), t), -A^*(t) D_{z(t)} f_\theta(z(t), t)]$
5     end
6     $[z(t_e), D_{z(t_e)}\psi^\star_\theta]$ = ManifoldODESolve($s_0$, augmented_dynamics, $t_s, t_e$)
7 return $z(t_e)$, $D_{z(t_e)}\psi^\star_\theta$
```

In the following we provide the proof for computing the differential of the inverse diffeomorphism (8). In § 3.2, we "discretize" the diffeomorphism using (6), which overcomes the need for integrators on Riemannian manifolds. Analogously, we can formulate the inverse diffeomorphism

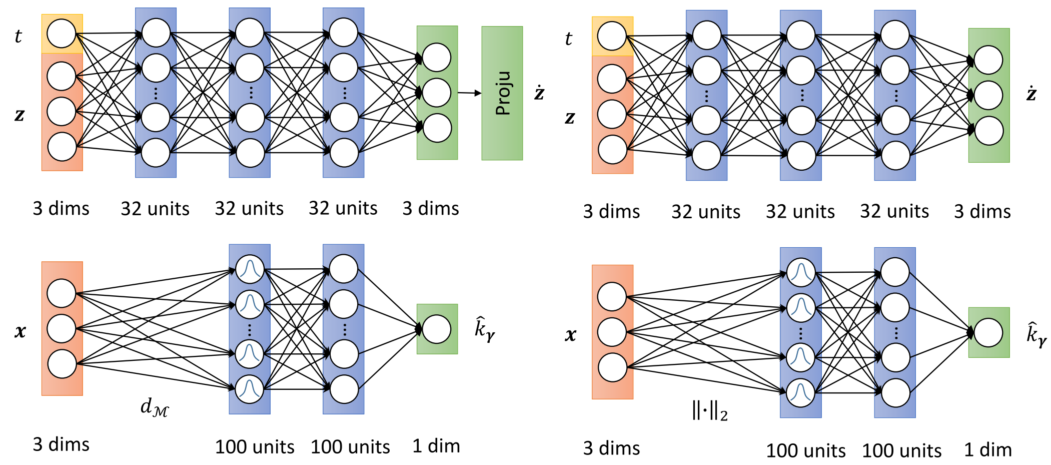


Figure 8: Neural network architectures for *RSDS* and *EuclideanFlow*. **Left**: In the *RSDS* setting, FCNN uses an extra projection operator in the output layer, and the RBF network uses the Riemannian distance $d_{\mathcal{M}}$. **Right**: In the *EuclideanFlow* setting, the RBF network uses the Euclidean distance computed by the L_2-norm.

$\psi_{\boldsymbol{\theta}}^{-1} : \boldsymbol{y} = \boldsymbol{z}_k \mapsto \boldsymbol{z}_0 = \boldsymbol{x}$ as,

$$\begin{aligned} \psi_{\boldsymbol{\theta}}^{-1} &= \mathrm{Log}_{\boldsymbol{z}_0}^{-1} \circ \hat{\psi}_{\boldsymbol{\theta},0}^{-1} \circ \mathrm{Exp}_{\boldsymbol{z}_0}^{-1} \circ \ldots \circ \mathrm{Log}_{\boldsymbol{z}_{k-1}}^{-1}, \circ \hat{\psi}_{\boldsymbol{\theta},k-1}^{-1} \circ \mathrm{Exp}_{\boldsymbol{z}_{k-1}}^{-1}, \\ &= \mathrm{Exp}_{\boldsymbol{z}_0} \circ \hat{\psi}_{\boldsymbol{\theta},0}^{-1} \circ \mathrm{Log}_{\boldsymbol{z}_0} \circ \ldots \circ \mathrm{Exp}_{\boldsymbol{z}_{k-1}} \circ \hat{\psi}_{\boldsymbol{\theta},k-1}^{-1} \circ \mathrm{Log}_{\boldsymbol{z}_{k-1}} . \end{aligned} \tag{21}$$

Now, we focus on a single component $\boldsymbol{z}_i = (\mathrm{Exp}_{\boldsymbol{z}_i} \circ \hat{f}_i^{-1} \circ \mathrm{Log}_{\boldsymbol{z}_i})\boldsymbol{z}_{i+1}$ from (21), and analyze its derivatives, i.e. $D_{\boldsymbol{z}_{i+1}}\boldsymbol{z}_i = D_{\boldsymbol{w}_i(t_{i,s})} \mathrm{Exp}_{\boldsymbol{z}_i} \circ D_{\boldsymbol{w}_i(t_{i,e})} \hat{\psi}_{\boldsymbol{\theta},i}^{-1} \circ D_{\boldsymbol{z}_{i+1}} \mathrm{Log}_{\boldsymbol{z}_i}$. Since $\hat{\psi}_{\boldsymbol{\theta},i} : \boldsymbol{w}_i(t_{i,s}) \mapsto \boldsymbol{w}_i(t_{i,e})$ and $\hat{\psi}_{\boldsymbol{\theta},i}^{-1} : \boldsymbol{w}_i(t_{i,e}) \mapsto \boldsymbol{w}_i(t_{i,s})$ respectively correspond to solving the equivalent ODE (3) forwards and backwards in time in a tangent space (i.e. a Euclidean space), the differential of these mappings corresponds to classical partial derivatives. This allows us to leverage Algorithm 2 with the classical ODE solver in Euclidean space to compute $\hat{\psi}_{\boldsymbol{\theta},i}^{-1}$ and its partial derivatives.

