



HAL
open science

Adaptive Asynchronous Control Using Meta-Learned Neural Ordinary Differential Equations

Achkan Salehi, Steffen Rühl, Stephane Doncieux

► **To cite this version:**

Achkan Salehi, Steffen Rühl, Stephane Doncieux. Adaptive Asynchronous Control Using Meta-Learned Neural Ordinary Differential Equations. *IEEE Transactions on Robotics*, 2023, 40, pp.403-420. 10.1109/TRO.2023.3326350 . hal-04723191

HAL Id: hal-04723191

<https://hal.science/hal-04723191v1>

Submitted on 7 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Adaptive Asynchronous Control Using Meta-learned Neural Ordinary Differential Equations

Achkan Salehi[†], Steffen Rühl[‡] and Stephane Doncieux[†]

Abstract—Model-based Reinforcement Learning and Control have demonstrated great potential in various sequential decision making problem domains, including in robotics settings. However, real-world robotics systems often present challenges that limit the applicability of those methods. In particular, we note two problems that jointly happen in many industrial systems: 1) Irregular/asynchronous observations and actions and 2) Dramatic changes in environment dynamics from an episode to another (e.g. varying payload inertial properties). We propose a general framework that overcomes those difficulties by meta-learning adaptive dynamics models for continuous-time prediction and control. The proposed approach is task-agnostic and can be adapted to new tasks in a straight-forward manner. We present evaluations in two different robot simulations and on a real industrial robot.

Index Terms—Model Learning for Control, Robust/Adaptive Control of Robotic Systems, Learning and Adaptive Systems, Industrial Robots

I. INTRODUCTION

Machine Learning methods have increasingly been studied and applied for decision making and control in robotic systems during the past few years. Model-free Reinforcement Learning (RL) methods [1], [2], [3] have shown great success in various robotics settings, and model-based Reinforcement Learning and Planning [4], [5] have garnered considerable attention due to their increased data-efficiency. More recently, population based methods such as model-free and model-based Quality-Diversity (QD) algorithms [6], [7] have also shown promise in contexts where behavioral diversity is essential.

However, the vast majority of those methods have been applied in contexts where the difficulties and constraints present in real-world systems are partially relaxed. In particular:

- 1) In a large number of systems, observations and actions are irregular/asynchronous. This contrasts with the assumptions made by the majority of works in RL/QD, in which actions are applied at a given state and observations are received at regular intervals.
- 2) The dynamics of the environments in which real-world systems operate are often subject to dramatic discontinuous changes between episodes. This can be for example because of the change in the geometry or mass distribution of a payload, or an unexpected malfunction of some component.

An example in which those difficulties can be encountered is the industrial SOTO2 robot (figure 1), which is manufactured by Magazino GmbH¹ and deployed in production or assembly lines in order to autonomously manipulate and transport payloads of varying dimensions and mass distributions. In particular, its gripper, which leverages two independent conveyor belts (figure 1 top right and bottom) receives commands irregularly. Those commands are not

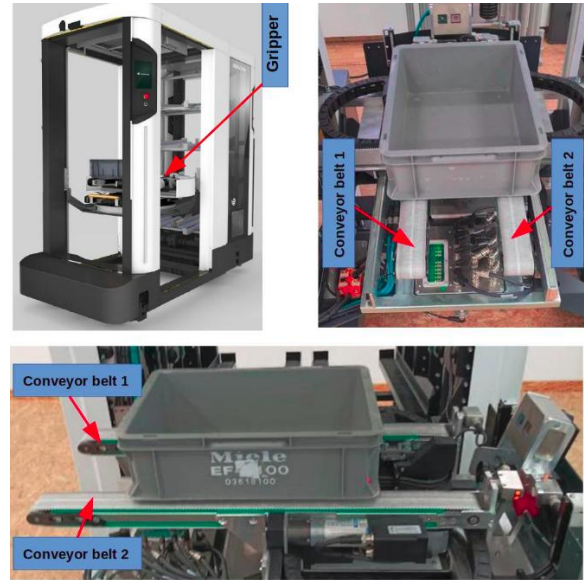


Fig. 1: (**Top left**) The SOTO2 robot manufactured by Magazino GmbH, which autonomously navigates in storage facilities such as warehouses and uses a gripper essentially composed of two independently controlled conveyor belts to manipulate boxes. (**Top right and bottom**) Different views of the gripper, the control of which requires addressing the two problems mentioned in the introduction: first, commands and observations are sent and received in an irregular/asynchronous manner. Second, the distributions of mass of the boxes that are manipulated can dramatically vary from an episode to the next.

synchronized with the —potentially also irregular— observations that are provided by its different sensors. Furthermore, a controller aiming to achieve non-trivial goals using those conveyor belts should be able to adapt to the inertial properties of previously unseen payloads.

In addition to adapting to changes in environment dynamics, a desirable property for autonomous robots is the ability to adapt to new, potentially unseen tasks. For example, given the same payload, the SOTO2 might be tasked to rotate it by some desired angle or instead displace it according to a succession of affine transformations or splines with constraints on velocity/accelerations. Similarly, a mobile robot tasked with navigating to specific goals might in the future be required to explore/search its environment instead. Our aim in this paper is to propose a solution that jointly handles the problems of asynchronous/irregular inputs and changes in environment dynamics, while retaining flexibility with respect to downstream tasks.

Current works in RL/QD and Model-Predictive Control (MPC) in general do not jointly address all of the aforementioned aspects. For example, many RL/QD systems that incorporate meta-learning [8], [2], [9], [10], [11], [12] or system identification [3] are able to adapt to unseen environment and/or tasks with minimal adaptation. However,

[†] Achkan Salehi and Stephane Doncieux are with the ISIR, Sorbonne University, CNRS, Paris, France (email: achkan.salehi@sorbonne-universite.fr, stephane.doncieux@sorbonne-universite.fr).

