Robotics: Science and Systems 2021
Held Virtually, July 12–16, 2021

RMP^2: A Structured Composable Policy Class for Robot Learning

Anqi Li[1]*, Ching-An Cheng[2]*, M. Asif Rana[3], Man Xie[3], Karl Van Wyk[4], Nathan Ratliff[4], and Byron Boots[1,4]
[1]University of Washington, [2]Microsoft Research, [3]Georgia Institute of Technology, [4]NVIDIA
*Equal contribution

***Abstract*—We consider the problem of learning motion policies for acceleration-based robotics systems with a structured policy class. We leverage a multi-task control framework called RMPflow which has been successfully applied in many robotics problems. Using RMPflow as a structured policy class in learning has several benefits, such as sufficient expressiveness, the flexibility to inject different levels of prior knowledge as well as the ability to transfer policies between robots. However, implementing a system for end-to-end learning of RMPflow policies faces several computational challenges. In this work, we re-examine the RMPflow algorithm and propose a more practical alternative, called RMP^2, that uses modern automatic differentiation tools (such as TensorFlow and PyTorch) to compute RMPflow policies. Our new design retains the strengths of RMPflow while bringing in advantages from automatic differentiation, including *1)* simple programming interfaces to designing complex transformations; *2)* support of general directed acyclic graph (DAG) transformation structures; *3)* end-to-end differentiability for policy learning; *4)* improved computational efficiency. Because of these features, RMP^2 can be treated as a structured policy class for efficient robot learning that is suitable for encoding domain knowledge. Our experiments show that using the structured policy class given by RMP^2 can improve policy performance and safety in reinforcement learning tasks for goal reaching in cluttered space. The video for our experimental results can be found at https://youtu.be/dliQ-jsYhgI and the code is available at https://github.com/UWRobotLearning/rmp2.**

I. Introduction

Generating reactive motion policies is a fundamental problem in robotics. This problem has been tackled by analytic control techniques and machine learning tools, but they lead to different compromises. Traditionally control techniques have been used, offering motion policies with analytical forms and desirable properties such as stability, safety, and performance guarantees [4, 16, 23, 35]. However, as tasks become more complex and the environment becomes less structured, synthesizing an analytical policy with such properties becomes difficult, and, even if one succeeds, the resulting policy may be highly sub-optimal [7]. In contrast to hand-designed control techniques, learning-based approaches (such as reinforcement learning [33, 34] or imitation learning [31, 32]) make minimal assumptions and promise to improve policy performance through interactions with the environment. However, such approaches often require many interactions to achieve reasonable results, especially when learning policies under sparse reward signals through reinforcement learning. Furthermore, most learning algorithms are sensitive to distribution shifts and have poor out-of-distribution generalization capability. For example, a policy that is trained to go to the left of a room in training often does not know how to go to the right, because such examples are never presented during training.

In practice, we desire motion policy optimization algorithms that possess *both* the non-statistical guarantees from the control-based approaches and the flexibility of the learning-based approaches. A promising direction is to use *structured policies* [8, 9, 12, 21, 22]. The main idea is to apply learning to optimize only within policy parameterizations that have provable control-theoretic properties (such as stability and safety guarantees). In other words, it uses data to optimize the hyperparameters of a class of controllers with provable guarantees. From a learning perspective, structured policies provide a way to inject prior knowledge about a problem domain into learning. More often than not, before running the robot in the field, we may know (approximate) kinematic and dynamic models, task constraints, and some notions of desired behaviors. For example, the kinematics of the robot is often provided by the manufacturer, and generally colliding with obstacles and hitting joint limits are undesirable. By using this information through a control framework to produce structured policies, we can ensure policies generated by a learning algorithm can always satisfy certain task specifications (such as safety) regardless how they are learned.

In this paper, we are interested in learning structured motion policies for acceleration-based robotics systems [35]. We adopt the RMPflow control framework [6] as the foundation of the structured policy class. RMPflow is a multi-task control algorithm that generates the acceleration-based motion policy by combining individually designed subtask reactive policies. Examples of subtasks include goal reaching, collision avoidance, maintaining balance, etc. RMPflow treats these subtask spaces as manifolds and provides a message passing algorithm to combine subtasks policies into a stable motion policy for all the subtasks [6]. Because of its control-theoretic guarantees and computational efficiency, RMPflow has been applied to a range of robotic applications [15, 18, 19, 21, 25, 36, 38].

Learning motion policies that can be expressed by the RMPflow framework, what we call the *RMPflow policies*, has several advantages: *1)* The task decomposition scheme in RMPflow provides an interface to inject different levels of prior knowledge. As an example, one can hand design safety-critical subtask policies for, e.g. collision avoidance, and

Algorithm	Time	Space	Req. Tree	Req. Auto.Diff.
RMP^2 (Algorithm 3)	$O(Nbd^3)$	$O(Ld^2 + Nd)$	✗	✓
Naïve Implementation (Algorithm 4)	$O(Nbd^3L)$	$O(NLd^2)$	✗	✓
RMPflow (Algorithm 1) [6]	$O(Nbd^3)$	$O(Ld^2 + Nd^2)$	✓	✗[1]

TABLE I: Comparison between different implementations of the RMPflow policy for a graph/tree with N nodes of dimension at most d, where $L \leq N$ nodes are leaf nodes and the maximum branching factor is b. We assume the automatic differentiation library is based on reverse-mode automatic differentiation. Computing the task map has time complexity of $O(Nbd^2)$ and memory complexity of $O(Nd)$. Calling the Gradient Oracle to compute the derivative of a scalar function with respect to a variable in $O(d)$ requires time complexity in $O(Nbd^2)$ and space complexity $O(Nd)$. See Appendix B for details.

learn other subtask policies to improve the overall performance; *2*) Hand-designed RMPflow policies have been applied to solve many real-world robotics applications that require complex motions, so RMPflow policies are sufficiently expressive for motion control problems; *3*) The RMPflow policies learned on one robot can be transferred to other robots because of its differential geometry centered design [6]. This allows us to easily adopt existing hand-designed subtask policies in the literature to partially parameterize the RMPflow policy to help warm-start the learning process.

The RMPflow message passing algorithm was originally designed for reactive control rather than learning. Despite above-mentioned promises, implementing a system for learning RMPflow policies faces several practical computational challenges. First, RMPflow uses a rather complicated user interface requiring a tree data structure, which is often non-trivial for the user to specify. Second, perhaps more importantly, it is hard and computationally expensive to trace the gradient flow in the message passing algorithm of RMPflow, and yet tracing gradient flows is commonly required for end-to-end learning. Although recent work has looked into learning with RMPflow [21, 22, 27], these methods either largely simplify the learning problem so that differentiating through RMPflow is not needed [2, 21, 27] or only allow for a very limited parameterization of the policy [22]. For example, Meng et al. [21] and Rana et al. [27] learn the subtask policies independently through imitation and then use RMPflow to combine the learned subtask policies with other hand-specified ones; and Mukadam et al. [22] learn scalar weight functions for pre-defined subtask polices. Recently Aljalbout et al. [2] explored learning collision avoidance policies with RMPflow through reinforcement learning, where differentiating through RMPflow is not required as the gradients can be estimated through samples and value function estimate.

In this work, we propose a simple alternative algorithm, called RMP^2 (RMPflow Reactive Motion Policy), to replace the message passing algorithm in RMPflow for computing RMPflow policies, so that end-to-end learning RMPflow polices can be more easily implemented and scaled up in practice. We emphasize that we *do not* propose a new structured policy class, but a more efficient and flexible implementation of RMPflow policies. Policies realized by either the original message passing algorithm of RMPflow or our new RMP^2 algorithm are therefore the same, and learning with them would lead to the same results statistically. For clarity, we will refer to the RMPflow algorithm as the message passing routine in RMPflow (which our RMP^2 algorithm replaces) and the RMPflow policy as the effective motion policy that RMPflow represents, which our RMP^2 algorithm also represents.

RMP^2 realizes the RMPflow policy by querying the Gradient Oracle [13] in an automatic differentiation library (such as TensorFlow [1] or PyTorch [24]) instead of using the tree data structure and message passing steps of the RMPflow algorithm. In comparison, RMP^2 has several advantages over the original RMPflow algorithm: *1*) RMP^2 allows for a simper user interface. The user only needs to specify the task map using an automatic differentiation library, the automatically constructed (directed acyclic) computational graph can then be used for computing the RMPflow policy. *2*) RMP^2 relaxes the assumption on task map structure from tree structure in the RMPflow algorithm [6] to *any* directed acyclic graphs (DAGs). *3*) RMP^2 is much easier to implement in conjunction with learning algorithms: As RMP^2 is implemented using operators supported by automatic differentiation libraries, it is convenient to take the gradient, or higher order derivatives, of *any functions* with respect to the parameters in subtask policies and task maps. *4*) RMP^2 uses a smaller memory footprint than RMPflow, while having the same time complexity (see Table I).

These computational advantages make RMP^2 generally applicable to many end-to-end learning scenarios and algorithms. In the rest of the paper, we provide the details of our new algorithmic design. At the end of the paper, we validate our RMP^2 algorithm in applications of learning acceleration-based motion control policies with reinforcement learning in simulated reaching tasks with a three-link robot arm and a Franka robot arm.

II. Background

A. Acceleration-based Motion Control

We focus on learning policies for controlling the motion of acceleration-based robotics systems. Typically this type of kinematic control problems arises when one wishes to reactively generate smooth reference trajectories for a low-level tracking controller, or wants to control a robot that is fully actuated and feedback linearized (e.g., by an inverse dynamics model).

