Riemannian Motion Policy Fusion through Learnable Lyapunov Function Reshaping

Mustafa Mukadam[1], Ching-An Cheng[1], Dieter Fox[2,3], Byron Boots[2,3], and Nathan Ratliff[3]

[1]Georgia Institute of Technology, USA
[2]University of Washington, USA
[3]NVIDIA, USA

Abstract: RMPflow is a recently proposed policy-fusion framework based on differential geometry. While RMPflow has demonstrated promising performance, it requires the user to provide sensible subtask policies as Riemannian motion policies (RMPs: a motion policy and an importance matrix function), which can be a difficult design problem in its own right. We propose RMPfusion, a variation of RMPflow, to address this issue. RMPfusion supplements RMPflow with weight functions that can hierarchically reshape the Lyapunov functions of the subtask RMPs according to the current configuration of the robot and environment. This extra flexibility can remedy imperfect subtask RMPs provided by the user, improving the combined policy's performance. These weight functions can be learned by back-propagation. Moreover, we prove that, under mild restrictions on the weight functions, RMPfusion always yields a globally Lyapunov-stable motion policy. This implies that we can treat RMPfusion as a structured policy class in policy optimization that is guaranteed to generate stable policies, even during the immature phase of learning. We demonstrate these properties of RMPfusion in imitation learning experiments both in simulation and on a real-world robot.

Keywords: Reactive motion generation, Structured end-to-end learning

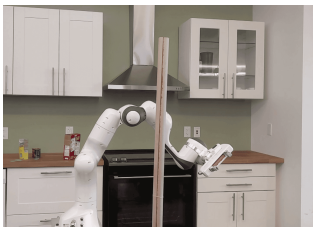
1 Introduction

Motion planning and control are core techniques to robotics [1, 2, 3]. Ideally a good algorithm must be both computationally efficient and capable of navigating a robot safely and stably across a wide range of environments. Several systems were recently proposed to address this challenge [4, 5, 6] through closely integrating planning and control techniques. In particular, RMPflow [6] is designed to combine reactive policies [7, 8, 9, 10, 11] and planning [12]. Based on differential geometry, RMPflow offers a unified treatment of the nonlinear geometries arising from a robot's internal kinematics and task spaces (e.g. environments with obstacles). Given user-provided subtask motion policies expressed in the form of Riemannian Motion Policies (RMPs) [13] (i.e. a second-order motion policy along with a matrix function that acts as a directional importance weight), RMPflow can synthesize a global motion policy for the full task in an efficient and geometrically consistent manner and has desirable properties such as stability and being coordinate-free [6].

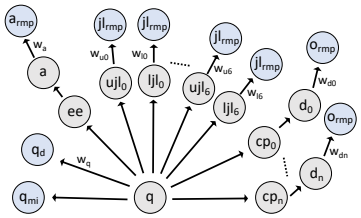
RMPflow has been successfully applied in many applications [14, 15, 16, 17, 18]. But RMPflow is not perfect. Despite its advancement, practical usage difficulties remain. For instance, the user must provide RMPs with matrix functions that properly describe the characteristics of subtask motion policies in order to build an effective RMPflow system. Otherwise, the final global policy may have unsatisfactory performance, though still being geometrically consistent (with respect to some meaningless geometric structure). This poses a challenge for practitioners who are inexperienced in control systems, or for designing policies of tasks where the full state is hard to describe.

In this paper we introduce a hierarchical Lyapunov function reshaping scheme into RMPflow to remedy the requirement of providing high-quality subtasks RMPs from the user. The modified algorithm, called RMPfusion, adds a set of multiplicative weight *functions* in the policy fusion step of RMPflow, which can be manually parametrized or modeled by function approximators (like neural networks). In a high level, these weight functions let RMPfusion adapt between multiple versions of RMPflow according to the robot's configuration and the environment. (RMPflow is

3rd Conference on Robot Learning (CoRL 2019), Osaka, Japan.



(a) A snapshot of the experiment.



(b) The RMP-tree* used for the Franka robot.

Figure 1: Franka robot navigating around an obstacle using RMPfusion with the RMP-tree*. Gray nodes show task spaces, blue nodes show subtask RMPs, and weight functions are shown on the respective edges where they are defined. See Section 4.2 for details.

RMPfusion with constant weights.) Therefore, an immediate benefit of our new algorithm is the extra design flexibility added to RMPflow. Compared with RMPflow, RMPfusion allows the user to start with simpler subtask RMPs and gradually build up more complex behaviors through the use of weight functions.

Interestingly, these weight functions in general do not just linearly combine outputs of motion policies as in [19, 20]. Instead they hierarchically reshape the inherent Lyapunov functions of the provided subtask policies, overall giving a nonlinear effect on the global policy RMPfusion creates. We prove that RMPfusion produces a policy that is Lyapunov-stable with respect to this reshaped Lyapunov function given by the weight functions. Therefore, the overall the system is stable, as long as the weight functions are non-negative and non-degenerate.

These properties suggest that we can treat RMPfusion as a structured policy class in reinforcement/imitation learning and optimize the weight functions to improve the combined policy's performance. Importantly, as RMPfusion remains stable under minor restriction on weight functions, we arrive at a policy class that is guaranteed to be stable, even during the immature phase of learning. Thus, RMPfusion is suitable for learning with safety constraints; for example, we can ensure that certain safe policies (like collision avoidance) are the only ones activated when the robot is facing extreme conditions. These theoretical properties of RMPfusion are verified in imitation learning tasks, in both simulations and on a real-world robot (Figure 1). Not only did RMPfusion learn to mimic the expert policy, but it also yielded stable policies throughout the learning.

Notation We use boldface to denote vectors and matrices. For derivatives, we use both the symbols, ∇ and ∂, which are transpose to each other: for $\mathbf{x} \in \mathbb{R}^m$ and a differential map $\mathbf{y} : \mathbb{R}^m \to \mathbb{R}^n$, we write $\nabla_{\mathbf{x}}\mathbf{y}(\mathbf{x}) = \partial_{\mathbf{x}}\mathbf{y}(\mathbf{x})^\top \in \mathbb{R}^{m\times n}$. For convenience we use $[\cdot]^:$ to denote horizontal concatenation in composing a matrix; for example, we write $\mathbf{M} = [\mathbf{m}_i]_{i=1}^m \in \mathbb{R}^{m\times m}$ for $\mathbf{m}_i \in \mathbb{R}^m$. We use $\mathbb{R}_+^{m\times m}$ to denote symmetric, positive semi-definite matrices in $\mathbb{R}^{m\times m}$. We assume all manifolds and maps are sufficiently smooth. Coordinates of manifolds mentioned here will be assumed local.

2 Background

2.1 Motion Policies

We treat a robot's configuration as a point on some smooth manifold and model its motion through differential equations. Assume the robot has been feedback linearized. We are interested in motions that can be described by second-order differential equations. We call these differential equations, *motion policies*, as they essentially define how the robot reacts given the current state (i.e. the configuration and the velocity). Suppose the robot lives on a d-dimensional manifold $\mathcal{C}$ (the configuration space) with coordinate $\mathbf{q} \in \mathbb{R}^d$. We say a map π is a motion policy on $\mathcal{C}$ if the robot travels according to the differential equation $\ddot{\mathbf{q}} = \pi(\mathbf{q}, \dot{\mathbf{q}})$, where $\dot{}$ denotes time derivative.

While motion policies can be specified directly on the configuration space $\mathcal{C}$, it is often more natural to define them indirectly on the *task space* $\mathcal{T}$ (another manifold) that describes the target application and then transform them back to the configuration space $\mathcal{C}$. This is the central idea underlined in operational space control [7]. For instance, suppose $\mathcal{T}$ has a coordinate $\mathbf{x}$ that is related to $\mathcal{C}$ through a task map ψ (i.e. $\mathbf{x} = \psi(\mathbf{q})$). A popular way to design motion policies is through the analogy of a mass-spring-damper system [7, 8, 9, 10]. These policies can be written in the task space $\mathcal{T}$ as

$$\mathbf{M}(\mathbf{x})\ddot{\mathbf{x}} + \mathbf{C}(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}} = -\mathbf{K}(\mathbf{x} - \mathbf{x}_g) - \mathbf{B}\dot{\mathbf{x}} \tag{1}$$

where $\mathbf{M}(\mathbf{x}) \succ 0$ is the inertia matrix (this inertia might not necessarily be the physical inertia of the robot), $\mathbf{C}(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}}$ is the Coriolis term with respect to $\mathbf{M}$, $\mathbf{K} \succeq 0$ is the stiffness matrix, $\mathbf{B} \succ 0$ is the

damping matrix, and $\mathbf{x}_g$ is the goal in $\mathcal{T}$. Using (1), the robot's behavior can be easily understood: it travels toward $\mathbf{x}_g$ along a trajectory regulated by damper $\mathbf{B}$.

2.2 RMPflow

However, specifying a global task-space policy, like above, can sometimes still be a daunting task, as the task requirement becomes complicated. RMPflow is designed to address this issue [6]. Rather than asking the user to provide a *global* task-space policy, RMPflow asks for only motion policies for *subtasks* of the original problem. This potentially can be much simpler. For instance, directly designing a reaching policy for a cluttered environment is non-trivial, but individually specifying policies for obstacle-free goal reaching and collision avoidance is more straightforward.

Inspired by geometric control theory [21], RMPflow provides a rigorous framework for policy fusion with theoretical guarantees, such as stability and geometric consistency. In implementation, RMPflow is realized by a data structure called *RMP-tree*, and a set operations called *RMP-algebra*. Below we highlight major features of each component.