[‡] Steffen Rühl is with the perception team at Magazino GmbH (email: ruehl@magazino.eu).

¹<https://www.magazino.eu>

they do not address the problem of irregular actions and observations. On the other hand, RL methods that are designed to handle those issues and/or operate in continuous-time scenarios [13], [14] are applied to environments that do not change significantly between episodes. Similarly, the incorporation of Neural Ordinary Differential Equations (N-ODEs) by a growing body of MPC-based methods [15], [16], [17] makes them able to process irregular/asynchronous inputs in a natural manner. However, they often focus on accuracy in a single environment (e.g. [15]), and recent efforts towards adaptive MPC with N-ODEs [16], [17] either adapt to novel environments while gradually forgetting previous ones [16] or learn controllers and models that are specific to trajectory-tracking objectives [17] which results in reduced flexibility in terms of adaptation to novel tasks, such as exploration.

In this paper, we propose a general framework, dubbed **ACUMEN**, for Adaptive Control in AsynchronoUs probleM sEttiNgs, which is capable of handling irregular/asynchronous observations and actions and changes in environment dynamics in a *task-agnostic* manner. It is based on incorporating meta-learned Neural Ordinary Differential Equations (N-ODEs) [18] that model environment dynamics into a sampling-based MPC pipeline.

We perform experiments in two different robot simulations as well as on the real SOTO2 robot. The first simulation, which we have developed for this work, is a light-weight pybullet [19] environment that provides a simplified model of the gripper from the SOTO2 robot. The second set of experiments is based on the Gazebo Turtlebot3 simulation [20] which relies on ROS² for asynchronous communications with the controller. Finally, our experiments on the real SOTO2 robot demonstrate the feasibility of using the proposed neural-ODE based model predictive approach in real-world robotic systems. However, those live experiments do not include meta-learning due to limited access to the robot. Instead, meta-learning results are provided based on off-line data from robots that are deployed at various customer sites.

The paper is structured as follows. The following section (§II) is dedicated to problem formalization and notations. We position our work with respect to the literature in §III and describe the proposed method in §IV. Experimental results in simulated environments and on the real SOTO2 robot are respectively presented in §V and §VI. Some advantages of the proposed system, its limitations and future directions are discussed in section §VII. Closing remarks follow in §VIII.

II. PROBLEM FORMULATION AND NOTATIONS

We consider robotic systems with irregular/asynchronous actions and observations that operate in episodic fashion, and where the unknown dynamic system governing state transitions is sampled from an unknown distribution for each new episode. A known reward function, defining the task, can also be sampled for each episode. We formalize those problems below.

Irregular/asynchronous actions and observations. In most of the RL literature, actions and observations are regular and synchronized: one performs an action a_t in some state z_t , and receives an observation ω_{t+1} . In contrast, we are interested in the general case in which observations and actions are given as $\omega_{t_1}, \dots, \omega_{t_k}$ and $a_{t'_1}, \dots, a_{t'_l}$ with $l \neq k$, $t'_i \neq t_i$. Typically, there might be many actions between two observations, and vice-versa.

We frame such problems as Continuous Time, Partially Observable Markov Decision Processes (CTPOMDP) in which the observation function is time-dependent. More formally, we consider the tuple $\mathcal{T} \triangleq \langle Z, \mathcal{A}, \mathcal{F}, R, \Omega, \Phi, \gamma, T \rangle$ where $Z \subset \mathbb{R}^N$ and $\mathcal{A} \subset \mathbb{R}^M$ respectively denote the state and action spaces. Let us denote $u : \mathbb{R} \rightarrow \mathcal{A}$ the function defining the actions to be applied at time t . Then \mathcal{F} defines the evolution of the dynamic system:

$$z(t_0 + \Delta t) = z(t_0) + \int_{t_0}^{t_0 + \Delta t} \mathcal{F}(z(t), t, u(t)) dt. \quad (1)$$

The possible time intervals over which the system operates are noted T and the function $\Phi : Z \times T \times \Omega \rightarrow [0, 1]$ defines the probability $p(\omega|z, t)$ of an element ω of the observation space Ω given a state-time pair. In what follows, a_t will denote an action applied at time t . Likewise, ω_t will indicate an observation made at time t . The function $R : Z \times \mathcal{A} \rightarrow \mathbb{R}$ is a dense reward function and $\gamma \in [0, 1]$ is a discount factor. During an episode associated to the tuple \mathcal{T} , our aim is to find $u(t)$ that maximizes the discounted cumulative reward $\int_0^\infty \gamma R(z(t), u(t)) dt$.

Changes in environment dynamics and tasks. We consider the case in which environment dynamics remain stationary through the duration of an episode, but dramatically and discontinuously change in between different episodes. More formally, each environment \mathcal{T} is sampled from a distribution of the form

$$P(\mathcal{T}) = P(\langle Z, \mathcal{A}, \mathcal{F}, R, \Omega, \Phi, \gamma, T \rangle) \triangleq P(\mathcal{F}, R). \quad (2)$$

In other words, two episodes associated to two distinct environments $\mathcal{T}_i, \mathcal{T}_j$ only differ in \mathcal{F}_i, R_i and \mathcal{F}_j, R_j , while the rest remains unchanged. Our objective is to find an adaptive control process that leverages previous learning experience to adapt to new, unseen environments from $P(\mathcal{T})$, in a manner that is robust to changes in R .

III. RELATED WORK

The proposed solution is made of: 1) a sampling based MPC module 2) A dynamics modelling component and 3) a meta-learning framework. In this section, we first discuss positioning w.r.t each of those building blocks separately (§III-A, III-B, III-C). Then, in section §III-D, we discuss related solutions from the RL/Control literatures that combine similar modules to solve problems related to adaptivity and irregularity in the model-based setting.