Suppose the robot's configuration space $\mathcal{C}$ (e.g. the joint space of a robot with revolute joints) is a d-dimensional smooth

[1]Although RMPflow does not necessarily require automatic differentiation, it is commonly used to implement to Jacobian between two nodes in RMPflow.

manifold that can be described by generalized coordinates $\mathbf{q} \in \mathbb{R}^d$. We can view the acceleration-based motion control problem as a continuous-time deterministic Markov decision process (MDP): the state is the position-velocity $(\mathbf{q}, \dot{\mathbf{q}})$, the action is the acceleration $\ddot{\mathbf{q}}$, the transition is the integration rule, and the reward is defined to encourage desired behaviors for a task (such as smooth, collision-free motions). Our goal is to find an acceleration-based policy π such that the system following $\ddot{\mathbf{q}} = \pi(\mathbf{q}, \dot{\mathbf{q}})$ would exhibit good performance for the tasks of interest.

In these motion control problems, the desired behavior of a task (equivalently the reward function) is often not directly described in the generalized coordinates $\mathbf{q} \in \mathbb{R}^d$, but in terms of another set of task coordinates $\mathbf{x} \in \mathbb{R}^m$ that are related to the generalized coordinates through a nonlinear mapping ψ, i.e. $\mathbf{x} = \psi(\mathbf{q})$. For example, in controlling an anthropomorphic robot, we wish to control the torso's motion to maintain the stability and the reachable region of the hands, whose performance is more easily described in the coordinates of the torso and the hand rather than directly in the joint space (i.e. the configuration space). We call this mapping ψ the *task map* and refer to the image manifold of $\mathcal{C}$ under ψ as the *task space*, which is denoted as $\mathcal{T}$. As shown in the above example, the task here is often multi-objective in nature, requiring the robot to satisfy various performance criteria. Mathematically, this implies that the overall task space $\mathcal{T}$ is a composition of many subtask spaces, such as $\mathcal{T} = \prod_{k=1}^{K} \mathcal{T}_k$ in a K-task control problem based on subtask manifolds $\{\mathcal{T}_k\}_{k=1}^K$. These subtask coordinates are often not independent but intertwined together as the image of the common configuration space $\mathcal{C}$ under the task map ψ (in the previous example, moving the torso affects the position of the robot hand). Therefore, generally, policies designed for each subtask cannot be trivially combined together (e.g., with using a convex combination) to generate a good policy for the full task.

B. RMPflow: A Framework for Multi-task Problems

RMPflow [6] is a control-theoretic computational framework designed to address the multi-task control problem mentioned above. Inspired by geometric control theory [4], RMPflow handles the trade-off between different subtasks by describing each subtask policy as a Riemannian Motion Policy (RMP) [30]. An RMP associates a subtask (e.g. the k-th subtask) not only with the desired acceleration $\mathbf{a}_k^d(\mathbf{x}_k, \dot{\mathbf{x}}_k)$, but also with a positive semi-definite matrix function $\mathbf{M}_k(\mathbf{x}_k, \dot{\mathbf{x}}_k)$ that depends on the state (i.e. the position and the velocity) of the subtask. Given RMPs for the subtasks, RMPflow generates the policy π on the configuration space by combining these subtask RMPs through message passing on a tree data structure of manifolds (called the RMP-tree), where the root and leaf nodes correspond to the configuration space $\mathcal{C}$ and the subtask spaces $\{\mathcal{T}_k\}_{k=1}^K$, and an edge represents a smooth map from a parent node manifold to its child node manifold.

This message passing scheme of RMPflow effectively realizes a differential-geometric operation, called the *pullback*, which propagates differential forms from the subtask manifolds to the configuration space manifold. As a result, it can be proved that the final policy π output by RMPflow is Lyapunov stable, when $\mathbf{M}_k$ is derived appropriately from a Riemannian metric that describes the motion induced by $\mathbf{a}_k^d$ for each policy [6, 18]. This nice control-theoretical property makes RMPflow a promising candidate of the structured policy class for acceleration-based motion control.

Algorithm 1 The RMPflow Algorithm (Message Passing) [6]

Input: root state $(\mathbf{q}, \dot{\mathbf{q}})$, RMP-tree `T`, RMPs `rmp_eval`
Return: motion policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$
`nodes` ← `T.topologically_sorted_nodes()`
// forward pass
For `node` **in** `nodes`: *// from root to leaves*
  For `child` **in** `node.children`:
    `child.state` ← `pushforward(node.state)`
// evaluate leaf RMPs
For `node` **in** `T.leaves`:
  `node.rmp` ← `rmp_eval(node.state)`
// backward pass
For `node` **in** `reversed(nodes)`: *// from leaves to root*
  `node.rmp` ← `pullback(node.child.rmps)`
// resolve for the motion policy
$\pi(\mathbf{q}, \dot{\mathbf{q}})$ ← `resolve(T.root.rmp)`

Let us provide some intuitions as to why RMPflow works. Here we take an optimization viewpoint recently made in [5] (rather than the geometric control viewpoint commonly used in the literature) and show the optimization problem that RMPflow effectively solves when generating the multi-task control policy. This insight explains the properties of RMPflow, which is more directly related to our proposed algorithm RMP^2, without going through its complex algorithmic steps. The message passing procedure of RMPflow is listed in Algorithm 1 and a detailed description can be found in Appendix A.

Consider an RMP-tree with a set of nodes $\mathcal{V}$. Let $\mathcal{L} := \{\mathtt{l}_k\}_{k=1}^K \subset \mathcal{V}$ be the set of leaf nodes and $\mathtt{r}$ be the root node. Cheng [5, Chapter 11.7] observed that there is a connection between the message passing algorithm of RMPflow and sparse linear solvers: the RMPflow policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$ is the solution to the following least-squares problem, and the message passing routine of RMPflow effectively uses the duality of (1) and the sparsity in the task map to efficiently compute its solution.

$$\min_{\{\mathbf{a}_\mathtt{v} : \mathtt{v} \in \mathcal{V}\}} \quad \sum_{k=1}^{K} \frac{1}{2} \|\mathbf{a}_{\mathtt{l}_k} - \mathbf{a}_k^d\|_{\mathbf{M}_k}^2, \tag{1}$$

$$\text{s.t.} \quad \mathbf{a}_\mathtt{v} = \mathbf{J}_{\mathtt{v};\mathtt{u}} \mathbf{a}_\mathtt{u} + \dot{\mathbf{J}}_{\mathtt{v};\mathtt{u}} \dot{\mathbf{x}}_\mathtt{u}, \tag{2}$$

$$\forall\, \mathtt{v} \in \mathcal{V} \setminus \mathtt{r}, \quad \mathtt{u} = \mathtt{parent}(\mathtt{v})$$

where, for a leaf node $\mathtt{l}_k$, $\mathbf{a}_k^d$ and $\mathbf{M}_k$ together are a leaf-node RMP, $\mathbf{J}_{\mathtt{v};\mathtt{u}}$ denotes the Jacobian of the task map from node $\mathtt{u}$ to node $\mathtt{v}$, and $\dot{\mathbf{J}}_{\mathtt{v};\mathtt{u}}$ is the time-derivative of Jacobian $\mathbf{J}_{\mathtt{v};\mathtt{u}}$. The objective in (1) is the sum of deviations between the acceleration $\mathbf{a}_{\mathtt{l}_k}$ on each leaf space and the desired one, $\mathbf{a}_k^d$, weighted by the importance matrix $\mathbf{M}_k$. The constraints (2) enforce the accelerations to be consistent with the maps between spaces. In other words, the leaf-node RMPs in RMPflow defines a trade-off between different subtask control

schemes and the policy of RMPflow is the optimal solution that can be realized under the geometric constraints between the subtask spaces and the configuration space.

Implementing a system for learning RMPflow policies, however, can be challenging, because the user needs to implement the data structure and the complex message passing algorithm, both described in Appendix A (which we omitted here due to its complexity). Moreover, the user may need to trace the gradient flow through this large computational graph. This difficulty has limited the applicability of existing work on end-to-end learning of RMPflow policies [21, 22, 27, 29]. Improving the efficiency and simplicity of implementing RMPflow policies is hence the main objective of our paper.

III. RMP^2 BASED ON AUTOMATIC DIFFERENTIATION

We propose an alternate algorithm to implement the RMPflow policy. Our new algorithm, RMP^2, works by calling the basic oracles (such as evaluation and the Gradient Oracle) of an automatic differentiation library, without using the message passing algorithm of RMPflow in Appendix A. The result is an easy-to-use and computationally efficient framework suitable for learning RMPflow policies end-to-end.

A. Key Idea

We observe that the constrained optimization problem (1)–(2) is equivalent to the following unconstrained least-squares optimization problem if the constraints are aggregated:

$$\min_{\mathbf{a}'_{\mathtt{r}} \in \mathbb{R}^d} \quad \sum_{k=1}^{K} \frac{1}{2} \left\| \mathbf{J}_{\mathtt{l}_k;\mathtt{r}} \mathbf{a}'_{\mathtt{r}} + \dot{\mathbf{J}}_{\mathtt{l}_k;\mathtt{r}} \dot{\mathbf{q}} - \mathbf{a}_k^d \right\|^2_{\mathbf{M}_k}, \tag{3}$$

where $\mathbf{J}_{\mathtt{l}_k;\mathtt{r}}$ is the Jacobian matrix of the subtask map from the root node $\mathtt{r}$ to the kth leaf node $\mathtt{l}_k$. Note that the Jacobians and velocities here are treated as constants in the optimization as they are only dependent on the state (not the accelerations).