RMP-tree An RMP-tree (e.g. Fig. 1b but without the weights $w.$) is a directed tree, which expresses the task map ψ as a sequence of basic maps. The RMP-tree serves two major purposes: (i) it provides a language for the user to specify the connections between different subtasks, and (ii) it allows RMPflow to reuse those basic computations inside ψ to achieve efficient policy fusion. In the RMP-tree, each node represents an RMP and its state; and each edge represents a transformation between manifolds in the user given decomposition of ψ. Particularly, the leaf nodes consist of the user-defined subtask RMPs, and the root node maintains the RMP of the global policy π on $\mathcal{C}$.

RMP-tree uses RMP [13] to describe motion policies on manifolds. Consider an m-dimensional manifold $\mathcal{M}$ with coordinate $\mathbf{x} \in \mathbb{R}^m$ and a motion policy $\mathbf{a}$ on $\mathcal{M}$ (i.e. $\ddot{\mathbf{x}} = \mathbf{a}(\mathbf{x}, \dot{\mathbf{x}})$). An RMP pairs the motion policy $\mathbf{a}$ with an *abstract* inertia matrix $\mathbf{M}(\mathbf{x}, \dot{\mathbf{x}}) \in \mathbb{R}_+^{m \times m}$, a function of *both* $\mathbf{x}$ and $\dot{\mathbf{x}}$ that describes the directional importance of $\mathbf{a}$ at the current state $(\mathbf{x}, \dot{\mathbf{x}})$ (see [6] for details). The RMP of $\mathbf{a}$ can be written in the canonical form $(\mathbf{a}, \mathbf{M})^{\mathcal{M}}$ or in the natural form $[\mathbf{f}, \mathbf{M}]^{\mathcal{M}}$, in which $\mathbf{f} = \mathbf{M}\mathbf{a}$ is called the force map. Note that $\mathbf{f}$ and $\mathbf{M}$ are not necessarily physical quantities, and that the motion policy in an RMP is not necessarily in the form of (1).

RMP-algebra RMPflow uses the RMP-algebra to combine the subtask policies at leaf nodes into a global policy on the configuration space at the root node. RMP-algebra consists of three operators:

(i) `pushforward` propagates the state $(\mathbf{x}, \dot{\mathbf{x}})$ of a node in the RMP-tree to update the states of its K child nodes. The state of its ith child node is computed as $(\mathbf{y}_i, \dot{\mathbf{y}}_i) = (\psi_i(\mathbf{x}), \mathbf{J}_i(\mathbf{x})\dot{\mathbf{x}})$, where ψ_i is the transformation $\mathbf{y}_i = \psi_i(\mathbf{x})$ and $\mathbf{J}_i = \partial_{\mathbf{x}}\psi_i$ is the Jacobian matrix.

(ii) `pullback` propagates the RMPs from the K child nodes to the parent node as $[\mathbf{f}, \mathbf{M}]^{\mathcal{M}}$ with

$$\mathbf{f} = \textstyle\sum_{i=1}^{K} \mathbf{J}_i^\top (\mathbf{f}_i - \mathbf{M}_i \dot{\mathbf{J}}_i \dot{\mathbf{x}}) \qquad \text{and} \qquad \mathbf{M} = \textstyle\sum_{i=1}^{K} \mathbf{J}_i^\top \mathbf{M}_i \mathbf{J}_i \tag{2}$$

where $[\mathbf{f}_i, \mathbf{M}_i]^{\mathcal{N}_i}$ is the RMP of the ith child node in the natural form.

(iii) `resolve` maps an RMP from its natural form $[\mathbf{f}, \mathbf{M}]^{\mathcal{M}}$ to its canonical form $(\mathbf{a}, \mathbf{M})^{\mathcal{M}}$ with $\mathbf{a} = \mathbf{M}^\dagger \mathbf{f}$, where $\dagger$ denotes Moore-Penrose inverse.

To compute the global policy π at time t, RMPflow first performs a forward pass by recursively calling `pushforward`. Then it performs a backward pass by recursively calling `pullback` and computes $[\mathbf{f}_r, \mathbf{M}_r]^{\mathcal{C}}$ at the root. Finally, the global policy $\pi = \mathbf{a}_r$ is generated by using `resolve`. Loosely speaking, the global policy π can be viewed as a weighted combination of the subtask policies. This can be seen by rewriting (2) as $\mathbf{a} = \mathbf{M}\mathbf{f} = (\sum_{i=1}^{K} \mathbf{J}_i^\top \mathbf{M}_i \mathbf{J}_i)^{-1} \mathbf{J}_i^\top (\mathbf{M}_i \mathbf{a}_i - \mathbf{M}_i \dot{\mathbf{J}}_i \dot{\mathbf{x}})$ (which is linear combination of child policies $\mathbf{a}_i$ plus some curvature correction due to $\dot{\mathbf{J}}_i$).

2.3 Theoretical Properties of RMPflow and GDSs

RMPflow is proved to be Lyapunov stable and coordinate-free, when the subtask policies belong to Geometric Dynamical Systems (GDSs) [6]. GDSs are a family of dynamical systems on manifolds that generalizes (1) to have *velocity-dependent* inertias. Let $\mathcal{M}$ be an m-dimensional manifold with coordinate $\mathbf{x} \in \mathbb{R}^m$. Let $\mathbf{G} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}_+^{m \times m}$ be a *metric* matrix, $\mathbf{B} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}_+^{m \times m}$ be a *damping* matrix, and $\Phi : \mathbb{R}^m \to \mathbb{R}$ be a *potential* function, which is lower bounded. A dynamical system on $\mathcal{M}$ is said to be a *GDS* $(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)$ if it follows

$$\mathbf{M}(\mathbf{x}, \dot{\mathbf{x}})\ddot{\mathbf{x}} + \boldsymbol{\xi}_{\mathbf{G}}(\mathbf{x}, \dot{\mathbf{x}}) = -\nabla_{\mathbf{x}}\Phi(\mathbf{x}) - \mathbf{B}(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}}, \tag{3}$$

in which $\mathbf{M}(\mathbf{x},\dot{\mathbf{x}}) \coloneqq \mathbf{G}(\mathbf{x},\dot{\mathbf{x}}) + \boldsymbol{\Xi}_{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}})$, $\boldsymbol{\Xi}_{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}}) \coloneqq \frac{1}{2}\sum_{i=1}^{m} \dot{x}_i \partial_{\dot{\mathbf{x}}} \mathbf{g}_i(\mathbf{x},\dot{\mathbf{x}})$, $\boldsymbol{\xi}_{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}}) \coloneqq \overset{\times}{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}})\dot{\mathbf{x}} - \frac{1}{2}\nabla_{\mathbf{x}}(\dot{\mathbf{x}}^\top \mathbf{G}(\mathbf{x},\dot{\mathbf{x}})\dot{\mathbf{x}})$, and $\overset{\times}{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}}) \coloneqq [\partial_{\mathbf{x}} \mathbf{g}_i(\mathbf{x},\dot{\mathbf{x}})\dot{\mathbf{x}}]_{i=1}^{m}$. The term $\mathbf{M}$ is again called the inertia matrix, despite being a function of both $\mathbf{x}$ and $\dot{\mathbf{x}}$. The *curvature* terms $\boldsymbol{\Xi}(\mathbf{x},\dot{\mathbf{x}})$ and $\boldsymbol{\xi}(\mathbf{x},\dot{\mathbf{x}})$ are generated from the dependency of $\mathbf{G}(\mathbf{x},\dot{\mathbf{x}})$ on $\mathbf{x}$ and $\dot{\mathbf{x}}$; if $\mathbf{G}(\mathbf{x},\dot{\mathbf{x}}) = \mathbf{G}(\mathbf{x})$, then $\mathbf{G}(\mathbf{x}) = \mathbf{M}(\mathbf{x})$ and $\boldsymbol{\xi}_{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}}) = \mathbf{C}(\mathbf{x},\dot{\mathbf{x}})\dot{\mathbf{x}}$ in (1). In view of this, a GDS extends (1) to have general potentials and velocity-dependent metrics, which is useful in modeling collision avoidance behaviors [6].

The behavior of a GDS $(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)$ is characterized by the Lyapunov function

$$V(\mathbf{x},\dot{\mathbf{x}}) = \tfrac{1}{2}\dot{\mathbf{x}}^\top \mathbf{G}(\mathbf{x},\dot{\mathbf{x}})\dot{\mathbf{x}} + \Phi(\mathbf{x}). \tag{4}$$

It can be shown that the stability property of RMPflow is governed by a Lyapunov function in a similar form [6], when the leaf-node policies are GDSs. An RMP $(\mathbf{a},\mathbf{M})^{\mathcal{M}}$ is a GDS if its motion policy is $\mathbf{a} = \mathbf{M}(\mathbf{x},\dot{\mathbf{x}})^{-1}(-\nabla_{\mathbf{x}}\Phi(\mathbf{x}) - \mathbf{B}(\mathbf{x},\dot{\mathbf{x}})\dot{\mathbf{x}} - \boldsymbol{\xi}_{\mathbf{G}}(\mathbf{x},\dot{\mathbf{x}}))$.