A. Model-based Reinforcement Learning and Control.

Model-based RL (MBRL) [22] with learned models has been demonstrated to be superior in terms of data-efficiency compared to model-free RL (MFRL), particularly in low-data regimes [23], [24], [4]. This is particularly important on robotic systems, where the large number of interactions required by model-free methods are often impractical. While it has been observed that MBRL can suffer from lower asymptotic performance compared to MFRL [24], [23], recent works [25] demonstrate that this performance gap can be closed by accounting for model uncertainty, which results in better exploration.

Closely related to MBRL are Model Predictive Control (MPC) methods, which consist in finding a sequence of controls that optimize a trajectory cost function, given a dynamics model traditionally derived from first principles. Considering such models as prior knowledge, some modern methods leverage machine learning techniques to construct more complex environment models that account for uncertainty [15] or inaccuracies [26], while others learn the dynamics model from scratch [27]. We note that the majority of MPC methods minimize a cost function that depends on a target state-space trajectory. However, not all tasks can be specified by a desired path in a straight-forward manner, and it can be preferable—from both feasibility and generalizability perspectives—to learn how to infer such trajectories without explicit planning. This is for example the case for tasks that require exploration or precise interactions with the environment. In such cases, the flexibility of RL methods makes them better candidates.

A popular approach that lies at the frontier between RL and MPC is to sample action-state trajectories from a learned model, and score each of them according to a reward that is (partially) dependent on

²The Robot Operating System[21].

the predicted states. The N first actions from the best trajectory are then applied on the real system. An example of such an action selection scheme is the Cross Entropy Method (CEM) [28], [5]. As a population-based algorithm, the model-based CEM is robust to model inaccuracies, and has been shown to produce results that are on-par with MFRL [25], [29]. Furthermore, CEM, unlike most of the work based on learned policies, is not tied to a particular reward function: indeed, the reward function can be changed at any time during or in-between episodes, without any need for additional training. While vanilla CEM suffers from poor data-efficiency in high-dimensional spaces, recent efforts [5] demonstrate that this aspect can be significantly improved, in particular via sampling time-correlated action sequences. We use CEM-based planning in some of our experiments, and in others, further simplify action selection by sampling from a discrete set of velocity controls.

B. Irregular/asynchronous actions and observations.

The vast majority of the RL literature considers discretized time-steps with regular observations and actions, and most prior work on continuous time systems make simplifying assumptions such as partially known and/or linearized dynamics systems [30], [31], [32], or are applied in model-free settings [33], [34]. However, more recent efforts build on Neural Ordinary Differential Equations (N-ODEs) which outperform discrete approaches to modeling continuous-time dynamics [18]. In their work, Du et al. [14] model environment dynamics using N-ODEs in order to learn policies in semi-MDP problem settings, while the continuous-time RL framework of Yildiz et al. [13] is notable for providing state uncertainty estimates. Although our use of N-ODEs is similar to what was done in those works, it should be noted that our focus is on an orthogonal direction: our aim is to investigate the combination of meta-learning and N-ODEs in order to simultaneously address the problem of irregular/asynchronous actions and observations and discontinuous changes in system dynamics on a per episode basis. This distinction is reflected in our experimental setup, which is closer to real-world applications. We also note that several recent works from the control literature have also incorporated neural ODEs [15], [16], [17] which equips them for dealing with irregular inputs³. The first two of these methods aim at increasing accuracy in a single environment, and while the two others [16], [17] do target adaptivity to different environments, they both —as we will discuss in more detail in section §III-D—are limited to specific types of tasks and trade-off flexibility for accuracy.

C. Meta-learning.

Leveraging previous experience to improve the learning process has been extensively studied in the meta-learning [35], [36] literature. As even when considering the most restrictive definitions of meta-learning in which identical train/test conditions, end-to-end optimization and sample splitting are essential [35], one is left with a plethora of methods that differ, among other things, in what meta-parameters they optimize. For example, many meta algorithms learn a recurrent network that models policies [37] or optimizers/update rules [38], while others optimize hyperparameters [39] or metrics [40]. It is thus important to note that the meta-parameter that we seek to optimize is a prior dynamics model (more precisely, a neural ODE), which when used as an initialization in novel control tasks, should be able to quickly adapt to represent the dynamics of that environment. Learning adaptive priors is precisely what algorithms in the MAML family [8], [41] are designed for. In particular, we update our parameters in a manner similar to ES-MAML [41], as it alleviates the need for the estimation of second order.

³We note in passing that this ability is more often than not a by-product of their methodology, as their focus is essentially on accurate dynamics modeling rather than on integrating data from different sources that is produced at different rates (e.g. from asynchronous publication to various ROS topics), which is our principal motivation.

D. Adaptive MB-RL / MPC

Controlling an agent in a manner that is adaptive to different tasks and/or environment dynamics has been the subject of several works in the RL literature. For example, the work of Sekar *et al.* builds a global world model that is used in order to adapt to unseen downstream tasks. Closer to our work is the method proposed by Nagabandi *et al.* [9], which uses MAML-based meta-learning to adapt the model to novel environment dynamics in a manner that is not dependent on a particular reward function. Other works have extended PETS[25] with a meta-learning component [12]. More recently, transformer-based in context-meta learning in the style of RL² [37] has been coupled with model-based RL in order to train an RL agent at scale (akin to an early foundation model for RL). While many of those approaches do indeed result in remarkable adaptivity, they all assume the classical RL settings in which actions and observations are synchronized.

On the other hand, some recent MPC methods that aim to learn an adaptive control process [16], [17] leverage neural ODEs to model the underlying dynamic systems. While they are therefore naturally equipped to handle irregular inputs, they are less flexible than their RL-based counterparts. In particular, the model learned in the work of Jiahao *et al.*[16] results from adapting a weighted average over a history of learned models, which results in gradual forgetting of previous environments. While the method proposed by Richards *et al.*[17] results in a more versatile controller, it is supervised by a trajectory-tracking meta-objective, which reduces its flexibility in terms of adaptation to other tasks, such as exploration. Furthermore, online adaptation in their work consists in forward simulation using meta-parameters and approximate dynamics that are all learned offline. While this is an efficient approach, its adaptivity is limited by the trajectory-environment space that is covered by the offline dataset.