E Network Architecture

In the illustrative examples on $\mathcal{S}^2$, under the *EuclideanFlow* setting, we used a fully-connected neural network with an input vector on $\mathbb{R}^4$ (i.e., the 3-dimensional state $\boldsymbol{x}$ and time), and 3 hidden layers each, with 32 hidden units. We used $\tanh$ as activation function to guarantee a $\mathcal{C}^1$-bounded mapping for modeling the Neural MODE. The RSDS architecture has an additional projection operator proju on the network head to project the output on the tangent space of manifolds (see App. A.1). The scaling factor $\hat{k}_{\gamma}$ is generated using a network composed of an RBF layer and a linear layer without bias. In the real robot experiments, we trained our model on $\mathbb{R}^3 \times \mathcal{S}^3$ (i.e. position and orientation of the end-effector) with the same architecture as our illustrative experiments, except that we use 16 hidden units for faster computations. The network architectures are shown in Fig. 8.

F Extended results

F.1 LASA dataset on $\mathcal{S}^2$

In this section, we provide an extended set of experiments to further support the results presented in § 4. The extended results include a complete set of experiments on datasets: P, G, W, and MultiModels, as displayed in Fig. 9. Moreover, we carried out three additional experiments on new letters from the LASA dataset, specifically: SharpC, Spoon and S, whose trajectories significantly differ from the datasets used in the main paper, as shown in Fig. 11. In addition, an extra set of results using *Projected Euclideanflow* is added to the comparison between RSDS and *EuclideanFlow*. As mentioned in § 4, this alternative method projects trajectories onto the manifold after computing

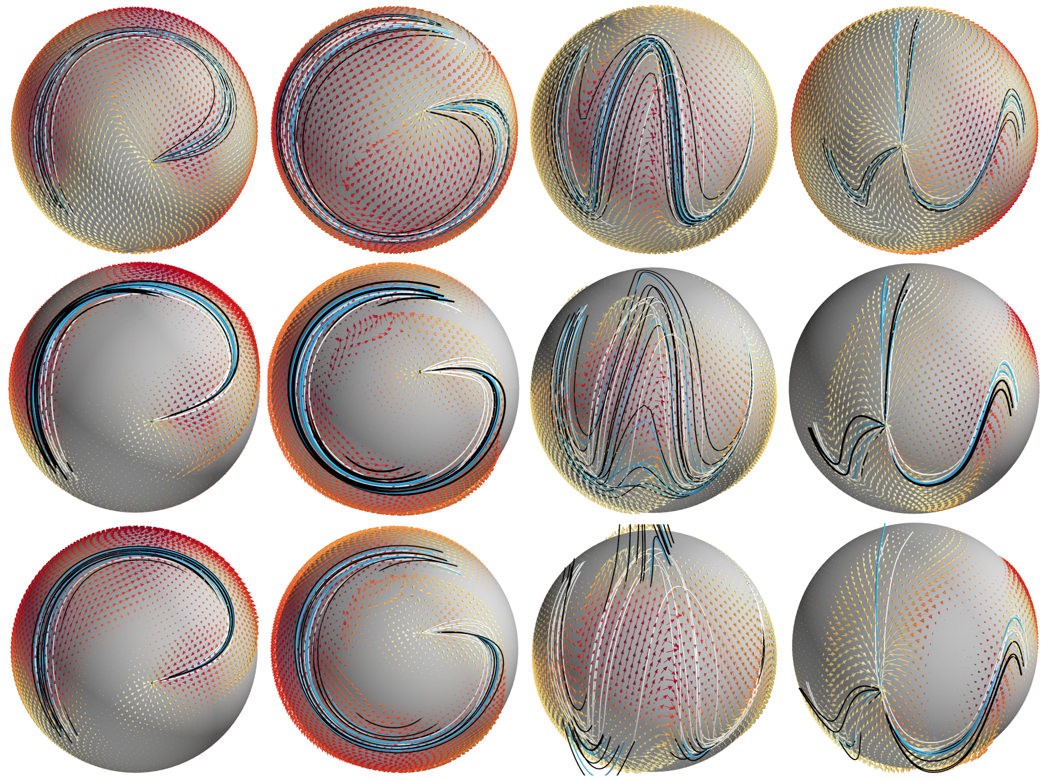


Figure 9: Experiments on LASA dataset on $\mathcal{S}^2$ for datasets: P, G, W, and MultiModels. The learned vector field is depicted by arrows (color-coded based on the magnitude), and the demonstrations are shown as white curves. The blue and black trajectories are reproductions starting at the same initial points as the demonstrations and randomly-sampled points around them, respectively. The first row shows the results for the RSDS approach, and the next two rows show the *Projected EuclideanFlow* and *EuclideanFlow* results.