Theorem 1. [6] *Suppose an RMP-tree has K leaf nodes of GDSs $(\mathcal{T}_k, \mathbf{G}_k, \mathbf{B}_k, \Phi_k)$ with Lyapunov function V_k in (4), for $k = 1,\dots,K$. Let $V_r = \sum_{k=1}^{K} V_k$ be a Lyapunov candidate.*

1. *If $\mathbf{M}_r$ of the root-node RMP on $\mathcal{C}$ is positive definite, then $\dot{V}_r = -\sum_{k=1}^{K} \dot{\mathbf{x}}_k^\top \mathbf{B}_k \dot{\mathbf{x}}_k \le 0$.*
2. *If further $\sum_{k=1}^{K} \mathbf{J}_k^\top \mathbf{G}_k \mathbf{J}_k \succ 0$ and $\sum_{k=1}^{K} \mathbf{J}_k^\top \mathbf{B}_k \mathbf{J}_k \succ 0$, the system converges forwardly to $\{(\mathbf{q},\dot{\mathbf{q}}) : \nabla_{\mathbf{q}}\Phi_r(\mathbf{q}) = 0, \dot{\mathbf{q}} = 0\}$, where $\mathbf{J}_k = \partial_{\mathbf{q}}\mathbf{x}_k$ and $\Phi_r(\mathbf{q}) = \sum_{k=1}^{K} \Phi_k(\mathbf{x}_k(\mathbf{q}))$.*

3 RMPfusion

RMPflow provides a control-theoretic framework for combining subtask policies. However, certain limitations exist. Particularly, it requires the user to provide sensible inertia matrices (cf. Section 2.2) to describe the subtask policies' characteristics in the leaf-nodes RMPs; failing to do so may result in a global policy with undesirable performance, albeit still being geometrically consistent with the meaningless geometric structure induced by the bad inertia matrices.

In this work, we propose a modified algorithm, RMPfusion, which adds extra flexibilities into RMPflow to address this difficulty. The main idea is to introduce an additional set of weight functions as gates to switch on and off the child-node policies in the RMP-tree, based on the current state of the robot and the environment. These functions can either be designed by hand, or be parameterized as function approximators (like neural networks) which are then learned end-to-end from data (see Section 3.4). As a result, RMPfusion can combine simpler/imperfect subtask RMPs into a better global policy, lessening the burden on the user to directly provide high-quality subtasks RMPs.

RMPfusion modifies RMP-tree and RMP-algebra into RMP-tree* and RMP-algebra*, respectively. RMP-tree* augments each node in RMP-tree with extra information and each edge with a weight function; RMP-algebra* replaces `pullback` with `pullback`*. Below we define these modifications. In addition, we show that RMPfusion retains the nice structural properties of RMPflow: under mild conditions on the weight functions, the global policy of RMPfusion is Lyapunov stable. Later in Section 3.4, we will show how to learn the weight functions in RMPfusion from data.

3.1 RMP-tree* and RMP-algebra*

Modified node In addition to the RMP and its state, each node in RMP-tree* also stores the *values* of a scalar function L and the metric matrix $\mathbf{G}$. When a leaf-node RMP is a GDS, $\mathbf{G}$ is defined as (3) and $L = \frac{1}{2}\dot{\mathbf{x}}^\top \mathbf{G}\dot{\mathbf{x}} - \Phi(\mathbf{x})$ (analogue of the Lagrangian in mechanical systems).

Modified edge Each edge in an RMP-tree* has in addition a weight function. This weight is a function of the parent-node configuration and some auxiliary state (which describes the task at hand, e.g., the location of the goal in a reaching task).

Modified pullback We modify `pullback` into `pullback`* to use the weight functions on edges to combine child-node RMPs. For the parent and child nodes given in (2), we set instead

$$\mathbf{f} = \sum_{i=1}^{K} w_i \mathbf{J}_i^\top (\mathbf{f}_i - \mathbf{M}_i \dot{\mathbf{J}}_i \dot{\mathbf{x}}) + \mathbf{h}_i, \quad \mathbf{M} = \sum_{i=1}^{K} w_i \mathbf{J}_i^\top \mathbf{M}_i \mathbf{J}_i, \quad \mathbf{G} = \sum_{i=1}^{K} w_i \mathbf{J}_i^\top \mathbf{G}_i \mathbf{J}_i, \quad L = \sum_{i=1}^{K} w_i L_i \tag{5}$$

where $\mathbf{h}_i = L_i \nabla_{\mathbf{x}} w_i - (\dot{\mathbf{x}}^\top \nabla_{\mathbf{x}} w_i)\mathbf{J}_i^\top \mathbf{G}_i \mathbf{J}_i \dot{\mathbf{x}}$. From (5), we see that `pullback`* does *not* simply linearly combine child-node motion policies. It adds a correction term $\mathbf{h}_i$, which is designed to anticipates the change of weighting w_i so that the system remains stable. When applied recursively in policy generation, it would *hierarchically* reshape the Lyapunov functions (see Section 3.3).

3.2 Stability

We show RMPfusion is also Lyapunov stable like RMPflow. To state the stability property, let us introduce additional notation to describe the functions in the RMP-tree*. We will use $(i;j)$ to denote the ith node in depth j of an RMP-tree* and we use $C_{(i;j)}$ to denote the indices of its child nodes. For example, node $(1;0)$ denotes the root node (also denoted as `r`). In addition, we will refer to the functions on the edges using the indices of the child nodes, e.g., the Jacobian of the transformation to the ith node in depth j is denoted as $\mathbf{J}_{(i;j)}$. We show the stability property of RMPfusion when all the leaf nodes are of GDSs, like Theorem 1. The proof is given in Appendix A.

Theorem 2. *Suppose an RMP-tree* has leaf-node policies as GDSs with Lyapunov functions given as (4). Define $V_{(i;j)}$, $\mathbf{B}_{(i;j)}$, and $\Phi_{(i;j)}$ on the RMP-tree* through the recursion*

$$V_{(i;j)} = \sum_{k\in C_{(i;j)}} w_{(k;j+1)} V_{(k;j+1)}, \quad \mathbf{B}_{(i;j)} = \sum_{k\in C_{(i;j)}} w_{(k;j+1)} \mathbf{J}_{(k;j+1)}^\top \mathbf{B}_{(k;j+1)} \mathbf{J}_{(k;j+1)}$$
$$\Phi_{(i;j)} = \sum_{k\in C_{(i;j)}} w_{(k;j+1)} \Phi_{(k;j+1)} \tag{6}$$

in which the boundary condition is given by the leaf-node GDSs. Let $V_{\mathtt{r}}$ be a Lyapunov candidate.

1. *If $\mathbf{M}_{\mathtt{r}} \succ 0$, then $\dot{V}_{\mathtt{r}} = -\dot{\mathbf{q}}^\top \mathbf{B}_{\mathtt{r}} \dot{\mathbf{q}} \leq 0$.*
2. *If further $\mathbf{G}_{\mathtt{r}}, \mathbf{B}_{\mathtt{r}} \succ 0$, the system converges forwardly to $\{(\mathbf{q}, \dot{\mathbf{q}}) : \nabla_{\mathbf{q}} \Phi_{\mathtt{r}}(\mathbf{q}) = 0, \dot{\mathbf{q}} = 0\}$.*

Theorem 2 shows that the system is Lyapunov stable with respect to $V_{\mathtt{r}}$. To satisfy the conditions required in Theorem 2, a sufficient condition is to select leaf-node GDSs with certain monotone metrics [6, Theorem 2] and have positive weight functions on edges. Therefore, in addition to the conditions needed by RMPflow, RMPfusion only imposes mild constraints on the weight functions. This is a useful feature when the weight functions are learned from data, because Theorem 2 essentially guarantees the output policy is always stable even in the premature stage of learning.

Note that it is straightforward to extend RMP-tree* to include, in (2), an extra time-varying term that vanishes as $t \to \infty$ (like the one used in DMPs [10]) and to consider time-varying potentials (e.g. in tracking applications). We omit discussions about these generalizations due to space limitation.

3.3 Advantages of RMPfusion over RMPflow

RMPfusion strictly generalizes RMPflow. When each weight is constant one, RMPfusion becomes RMPflow (i.e. `pullback`* is the same as `pullback` and Theorem 2 reduces to Theorem 1). More generally, RMPfusion allows mixing policies through reweighting their Lyapunov functions, while retaining the nice structural properties of RMPflow, as shown in Theorem 2.

In comparison, RMPfusion has a more flexible way to express policies and compose the subtask Lyapunov functions into the Lyapunov candidate $V_{\mathtt{r}}$ in (7). Whereas Theorem 1 uses the simple summation of subtask energies $V_{\mathtt{r}} = \sum_{i=1}^K V_i$, Theorem 2 effectively uses the Lyapunov function

$$V_{\mathtt{r}} = \sum_{k_1\in C_{(1;0)}} w_{(k_1;1)} \sum_{k_2\in C_{(k_1;1)}} \cdots \sum_{k_D\in C_{(k_{D-1};D-1)}} w_{(k_D;D)} V_{(k_D;D)} \tag{7}$$

for a depth-D RMP-tree* (cf. (6)) and each weight $w_{(i;j)}$ can be a function of the configuration and auxiliary state of the parent of node $(i;j)$. Therefore, from (6) and (7), RMPfusion can be viewed as a form of hierarchical Lyapunov function reshaping scheme along the hierarchy structure induced by the RMP-tree*. Consequently, the recursive formulation of RMPfusion allows the user only to provide basic subtask policies and gradually increase their expressiveness by the weight functions. In contrast, using RMPflow requires directly specifying subtask policies with complicated behaviors. We include a concrete example to illustrate the benefit of this extra flexibility in Appendix B.

3.4 Learning RMPfusion

We presented a new computational graph, RMPfusion, which supplements RMPflow with a set of multiplicative weight functions to achieve extra flexibility in policy fusion. In Appendix C, we show these weight functions can be learned by back-propagation, and therefore RMPfusion can be treated as a parameterized policy class in policy optimization by using computational graph libraries like tensorflow [22] or pytorch [23]. Finally, it is important to note that we do not have to learn all the weight functions in a RMP-tree*. If we know that certain leaf-node RMPs have to be turned on, we can adopt a semi-parametric scheme of weight functions. For example, we can design parameterization of the weight functions such that only collision avoidance RMPs are turned on, when the robot is extremely close to an obstacle. This property is due to the structure of RMP-tree*, which is interpretable, unlike policies purely based on general function approximators. Interpretability allows for prior knowledge (like constraints and preferences) to be easily incorporated into the policy structure. This feature is particularly valuable for policy learning with safety constraints [24].