IV. ACUMEN

We propose an algorithm based on Model Predictive Control (MPC) [5], [22] with a learned and dynamically adapted environment model based on the formalism of Neural Ordinary Differential Equations (N-ODE) [18]. In order to ensure performance on unseen tasks, learning experience from previous episodes is used to maintain a prior on the weights of the N-ODE with which each new episode is initialized. Each of the following subsections is dedicated to one of those components. A high-level overview of our approach is given in figure 2, and pseudo-code is given in algorithm 1 .

A. Handling irregular/asynchronous observations and actions

As the state in the problem settings that we discussed in section §II is partially observable, we assume the existence of a learned or hand-designed function $\xi_n : \omega_0 \times \omega_1 \times \dots \times \omega_n \rightarrow Z$ that maps a sliding window of observations $W = \{\omega\}_{i=1}^n$ to a state approximation, *i.e.* $\xi(W) = \hat{z} \in Z$. When clear from context, we will drop the window size from the notation and simply write ξ . We then use a neural ODE $\hat{\mathcal{F}}$, parametrized by weights θ to approximate the dynamics evolution function \mathcal{F} :

$$\hat{z}(t_0 + \Delta t) \approx \hat{z}(t_0) + \int_{t_0}^{t_0 + \Delta t} \hat{\mathcal{F}}(\hat{z}(t), t, u(t), \theta) dt. \quad (3)$$

As we are interested in irregular actions, the function $u(\cdot)$ will in effect be the composition of two functions: a decision process $\pi(t_0 - s, t_0 + \Delta t)$ that outputs irregular actions at discrete timestamp falling in $[t_0 - s, t_0 + \Delta t]$, and an interpolation function that interpolates those actions to produce action values at continuous timestamps⁴. More formally, u can be written as

⁴In our work, π is not a trained policy, but is instead sampled from a distribution P_ψ . To keep notations concise, we have omitted the dependency of π on that distribution in the notation $\pi(t_0 - s, t_0 + \Delta t)$.

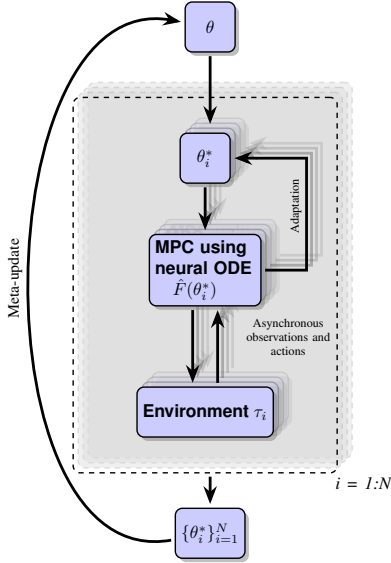


Fig. 2: A high-level view of the proposed algorithm. At each meta-iteration, N different environments τ_1, \dots, τ_N are sampled and each τ_i is paired with a neural ODE (noted $\hat{F}(\theta_i^*)$ in the figure). Episode i then consists in a sampling-based model-predictive pipeline in environment τ_i (see sections IV-B and algorithm 2). During such an episode, $\hat{F}(\theta_i^*)$ is used as an approximate world model for state predictions, and θ_i^* is continuously refined in order to improve the predictions of $\hat{F}(\theta_i^*)$. After all N episodes have been completed (either in parallel or sequentially), the optimized $\{\theta_i^*\}_{i=1}^N$ are used to estimate the gradient for the meta-update (equations 5, 6 in the main text, and line 30 in algorithm 1).

$$u(t) = (\mathcal{I} \circ \pi(t_0 - s, t_0 + \Delta t))(t) \quad \forall t \in [t_0 - s, t_0 + \Delta t] \quad (4)$$

for some (small) real value s . In our work, the interpolation \mathcal{I} is the linear interpolation function.

In our action-sampling based system (algorithm 2), N_p different action sequences, each of length H , are sampled and considered as possible continuations of the current history of actions. Each of these action sequences can be thought of as a decision process π_i (with $i \in \{1, \dots, N_p\}$). Since the interpolation function \mathcal{I} is fixed, it is equivalent to say that we sample N_p functions $u_i \triangleq \mathcal{I} \circ \pi_i$.

B. Planning with a continuously updated model

The proposed method is based on Model Predictive Control (MPC) with a learned model, that is updated throughout the duration of an episode. At each iteration of the algorithm (lines 11-28 in algorithm 1), a sliding window consisting of the last M observations $\{\omega_1, \dots, \omega_M\}$ is mapped to a state estimation $\hat{z}(t)$. Actions $\{c_1, \dots, c_l\}$ (in general, $l \neq M$) that have been applied since the time at which ω_1 was observed are gathered. All observations and actions are added to a buffer with the aim of updating the model.

We determine an action sequence $a_{1:H}^*$, that when appended to $\{c_1, \dots, c_l\}$, would result in the most promising future state according to the predictions of the learned model. The sampling and selection process is detailed in algorithm 2. As previously discussed in detail in section III, our decision to rely on command sampling is not only motivated by its demonstrated efficacy and robustness, but also by its simplicity.

Once a sequence $a_{1:H}^*$ has been selected, its first action is applied on the system. The learned model is updated every $train_freq$ iterations. Note that the loss function for training the neural ODE and the dense reward function $R(\cdot)$ are both application dependent. See appendices A and B for details about those used in our experiments.

Algorithm 1: The ACUMEN algorithm.