the integration in the Euclidean space. Figure 10 shows the comparison for learning efficiency between the two methods, RSDS and *EuclideanFlow*. Here, the RSDS method generally requires fewer training epochs than *EuclideanFlow*. Note that the results regarding *Projected Euclideanflow* and *Euclideanflows* share the same models, therefore, *Projected EuclideanFlow* is omitted from this comparison.

Figure 9 shows the demonstrations as white curves for all datasets, the corresponding learned vector fields and the reproduced trajectories. The latter, depicted as blue and black trajectories, are rollouts starting from the same initial position as the demonstrations and randomly-sampled points around them, respectively. From top to bottom, the rows correspond to trajectories computed using RSDS, *Projected EuclideanFlow*, and *EuclideanFlow*. In Fig. 9, the datasets are ordered from left to right (i.e. from P to W) based on their difficulty. For example, the W dataset contains multiple sharp turns, demanding a more expressive learning model in comparison to P dataset. Note that the more complex the vector field is, the more prominent the shortcomings of Euclidean approaches are.

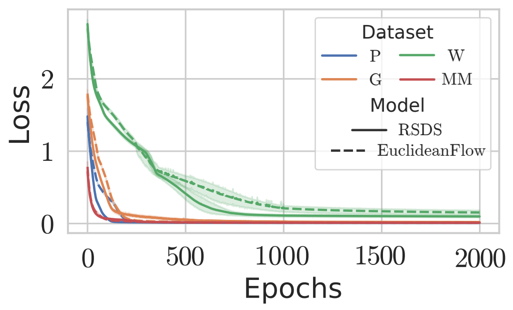


Figure 10: Loss values over 2000 epochs for datasets: P, G, W, and MultiModels. Several models were trained for each dataset, with the lines and the shaded regions representing the mean and standard deviation.

To quantitatively demonstrate the differences between *RSDS* and Euclidean approaches (i.e., *EuclideanFlow* and *Projected EuclideanFlow*), Fig. 12-*bottom-left*, and -*top-right* show the *dynamic*

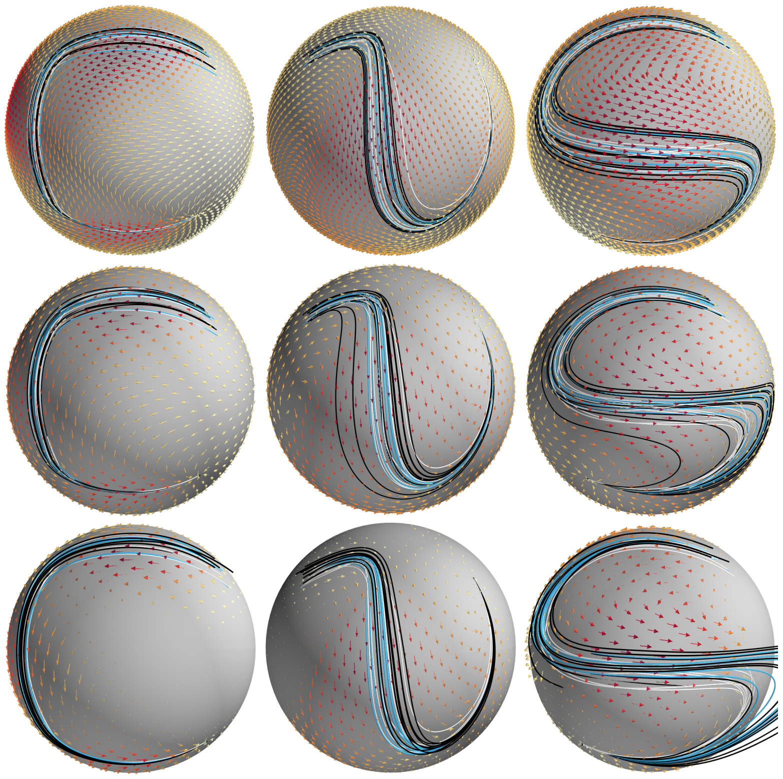


Figure 11: Additional experiments on LASA dataset on $\mathcal{S}^2$ for datasets: SharpC, Spoon, and S. The learned vector field is depicted by arrows (color-coded based on the magnitude), and the demonstrations are shown as white curves. The blue and black trajectories are reproductions starting at the same initial points as the demonstrations and randomly-sampled points around them, respectively. The first row shows the results for the RSDS approach, and the next two rows show the *Projected EuclideanFlow* and *EuclideanFlow* results.