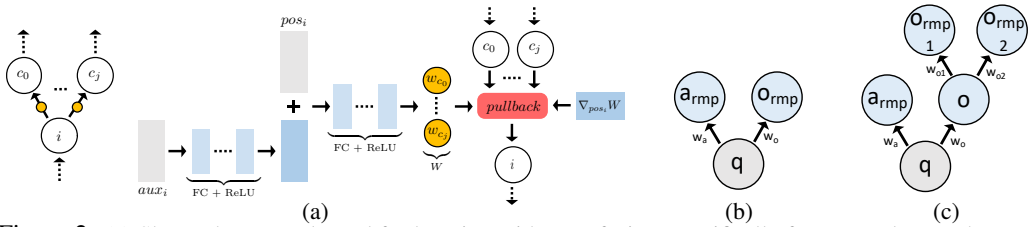


Figure 2: (a) Shows the network used for learning with RMPfusion, specifically for any node i on the RMP-tree*, with children $c_0, \ldots, c_j$. If i is a leaf node, then it is evaluated from the designed RMP policy. The global policy is obtained by applying `resolve` on the root node RMP. RMP-tree* used in experiments for (b) `2d1level` and (c) `2d2level`.

4 Experiments

We validate our approach with experiments of imitation learning. The goal is to show that RMPfusion with an RMP-tree* that is parametrized by randomly initialized neural networks (as in Figure 2) is able to mimic the expert policy's behavior by observing expert demonstrations. This setup simulates the situation where the user of RMPfusion only provides imperfect subtask policies. We also use these experiments to validate the stability properties of RMPfusion by studying if the Lyapunov function of the policies generated by RMPfusion (even the premature ones obtained before learning converges) decay monotonically over time. We perform these experiments with a 2D particle robot and with a Franka Panda 7-DOF robot (video of experiments is available at https://youtu.be/McSrpW-mIq4).

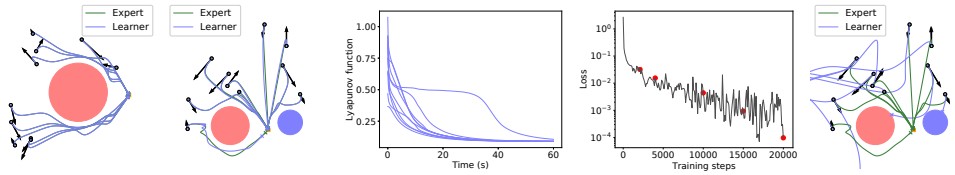
As our aim it not to invent a new imitation learning algorithm, we adopt the most basic approach, behavior cloning [25], in which the demonstrations are purely generated by running the expert policy alone without any active intervention from the learner. The objective of these experiments is to study how well RMPfusion can recover the behaviors of an expert that is within its effective policy class, and therefore we use a known RMP-tree* with fixed weights as the expert policy. We choose this setting to rule out bias due to mismatches between policy classes, because properly handling policy class biases in imitation learning is a non-trivial research question on its own right [26, 27, 28, 29]. Note that any policy optimization technique can be used to train RMPfusion, including online imitation learning and policy gradient methods, etc.

4.1 2D Robot

We first validate our approach on two problems where a 2D robot is tasked with reaching a goal while avoiding one obstacle (`2d1level`) or two obstacles (`2d2level`). The RMP-tree* for these problems are shown in Figure 2b-2c and are detailed in Appendix D. The tree structure here is heuristically chosen based on the problem domain, as in RMPflow and typically follows the robots kinematic chain and then extends into the workspace and abstract task spaces. In the `2d1level` problem, the aim is to show near-perfect recovery of the weights given that the problem is convex in the weight functions. The `2d2level` problem adds extra complexity to the learning process. It introduces multiplication between weights so the weights cannot be uniquely identified. The aim here is to show that close-to-expert behavior can still be achieved.

Data For each problem, the expert policy is generated by the respective RMP-tree* with some fixed assigned weights, which are unknown to the learner. The training data consist of 20 *randomly selected environments* with varying placements and sizes of obstacles. In each environment, the expert is run to generate 50 trajectories from unique initial states, and 60 temporally equidistant data points on each trajectory are recorded. Each data point is a pair of input and output: the input consists of the state (position and velocity) of the 2D particle and the auxiliary state (obstacle location and dimension, goal location) i.e. the meta information about the environment; the output consists of the action (acceleration) as specified by the expert given the input state visited by running the expert policy. Test data are collected by repeating this process with 5 new environments with 10 trajectories in each environment.

Unstructured network For `2d2level` we also compare our RMPfusion `learner-rmp` with an unstructured neural network `learner-un`. This is a fully connected feed forward network with similar number of learnable parameters compared to `learner-rmp`. This network takes robot state and auxiliary state as the inputs, and outputs the acceleration. Our aim with this comparison is to show that an unstructured approach cannot offer any stability or safety guarantees, and with the same amount of data and training underperforms compared to the structured approach.



(a) `2d1level` (b) `2d2level` (c) (d) (e) `2d2level`

Figure 3: Trajectories generated in by (a)-(b) `learner-rmp` and (e) `learner-un`, compared to the expert are shown. Initial state is a black circle for position and black arrow for velocity. The environment has obstacles (red and blue) and goal (orange square). (c) shows the corresponding Lyapunov function for `learner-rmp` trajectories in (b) while (d) shows its learning curve.

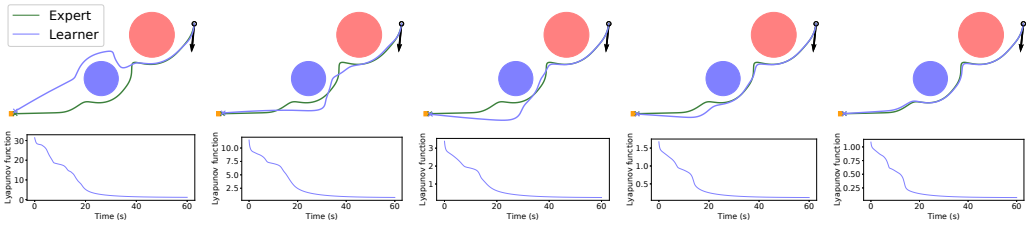


Figure 4: Improvement of the behavior produced by `learner-rmp` at various stages during training for `2d2level`. The top row shows the trajectories and the bottom row shows the corresponding Lyapunov function. From left to right these plots correspond to the red dots from left to right on the training curve in Figure 3d.

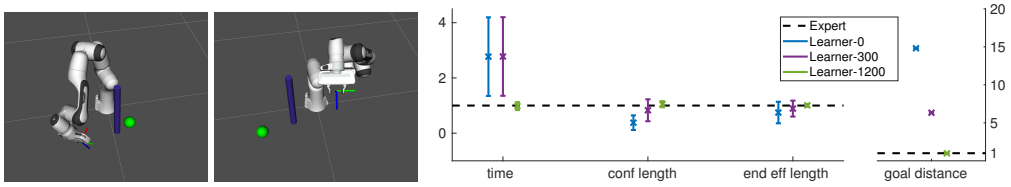
Training We use the mean squared error between the action generated by any learner and the action specified by the expert as the loss function for imitation learning. All learners are trained using RMSprop [30] with a minibatch size of 200 for 20,000 iterations. The number of iterations were chosen such that learning roughly converged and over-fitting had not happened.

Results We report two types of test loss: the batch-loss is the average loss on the entire test dataset generated by the expert policy, and the online-loss is the average loss at every time step (1 second interval) on the trajectories generated by the learner's policy starting from the initial states in the test dataset. In `2d1level`, the batch-loss is 5.42×10^{-5} and the online-loss is 5.82×10^{-5}. In `2d1level`, for `learner-rmp` the batch-loss is 2.45×10^{-4} and the online-loss is 2.78×10^{-4}, while for `learner-un` the batch-loss is 0.111 and the online-loss is 12.203. The higher batch-loss for `learner-un` indicates that with the same amount of data and training the network is unable to learn the policy from the expert, while the much worse online-loss indicates that it cannot generalize well and succumbs to covariate shift problems.

Figures 3a, 3b and 3e show the evaluation of the trained networks on an example test environment. These results show that RMPfusion can perfectly match the behavior of the expert in the convex case (`2d1level`), while achieving near-expert performance in the non-identifiable case (`2d12evel`). From the overall results we also observe that `learner-un` is never able to reach the goal and also has a collision rate of 28% (e.g. Figure 3e), whereas `learner-rmp` successfully finishes the task 100% of the time. We also tried a unstructured network with 5.8 times the number of learnable parameters. While the loss values improved with a small drop in collision rate, it was still never able to complete the task (please see Appendix D for more details). Figure 4 shows the improving progression of `learner-rmp` during training, in which each snapshot corresponds to an associated point on the training curve in Figure 3d. This verifies that with training we can progressively improve the behavior of the learner. In addition, we verify that the stability properties of RMPfusion in the associated Lyapunov functions in Figure 3c and Figure 4. We see that, regardless of the setting, the Lyapunov functions always decays monotonically as indicated by Theorem 2. This suggests RMPfusion produces a stable policy even when the learned weight functions are premature before the learning has converged (Figure 4). On the other hand, `learner-un` does not always avoid collision or provide any stability during or after training (see Figure 7 in Appendix D).

4.2 Franka Robot

We also validate our approach in a more realistic setup with a Franka Panda 7-DOF robot arm. In these experiments, the task is to reach a goal while navigating around an obstacle. The RMP-tree* used is shown in Figure 1b, where the configuration space of the robot is the root node, and weights functions are shown on the edges where they are defined. Please see Appendix D for details.