Input: Environment distribution $\mathcal{P}(\tau)$, number of environments to sample for each meta update N , train/validation split ratio $r_{split} \in (0, 1)$, maximum number of iterations for inner loop (neural ODE) optimizations N_{it} , optionally prior parameters of the N-ODE θ_0 , hyperparameters $action_selection_hyperparams$ (see algorithm 2), function ξ mapping sliding windows of observations to state approximations, size of sliding window of observations M , learning rate for the meta update α , standard deviation σ for the estimation of the meta update

Output: prior parameters to use for future sessions θ

```

1   $\theta \leftarrow \text{RandomInit}()$  if  $\theta_0$  is not given; else  $\theta \leftarrow \theta_0$ 
2  while session no over do
3    // sample i.i.d vectors for the ES update of the meta-parameters
4     $\epsilon_1, \dots, \epsilon_N \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
5    for  $i \in \{1, \dots, N\}$  do
6       $\tau_i \sim \mathcal{P}(\tau)$ 
7       $\mathcal{D}_i^{train} \leftarrow \emptyset$ 
8       $\mathcal{D}_i^{val} \leftarrow \emptyset$ 
9       $\theta_i^* \leftarrow \theta + \epsilon_i \sigma$ 
10      $C_{hist} \leftarrow \emptyset$  // history of applied actions
11     while not ( $\tau_i.success$  or  $\tau_i.timeout$ ) do
12       // get the  $M$  most recent observations
13        $\omega_1, \dots, \omega_M \leftarrow \tau_i.GetSlidingWindow(M)$ 
14       // get all actions sent since the oldest observation in the window
15        $c_1, \dots, c_l \leftarrow \text{GetAllActionsSince}(\omega_1.timestamp, C_{hist})$ 
16       // Map the sliding window to a state approximation
17        $\hat{z} \leftarrow \xi(\omega_1, \dots, \omega_M)$ 
18       // Choose the next action sequence (see algorithm 2 for details)
19        $seq \leftarrow \text{SelectNewCommand}(\theta_i^*, \hat{z}, \{c_1, \dots, c_l\}, action\_selection\_hyperparams)$ 
20       // Apply the  $K$  first actions in the sequence
21        $\tau_i.Apply(seq_{1:K})$ 
22       // Add the applied actions to command history
23        $C_{hist}.Append(seq_{1:K})$ 
24       // split gathered data into train/validation sets
25        $d_{train}, d_{val} \leftarrow \text{SampleSplit}([\omega_1, \dots, \omega_M], [c_1, \dots, c_l], r_{split})$ 
26        $\mathcal{D}_i^{train}.Append(d_{train})$ 
27        $\mathcal{D}_i^{val}.Append(d_{val})$ 
28       // optimize the prior if necessary (see appendix A for details on OptimizeNODE)
29       if  $train\_freq$  then
30          $\theta_i^* \leftarrow \text{OptimizeNODE}(\theta_i^*, \mathcal{D}_i^{train}, N_{it})$ 
31       end
32     end
33   end
34   // update the meta-parameters  $\theta$ 
35    $\theta \leftarrow \theta - \frac{\alpha}{N\sigma} \sum_{i=1}^N \mathcal{L}_i(\theta_i^*, \mathcal{D}_i^{val}) \epsilon_i$ 
36 end

```

C. Learning a neural ODE prior

Our objective of obtaining a point-wise estimate of weights that could serve as an adaptive prior in environments with previously unseen dynamics can be formulated via the following equation:

$$\theta \leftarrow \theta - \alpha \nabla \sum_{i=1}^N \mathcal{L}(\mathcal{T}_{val}^i, \theta - \lambda \nabla \mathcal{L}(\mathcal{T}_{train}^i, \theta)) \quad (5)$$

in which $\mathcal{L}(\cdot, \theta)$ denotes the training loss of a neural ODE on some dataset. As is classically done in the meta-learning literature for the sake of conciseness in notations, each of the inner and outer

Algorithm 2: An action selection step using the learned neural ODE. While predicted actions $a_{1:H}^i$ are regularly sampled, their application on the system will be irregular. Thus, the history of actions c_1, \dots, c_n will be composed of irregularly applied actions, hence the advantage of using the neural-ODE formalism. Note that while the action sampling process outlined here includes CEM-based sampling with time-correlated actions (as in [5]) as a special case, it can also be reduced to a simple random shooting depending on the choice of $P_\psi(A)$ and other hyperparameters. We will adjust those on a per application basis in our experiments. See sections V, VI and appendix B for more details.

Input: Neural ODE weights θ , state approximation $\hat{z}(t_0)$ at time t_0 , desired duration of state propagation Δt , previous actions c_1, \dots, c_l (augmented with their timestamp info), number of action sequences N_p to sample, number of elite action sequences N_e , length of action sequence to sample H , reward function $R(\cdot)$, prior distribution on actions $P_\psi(A)$ (parametrized by ψ), initial parameters ψ_0 for $P_\psi(A)$, convergence criterion \mathcal{Y}

Output: action sequence $a_{1:H}^*$

```

1 Function SelectNewCommand( $\theta, \hat{z}(t_0), \Delta t, \{c_1, \dots, c_l\}, N_p,$ 
   $N_e, H, R(\cdot), \mathcal{Y}$ ):
2    $\psi \leftarrow \psi_0$ 
  // initialize elite set
3   elites  $\leftarrow \emptyset$ 
4   while not  $\mathcal{Y}$  do
  // sample  $N_p$  action sequences of length  $H$  at regular
  // intervals in  $[t_0, t_0 + \Delta t]$ 
5    $a_{1:H}^1, \dots, a_{1:H}^{N_p} \sim P_\psi(A)$ 
6   for each  $a_{1:H}^i$  do
7      $\pi_i \leftarrow [c_1, \dots, c_l].\text{concatenate}(a_{1:H}^i)$ 
  // Let  $u_i(t)$  the function that computes actions via
  // interpolating elements of  $\pi_i$ 
8      $u_i \leftarrow \mathcal{I} \circ \pi_i$ 
  // propagate the state
9      $\hat{z}_i \leftarrow \hat{z}(t_0) + \int_{t_0}^{t_0 + \Delta t} \hat{\mathcal{F}}(\hat{z}(t), u_i(t), \theta) dt.$ 
  // compute associated reward
10     $r_i \leftarrow R(\hat{z}_i)$ 
11  end
12  elites  $\leftarrow$  select best  $N_e$  action sequences
  according to the  $r_i$ 
  // Update prior distribution  $P_\psi(A)$  on commands
13   $\psi \leftarrow \text{updatePriorDistrib}(a_{1:H}^1, \dots, a_{1:H}^{N_p}, r_1, \dots, r_{N_p})$ 
14 end
  // return the best action sequence (alternatively, re-sample from
  //  $P_\psi(A)$ )
15 return elites