This observation implies that we can compute the RMPflow policy by solving (3), which has a closed-form solution:

$$\mathbf{a}_{\mathtt{r}} = \underbrace{\left(\sum_{k=1}^{K} \mathbf{J}_{\mathtt{l}_k;\mathtt{r}}^\top \mathbf{M}_k \mathbf{J}_{\mathtt{l}_k;\mathtt{r}} \right)^\dagger}_{\mathbf{M}_{\mathtt{r}}^\dagger} \underbrace{\left(\sum_{k=1}^{K} \mathbf{J}_{\mathtt{l}_k;\mathtt{r}}^\top \mathbf{M}_k (\mathbf{a}_k^d - \dot{\mathbf{J}}_{\mathtt{l}_k;\mathtt{r}} \dot{\mathbf{q}}) \right)}_{\mathbf{f}_{\mathtt{r}}}. \tag{4}$$

This means that if we can compute the solution in (4) efficiently for a large set of sparsely-connected manifolds, then we can realize the RMPflow Policy without the RMPflow message passing algorithm (Algorithm 1).

B. RMP^2 based on Reverse Accumulation

We propose RMP^2 (Algorithm 3), an efficient technique that computes the closed-form solution (4) using automatic differentiable libraries, which are commonly used in modern machine learning. As we will show in Appendix B, RMP^2 has the same time complexity of as RMPflow [6] while enjoying a smaller memory footprint. More importantly, RMP^2 provides a simpler and more intuitive interface for learning.

Algorithm 2 Jacobian-vector product [10] `jvp(v, u, w)`

1: **Input:** $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$
2: **Return:** $(\partial_{\mathbf{u}} \mathbf{v})\, \mathbf{w}$
3: $\boldsymbol{\lambda} \leftarrow \mathbf{1}$ *// dummy variable for reverse accumulation*
4: $\mathbf{g} \leftarrow$ `gradient(`$\boldsymbol{\lambda}^\top \mathbf{v}$`, u)` *// sum of partial derivatives*
5: compute Jacobian-vector product

$$(\partial_{\mathbf{u}} \mathbf{v})\, \mathbf{w} \leftarrow \texttt{gradient}(\mathbf{g}^\top \mathbf{v}, \boldsymbol{\lambda})$$

Algorithm 3 RMP^2

1: **Input:** root state $(\mathbf{q}, \dot{\mathbf{q}})$, `task_map`, `rmp_eval`
2: **Return:** motion policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$
// forward pass
3: $\{\mathbf{x}_k\}_{k=1}^K \leftarrow$ `task_map(`$\mathbf{q}$`)`
4: $\{\dot{\mathbf{x}}_k\}_{k=1}^K \leftarrow$ `jvp(`$\{\mathbf{x}_k\}_{k=1}^K, \mathbf{q}, \dot{\mathbf{q}}$`)` *// leaf node velocity*
5: $\{\mathbf{c}_k\}_{k=1}^K \leftarrow$ `jvp(`$\{\dot{\mathbf{x}}_k\}_{k=1}^K, \mathbf{q}, \dot{\mathbf{q}}$`)` *// curvature terms*
// evaluate leaf RMPs
6: $\{(\mathbf{M}_k, \mathbf{a}_k^d)\}_{k=1}^K \leftarrow$ `rmp_eval(`$\{(\mathbf{x}_k, \dot{\mathbf{x}}_k)\}_{k=1}^K$`)`
// backward pass
7: $\mathbf{q}' =$ `copy(`$\mathbf{q}$`)`, $\mathbf{q}'' =$ `copy(`$\mathbf{q}$`)` *// copies without gradient*
8: $\{\mathbf{x}'_k\}_{k=1}^K \leftarrow$ `task_map(`$\mathbf{q}'$`)`, $\{\mathbf{x}''_k\}_{k=1}^K \leftarrow$ `task_map(`$\mathbf{q}''$`)` *// mirrored images*
9: $r \leftarrow \sum_{k=1}^K (\mathbf{x}'_k)^\top \mathbf{M}_k \mathbf{x}''_k, \quad s \leftarrow \sum_{k=1}^K (\mathbf{x}'_k)^\top \mathbf{M}_k (\mathbf{a}_k^d - \mathbf{c}_k)$ *// auxiliary variables*
10: $\mathbf{M}_{\mathtt{r}} \leftarrow$ `jacobian(gradient(`$r, \mathbf{q}$`), `$\mathbf{q}'$`)` *// root matrix*
11: $\mathbf{f}_{\mathtt{r}} \leftarrow$ `gradient(`$s, \mathbf{q}$`)` *// root force*
// resolve for the motion policy
12: $\pi(\mathbf{q}, \dot{\mathbf{q}}) \leftarrow \mathbf{M}_{\mathtt{r}}^\dagger \mathbf{f}_{\mathtt{r}}$

In automatic differentiation libraries, a computation graph is automatically built as functions are specified, and derivatives are computed through message passing on the computation graph. RMP^2 leverages this feature so that no additional data structure and message passing routines are needed, whereas RMPflow requires the user to specify an RMP-tree and implement the message passing algorithm.

RMP^2 uses the following common functionalities provided by automatic differentiation libraries:

- `gradient(`$s, \mathbf{u}$`)`: the gradient operator. It computes the gradient of a scalar graph output $s \in \mathbb{R}$ with respect to the graph input vector $\mathbf{u} \in \mathbb{R}^n$ through back-propagation;
- `jacobian(`$\mathbf{v}, \mathbf{u}$`)`: the Jacobian operator. It computes the Jacobian matrix $\partial_{\mathbf{u}} \mathbf{v} \in \mathbb{R}^{m \times n}$ through back-propagation; `jacobian` is equivalent to multiple calls of `gradient`.
- `jvp(`$\mathbf{v}, \mathbf{u}, \mathbf{w}$`)`: the Jacobian-vector product. It computes $(\partial_{\mathbf{u}} \mathbf{v})\, \mathbf{w} \in \mathbb{R}^m$ given the graph input vector $\mathbf{u} \in \mathbb{R}^n$, the output vector $\mathbf{v} \in \mathbb{R}^m$, and an addition vector $\mathbf{w} \in \mathbb{R}^n$. The Jacobian-vector product can be efficiently realized by `gradient` (see Algorithm 2) using a technique, called reverse accumulation [10] (also known as double backward). The algorithm computes Jacobian-vector product through 2 passes. An auxiliary all-ones vector $\boldsymbol{\lambda}$ is created and the gradient with respect to $\boldsymbol{\lambda}$ is tracked (line 2).

By using the `gradient`, `jacobian`, `jvp` operators, RMP^2 in Algorithm 3 efficiently computes quantities needed for

evaluating (4). In the forward pass of RMP^2 (Algorithm 3, line 3–5), it evaluates subtask maps and compute their velocities and curvature terms using Jacobian-vector products. Specifically, for the k-th subtask map $\psi_{\mathtt{l}_k;\mathtt{r}} : \mathbf{q} \mapsto \mathbf{x}_k$ (which we may think as the map from the joint space of a robot manipulator to the workspace), the velocity $\dot{\mathbf{x}}_k := \mathbf{J}_{\mathtt{l}_k;\mathtt{r}}\dot{\mathbf{q}}$ and curvature term $\mathbf{c}_k := \dot{\mathbf{J}}_{\mathtt{l}_k;\mathtt{r}}\dot{\mathbf{q}}$ on the subtask space can be computed through Jacobian-vector products:

$$\dot{\mathbf{x}}_k = \texttt{jvp}\,(\mathbf{x}_k, \mathbf{q}, \dot{\mathbf{q}})\,, \quad \text{and} \quad \mathbf{c}_k = \texttt{jvp}\,(\dot{\mathbf{x}}_k, \mathbf{q}, \dot{\mathbf{q}})\,. \tag{5}$$

Using $\{(\mathbf{x}_k, \dot{\mathbf{x}}_k)\}$, RMP^2 then evaluates the values of the leaf RMPs (line 6), which implicitly define the objective of the weighted least-squares problem in (3). Next, in the backward pass (line 7–11), RMP^2 computes the pullback force $\mathbf{f}_\mathtt{r}$ and importance weight matrix $\mathbf{M}_\mathtt{r}$ in (4) using reverse accumulation (i.e., the technique used in `jvp` in Algorithm 2). This is accomplished by creating auxiliary variables $\mathbf{q}'$ and $\mathbf{q}''$, which have the same numerical value as $\mathbf{q}$ (but are different nodes in the computational graph of automatic differentiation), and their mirrored task images (line 7 and 8), and then querying the gradients and Jacobians:

$$\begin{aligned}
\mathbf{f}_\mathtt{r} &= \texttt{gradient}\Big(\sum_{k=1}^{K}(\mathbf{x}_k')^\top \mathbf{M}_k(\mathbf{a}_k^d - \mathbf{c}_k), \mathbf{q}'\Big), \\
\mathbf{M}_\mathtt{r} &= \texttt{jacobian}\Big(\texttt{gradient}\big(\sum_{k=1}^{K}(\mathbf{x}_k')^\top \mathbf{M}_k \mathbf{x}_k'', \mathbf{q}'\big), \mathbf{q}''\Big),
\end{aligned} \tag{6}$$

where $\mathbf{x}_k' = \psi_k(\mathbf{q}')$, $\mathbf{x}_k'' = \psi_k(\mathbf{q}'')$, and $\mathbf{q} = \mathbf{q}' = \mathbf{q}''$.