(a) (b)

Figure 5: (a) An example from the training dataset (left) and the test dataset (right). The robot is shown in its start configuration with an obstacle (cylinder) and a goal (sphere). (b) Learner's performance with respect to the expert on the test dataset for the experiments with the Franka robot.

Data and training The expert policy is given by the RMP-tree* with some fixed but unknown weights, while the learner's policy is defined by the RMP-tree* with neural network weight functions that will be learned through behavior cloning. For training data we place an obstacle in a fixed location near the robot and sample different start configurations and goal locations that are in a region in front of the obstacle from the robot's perspective, so that the robot is forced to interact with the obstacle while trying to reach the goal. We run the expert to generate 110 unique trajectories for the training data. The trajectories are 5-10 seconds long and data is collected every 0.1 seconds; a data point consists of the state (configuration position and velocity of the robot), the auxiliary state (distances to goal and obstacle), and the expert action (acceleration). In a new environment with a different placement of the obstacle, this process is repeated to gather the test dataset where the expert is used to generate 20 unique trajectories. An example from the training and test dataset is shown in Figure 5a. The loss function is the same as in the experiments with the 2D robot and we train the policy using ADAM [31] with a minibatch size of 200 for 1500 iterations. The number of iterations were chosen such that learning roughly converged and over-fitting had not happened.

Results We compare the performance of the learner, against the expert, at various stages of training: `learner-0` at no training (the neural network is initialized with random weights), `learner-300` at 300 iterations, and `learner-1200` at 1200 iterations when the learning converges. We record the following metrics on the test dataset for the expert and all the learners: (i) time: the time to reach within a precision of $0.05m$ of the goal; we time-out the execution at 10 seconds, (ii) conf length: the distance traveled in configuration space, (iii) end eff length: the distance traveled by the end effector in workspace, and (iv) goal distance: the distance to the goal from the end effector at the end of an execution.

Figure 5b shows the performance of the learners relative to the expert on the test dataset (it plots the mean and the standard deviation of the ratios of the learner's metric and the expert's metric across trajectories; the expert is shown as the dotted horizontal baseline). From these results we see that, when the learner is not trained, the robot does not move much and incurs a high goal distance before timing out. With more training, the goal error reduces as the robot start traveling towards the goal but it still often times out. As the learning converges so does the performance of the learner towards the expert's performance. In all the trajectories across all the learners there are no collision, which verifies that constraints like safety can be incorporated through the structured learning approach that RMPfusion allows. We do a qualitative comparison on an example execution with the expert and the learners and also verify the stability properties of RMPfusion (even during learning) with the monotonically decreasing Lyapunov functions on these executions. Please see Appendix D and Figure 9 therein for details.

5 Conclusion

We introduce extra parametrization flexibility into RMPflow and propose a new algorithm called RMPfusion. RMPfusion features a set of learnable weight functions that specifies the importance of subtask policies based on the robot's configuration and the environment. Consequently, RMPfusion can combine imperfect subtask policies into a global policy with good performance, where the original RMPflow fails. We demonstrate the ability of RMPfusion to learn weight functions for policy fusion in experiments, and further theoretically prove that RMPfusion inherits the Lyapunov-type stability from RMPflow with only mild conditions on the weight functions. These structural properties and encouraging experimental results of RMPfusion suggest that RMPfusion can be treated as a class of structural policies suitable for policy learning with safety and interpretability requirements. Important future work includes designing more expressive policies based on RMPfusion, e.g., we can modify RMPfusion slightly to also learn part of the subtask policies and extra perturbations.

Acknowledgments

This research was partially supported by NSF CAREER award 1750483 and NSF NRI award 1637758.

References

[1] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[2] M. Johnson, B. Shrewsbury, S. Bertrand, T. Wu, D. Duran, M. Floyd, P. Abeles, D. Stephen, N. Mertins, A. Lesman, et al. Team ihmc's lessons learned from the darpa robotics challenge trials. *Journal of Field Robotics*, 32(2):192–208, 2015.

[3] N. Correll, K. E. Bekris, D. Berenson, O. Brock, A. Causo, K. Hauser, K. Okada, A. Rodriguez, J. M. Romano, and P. R. Wurman. Analysis and observations from the first amazon picking challenge. *IEEE Transactions on Automation Science and Engineering*, 15(1):172–188, 2018.

[4] D. Kappler, F. Meier, J. Issac, J. Mainprice, C. Garcia Cifuentes, M. Wüthrich, V. Berenz, S. Schaal, N. Ratliff, and J. Bohg. Real-time perception meets reactive motion generation. *https://arxiv.org/abs/1703.03512*, 2017.

[5] M. Mukadam, C.-A. Cheng, X. Yan, and B. Boots. Approximately optimal continuous-time motion planning and control via probabilistic inference. In *Proceedings of the 2017 IEEE Conference on Robotics and Automation (ICRA)*, 2017.

[6] C.-A. Cheng, M. Mukadam, J. Issac, S. Birchfield, D. Fox, B. Boots, and N. Ratliff. RMPflow: A computational graph for automatic motion policy generation. In *The 13th International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2018.

[7] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[8] J. Nakanishi, R. Cory, M. Mistry, J. Peters, and S. Schaal. Operational space control: A theoretical and empirical comparison. *IJRR*, 6:737–757, 2008.

[9] J. Peters, M. Mistry, F. E. Udwadia, J. Nakanishi, and S. Schaal. A unifying framework for robot control with redundant dofs. *Autonomous Robots*, 1:1–12, 2008.

[10] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. Dynamical movement primitives: Learning attractor models for motor behaviors. *Neural Computation*, 25(2):328–373, Feb 2013.

[11] S.-Y. Lo, C.-A. Cheng, and H.-P. Huang. Virtual impedance control for safe human-robot interaction. *Journal of Intelligent & Robotic Systems*, 82(1):3–19, 2016.

[12] N. Ratliff, M. Toussaint, and S. Schaal. Understanding the geometry of workspace obstacles in motion optimization. In *IEEE ICRA*, 2015.

[13] N. D. Ratliff, J. Issac, D. Kappler, S. Birchfield, and D. Fox. Riemannian motion policies. *arXiv preprint arXiv:1801.02854*, 2018.

[14] X. Meng, N. Ratliff, Y. Xiang, and D. Fox. Learning latent space dynamics for tactile servoing. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[15] C. Paxton, N. Ratliff, C. Eppner, and D. Fox. Representing robot task plans as robust logical-dynamical systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[16] G. Sutanto, N. Ratliff, B. Sundaralingam, Y. Chebotar, Z. Su, A. Handa, and D. Fox. Learning latent space dynamics for tactile servoing. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[17] A. Li, M. Mukadam, M. Egerstedt, and B. Boots. Multi-objective policy generation for multi-robot systems using riemannian motion policies. In *Proceedings of the International Symposium on Robotics Research (ISRR)*, 2019.

[18] A. Li, C.-A. Cheng, B. Boots, and M. Egerstedt. Stable, concurrent controller composition for multi-objective robotic tasks. In *Proceedings of Conference on Decision and Control (CDC)*, 2019.

[19] S. B. Slotine. A general framework for managing multiple tasks in highly redundant robotic systems. In *Proceeding of 5th International Conference on Advanced Robotics*, volume 2, pages 1211–1216, 1991.

[20] R. C. Arkin. Governing lethal behavior: embedding ethics in a hybrid deliberative/reactive robot architecture. In *Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, pages 121–128. ACM, 2008.

[21] F. Bullo and A. D. Lewis. *Geometric control of mechanical systems: modeling, analysis, and design for simple mechanical control systems*, volume 49. Springer Science & Business Media, 2004.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[24] J. Garcıa and F. Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[25] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[26] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *International conference on artificial intelligence and statistics*, pages 627–635, 2011.

[27] S. Ross and J. A. Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

[28] C.-A. Cheng and B. Boots. Convergence of value aggregation for imitation learning. In *International Conference on Artificial Intelligence and Statistics*, volume 84, pages 1801–1809, 2018.

[29] C.-A. Cheng, X. Yan, N. Wagener, and B. Boots. Fast policy learning through imitation and reinforcement. In *Uncertainty in artificial intelligence*, 2018.

[30] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[31] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Appendix

A Proof of Theorem 2

We provide the proof of Theorem 2 for completeness. We use [6, Thoerem 1] as the main lemma in our proof.

A.1 Background

We first recall the definition of structured GDS [6], which augments a GDS with the information on how the metric matrix $\mathbf{G}$ factorizes, in order to state [6, Thoerem 1].

Definition 1. [6] Suppose $\mathbf{G}$ has a structure $\mathcal{S}$ that factorizes $\mathbf{G}(\mathbf{x}, \dot{\mathbf{x}}) = \mathbf{J}(\mathbf{x})^\top \mathbf{H}(\mathbf{z}, \dot{\mathbf{z}})\mathbf{J}(\mathbf{x})$, where $\mathbf{z} : \mathbf{x} \mapsto \mathbf{z}(\mathbf{x}) \in \mathbb{R}^n$ and $\mathbf{H} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_+^{n\times n}$, and $\mathbf{J}(\mathbf{x}) = \partial_{\mathbf{x}}\mathbf{z}$. The tuple $(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)_{\mathcal{S}}$ is a *structured GDS*, if

$$\mathbf{M}(\mathbf{x}, \dot{\mathbf{x}})\ddot{\mathbf{x}} + \boldsymbol{\eta}_{\mathbf{G};\mathcal{S}}(\mathbf{x}, \dot{\mathbf{x}}) = -\nabla_{\mathbf{x}}\Phi(\mathbf{x}) - \mathbf{B}(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}} \tag{8}$$

where $\boldsymbol{\eta}_{\mathbf{G};\mathcal{S}}(\mathbf{x}, \dot{\mathbf{x}}) \coloneqq \mathbf{J}(\mathbf{x})^\top(\boldsymbol{\xi}_{\mathbf{H}}(\mathbf{z}, \dot{\mathbf{z}}) + (\mathbf{H}(\mathbf{z}, \dot{\mathbf{z}}) + \boldsymbol{\Xi}_{\mathbf{H}}(\mathbf{z}, \dot{\mathbf{z}}))\dot{\mathbf{J}}(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}})$. Given two structures, $\mathcal{S}_a$ is said to *preserve* $\mathcal{S}_b$ if $\mathcal{S}_a$ has the factorization (of $\mathbf{H}$) made by $\mathcal{S}_b$.