```

optimization problems have been written as a single gradient descent update. However, in practice the number of updates is arbitrary.

Notice that the inner optimizations, *i.e.* $\theta - \lambda \nabla \mathcal{L}(\mathcal{T}_{train}^i, \theta)$, correspond to model-predictive episodes (lines 11-28 in algorithm 1).

Computing the gradient update for the outer level optimization problem requires differentiating through the gradient computed in the inner optimization. As this would necessitate computing higher order derivatives of neural ODEs, we simplify the outer level update by using an estimator based on Evolution Strategies [42]:

$$\theta \leftarrow \theta - \frac{1}{\sigma} \mathbb{E}_{\tau \sim P(\tau), \epsilon \sim \mathcal{N}(0, I)} [\mathcal{L}(\mathcal{T}_{val}^\tau, \theta + \epsilon - \lambda \nabla \mathcal{L}(\mathcal{T}_{train}^\tau, \theta + \epsilon))] \quad (6)$$

Note that this update, approximated using N sampled environments in algorithm 1 (line 30) is equivalent to the zero-order ES-MAML update [41].

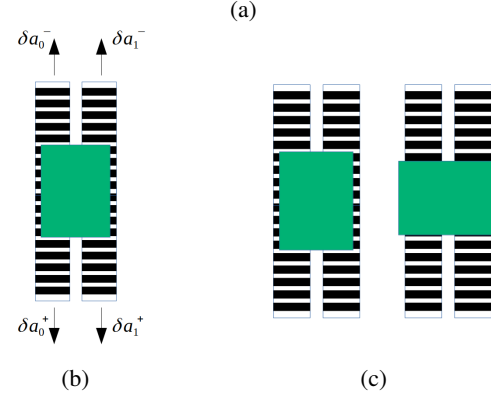
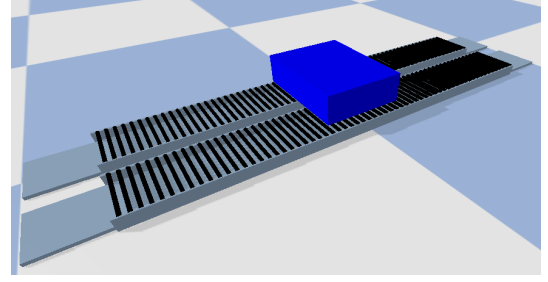


Fig. 3: **(a)** Screenshot of the developed robot simulation, based on simulating two roller conveyors. **(b)** Schematic view of the system, showing a green box on top of two conveyor belts. The conveyor belts are independent, and the motion of each one is controlled via a one-dimensional velocity command in $[-1, 1]$. Positive and negative actions, *e.g.* δa_0^+ , δa_0^- respectively move the left conveyor in the top or bottom direction. Commands such as δa_1^+ , δa_1^- should affect the right-hand conveyor belt in similar fashion. **(c, left)** Illustration of the initial conditions in each episode. **(c, right)** Target pose that we aim to reach through controlling the conveyor belts. Note that the mass and inertial properties significantly and discontinuously vary in-between episodes.

V. EXPERIMENTAL VALIDATION IN SIMULATED ENVIRONMENTS

We first present experiments on two simulated robots. The first one is inspired by the SOTO2 robot manufactured by Magazino GmbH, and uses two conveyor belts to manipulate boxes of different inertial properties. As the simulation is developed in pybullet [19], command and observation irregularities are simulated by randomly dropping observations and by applying interpolated commands at random timestamps. The second experiment is based on the Gazebo Turtlebot3 simulation[20]. As communicating with the Gazebo simulator is done via ROS topics and services, the observations and actions already suffer from some irregularity that we further accentuate by dropping random observations in order to better highlight the advantages of using neural ODEs. For both robots, the physical properties of the environment are randomly sampled at the beginning of an episode.

In addition to evaluating the ability of the ACUMEN algorithm to jointly address asynchronous/irregular actions and observations as well as discontinuous changes in environment dynamics, we also highlight the effect of its different components through ablation studies. In particular, we highlight the advantages of using neural-ODEs. Classical model-based RL approaches usually learn a model of the form $\Delta z_{t+1} = M(z_t, a_t)$, which corresponds to (PO)MDPs. A simple extension to that model for the case of irregular/asynchronous actions can be obtained by considering a recurrent neural network which would take as input the sequence of tuples $(a_0, \Delta t_0), \dots, (a_n, \Delta t_n)$ with Δt_i the elapsed time between a_i and a_{i+1} for $i \neq n$, and Δt_n the elapsed time between a_n and