time warping distance (DTWD) as a measure of reproduced position trajectory accuracy, and the *mean squared error* (MSE) of the velocities reproduction. Additionally, the success rate (i.e., convergence to the attractor) of our approach and the baseline can be seen in Fig. 12-*top-left*, where *RSDS* always reproduces stable trajectories. As mentioned in § 4, for a fair comparison in stability analysis, we used the *Projected EuclideanFlow* instead of *EuclideanFlow*. To clearly show the intention behind this, Fig. 13 and Fig. 14 provide the results computed by *EuclideanFlow* in the last row. Figures 13 and 14 show the stability evaluation of the reproduced trajectories on the learned vector field. Here, 1000 trajectories were generated using initial states uniformly sampled on $\mathcal{S}^2$. The successful and failed trajectories are colored as green and red, respectively. From top to bottom, the rows correspond to trajectories computed using RSDS, *Projected EuclideanFlow*, and *EuclideanFlow*, respectively. It is evident that all the *RSDS* trajectories succeeded, while some the *EuclideanFlow* trajectories failed to converge despite the projection.

In order to further show how unstable trajectories arise, we provide additional plots displaying a different view of some examples of *Projected EuclideanFlow* in Fig 15. We can see that additional spurious attractor points can appear on $\mathcal{S}^2$ if we do not explicitly consider the geometric constraints of the data. Then, instead of converging to the true attractor, unstable trajectories diverge to these undesired points. This phenomenon might not occur when learning vector fields displaying simple dynamics, but in general, geometry-unaware methods may be prone to it. Therefore, when using *EuclideanFlow*, we cannot provide stability guarantees on Riemannian manifolds and this may lead to catastrophic results for real-world applications. However, this issue can be theoretically and practically overcome by *RSDS*. Additionally, when compared against *projected EuclideanFlow*, *RSDS* usually generates more consistent and smoother trajectories around the demonstrations, which can be clearly observed, for example, for the S shape.

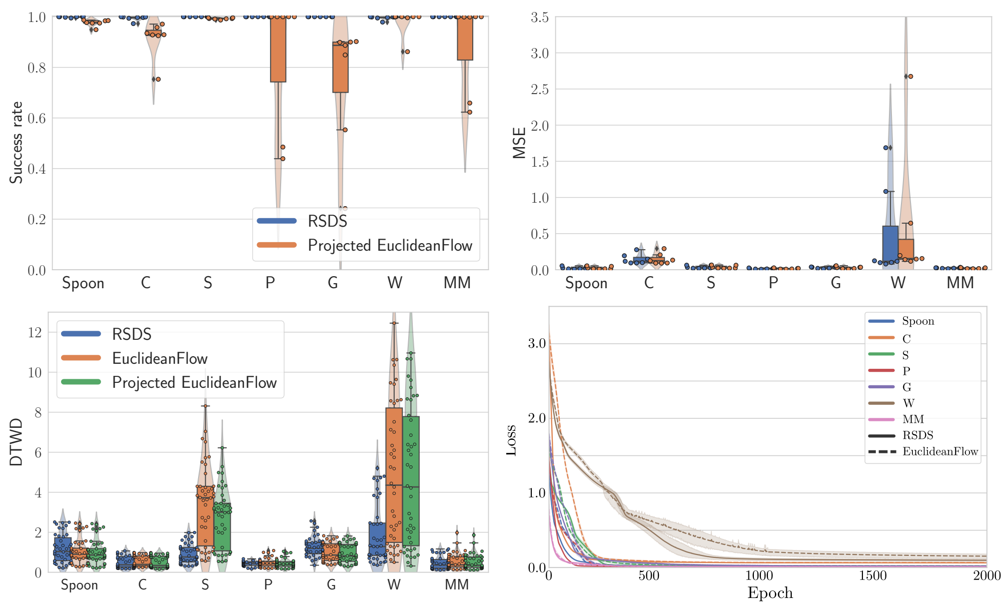


Figure 12: *Top-left*: Average success rate of *RSDS* and *EuclideanFlow* over randomly-sampled initial points on $\mathcal{S}^2$ using 7 different trained models indicated as points. *Top-right:* Average mean square error (MSE) between observed and predicted velocity over data points in the test trajectories indicated as points. *Bottom-left:* Dynamic time warping distance (DTWD) between demonstrations and reproductions. *Bottom-right*: Loss values over $2000 epochs$ for all tested LASA datasets. Several models were trained for each dataset, with the lines and the shaded regions representing the mean and standard deviation.