C. Complexity of RMP^2

In Appendix B, we analyze the time and space complexities of RMP^2. We show that RMP^2 has a time complexity of $O(Nbd^3)$ and a memory complexity of $O(Nd + Ld^2)$, where N is the total number of nodes, L is the number of leaf nodes, b is the maximum branching factor, and d is the maximum dimension of nodes. In comparison, we prove, also in Appendix B, that the original RMPflow algorithm (Algorithm 1) by [6] has a time complexity of $O(Nbd^3)$ and a *worse* space complexity of $O(Nd^2 + Ld^2)$. Please see Table I for a summary.

D. Discussion: A Naïve Alternative Algorithm

With the RMPflow policy expression in (4), one may attempt to explicitly compute the matrices and vectors listed in (4) using automatic differentiation and then combine them to compute the RMPflow policy. This idea leads to a conceptually simpler algorithm, shown in Algorithm 4, which directly computes the Jacobians for the task maps through the `jacobian` operator provided by the automatic differentiation library, and then compute the root RMPs $\mathbf{M}_\mathtt{r}$ and $\mathbf{f}_\mathtt{r}$ based on (4).

However, this naïve approach turns out to be not as efficient as RMP^2 and RMPflow. Because the Jacobian matrix $\mathbf{J}_k := \mathbf{J}_{\mathtt{l}_k;\mathtt{r}}$ is constructed here (line 4 in Algorithm 4), the time complexity of the naïve algorithm is $O(L)$ times larger than RMP^2, where L is the number of the leaf nodes. For a binary tree with N nodes, this means the time complexity is in $O(N^2)$, not the $O(N)$ of RMP^2 and RMPflow.

Algorithm 4 A Naïve Implementation

1: **Input:** root state $(\mathbf{q}, \dot{\mathbf{q}})$, `task_map`, `rmp_eval`
2: **Return:** motion policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$
// forward pass
3: $\{\mathbf{x}_k\}_{k=1}^K \leftarrow \texttt{task\_map}\,(\mathbf{q})$
4: $\{\mathbf{J}_k\}_{k=1}^K \leftarrow \texttt{jacobian}\,(\{\mathbf{x}_k\}_{k=1}^K, \mathbf{q})$
5: $\{\dot{\mathbf{x}}_k\}_{k=1}^K \leftarrow \{\mathbf{J}_k\dot{\mathbf{q}}_k\}_{k=1}^K$ *// leaf space velocity*
6: $\{\mathbf{c}_k\}_{k=1}^K \leftarrow \texttt{jvp}\,(\{\dot{\mathbf{x}}_k\}_{k=1}^K, \mathbf{q}, \dot{\mathbf{q}})$ *// curvature terms*
// evaluate leaf RMPs
7: compute leaf RMPs

$$\{(\mathbf{M}_k, \mathbf{a}_k^d)\}_{k=1}^K \leftarrow \texttt{rmp\_eval}\,(\{(\mathbf{x}_k, \dot{\mathbf{x}}_k)\}_{k=1}^K)$$

// backward pass
8: compute root RMP

$$\mathbf{M}_\mathtt{r} \leftarrow \textstyle\sum_{k=1}^K \mathbf{J}_k^\top \mathbf{M}_k \mathbf{J}_k, \quad \mathbf{f}_\mathtt{r} \leftarrow \sum_{k=1}^K \mathbf{J}_k^\top \mathbf{M}_k(\mathbf{a}_k^d - \mathbf{c}_k)$$

// resolve for the motion policy
9: $\pi(\mathbf{q}, \dot{\mathbf{q}}) \leftarrow \mathbf{M}_\mathtt{r}^\dagger \mathbf{f}_\mathtt{r}$

Moreover, this naïve algorithm also has a worse space complexity of $O(NLd^2)$. This large space usage is created by the computation of the curvature term: the curvature term here is computed through differentiating velocities $\{\dot{\mathbf{x}}_k\}$ that are computed by the explicit Jacobian vector product in line 5; as a result, the intermediate graph created by the Jacobian needs to be stored, which is the source of high memory usage. A simple fix to this memory usage is to compute the curvature term instead by two calls of Jacobian-vector-product (as in RMP^2). This modified algorithm has the time complexity of $O(Nbd^3L)$ (still $O(L)$ times larger than RMP^2 and RMPflow) but a space complexity of $O(Nd + Ld^2)$ (same as RMP^2).

E. Key Benefits of RMP^2

Simpler User Interface: The major benefit of RMP^2 is that it is much easier to implement and apply than RMPflow, while producing the same policy and having the same time complexity. For RMP^2, the user no longer needs to construct the RMP-tree data structure (just like implementing a neural network architecture from scratch) or implement the message passing algorithm. Instead, the user only needs to specify the task maps with automatic differentiation libraries, and the policy can be computed through standard operators in automatic differentiation libraries. See Appendix D for a case study.

More General Taskmaps: Another benefit of RMP^2 is that it supports task maps described by *arbitrary* directed acyclic graphs (DAGs), whereas RMPflow is limited to tree-structured task maps. While Cheng et al. [6] show that every task map has a tree representation, not all motion control problems have an *intuitive* RMP-tree representation (e.g. multi-robot control [19]). If they are implemented using a tree structure, extra high-dimensional nodes would be induced and the user interface becomes tedious.

Differentiable Policies: Since RMP^2 is implemented using automatic differentiation libraries, computational graphs can be automatically constructed while calculating the policy. This allows for convenient gradient calculation of *any functions* with

respect to *any parameters* in, e.g., parameters used to describe task maps and RMPs. This fully differentiable structure is useful to end-to-end learning in many scenarios.

Smaller Memory Footprint: As is analyzed in Appendix B, RMP^2 has a memory footprint of $O(Nd + Ld^2)$, which is smaller than RMPflow (see Table I).

In summary, RMP^2 is an efficient algorithm that is easier to implement and apply, while providing a more convenient interface for learning applications.

IV. RMP^2 for Learning

In this section, we discuss various options of using RMP^2 to parameterize structured policies for learning. We note that some examples below have already been explored by existing work using the RMPflow algorithm. However, in most cases, realizing these ideas with the new RMP^2 algorithm instead of the message passing algorithm of RMPflow would largely simplify the setup and implementation, as RMP^2 provides a more natural interface for learning. Moreover, RMP^2 enables graph-structured task maps, whereas the RMPflow algorithm works only with tree-structured task maps.

A. Parameterizing RMP^2 Policies

RMP^2 policies are alternate parameterizations of RMPflow policies. They differ only in the way how (4) is computed (RMP^2 uses automatic differentiation whereas RMPflow uses a message passing routine). These two paramertizations therefore have the same representation power, but potentially the RMP^2 policies are more computationally efficient, because the tree structure used in RMPflow may not be the most natural way to describe the task map.

Learnable leaf RMPs: One way to parameterize RMP^2 policies is through parameterizing leaf RMPs in RMP^2. There have been existing work exploring various ways to represent RMPs with neural networks so that the resulting policy can have certain theoretical properties, e.g. positive-definiteness of importance weight matrices [27], Lyapunov-type stability guarantees [22, 27], etc.

Learnable task maps: Perhaps less obviously, one can also learn the task maps. For example, existing work has developed task map learning techniques such that the latent space policies take in simple forms or are easier to be learned [28, 37]. These task map learning techniques can be easily realized when learning RMP^2 polices.

B. Learning Setups

Because RMP^2 is based on automatic differentiation, it can be implemented naturally within the typical machine learning pipelines. Below we discuss common scenarios.

Supervised Learning: When there is an expert policy, one common scenario for robot learning is behavior cloning [26], which minimizes the empirical difference between the learner and expert policies on a dataset collected by the expert policy. Gradient-based algorithms are often used to minimize the error, which requires computing the derivative of the acceleration-based policy with respect to the parameters. Due to the difficulty in differentiating through the RMPflow algorithm, most existing work on learning the RMPs with supervised learning either differentiates through an approximate algorithm (e.g. without the curvature terms) [21], or learn with a trivial task map [27]. By contrast, using our proposed RMP^2 algorithm, we can easily combine gradient-based learning algorithm with arbitrary task map or RMP parameterizations.

Reinforcement Learning: In reinforcement learning (RL) applications, one can choose whether to differentiate through the RMP^2 algorithm: One can either choose RMP^2 as part of the policy, or as a component of the environment dynamics. This choice can be made in consideration of the policy parameterization. For example, if a large number of leaf RMPs is parameterized, it could be beneficial to consider RMP^2 as part of the policy and differentiate through it, because otherwise it will result in a high-dimensional action space for RL. On the other hand, if RMP^2 is considered as part of the environment, the output of the parameterized RMPs or parameterized task maps are treated as the action in RL, and there is no need to differentiate through RMP^2. When there is only a single low-dimensional leaf RMP to be learned, it might be convenient to consider RMP^2 as part of the environment so that policy update is faster.[2] Recently Aljalbout et al. [2] explored learning collision avoidance RMPs with the RMPflow algorithm as a component of the environment.

C. Learning with Residuals

For many robotics tasks, hand-crafted RMPs [6] can provide a possibly sub-optimal but informative prior solution to the task or some subtasks (e.g. avoiding collision, respecting joint limits, etc.). In many cases, making use of these hand-crafted RMPs within RMP^2 policies can benefit learning, as it could provide a reasonable initialization for the learner and, for RL, an initial state visitation distribution with higher rewards.

Residual Acceleration Learning: Perhaps the most straightforward solution is to learn the residual policy of the RMP^2 policy using universal functional approximators, e.g. neural networks [14]. As we show in the experiments, this can often provide a significant improvement to the performance compared to randomly initialized policies, especially when the tasks are more challenging.