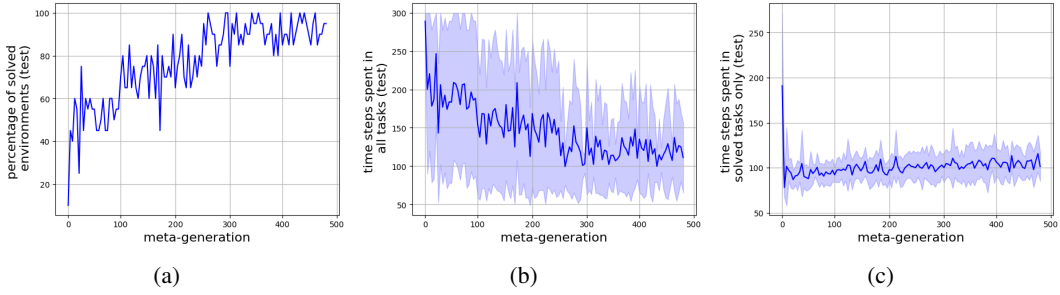


Fig. 4: **(a)** The evolution of the percentage of solved environments among the $M_{test} = 20$ ones that are sampled from the test distribution $P(\mathcal{T}_{box})$ at each meta-generation, starting from a random neural ODE. **(b)** In this figure, the value displayed at each meta-generation is the time spent in all environments (solved and unsolved) that have been sampled at this meta-iteration. **(c)** Same as in **(b)**, but for *solved* environments only. Notice the large drop in the number of necessary timesteps in the ~ 10 first meta-generations.

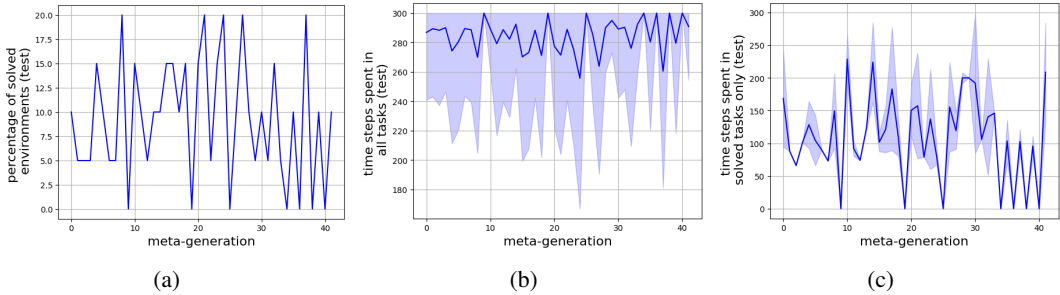


Fig. 5: Result of meta-update ablation. **(a)** It can be seen that a random neural ODE can be adapted to solve an average of $\sim 9\%$ of the sampled environments in the given time limit of $H_{max} = 300$. **(b)** Naturally, without the meta-update, there is no reduction in the average time taken to solve the environments. **(c)** The average and the variance of time-steps spent *solved* environments only. Note that *values of zero indicate that none of the environments were solved*. The remaining values are in general larger than those that were reached at convergence with the meta-update enabled (figure 4(c)).

the next observation (or during planning, the next predicted state). In this paper, we implement this model as a vanilla stacked RNN. Note that this is a slight generalization of state propagation of the form $z_t = z_{t-1} + M(z_{t-1}, a_{t-1}, \Delta t_{t-1})$, and that the latter has been shown in prior work [13] to be inferior to neural ODEs in continuous time settings. In the rest of the paper, we will use the terms RNN and stacked RNN and recurrent model interchangeably when the context allows it.

Details regarding loss and reward functions as well as various hyper-parameters can be found in the appendixes.

A. Simulated box rotation using conveyor belts

This robotic simulation is inspired by the SOTO2 robot manufactured by Magazino GmbH. Before presenting our results, we detail the simulation as well as some algorithmic choices in the following section. Results from the complete system are presented in section V-A2, and ablation studies follow in sections V-A3.

1) *Simulated Box rotation: experiment description:* The simulation, developed using the bullet physics engine [19], defines the following problem: given two parallel and independently controlled conveyor belts separated by some distance (figure 3(a)), the aim is to rotate a parcel by $\frac{\pi}{2}$ radians. The default problem setting is illustrated in figure 3(c). At the beginning of an episode, a box of varying inertial properties is sampled from a distribution $P(\mathcal{T}_{box})$, and is positioned (plus or minus some gaussian noise) at the pose shown in figure 3(c, left). The task is considered solved if the system reached (with some tolerance) the state given in the right hand of figure 3(c). The environment, shown in 3(b), is based on simulated roller conveyors [43].

The box distribution $P(\mathcal{T}_{box})$ is defined by a random choice of mass in the interval $[0.1kg, 3kg]$ and a mass distribution given by random $3d$ Gaussian distributions, which can have dense covariances. To

simplify matters, we use ground truth $6d$ box poses as observations. Given the last two observed box poses ω_{t-1}, ω_t , we approximate the state to propagate as $\hat{z} = \xi(\omega_t, \omega_{t-1}) \triangleq [\omega_t, \frac{\omega_t - \omega_{t-1}}{\Delta t}]$. Note that the action space is two-dimensional and defined by $[-1, 1] \times [-1, 1]$, as each of the conveyors can receive velocity controls independently of the other one. As specified in appendix B-A, the distance from a given pose and the target pose is used as a dense reward signal to guide the action selection process. The action sampling mechanism that is used is a particular instantiation of algorithm 2 which is equivalent to the CEM-based control method with colored noise from [5]. Details are given in appendix B.

For benchmarking purposes, we define each timestep as the time elapsed between applying actions in the physics engine, which is done at regular intervals. We however emphasize that those actions are not used by our control algorithm, which only receives actions that the result of linear interpolations at random time-steps. The maximum number of timesteps for each episode is set to $H_{max} = 300$.

In all experiments, the RK45 solver [44] (commonly referred to as `dopri5`) is used.

Irregular/asynchronous observations and actions. We simulate irregular/asynchronous observations and actions by 1- Returning only one random observation among K_s successive observations, 2- returning actions at randomly interpolated time-stamps in between real actions and 3- Discarding selected observations with some probability η . In the following experiments, a value of $K_s = 3$ has been chosen, and $\eta = 0.05$. The colored noise parameter used in CEM-based sampling (algorithm 3) was set to $\beta = 2$.