F.2 Real robot experiments on $\mathbb{R}^3 \times \mathcal{S}^3$

F.2.1 Data Pre-processing:

Before using the demonstrations collected through kinesthetic teaching, a low-pass filter was applied to smooth out the trajectories. In addition, the trajectories are slightly shifted to guarantee that all demonstrations converge to common target. Although, this alteration in position trajectories can be achieved using classical parallel shifting, for quaternion trajectories the *parallel transport* operator on a 3-Sphere is required to accomplish the shifting operation. For sake of completeness, Fig. 16 shows the replay of one of the demonstrations (left picture), as well as the normal and perturbed reproductions shown in Fig. 5.

F.3 Stability on Robotic Tasks

To provide further evidence for the practical use of *RSDS*, we design new experiments on simulated robotic tasks, where we only require the end-effector to follow demonstrated quaternion trajectories while keeping its position fixed. The quaternion demonstrations are synthetically generated on $\mathcal{S}^3$. We apply the same experimental settings as the ones described for the illustrative LASA dataset experiments on $\mathcal{S}^2$, detailed in § 4.1, for both *RSDS* and *EuclideanFlow*, except that an additional hidden layer is added to improve models' expressiveness. We reproduce several quaternion trajectories, using the learned vector fields, starting from randomly-sampled initial points around the demonstrations for both *RSDS* and *Projected EuclideanFlow*. For the latter, we project the learned vector fields onto $\mathcal{S}^3$, similarly to the experiments reported in § 4.1 of the main paper. Note that this projection is necessary to guarantee that the resulting integral curves lead to proper unit quaternion references. Additionally, we use a Cartesian impedance controller to track the reference signals. Figure 17 shows the reconstructed trajectories on $\mathcal{S}^3$ using *Projected EuclideanFlow* corresponding to the G dataset.

In Fig. 17a, we can observe that some trajectories (depicted in red) generated by *Projected EuclideanFlow* do not to reach the target if the initial point moderately differs from the demonstrations, which often happens in real-world settings. In contrast, *RSDS* succeeds in reproducing stable

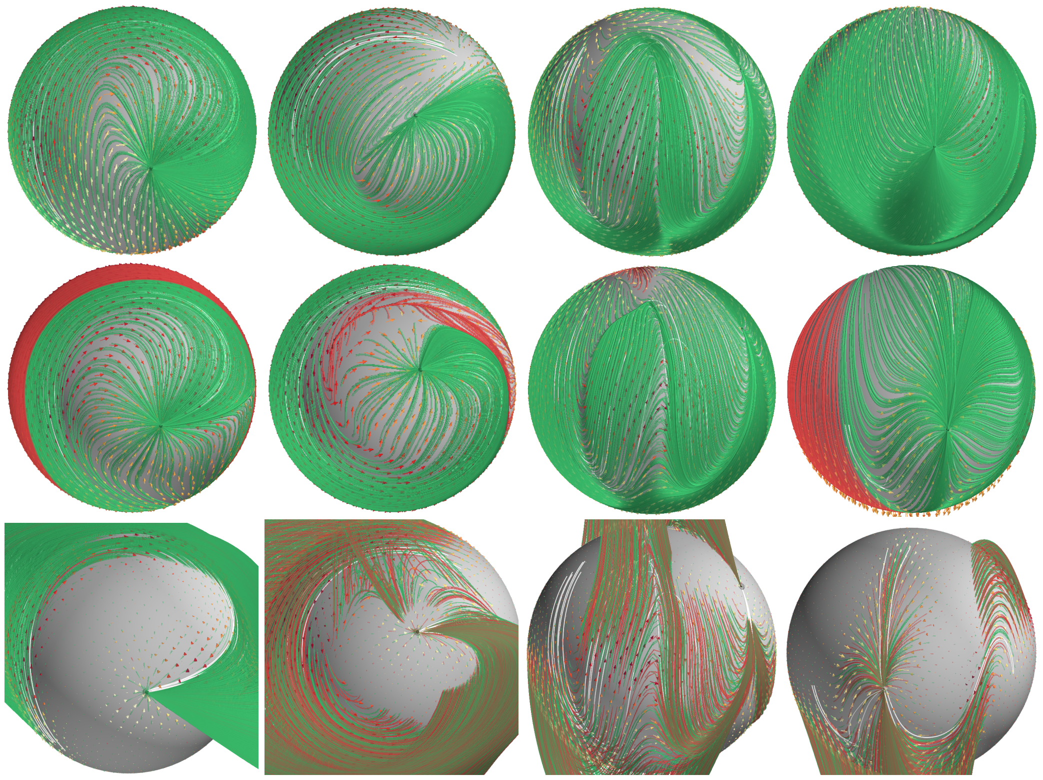


Figure 13: Evaluation of the reproduction stability on the learned vector fields. 1000 trajectories were generated using initial states uniformly sampled on $\mathcal{S}^2$. The path integral for each initial state was computed on the vector field during a fixed time. The successful and failed trajectories are indicated in green and red, respectively. From top to bottom, the rows corresponds to trajectories computed using RSDS, *Projected EuclideanFlow*, and *EuclideanFlow*. From left to right, each column corresponds to P, G, W, and MultiModels demonstrations.