Residual RMP learning: Another option is to learn a residual leaf RMP with a universal functional approximator (i.e. the leaf RMP is initialized as the hand-crafted RMP). In this way, the structure of the RMP^2 policy is preserved. As is shown in the experiments, residual RMP learning can perform significantly better than randomly initialized neural network policies, and can sometimes learn faster than the residual acceleration learning approach.

V. Experiments

A. Three-Link Robot Reaching

We first consider a three-link robot simulated by the PyBullet physics engine [11]. The robot has 3 links, each of length 0.25

[2]The time for differentiating through RMP^2 is a constant factor more than the time required for computing RMP^2, which is still reasonably fast.

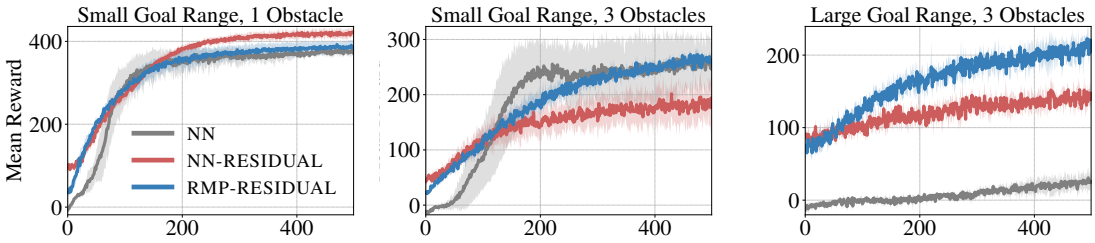


Fig. 1: Mean episode reward over training iterations for the three-link robot reaching task. See text for details.

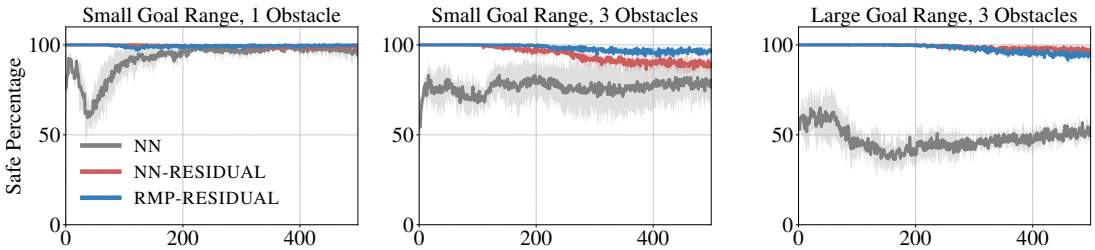


Fig. 2: Percentage of safe episodes over training iterations for the Franka robot reaching task. See text for details.

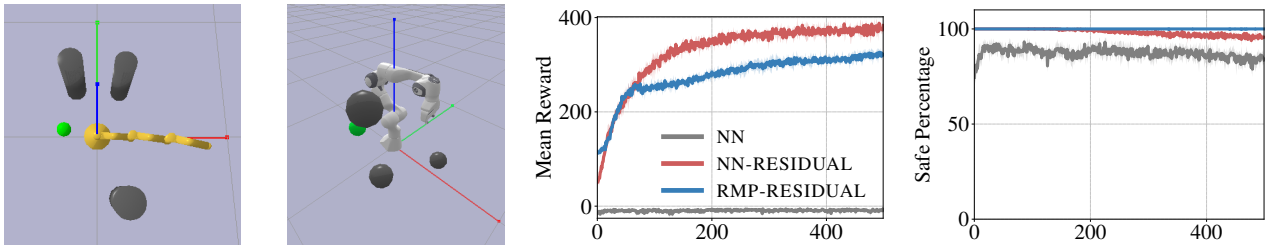


Fig. 3: (a)–(b) PyBullet simulation for the three-link robot reaching task (a) and the Franka robot reaching task (b). (c)–(d) Mean episode reward (c) and safe episode percentage (d) over training iterations for the Franka reaching task.

m. The workspace of the robot is a 2-dimensional disk of radius 0.75 m. The z-coordinates of all links are different so that the links cannot collide into one another. The objective here is for the robot to move the tip of the last link (i.e., the end-effector of the robot) to a randomly generated goal location while avoiding randomly generated cylinders, as shown in Fig. 3 (a). Acceleration-based control is realized through a low-level PD tracking controller, where the acceleration motion policy π generates the desired reference state for the PD controller. The robot does not have joint limits but have joint velocity limit of 1.0 rad/s for all joints.

Environment Setups: The robot is initialized at a random configuration within a small range (± 0.1rad) around the zero-configuration (all links pointing right) and at a random low joint velocity within $[-0.005, 0.005]$ rad/s.

The (2-dimensional) center of the base of each obstacle is sampled from an annulus centered at the origin with outer radius 0.9 m and inner radius 0.4 m, and the radius is sampled uniformly from $[0.05, 0.1]$ m. The height of the cylinder is fixed at 0.5 m, which would result in collision if the x, y-coordinates of any points on the robot intersect with the cylinder. The consideration behind this obstacle configuration is that their intersection with the workspace is usually non-zero, and they would not result in a configuration where the goal is not achievable. The initial configurations of the environment also ensures a minimum of 0.1 m between the goal and any obstacles as well as that between the initial configuration of the robot and any obstacles (otherwise, the initialization is rejected, and the goal and obstacle(s) are re-sampled).

We consider three environment setups for the three-link robot with increasing difficulty:

- `Env 1` (small goal range; 1 obstacle): the goal is sampled from the intersection of an octants (sector with central angle $\pi/2$) at the origin and an annulus with outer radius 0.275 m and inner radius 0.475 m. One obstacle is sampled from the procedure described above;
- `Env 2` (small goal range; 3 obstacles): the goal is sampled from the same region as Env 1. Three obstacles are sampled through the same procedure;

- `Env 3` (large goal range; 3 obstacles): the goal is sampled from the intersection of the left half-disk and an annulus with outer radius 0.125 m and inner radius 0.625 m.

`Env 2` is more difficult than `Env 1` as there are more obstacles. `Env 3` is the most complicated scenario as it has a larger range of random goals, which is known to be challenging for RL algorithms [3]. Moreover, although the goals here are generally closer than the previous 2 environment setups, this also increase the frequency of the scenarios where the obstacles are directly obstructing the way to the goal, requiring a more sophisticated policy.

Inspired by [17], we define the reward function as, $r = \exp\left(\frac{\|\mathbf{x}-\mathbf{g}\|_2^2}{2\sigma^2}\right) + \sum_{i=1}^{N} \max\left(0, 1 - \frac{d_i}{\delta}\right) - \lambda\|\tau\|^2$, where $\mathbf{x}$ is the position of the end-effector of the robot, $\mathbf{g}$ is the goal position, d_i is the distance between the robot and the ith obstacle, and τ is the torque applied to the robot by the low-level PD controller. The scalars σ, δ, and λ are the characteristic length scale of the goal reward, characteristic length scale of the obstacle cost, and the multiplier for the actuation cost. We choose $\sigma = 0.1$, $\delta = 0.05$, and $\lambda = 1 \times 10^{-5}$. Further, we clip the reward if it is smaller than -5 so that the reward is in the range of $[-5, 1]$ for each step.

The horizon of each episode is 600 steps, giving the episode reward a range of $[-3000, 600]$. For each step, the acceleration command is applied to the low-level PD-controller, and the simulation proceeds for 0.0125s (simulation time). Therefore, each episode is 7.5s (simulation time) of policy rollout. The episode can end early if the robot collides with an obstacle.

Policies: We compare the results for learning with the following 3 types of policies, all implemented in TensorFlow [1].

NN: A randomly initialized 3-layer neural network policy with relu activation function and hidden layers of sizes 256 and 128, respectively. The input to the neural network policy consists of $[\sin(\mathbf{q}), \cos(\mathbf{q}), \dot{\mathbf{q}}, \mathbf{g} - \mathbf{x}, \{\mathbf{v}_i\}_i, \{\mathbf{o}_i\}_i]$, where $\mathbf{v}_i$ is the vector pointing from the i-th obstacle and its closest point on the robot, and $\mathbf{o}_i$ denotes the center position and radius of the i-th obstacle. The output of the policy is the 3-dimensional joint acceleration.

NN-RESIDUAL: A residual neural network policy to a hand-designed RMP^2 policy. The input and neural network architecture here are the same as the randomly-initialized case above. However, the joint acceleration now is the sum of the output of the residual policy and the hand-designed RMP^2 policy. The hand-designed RMP^2 policy consists of a joint damping RMP, a joint velocity limit RMP, collision avoidance RMPs, and a goal attractor RMP [6].

RMP-RESIDUAL: A residual RMP policy on the 2-dimensional end-effector space, which is the residual to a hand-designed goal attractor RMP (same as the one used for NN-RESIDUAL). The residual RMP policy is parameterized as a 3-layer neural network with hidden layer sizes 128 and 64. We use elu activation function for the neural network. The input to the residual RMP policy consists of $[\mathbf{x}, \dot{\mathbf{x}}, \mathbf{g}, \{\mathbf{o}_i\}_i]$, which are the end effector position, velocity, and the information of goal and obstacle(s). The output of the neural network is (the concatenation of) a matrix $\mathbf{A}_r \in \mathbb{R}^{2\times 2}$ and a residual acceleration vector $\mathbf{a}_r \in \mathbb{R}^2$. Let $(\mathbf{M}_a, \mathbf{a}_a)$ be the output for the hand-designed attractor, the output of the residual RMP policy is then, $\mathbf{M} = (\mathbf{A}_r + \mathtt{chol}(\mathbf{M}_a))(\mathbf{A}_r + \mathtt{chol}(\mathbf{M}_a))^\top$ and $\mathbf{a} = \mathbf{a}_r + \mathbf{a}_a$, where $\mathtt{chol}(\cdot)$ is the lower-triangular Cholesky decomposition of the matrix. This parameterization ensures that the importance weight $\mathbf{M}$ is always positive-semidefinite, and the output is close to the hand-designed RMP when the neural network is initialized with weights close to zero. The output of the residual RMP policy, is combined with other hand-designed RMPs (the same set of RMPs with NN-RESIDUAL) with RMP^2 to produce the joint acceleration. Since we are learning a low-dimensional RMP, as is discussed in Section IV, it is more convenient to consider RMP^2 as part of the environment so that the policy update is faster.