As noted in [6], GDSs are structured GDSs with a trivial structure (i.e. $\mathbf{z} = \mathbf{x}$), and structured GDSs reduce to GDSs if $\mathbf{G}(\mathbf{x}, \dot{\mathbf{x}}) = \mathbf{G}(\mathbf{x})$, or if the manifold is one-dimensional.

Lemma 1. *[6, Thoerem 1] Suppose the ith child node follows $(\mathcal{N}_i, \mathbf{G}_i, \mathbf{B}_i, \Phi_i)_{\mathcal{S}_i}$ and has coordinate $\mathbf{y}_i$. Let $\mathbf{a}_i = (\mathbf{G}_i + \boldsymbol{\Xi}_{\mathbf{G}_i})^\dagger(-\boldsymbol{\eta}_{\mathbf{G}_i;\mathcal{S}_i} - \nabla_{\mathbf{y}_i}\Phi_i - \mathbf{B}_i\dot{\mathbf{y}}_i)$ and $\mathbf{M}_i = \mathbf{G}_i + \boldsymbol{\Xi}_{\mathbf{G}_i}$. Suppose $\mathbf{a}$ of the parent node is given by* `pullback` *with $\{(\mathbf{a}_i, \mathbf{M}_i)_C^{\mathcal{N}_i}\}_{i=1}^K$. Then $\mathbf{a}$ follows the* pullback structured GDS *$(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)_{\mathcal{S}}$, where $\mathbf{G} = \sum_{i=1}^K \mathbf{J}_i^\top \mathbf{G}_i \mathbf{J}_i$, $\mathbf{B} = \sum_{i=1}^K \mathbf{J}_i^\top \mathbf{B}_i \mathbf{J}_i$, $\Phi = \sum_{i=1}^K \Phi_i \circ \mathbf{y}_k$, $\mathcal{S}$ preserves $\mathcal{S}_i$, and $\mathbf{J}_i = \partial_{\mathbf{x}}\mathbf{y}_i$. That is, the parent node is $(\mathbf{a}, \mathbf{M})_C^{\mathcal{M}}$ such that $\mathbf{M} = \sum_{i=1}^K \mathbf{J}_i^\top(\mathbf{G}_i + \boldsymbol{\Xi}_{\mathbf{G}_i})\mathbf{J}_i$ and $\mathbf{a} = (\mathbf{G} + \boldsymbol{\Xi}_{\mathbf{G}})^\dagger(-\boldsymbol{\eta}_{\mathbf{G};\mathcal{S}} - \nabla_{\mathbf{x}}\Phi - \mathbf{B}\dot{\mathbf{x}})$.*

Lemma 1 shows that the original `pullback` operator preserves structured GDSs. Consequently, when all the leaf nodes are GDSs, the root node is a structured GDS, which implies the type of Lyapunov stability in Theorem 1.

A.2 Proof of Theorem 2

We prove the stability of RMPfusion using similar techniques as [6]. Using the recursive property, it is sufficient to show that `pullback`* preserves a family of structured GDSs, which are specified by the weight functions. Then the statement of Theorem 2 follows directly as in [6].

We proceed by first decoupling the `pullback`* into two steps. Let u be a parent node on manifold $\mathcal{M}$ and $\{v_k\}_{k=1}^K$ be its K child nodes on manifold $\{\mathcal{N}_k\}_{k=1}^K$ in an RMP-tree*. Between u and each v_k, we add an extra node $\tilde{v}_k$ on manifold $\mathcal{M}$ to create a new graph. In this new graph, u has K child nodes $\{\tilde{v}_k\}_{k=1}^K$ *with identity transformation and the original weight function* w_k, and $\tilde{v}_k$ has a single child which is v_k *with the original transformation from u to v_k and an identity weight function*. Under this construction, the `pullback`* operator in the original graph can then be realized in the new graph as

1. a `pullback`* operator from v_k to $\tilde{v}_k$ for each k
2. a `pullback`* operator from $\{\tilde{v}_k\}_{k=1}^K$ to u.

To verify this we rewrite (5) as

$$\begin{aligned}
\mathbf{f} &= \textstyle\sum_{i=1}^K w_i \mathbf{J}_i^\top(\mathbf{f}_i - \mathbf{M}_i\dot{\mathbf{J}}_i\dot{\mathbf{x}}) + \mathbf{h}_i =: \sum_{i=1}^K w_i\tilde{\mathbf{f}}_i + \tilde{\mathbf{h}}_i \\
\mathbf{M} &= \textstyle\sum_{i=1}^K w_i \mathbf{J}_i^\top \mathbf{M}_i \mathbf{J}_i =: \sum_{i=1}^K w_i\tilde{\mathbf{M}}_i \\
\mathbf{G} &= \textstyle\sum_{i=1}^K w_i \mathbf{J}_i^\top \mathbf{G}_i \mathbf{J}_i =: \sum_{i=1}^K w_i\tilde{\mathbf{G}}_i \\
L &= \textstyle\sum_{i=1}^K w_i L_i =: \sum_{i=1}^K w_i\tilde{L}_i
\end{aligned}$$

where we also has $\mathbf{h}_i = \tilde{\mathbf{h}}_i = \tilde{L}_i \nabla_{\mathbf{x}} w_i - (\dot{\mathbf{x}}^\top \nabla_{\mathbf{x}} w_i)\tilde{\mathbf{G}}_i \dot{\mathbf{x}}$. That is, node $\tilde{v}_k$ has the RMP $(\tilde{\mathbf{f}}_i, \tilde{\mathbf{M}}_i)_C^{\mathcal{M}}$, the metric matrix $\tilde{\mathbf{G}}_i$, and the Lagrangian $\tilde{L}_i$. From the equalities above, we verify the two-step decomposition of `pullback*` is valid.

Next we show that each step in the two-step decomposition yields a structured GDS like Lemma 1, which is sufficient condition we need to prove Theorem 2. In the first step from v_i to $\tilde{v}_i$, because the weight is constant identity, `pullback*` is the same as `pullback`. We apply Lemma 1 and conclude that $\tilde{v}_i$ follows $(\mathcal{M}, \tilde{\mathbf{G}}_i, \tilde{\mathbf{B}}_i, \tilde{\Phi}_i)_{\tilde{\mathcal{S}}_i}$, where $\tilde{\mathcal{S}}_i$ preserves $\mathcal{S}_i$.

Then we show the second step from $\{\tilde{v}_i\}_{i=1}^K$ to u has similar properties. This is summarized as Lemma 2 below.

Lemma 2. *If $\tilde{v}_i$ follows $(\mathcal{M}, \tilde{\mathbf{G}}_i, \tilde{\mathbf{B}}_i, \tilde{\Phi}_i)_{\tilde{\mathcal{S}}_i}$, then u follows $(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)_S$, where S preserves $\tilde{\mathcal{S}}_i$, $\mathbf{G} = \sum_{i=1}^K w_i \tilde{\mathbf{G}}_i$, $\mathbf{B} = \sum_{i=1}^K w_i \tilde{\mathbf{B}}_i$, and $\Phi = \sum_{i=1}^K w_i \tilde{\Phi}_i$.*

Proof of Lemma 2. This can be shown by algebraically comparing the dynamics of $(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)_S$ and the result of (5). Let $\mathbf{x}$ be a coordinate of $\mathcal{M}$ and, without loss of generality, let us consider w_k to be a function of only $\mathbf{x}$ (we ignore the dependency on the auxiliary state). By Definition 1, the dynamics of $(\mathcal{M}, \mathbf{G}, \mathbf{B}, \Phi)_S$ satisfies

$$\mathbf{M}(\mathbf{x}, \dot{\mathbf{x}})\ddot{\mathbf{x}} + \boldsymbol{\eta}_{\mathbf{G};\mathcal{S}}(\mathbf{x}, \dot{\mathbf{x}}) = -\nabla_{\mathbf{x}}\Phi(\mathbf{x}) - \mathbf{B}(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}} \tag{9}$$

We first show the recursion of $\mathbf{f}$ of `pullback*` satisfies (9). To this end, we rewrite $\boldsymbol{\eta}_{\mathbf{G};\mathcal{S}}$ by Definition 1 as

$$\begin{aligned}
&\boldsymbol{\eta}_{\mathbf{G};\mathcal{S}}(\mathbf{x}, \dot{\mathbf{x}}) = \textstyle\sum_{i=1}^K \boldsymbol{\xi}_{w_i \tilde{\mathbf{G}}_i}(\mathbf{x}, \dot{\mathbf{x}}) \\
&= \textstyle\sum_{i=1}^K w_i(\mathbf{x}) \boldsymbol{\eta}_{\tilde{\mathbf{G}}_i}(\mathbf{x}, \dot{\mathbf{x}}) + (\dot{\mathbf{x}}^\top \nabla_{\mathbf{x}} w_i(\mathbf{x}))\tilde{\mathbf{G}}_i(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}} - \tfrac{1}{2}\nabla_{\mathbf{x}} w_i(\mathbf{x}) \dot{\mathbf{x}}^\top \tilde{\mathbf{G}}_i(\mathbf{x}, \dot{\mathbf{x}})\dot{\mathbf{x}}
\end{aligned}$$

where in the first equality we use the trick we made that the transformation from u to $\tilde{v}_k$ is identity and we use the fact $\tilde{S}_i$ preserves S_i, so the structure S that preserves $\tilde{S}_i$ has a clean structure

$$\mathbf{G} = \begin{bmatrix} I & \dots & I \end{bmatrix} \begin{bmatrix} w_1 \tilde{\mathbf{G}}_1 & & \\ & \ddots & \\ & & w_K \tilde{\mathbf{G}}_K \end{bmatrix} \begin{bmatrix} I \\ \vdots \\ I \end{bmatrix}$$