trajectories despite the initial conditions being different to the training data, as shown in Fig. 17b. To visualize the effects that this may have in real-world applications, we show the final robot end-effector pose reached by our *RSDS* method and *Projected EuclideanFlow* for two different simulated experiments in Fig. 18, where Figs. 18a and 18b correspond to the experiments reported in Figs. 17a and 17b, respectively. As pointed out previously, *Projected EuclideanFlow* diverges and therefore reaches an undesired final end-effector pose. However, the final end-effector poses reached by *RSDS* are in sharp contrast to the *Projected EuclideanFlow* results, as our method successfully converges to the desired target (represented by the green robot arm in Fig. 18).

F.4 Runtime of RSDS

We measured the runtime for *RSDS* and *EuclideanFlows* on a PC with Intel Xeon W-1250P CPU and 31 Gigabytes of memory. Table 2 provides reference values of runtime and shows that *RSDS* runs around 2 times slower than *EuclideanFlow*, as the *RSDS* model requires solving IVPs on Riemannian manifolds. However, there is still a lot of room for improvement in terms of speeding up the *RSDS* implementation, which is worthwhile for future research.

	1	2	3	Mean
RSDS	198	200	197	198
EF	90	89	91	90

Table 2: The average runtime (in milliseconds) of *RSDS* and *EuclideanFlow* over 100 iterations in three separate experiments.

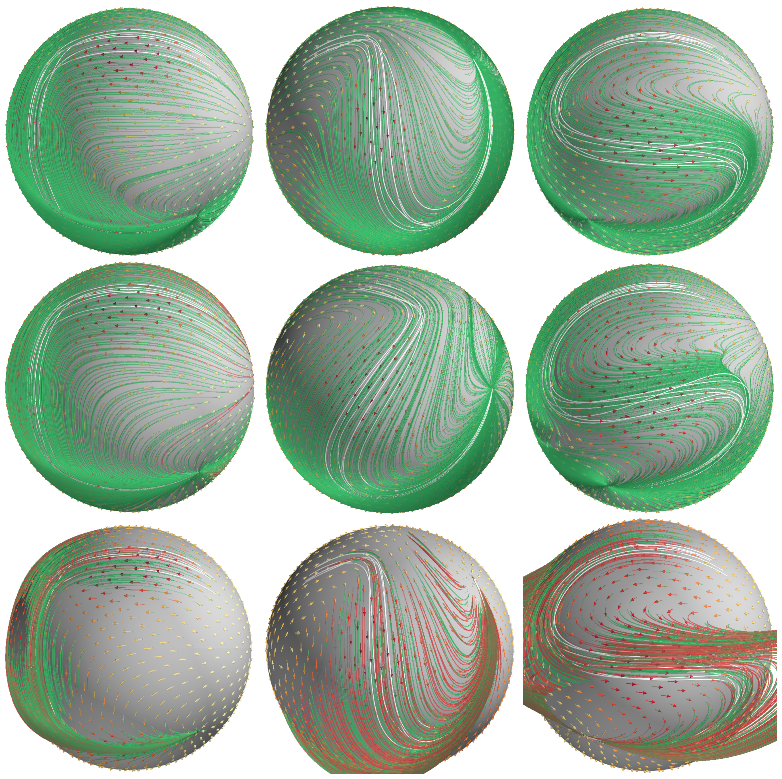


Figure 14: Additional evaluation of the reproduction stability on the learned vector fields. Here, 1000 trajectories were generated using initial states uniformly sampled on $\mathcal{S}^2$. The path integral for each initial state was computed on the vector field during a fixed time. The successful and failed trajectories are indicated in green and red, respectively. From top to bottom, the rows corresponds to trajectories computed using *RSDS*, *Projected EuclideanFlow*, and *EuclideanFlow*. From left to right, each column corresponds to SharpC, Spoon, and S demonstrations.