Training Details: We train the three policies under the three environment setups by Proximal Policy Optimization (PPO) [34], implemented by RLlib [20], for 500 training iterations. For each iteration, we collect a batch of 67312 interactions with the environment (112 episodes if there is not collision). We use a learning rate of 5×10^{-5}, PPO clip parameter of 0.2. To compute the policy gradient, we use Generalized Advantage Estimation (GAE) [33] with $\lambda = 0.99$. The value function is parameterized by a neural network with 2 hidden layers (of sizes 256 and 128, respectively) and $\tanh$ activation function.

Results: The mean episode reward and percentage of safe episodes over training iterations for the three environment setups are shown in Fig. 1 and Fig. 2, respectively. The curve and the shaded region show the average, and the region within 1 standard deviation from the mean over 4 random seeds.

For `env 1` with small goal range and 1 obstacle, all three policies manage to achieve high reward of around 400 (meaning that, on average, the robot stays very close to the goal for at least 400 steps out of the 600 steps) with a similar number of iterations. The random-initialized neural network (NN) conducts a large number of unsafe exploration, especially during the first 200 iterations, as shown in Fig. 2 (a). The other two policies manage to learn a good policy without many collisions as the red and blue curves stay close to 100% throughout training.

For `env 2` with 3 obstacles, the randomly-initialized neural network policy (NN) learns slightly faster than the other two polices, though it has a higher variance (shown by the large gray shaded region in Fig. 1 (b)). Notably, although the reward seems high, the resulting policies are not safe, as shown in Fig. 2 (b). This reflects the difficulty of reward design: as high reward policies do not necessarily have desirable behaviors. On the contrary, the residual RMP policy (RMP-RESIDUAL) achieves similar reward but is able to keep the number of collisions low. The residual neural network policy (NN-RESIDUAL) improves rather slowly, possibly due to the lack of knowledge about the internal structure of RMP^2.

For `env 3` with large goal range and 3 obstacles, the neural netowrk policy (NN) struggles to achieve reasonable performance under 500 training iterations, and the collision rate

remains high. The performance improvement for the reisual neural network policy (NN-RESIDUAL) is also slow, with a slope similar to the neural network policy, though it starts with a higher rewards. The residual RMP policy manages to improve the performance by more than one-fold.

Remark: One may notice that the initial reward for the residual policies is different for each experiment setup. For example, the initial rewards for `env 3` is significantly higher than `env 2`. This is because we used the *same* hand-designed RMP policy for all three setups, and in `env 3`, the goals are generally initialized closer to the robot than the other two steps (as discussed, this does not make `env 3` easier however).

B. Franka Robot Reaching

We additionally train the three aforementioned policies on a simulated 7-degree-of-freedom Franka manipulator on PyBullet. The environment setup is similar to the three-link robot reaching task, although the initial configuration is a centered position, as shown in Fig. 3 (b). We randomly sample 3 ball obstacles, where the center is sampled a half-torus of major radius 0.5 m, minor radius 0.3 m, and height 0.5 m; and the radius is uniformly sampled from $[0.05, 0.1]$ m. The goal is sampled from the same half-torus, but the distance between the goal and initial end-effector position needs to be at least 0.5 m. The Franka reaching task is more challenging than the three-link robot reaching task as the states are of higher-dimension; however, it is easier in the sense that it has higher degrees of freedom to avoid collision with obstacles.

Results The mean episode reward and safe percentage of the three polices for the Franka reaching task is shown in Fig. 3 (right 2). Again, the neural network policy (NN) struggles to learn a good policy and avoid collision with obstacles. The residual neural network policy (NN-RESIDUAL) achieves slightly higher reward than the residual RMP policy (RMP-RESIDUAL), possibly because it has higher degree-of-freedoms to control the robot and the learning of the residual RMP policy (RMP-RESIDUAL) has not fully converged. Notably, the residual RMP policy (RMP-RESIDUAL) maintain zero collisions throughout the training process.

ACKNOWLEDGMENTS

This work was supported in part by an NVIDIA Graduate Fellowship.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[2] E. Aljalbout, J. Chen, K. Ritt, M. Ulmer, and S. Haddadin. Learning vision-based reactive policies for obstacle avoidance. In *Conference on Robot Learning*, 2020.

[3] P. Aumjaud, D. McAuliffe, F. J. Rodríguez-Lera, and P. Cardiff. Reinforcement learning experiments and benchmark for solving robotic reaching tasks. In *Workshop of Physical Agents*, pages 318–331. Springer, 2020.

[4] F. Bullo and A. D. Lewis. *Geometric control of mechanical systems: modeling, analysis, and design for simple mechanical control systems*. Springer New York, 2005.

[5] C.-A. Cheng. *Efficient and principled robot learning: theory and algorithms*. PhD thesis, Georgia Institute of Technology, 2020.

[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 *International Workshop on the Algorithmic Foundations of Robotics*. Springer, 2018.

[7] C.-A. Cheng, M. Mukadam, J. Issac, S. Birchfield, D. Fox, B. Boots, and N. Ratliff. RMPflow: A geometric framework for generation of multi-task motion policies. *IEEE Transactions on Automation Science and Engineering*, 2021.

[8] J. Choi, F. Castañeda, C. J. Tomlin, and K. Sreenath. Reinforcement learning for safety-critical control under model uncertainty, using control Lyapunov functions and control barrier functions. In *Proceedings of Robotics: Science and Systems*, 2020.

[9] Y. Chow, O. Nachum, A. Faust, E. Duenez-Guzman, and M. Ghavamzadeh. Lyapunov-based safe policy optimization for continuous control. *arXiv preprint arXiv:1901.10031*, 2019.

[10] B. Christianson. Automatic Hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, 12(2): 135–150, 1992.

[11] E. Coumans and Y. Bai. PyBullet, a python module for physics simulation in robotics, games and machine learning, 2017.

[12] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*, 2018.

[13] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[14] T. Johannink, S. Bahl, A. Nair, J. Luo, A. Kumar, M. Loskyll, J. A. Ojea, E. Solowjow, and S. Levine. Residual reinforcement learning for robot control. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6023–6029. IEEE, 2019.

[15] 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. *IEEE Robotics and Automation Letters*, 3(3): 1864–1871, 2018. URL https://arxiv.org/abs/1703.03512.

[16] H. K. Khalil and J. W. Grizzle. *Nonlinear systems*,

volume 3. Prentice hall Upper Saddle River, NJ, 2002.

[17] V. Kumar, D. Hoeller, B. Sundaralingam, J. Tremblay, and S. Birchfield. Joint space control via deep reinforcement learning. *arXiv preprint arXiv:2011.06332*, 2020.

[18] A. Li, C.-A. Cheng, B. Boots, and M. Egerstedt. Stable, concurrent controller composition for multi-objective robotic tasks. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 1144–1151. IEEE, 2019.

[19] A. Li, M. Mukadam, M. Egerstedt, and B. Boots. Multi-objective policy generation for multi-robot systems using Riemannian motion policies. In *International Symposium on Robotics Research*, 2019.

[20] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[21] X. Meng, N. Ratliff, Y. Xiang, and D. Fox. Neural autonomous navigation with Riemannian motion policy. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8860–8866. IEEE, 2019.

[22] M. Mukadam, C.-A. Cheng, D. Fox, B. Boots, and N. Ratliff. Riemannian motion policy fusion through learnable Lyapunov function reshaping. In *Conference on Robot Learning*, pages 204–219, 2019.

[23] J. Nakanishi, R. Cory, M. Mistry, J. Peters, and S. Schaal. Operational space control: A theoretical and empirical comparison. *The International Journal of Robotics Research*, 27(6):737–757, 2008.

[24] 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. In *NIPS Autodiff Workshop*, 2017.

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

[26] D. A. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In *Proceedings of the 1st International Conference on Neural Information Processing Systems*, pages 305–313, 1988.

[27] M. A. Rana, A. Li, H. Ravichandar, M. Mukadam, S. Chernova, D. Fox, B. Boots, and N. Ratliff. Learning reactive motion policies in multiple task spaces from human demonstrations. In *Conference on Robot Learning*, 2019.

[28] M. A. Rana, A. Li, D. Fox, B. Boots, F. Ramos, and N. Ratliff. Euclideanizing flows: Diffeomorphic reduction for learning stable dynamical systems. In *Learning for Dynamics and Control*, pages 630–639. PMLR, 2020.

[29] M. A. Rana, A. Li, D. Fox, S. Chernova, B. Boots, and N. Ratliff. Towards coordinated robot motions: End-to-end learning of motion policies on transform trees. *arXiv preprint arXiv:2012.13457*, 2020.

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

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

[32] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[33] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[34] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[35] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

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

[37] J. Urain, M. Ginesi, D. Tateo, and J. Peters. ImitationFlow: Learning deep stable stochastic dynamic systems by normalizing flows. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[38] B. Wingo, C.-A. Cheng, M. Murtaza, M. Zafar, and S. Hutchinson. Extending Riemmanian motion policies to a class of underactuated wheeled-inverted-pendulum robots. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020.