Similarly, we rewrite $\nabla_{\mathbf{x}}\Phi(\mathbf{x}) = \sum_{i=1}^K w_i(\mathbf{x})\nabla_{\mathbf{x}}\tilde{\Phi}_i(\mathbf{x}) + \tilde{\Phi}_i \nabla_{\mathbf{x}} w_i(\mathbf{x})$. Substituting these two equalities into (9), we can write (with input dependency omitted)

$$\begin{aligned}
&\mathbf{M}\ddot{\mathbf{x}} = -\nabla_{\mathbf{x}}\Phi - \mathbf{B}\dot{\mathbf{x}} - \boldsymbol{\eta}_{\mathbf{G};\mathcal{S}} \\
&= \textstyle\sum_{i=1}^K -w_i \nabla_{\mathbf{x}}\tilde{\Phi}_i - \tilde{\Phi}_i \nabla_{\mathbf{x}} w_i - w_i \mathbf{B}_i \dot{\mathbf{x}} + \sum_{i=1}^K -w_i \boldsymbol{\eta}_{\tilde{\mathbf{G}}_i} - (\dot{\mathbf{x}}^\top \nabla_{\mathbf{x}} w_i)\tilde{\mathbf{G}}_i \dot{\mathbf{x}} + \tfrac{1}{2}\nabla_{\mathbf{x}} w_i \dot{\mathbf{x}}^\top \tilde{\mathbf{G}}_i \dot{\mathbf{x}} \\
&= \textstyle\sum_{i=1}^K w_i \tilde{\mathbf{f}}_i + \tfrac{1}{2}\nabla_{\mathbf{x}} w_i \dot{\mathbf{x}}^\top \tilde{\mathbf{G}}_i \dot{\mathbf{x}} - \tilde{\Phi}_i \nabla_{\mathbf{x}} w_i - (\dot{\mathbf{x}}^\top \nabla_{\mathbf{x}} w_i)\tilde{\mathbf{G}}_i \dot{\mathbf{x}} \\
&= \textstyle\sum_{i=1}^K w_i \tilde{\mathbf{f}}_i + \mathbf{h}_i
\end{aligned}$$

where we use the fact that $\tilde{\mathbf{f}}_i = -\nabla_{\mathbf{x}}\tilde{\Phi}_i - \tilde{\mathbf{B}}_i \dot{\mathbf{x}} - \boldsymbol{\eta}_{\tilde{\mathbf{G}}_i;\tilde{\mathcal{S}}_i}$ as $\tilde{v}_i$ follows $(\mathcal{M}, \tilde{\mathbf{G}}_i, \tilde{\mathbf{B}}_i, \tilde{\Phi}_i)_{\tilde{\mathcal{S}}_i}$ with $\tilde{\mathcal{S}}_i$ preserving $\mathcal{S}_i$. This is exactly the recursion of $\mathbf{f}$ when `pullback*` is applied between $\tilde{v}_i$ and u, i.e. $\mathbf{f} = \mathbf{M}\ddot{\mathbf{x}} = \sum_{i=1}^K w_i \tilde{\mathbf{f}}_i + \mathbf{h}_i$.

To establish the equivalence of the other recursions, we next rewrite $\mathbf{M}$ by definition in (3) as

$$\begin{aligned}
\mathbf{M}(\mathbf{x}, \dot{\mathbf{x}}) &= \mathbf{G}(\mathbf{x}, \dot{\mathbf{x}}) + \boldsymbol{\Xi}_{\mathbf{G}}(\mathbf{x}, \dot{\mathbf{x}}) \\
&= \textstyle\sum_{i=1}^K w_i(\mathbf{x}) \left(\tilde{\mathbf{G}}_i(\mathbf{x}, \dot{\mathbf{x}}) + \boldsymbol{\Xi}_{\tilde{\mathbf{G}}_i}(\mathbf{x}, \dot{\mathbf{x}}) \right) \\
&= \textstyle\sum_{i=1}^K w_i(\mathbf{x}) \tilde{\mathbf{M}}_i(\mathbf{x}, \dot{\mathbf{x}})
\end{aligned}$$

where we use the fact that w_i does not on the velocity $\dot{\mathbf{x}}$. The recursion for $\mathbf{G}$ and L can be derived similarly, so we omit them here. ■

So far we have shown that `pullback*` of RMPfusion retains the closure of structured GDSs as `pullback` in RMPflow. In addition, we show that the structured GDS created by `pullback*` has

a linearly weighted metric matrix, damping matrix, and potential function (cf. Lemma 2). By recursively applying the two-step decomposition above, from the leaf nodes to the root node, we conclude that the root node policy will be a structured GDS with a Lyapunov function given by the recursion in (6). The rest of the statement of Theorem 2 follows from the properties of structured GDSs as shown in [6].

B Benefits due to the Extra Flexibility of RMPfusion

We use an example to illustrate the extra flexibility offered by RMPfusion. Consider a simple Y-shape RMP-tree* with a root node and two child nodes with weight functions w_1 and w_2. For the child nodes, suppose they are GDS $(\mathcal{N}_i, \mathbf{G}_i, \mathbf{B}_i, \Phi_i)$ and have coordinate $\mathbf{y}_i$, for $i = 1, 2$. For simplicity, let us assume $\mathbf{G}_i$ only depends on the configuration $\mathbf{y}_i$. From Theorem 2, we see that the root node has an energy function $V_{\mathtt{r}} = \frac{1}{2}\dot{\mathbf{q}}^\top \mathbf{G}_r \dot{\mathbf{q}} + \Phi_r$, where $\mathbf{G}_r(\mathbf{q}) = w_1(\mathbf{q})\mathbf{G}_1(\mathbf{y}_1(\mathbf{q})) + w_2(\mathbf{q})\mathbf{G}_2(\mathbf{y}_2(\mathbf{q}))$ and $\Phi_r(\mathbf{q}) = w_1(\mathbf{q})\Phi_1(\mathbf{y}_1(\mathbf{q})) + w_2(\mathbf{q})\Phi_2(\mathbf{y}_2(\mathbf{q}))$ Because w_i is a function of $\mathbf{q}$ not $\mathbf{y}_i$ and the Lyapunov function of RMPflow only allows summing child-node functions, this example root node policy does not admit a tree structure decomposition in the original RMP-tree and can only be implemented as a single large node. Conversely, because of the weight function on the edges, RMP-tree* can further exploits potential sparsity inside the policy representation so that building complicated global polices with only basic elementary policies becomes possible.

We note that the example above does not imply that RMPfusion can generate more expressive policies than RMPflow. More precisely, RMP-tree* allows representing the same global policy using more basic leaf-node policies. This property has two implications: it suggests (i) RMPfusion can be more efficient to compute and (ii) RMPfusion can offload the difficulties of designing leaf-nodes policies into the weight functions, which are learnable.

C Learning RMPfusion

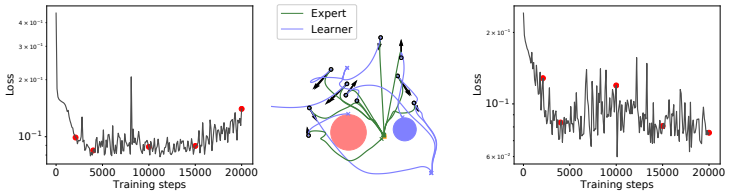
To show the weights are learnable, it is sufficient to check if we can differentiate through the output of the final policy $\pi = \mathbf{a}_{\mathtt{r}}$ with respect to the parameters that specify the weight functions. As the computation of $\mathbf{a}_{\mathtt{r}}$ is accomplished recursively in the backward pass using `pullback*`, we will only illustrate that `pullback*` is differentiable. This can be seen by treating `pullback*` as a computation graph, as illustrated in Figure 2. Take the nodes in (5) as an example. `pullback*` receives $\mathbf{f}_i$, $\mathbf{M}_i$, $\mathbf{G}_i$, $\mathbf{B}_i$, $\mathbf{J}_i$, $\dot{\mathbf{J}}_i$, L_i from the edges to the child nodes, the current state $(\mathbf{x}, \dot{\mathbf{x}})$ and the auxiliary state to define the weight function w_i and the correction term $\mathbf{h}_i$. As these inputs values do not depend on the weight functions $\{w_i\}$ at the current node (i.e. they do not form a loop), the derivative of $\mathbf{a}_r$ with respect to the weight functions in the RMP-tree* can be computed recursively by back-propagating the derivatives through each `pullback*` operator.

D Experimental Details

D.1 2D Robot

`2d1level` consists of a 2D particle that aims to reach a goal while avoiding an obstacle. The RMP-tree* for `2d1level` is of depth one (see Figure 2b), where the root node q (configuration space of the robot) has one child obstacle RMP node (o_{rmp}) and one child attractor RMP node (a_{rmp}). `2d2level` consists of a 2D particle that aims to reach a goal while avoiding two obstacles. The RMP-tree* for `2d2level` is of depth two (see Fig 2c), where the root node (q) has one child attractor RMP node (a_{rmp}) and one all-obstacle RMP (o) that is meant to combine two child obstacle RMPs (o_{rmp}, one for each obstacle). The respective weight functions are shown on the edges of both these trees. The tree structures here are heuristically chosen based on the problem domain, as in RMPflow and typically follow the robots kinematic chain and then extend into the workspace and abstract task spaces.