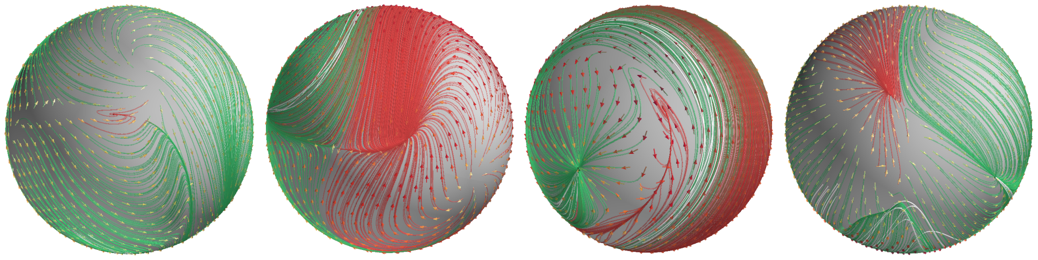


Figure 15: Spurious attractors on vector fields learned with geometry-unaware *Projected EuclideanFlow*. Plots show a different view angle of the reproduction stability on the learned vector fields using *Projected EuclideanFlow*. From left to right, each plot corresponds to Spoon, P, G, and W datasets.

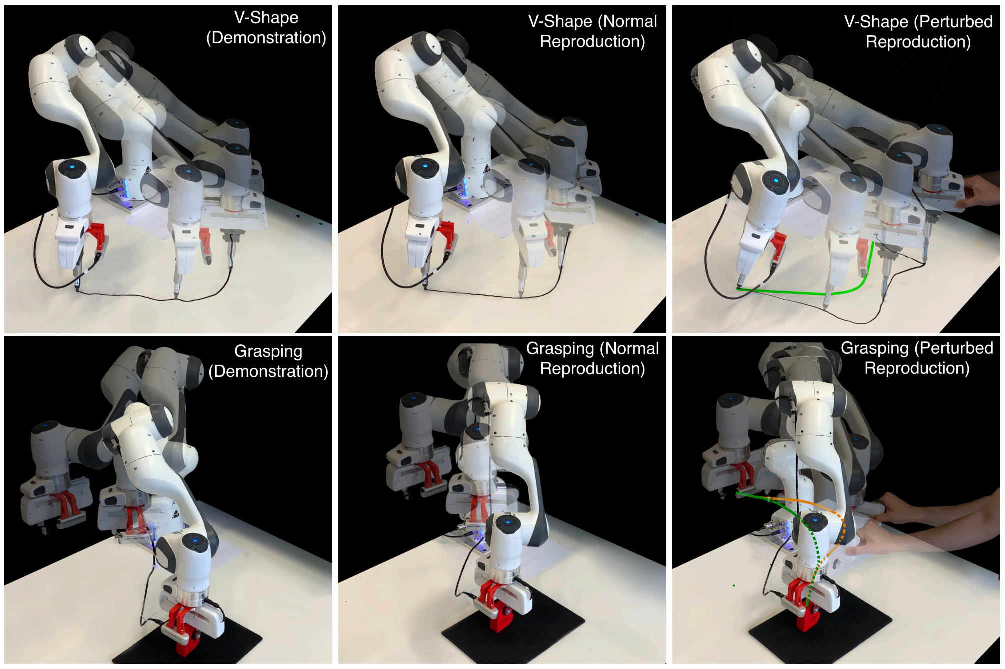
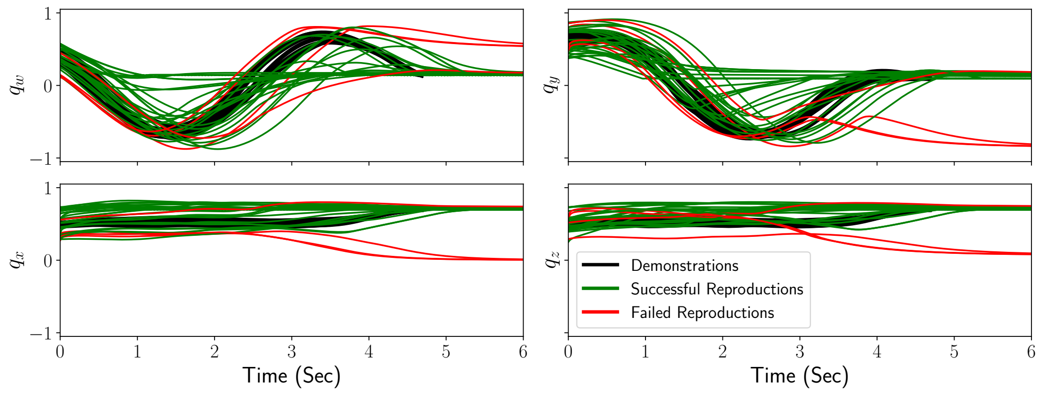
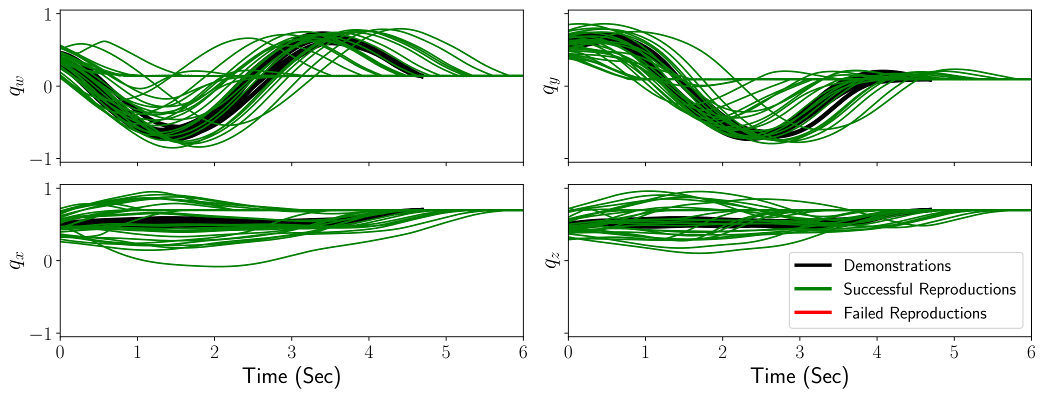

V-Shape (Demonstration)
V-Shape (Normal Reproduction)
V-Shape (Perturbed Reproduction)
Grasping (Demonstration)
Grasping (Normal Reproduction)
Grasping (Perturbed Reproduction)

Figure 16: *Left:* The demonstrated trajectories depicted by superimposing images from different time frames. The transparent robot arms depict the trace of the motion trajectory. Here, the robot performs V-shape DrawingTask (top row) and GraspingTask with 90 degrees rotation (bottom row). *Middle:* The motion reproduced using learned vector field on the product manifold $\mathbb{R}^3 \times \mathcal{S}^3$. *Right:* The motion reproduced using learned vector field on the product manifold $\mathbb{R}^3 \times \mathcal{S}^3$ when applying perturbation.

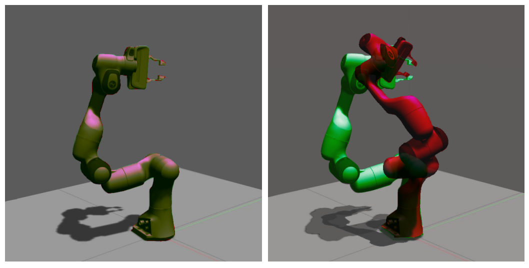