APPENDIX A
THE RMPFLOW ALGORITHM

Here we describe the message passing steps of RMPflow [6]. As is noted in [5], this message passing routine effectively uses the duality of (1) and the sparsity in the task map to efficiently compute $\mathbf{a}_\mathtt{r}$.

The RMPflow algorithm (Algorithm 1) is based on two components: *1*) the RMP-tree: a directed tree encoding the structure of the task map; and *2*) the RMP-algebra: a set of operations to propagate information across the RMP-tree.

In the RMP-tree, a node $\mathtt{u}$ stores the state $(\mathbf{x}_\mathtt{u}, \dot{\mathbf{x}}_\mathtt{u})$ on a manifold $\mathcal{M}_\mathtt{u}$ and the associated RMP $(\mathbf{a}_\mathtt{u}, \mathbf{M}_\mathtt{u})$. We define the *natural form* of an RMP as $[\mathbf{f}_\mathtt{u}, \mathbf{M}_\mathtt{u}]$, where $\mathbf{f}_\mathtt{u} = \mathbf{M}_\mathtt{u}\mathbf{a}_\mathtt{u}$ is the force. An edge $\mathtt{e}$ represents a smooth map $\psi_\mathtt{e}$ from the parent node manifold to a child node manifold. The root node of the RMP-tree (denoted as $\mathtt{r}$) and the leaf nodes correspond to the configuration space $\mathcal{C}$ and the subtask spaces $\{\mathcal{T}_k\}_{k=1}^K$ on which the subtask RMPs are hosted.

The RMP-algebra comprises of three operators, which are for propagating information on the RMP-tree. For illustration, we consider the node $\mathtt{u}$ on a manifold $\mathcal{M}$ with coordinates $\mathbf{x}$ and its M child nodes $\{\mathtt{v}_m\}_{m=1}^M$ with coordinates $\{\mathbf{y}_m\}_{m=1}^M$ on the RMP-tree.

(*i*) `pushforward` propagates the state of a node $(\mathbf{x}, \dot{\mathbf{x}})$ in the RMP-tree to update the states of its M child nodes $\{(\mathbf{y}_m, \dot{\mathbf{y}}_m)\}_{m=1}^M$. The state of its mth child node is computed as $(\mathbf{y}_m, \dot{\mathbf{y}}_m) = (\psi_{\mathtt{v}_m;\mathtt{u}}(\mathbf{x}), \mathbf{J}_{\mathtt{v}_m;\mathtt{u}}(\mathbf{x})\,\dot{\mathbf{x}})$, where ψ_m is the smooth map of the edge connecting the two nodes and $\mathbf{J}_{\mathtt{v}_m;\mathtt{u}} = \partial_\mathbf{x}\psi_{\mathtt{v}_m;\mathtt{u}}$ is the Jacobian matrix.

(*ii*) `pullback` propagates the RMPs of the child nodes in the natural form, $\{[\mathbf{f}_{\mathtt{v}_m}, \mathbf{M}_{\mathtt{v}_m}]\}_{m=1}^M$, to the parent node as $[\mathbf{f}_\mathtt{u}, \mathbf{M}_\mathtt{u}]$:

$$\begin{aligned} \mathbf{f}_\mathtt{u} &= \sum_{m=1}^M \mathbf{J}_{\mathtt{v}_m;\mathtt{u}}^\top (\mathbf{f}_{\mathtt{v}_m} - \mathbf{M}_{\mathtt{v}_m}\dot{\mathbf{J}}_{\mathtt{v}_m;\mathtt{u}}\dot{\mathbf{x}}), \\ \mathbf{M}_\mathtt{u} &= \sum_{m=1}^M \mathbf{J}_{\mathtt{v}_m;\mathtt{u}}^\top \mathbf{M}_{\mathtt{v}_m}\mathbf{J}_{\mathtt{v}_m;\mathtt{u}}. \end{aligned} \tag{7}$$

The natural form of RMPs are used here since they can be combined more efficiently.

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

RMPflow in Algorithm 1 computes the policy $\pi(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{a}_\mathtt{r}$ on the configuration space $\mathcal{C}$ through the following procedure. Given the state $(\mathbf{q}, \dot{\mathbf{q}})$ of the configuration space $\mathcal{C}$, the `pushforward` operator is first recursively applied to the RMP-tree to propagate the states up to the leaf nodes. Then, the subtask RMPs are evaluated on the leaf nodes and combined recursively backward along the RMP-tree by the `pullback` operator. The `resolve` operator is finally applied on the root node $\mathtt{r}$ to compute the desired acceleration $\pi(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{a}_\mathtt{r}$.

APPENDIX B
COMPLEXITY ANALYSIS

A. Complexity Analysis of RMP^2

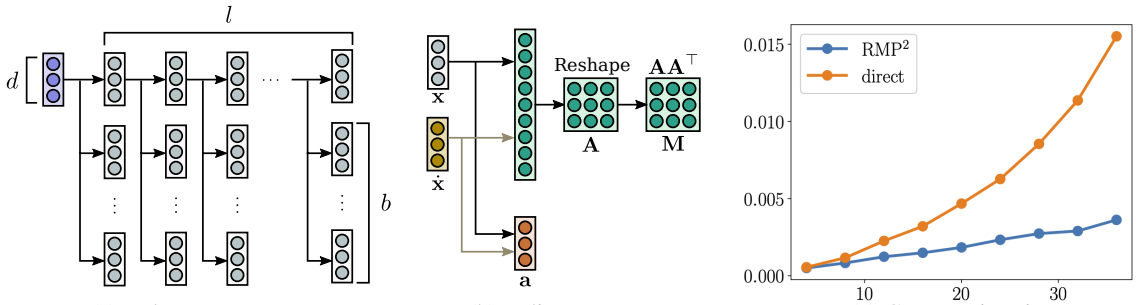
We analyze the time and space complexities of RMP^2. We show that RMP^2 has a time complexity of $O(Nbd^3)$ and a memory complexity of $O(Nd + Ld^2)$, where N is the total number of nodes, L is the number of leaf nodes, b is the maximum branching factor, and d is the maximum dimension of nodes. In comparison, we prove in Appendix B-B that the original RMPflow algorithm (Algorithm 1) by [6] has a time complexity of $O(Nbd^3)$ and a *worse* space complexity of $O(Nd^2 + Ld^2)$. Please see Table I for a summary.

Specifically, consider a directed-acyclic-graph-structured task map with N nodes, where each node has dimension in $O(d)$ and has at most b parents. We suppose that $L \leq N$ nodes are leaf nodes, and that the automatic differentiation library is based on reverse-mode automatic differentiation. We first analyze the complexity of task map evaluation and Jacobian-vector-product subroutine (Algorithm 2) based on reverse accumulation in preparation for the complexity analysis for RMP^2.

Task map evaluation: For each node in the graph, the input and output dimensions are bounded by $O(bd)$ and $O(d)$, respectively. Hence, evaluating each node has a time complexity in $O(bd^2)$. Because each node is evaluated exactly once in in computing the full task map, the total time complexity of task map evaluation is $O(Nbd^2)$. If the Gradient Oracle `gradient` will be called (as in RMP^2), the value of each node needs to be stored in preparation for the gradient computation. Overall this would require a space complexity in $O(Nd)$ to store the values in the entire graph.

Jacobian-vector product with L output nodes: Suppose that the output of the graph in Algorithm 2 is a collection of L nodes in the graph. During reverse accumulation, the task map is first computed, which, based on the previous analysis, has time and space complexity of $O(Nbd^2)$ and $O(Nd)$, respectively. The dummy variable $\boldsymbol{\lambda}$ is of size $O(Ld)$ and computing the inner product $\boldsymbol{\lambda}^\top \mathbf{v}$ requires $O(Ld)$ computation (i.e. it creats a new node of dimension 1 with $2L$ parent nodes of dimension in $O(d)$). By the reverse-mode automatic differentiation assumption, the first backward pass on the graph (line 4) has time complexity of $O(Nbd^2 + Ld) = O(Nbd^2)$ and space complexity of $O(Nd + Ld) = O(Nd)$ [13]. The final backward pass (line 5) is on a graph of size $O(N)$, as the first backward pass creates additional $O(N)$ nodes. With a similar analysis, the second backward pass have time complexity of $O(Nbd^2 + Ld) = O(Nbd^2)$ and space complexity of $O(Nd + Ld) = O(Nd)$. Therefore, the time and space complexity of Algorithm 2 is $O(Nbd^2)$ and $O(Nd)$.

Forward pass: The complexity of line 3–5 in Algorithm 3 follows the analyses above. Here the computation graph is always of size $O(N)$ (the original task map is in $O(N)$ and each call of Gradient Oracle `gradient` in the Jacobian-vector-product subroutine `jvp` creates additional $O(N)$ nodes in the computation graph). By previous analysis, the time and space complexity of the forward pass are $O(Nbd^2)$ and $O(Nd)$.



(a) task map structure (b) Policy structure (c) Computation time

Fig. 4: (a) task map and (b) policy structure for benchmarking the complexity of the algorithms. (c) The computation time of RMP^2 and the naïve implementation (direct) on chain-structured graphs with varying lengths. The results show that RMP^2 has a linear time complexity whereas the naïve implementation (direct) has a super-linearly time complexity.

Leaf evaluation: Assume, for each leaf node, $O(d^3)$ computation is needed for evaluating the importance weight matrix and $O(d^2)$ for acceleration in an RMP. The leaf evaluation step then has time complexity of $O(Ld^3)$ and space complexity of $O(Ld^2 + Ld) = O(Ld^2)$.