Figure 7 shows the progression of `learner-un` during training, in which each snapshot corresponds to an associated point on the training curve in Figure 6a. We see that `learner-un` is never able to reach the goal and often ends up in collision during and after training. We also compared with a unstructured network, `learner-un-large`, that has 5.8 times more learnable parameters



(a) (b) (c)

Figure 6: (b) Trajectories generated in `2d2level` by `learner-rmp-large` compared to the expert is shown. Initial state is a black circle for position and black arrow for velocity. The environment has obstacles (red and blue) and goal (orange square). Learning curves for (a) `learner-rmp` and (c) `learner-rmp-large` on `2d2level` is also shown.

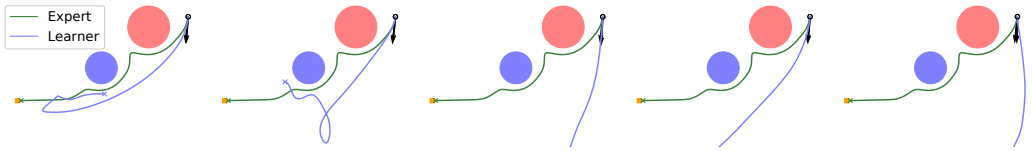


Figure 7: Trajectories produced by `learner-un` at various stages during training for `2d2level`. From left to right these plots correspond to the red dots from left to right on the training curve in Figure 6a.

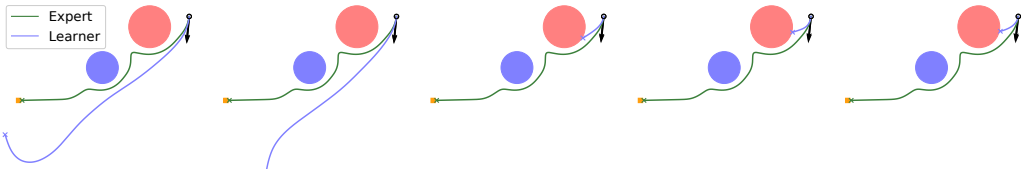


Figure 8: Trajectories produced by `learner-un-large` at various stages during training for `2d2level`. From left to right these plots correspond to the red dots from left to right on the training curve in Figure 6c.

compared to `learner-un`. We see improvement over loss values where the batch-loss is 0.065 and the online-loss is 0.393, and the collision rate decreases to 16%. However, it is still never able to complete the task (e.g. see Figure 6b). Figure 8 shows the progression of `learner-un-large` during training, in which each snapshot corresponds to an associated point on the training curve in Figure 6c.

D.2 Franka Robot

From the root node we have various task spaces, like the end-effector position (ee) on which the attractor space (a) is defined by a change of coordinates such that the goal position is at the origin. The attractor RMP (a_{rmp}) is then defined on the attractor space for a goal reaching subtask. Each joint of the robot is mapped to a one dimensional upper (ujl_i) and lower (ljl_i) joint limit space where a joint limit RMP (jl_{rmp}) is defined for joint limit avoidance subtasks. The root node is also mapped to a pre-specified number of control points on the robot (cp_i) such that they collectively approximate the robot's body and can be used for collision avoidance. On any control point space we add a distance space to the obstacle (d_i) where the obstacle RMP (o_{rmp}) is defined. Note that when multiple obstacles are present we can add distance spaces and the obstacle RMPs for each obstacle on every control point. Now, since the tree structure can change with the number of obstacles, in practice, shared weights can be specified across all obstacles on a given control point, such that training can be performed with only one obstacle to learn the weight function and then can be applied to arbitrary number of obstacles during execution. Finally, there are also native RMPs defined on the root node like a constant damper RMP (q_d) and an RMP which is just an identity metric (q_{mi}) with no learnable weight function to ensure the `resolve` operator is numerically stable.

Figure 9 shows a qualitative comparison on an example execution with the expert and the learners. We verify the stability properties of RMPfusion (even during learning) with the monotonically decreasing Lyapunov function plots on these executions. Note that the scale on the plot for

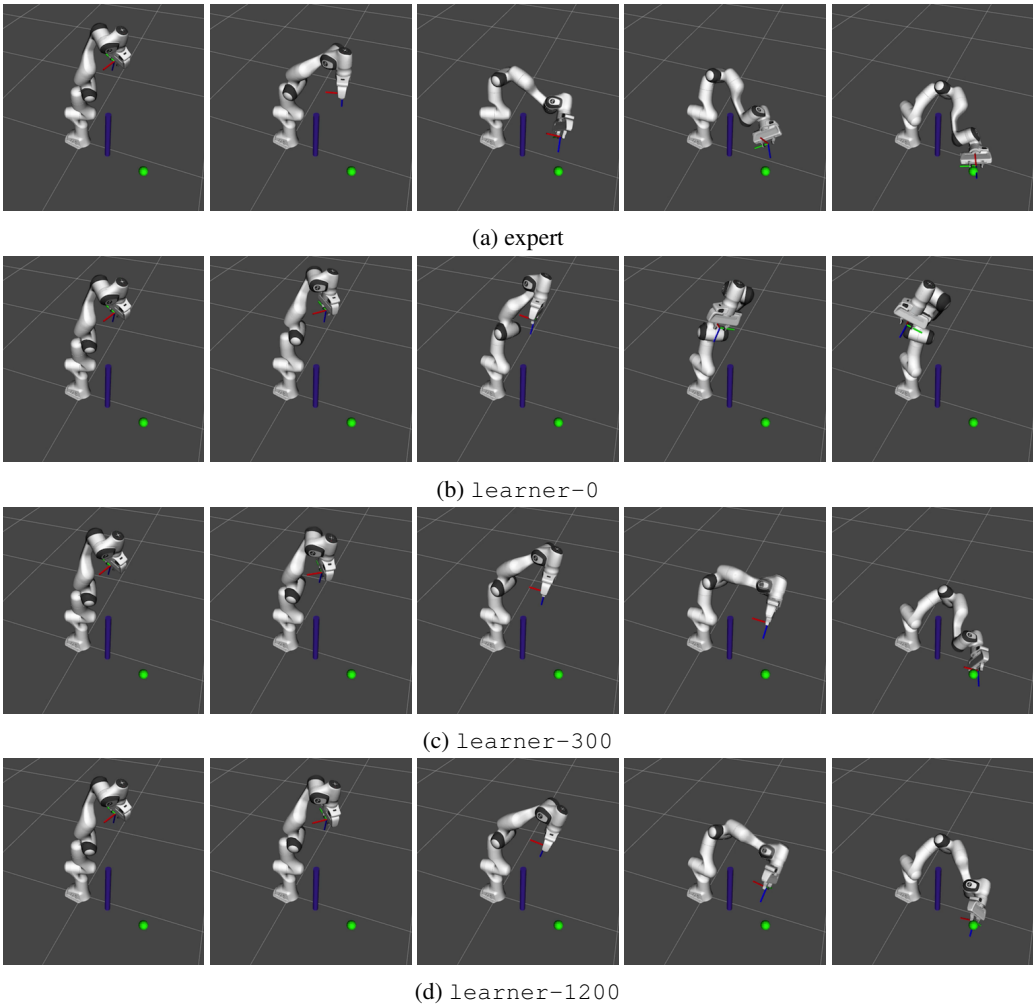


Figure 9: (a)-(d) An example execution (left to right) from the test dataset, comparing (a) the expert with (b) `learner-0`, (c) `learner-300`, and (d) `learner-1200`. (e) The respective Lyapunov function of the learners' trajectories (`learner-0` (left), `learner-300` (middle), `learner-1200` (right)).

`learner-0` is very small and the tiny kink on the plot is due to numerical issues with Euler integration.

D.3 Discussion

The experiments shown here were designed to study if RMPfusion can combine imperfect subtask RMPs, whose inertia weight functions are incorrectly specified while motion policies are sensibly designed with domain knowledge. While this setup does not emulate the full generality where everything is unknown, we think that it captures a representative and important scenario that often happens in practice. Weve had extensive literature in designing motion policies, whereas designing

the associated metrics/inertias for these policies is a fairly new and nontrivial concept, which is a major user burden imposed by RMPflow.

We address this issue by learning the weight functions, and show in the experiments that imperfect subtask RMPs with poorly designed metrics can still be compensated by our framework. Importantly, we emphasize that RMPfusion is designed for generality and does not assume the knowledge that only the inertias are wrongly specified. Therefore, though not tested in the current experiments, we do believe RMPfusion can be used in more general setups, so long as the user provides sufficiently rich subtask RMPs such that there exists a fusion that can generate the desired behavior. However, how to choose the subtask RMPs to start with is a domain specific problem, similar to specifying the size and structure of a neural network in general. Therefore, we consider it beyond the scope of the current paper, because our main focus here is to study and validate the theoretical benefits of RMPfusion (like stability during immature learning).

Generally, an RMPfusion policy with constant weights (not a function of the parent state, etc.) can be reduced into an RMPflow policy with the same tree structure. This can be seen from (5); when the weights are constant, we can effectively push all the weights of an RMP-tree* to the leaf-nodes to define modified inertia matrices on an RMP-tree (the motion policy doesn't change). In other words, in the experiments, the expert can be viewed as an RMPflow policy with some unknown inertia matrices and therefore RMPflow wasn't directly compared.

Using neural networks to parameterize the weight functions maybe is an overkill in our experiments. The reason for using general function approximators here is to show that our framework is practically feasible and can support situations where this will become necessary. For example, this allows for learning general differentiable representation for the weight functions, e.g., using images for auxiliary states. However, one should note also that while using expressive function approximators would add representation to the whole policy it could also potentially make learning more difficult.