(a) Reproduction of quaternion trajectories with *Projected EuclideanFlow*.

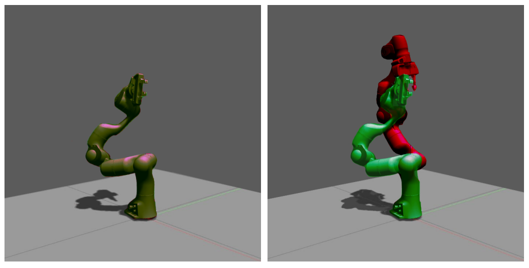


(b) Reproduction of quaternion trajectories with *RSDS*.

Figure 17: Time-series plot of quaternion trajectories obtained from the first-order dynamical systems learned by *Projected EuclideanFlow* and *RSDS*, corresponding to Experiment 1 (see Fig. 18a). The quaternion trajectories reproduced by *Projected EuclideanFlow* reveal several integral curves that diverge, therefore winding up distant from the target. These results show the inability to reconstruct stable quaternion trajectories when disregarding the geometric constraints arising from $\mathcal{S}^3$. In contrast, the *RSDS* integral curves succeed to converge and reach the target, demonstrate its ability to reconstruct stable quaternion trajectories on $\mathcal{S}^3$.



(a) Experiment 1



(b) Experiment 2

Figure 18: Stability experiments on a simulated Franka-Emika robot. The green robotic arm represents the end-effector pose target extracted from demonstrations, and the red robot displays the final end-effector pose achieved during reproduction. For each experiment, the **left** snapshot displays the *RSDS* reproduction, and the **right** picture shows the *EuclideanFlow* result.