Backward pass: By previous analyses, task map evaluation requires $O(Nbd^2)$ computation and $O(Nd)$ space. The vector-matrix-vector product for computing auxiliary variables r and s has time complexity of $O(Ld^2)$. To compute the importance weight matrix at the root, $\mathbf{M_r}$, the first backward pass, `gradient(r,q)`, needs $O(Nbd^2)$ time and $O(Nd)$ space as it operates on a graph of size $O(N)$ where the number of parents of each node is in $O(b)$[3]. The `jacobian` operator in line 10 is done by $O(d)$ sequential calls of the Gradient Oracle `gradient`. Hence, it has a time complexity of $O(Nbd^3)$ and a space complexity of $O(Nd)$. (If we are not taking further derivatives, the values of the new graphs created in calling the `jacobian` operator do not need to be stored.) With a similar analysis, computing $\mathbf{f_r}$ requires $O(Nbd^3)$ computation and $O(Nd)$ space.

Resolve: The matrix inversion has time complexity of $O(d^3)$ and space complexity of $O(d^2)$.

In summary, the time complexity of RMP^2 is $O(Nbd^2 + Ld^3 + Nbd^3 + d^3) = O(Nbd^3)$ and the space complexity is $O(Nd + Ld^2)$.

B. Complexity of RMPflow

First we need to convert a graph with $O(b)$ parent nodes into a tree. This can be done by creating meta nodes that merge all the parents of a node into a single parent node; inside the mega node, each component is computed independently. Therefore, for a fair comparison, in the following analysis, we shall assume that in the tree version evaluating each node would need a time complexity in $O(bd^2)$. We suppose the space complexity to store all the nodes is still in $O(Nd)$ because the duplicated information resulting from the creation of the meta nodes can be handled by sharing the same memory reference in a proper implementation [6].

[3]Except the final node aggregating L outputs. However, it does not change the complexity as it adds a complexity in $O(Ld^2) < O(Nbd^2)$.

Forward pass: During the forward pass, similar to the reverse accumulation analysis, $O(bd^2)$ per node is needed for `pushforward`, i.e. computing the pushforward velocity through reverse accumulation, yielding a time complexity of $O(Nbd^2)$. The space complexity is $O(Nd)$ for storing the state at every node.

Leaf evaluation: Same as RMP^2.

Backward pass: Computing the metric in (7) requires $O(bd^3)$ computation per node, yielding time complexity of $O(Nbd^3)$ and space complexity of $O(Nd^2)$. The curvature term $\dot{\mathbf{J}}_{\mathbf{v}_m;\mathbf{u}}\dot{\mathbf{x}}$ can be computed through reverse accumulation similar to RMP^2, which has time complexity of $O(bd^2)$ per node. The matrix-vector products to compute the force requires $O(bd^2)$ space and computation for each node.

Resolve: Same as RMP^2.

Thus, the time complexity of RMPflow is $O(Nbd^2 + Ld^3 + Nbd^3 + Nbd^2 + d^3) = O(Nbd^3)$ and the space complexity is $O(Nd + Ld^2 + Nd^2 + bd^2 + d^2) = O(Nd^2 + Ld^2)$.

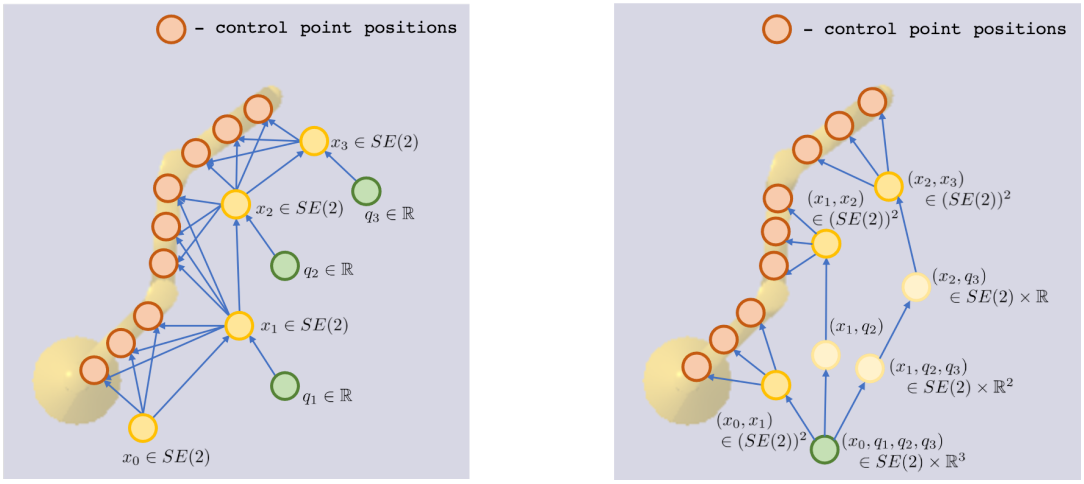
Appendix C
Experimental Validation of Time Complexity

We validate the time complexity analysis for RMP^2 and the naïve implementation (Algorithms 3 and 4, respectively). In particular, we are interested in how the two algorithms scale with respect to the number of nodes in the graph.

We consider a directed chain-like graph of length l: Each node on the chain is connected to b leaf nodes; every node in the graph is of dimension d. Overall such a graph has $1 + (b+1)l$ nodes, where bl nodes are leaf nodes. The task map structure for the time complexity experiment is shown in Fig. 4. In the experiment, we vary the length of the chain while fixing the branching factor[4] and the dimension of the nodes ($b = d = 3$). The map associated with each edge is implemented as a single-layer neural network with `tanh` activation function. Both algorithms as well as the graph structure are implemented in TensorFlow [1]. We consider chain length in $[4, 8, 12, \ldots, 36]$.

Fig. 4(c) shows the computation time of the two algorithms. The computation time of RMP^2 increases linearly with respect

[4]The branching factor and the chain length have similar effect to the size of the graph and hence only the effect of chain length is evaluated.



(a) The computational graph used by RMP^2

(b) An example RMP-tree which can be used by RMPflow

Fig. 5: (a) The computation graph automatically built by automatic differentiation libraries when computing control point positions. RMP^2 directly operates on this graph. (b) An example RMP-tree constructed for the same problem under similar strategy as [6, Appendix D]. It introduces intermediate high-dimensional nodes as well as redundant computation. In both figures, q_i is the joint angle of the i-th joint, x_i is the pose (position and orientation) of the i-th link, and x_0 is the base pose.

to the size of the graph whereas the naïve implementation (direct) suffers from a super-linear growth. The computation time reported in Fig. 4(c) is the average over 1000 runs of the algorithms with random input on a static computational graph in TensorFlow [1]. The *constant* time required to compile the static computational graph is not included in the reported average computation time.

Appendix D
RMP^2 versus RMPflow: A Case Study

In this appendix, we demonstrate how RMP^2 provides a more convenient interface for the user. Consider the planar three-link robot from the experiment section (Fig. 3 (a)). Assume that the subtask spaces are with the positions of control points along the robot arm. In Fig. 5, for example, there are 9 spaces, each corresponding to the position of one control point on the robot. This type of subtask space is useful for specifying behaviors such as collision avoidance, where we need each control point to avoid collision with obstacles in the environment.

Intuitively, to compute control point positions along the robot, we can first compute the pose of each link, $\{x_i\}_{i=1}^3$, through the kinematic chain of the robot, where $x_i \in SE(2)$ denotes the position and orientation of the i-th link. Then, control point positions can be obtained through interpolating the positions of any two adjacent links. Fig. 5(a) shows the computational graph that is automatically built through the above computation, where green nodes denote joint angles, yellow nodes denote link poses, and orange nodes denote control point positions. As is introduced in Section III, RMP^2 directly uses the computational graph as the core data structure and compute the policy through calling the Gradient Oracles provided by automatic differentiation libraries.

In contrast, RMPflow (see Appendix A) requires the user to specify a tree data structure called the RMP-tree. In the RMP-tree structure, it is required that the states in the parent node contain sufficient information for computing the child node states (see the `pushforward` operator in Appendix A). Here we use an RMP-tree structure (Fig. 5(b)) similar to what is introduced in [6, Appendix D]. The root node includes x_0, the pose of the base, as well as all joint angles $\{q_i\}_{i=1}^3$ as we need all these quantities to compute control point positions. Then, the RMP-tree branches out to compute the control points on each link. To compute the control point positions for the first link, we need the poses[5] of the base link and link 1, $(x_0, x_1) \in (SE(2))^2$, similarly for the other two links. This gives us an RMP-tree structure shown in Fig. 5(b). Note that computationally, to compute the poses of link 3, for example, we need to compute the poses for all previous links, as is shown in the light yellow nodes on the path for the second and third links. Therefore, not only does the construction of RMP-tree cost additional effort, it also introduces nodes with high dimensions (e.g. $SE(2) \times \mathbb{R}^2$) and redundant computation (the forward mapping for the first link is computed by all three branches) under less careful design choices. These, in turn, impair the computational efficiency of RMPflow as the time complexity is a function of node dimension and node number.

Moreover, RMP^2 allows us to specify complicated task maps more easily. For example, if one would like to additionally consider self-collision avoidance (even though it is not relevant to the planar three-link robot). For RMP^2, one can directly compute the distance between any two control points from different links, creating child nodes to pairs of control point position nodes. However, RMPflow requires the user to redesign the RMP-tree structure entirely, creating even more intermediate nodes due to the limitation of the tree structure.

[5]Only the positions are sufficient. However, we use poses here to make the notation more compact.