2) *Simulated box rotation: system results:* The objective of this subsection is to validate the complete system in the presented simulation, where irregular/asynchronous actions and observations have been simulated, and where environment dynamics vary from one episode to the next due to the random sampling of box mass and inertial properties.

A neural ODE is initialized with random weights, and receives successive meta updates in order to form the learned prior. In each meta iteration, $N = 20$ environments τ_1, \dots, τ_N are sampled from the environment distribution. As detailed in algorithm 1, the neural ODE is independently adapted to each of the τ_i using data gathered during the episode, the validation split of which is then used in the meta update.

The result of the experiment are plotted in figure 4. As figure 4(a) shows, at initialization, model-predictive control with the random neural ODE is only able to solve about 5% of the environments that are sampled from the test distribution. In this first meta iteration, the variance of episode lengths (figure 4(b)) is large, with most episode requiring many updates to the neural ODE. Hence, the large mean episode length reported in that same figure.

It can be seen that as the learned prior is optimized during successive meta-iteration, the percentage of successfully solved test environments increases until eventually reaching a point where it oscillates between 95 – 100% (figure 4(a)). Simultaneously, the number of steps spent across all environments (solved and unsolved) decreases as the optimization of the prior progresses (figure 4(b)).

Figure 4(c) shows the number of timesteps spent in environments that the system was able to solve at a given meta iteration. After an initial large drop in mean and variance in the first few (< 10) meta iterations, the number of necessary steps for solving an environment stabilizes to around 100 timesteps, while the number of success (figure 4(a)) continues to grow. A possible explanation for this phenomenon comes from observing the behavior of models specialized for an environment, an example of which will be discussed in the next subsection. Models that are specialized to an environment result in near-optimal state-action trajectories connecting the initial box pose to the target pose, and take less than 50 timesteps⁵. A general prior would then be positioned in some area of parameter space minimizing its average distance to several specialist networks, hence the higher number of updates necessary for its adaptation. In addition to that, we note that our meta-update does not impose hard constraints on the adaptation speed.

3) *Simulated box rotation: Ablations:* Our principal aim in this paper is to investigate the advantages of combining meta-learning and neural-ODEs in the contexts that were previously discussed. Therefore, we consider two ablation experiments. In the first one, we consider two different manners in which the meta-update component can be removed. In the second ablation study, we compare a specialized neural ODE to a specialized Recurrent network.

Meta-update ablation. We first naively disable the meta-update in ACUMEN. The results of this experiment are reported in figure 5. Unsurprisingly, without the meta update, the adaptations that the neural ODE goes through within distinct episodes is only sufficient for solving on average about $\sim 9\%$ of the sampled environments in the given time limit of $H_{max} = 300$ (which is the same as in all our simulation-based experiments). The number of timesteps solved in all environments (solved or unsolved) displays significantly higher mean and variance.

To further investigate the advantages of the learned prior, we compare its performance with that of a specialist model trained until convergence on a single "average" environment, henceforth noted E_1 , given by a box with mass $1.5kg$ and uniform mass distribution. The results are compiled in table I. As expected, the learned prior is able to adapt to and solve 91.3% of the environments within the (per episode) time limit of H_{max} , while the specialized model is able to solve 77.8% of the environments. The number of timesteps spent across all environments (solved and unsolved) displays significantly higher mean and variance. It is however interesting to note that the specialized model can more easily solve environments that are

close to the dynamics it has been trained for. This is why \min_{time} , the minimum number of necessary timesteps across all environments is lower for the specialized environment in table I.

	#sampled envs	success	μ_{time}	σ_{time}	\min_{time}
general prior	600	548 (91.3%)	121.943	57.85	58
specialized model	600	467 (77.8%)	127.513	92.26	40

TABLE I: Comparison between the prior learned by ACUMEN and a specialized model trained in an average environment on 600 randomly sampled boxes. Here, $\mu_{time}, \sigma_{time}$ denote the mean and average of the number of timesteps spent in a given environment, which are both lower for the learned prior. The minimum number of time-steps used to solve an environment is denoted \min_{time} .

	Neural ODE	RNN
#episodes	30	30
success	24 (80%)	15 (50%)
μ_{time}	58.125	109.26
σ_{time}	10.89	34.93
med_{time}	59.15	107.0

TABLE II: Comparison between a specialized neural ODE and a specialized RNN.

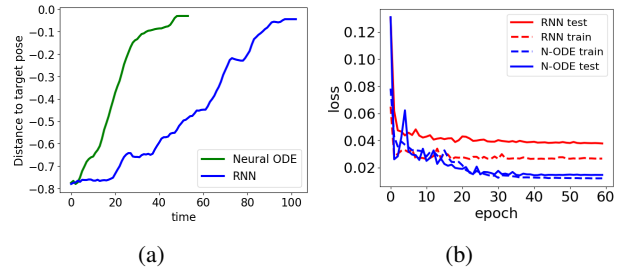


Fig. 6: (a) Comparison between typical trajectories resulting from neural ODEs vs RNNs. (b) Train and test losses on the E_2 environment.

Neural-ODE ablation. As discussed in the beginning of section V, we compare the results obtained with a neural ODE with those obtained with a stacked RNN, which takes as input the sequence of tuples $(a_0, \Delta t_0), \dots, (a_n, \Delta t_n)$ with Δt_i the elapsed time between a_i and a_{i+1} for $i \neq n$, and Δt_n the elapsed time between a_n and the next observation (or during planning, the next predicted state). Note that the RNN and the neural ODE used in this section are comparable in terms of number of parameters ($\sim 9k$ for the RNN vs $\sim 8.5k$ for the neural ODE, see appendix D for more details).

For this experiment, we considered an environment with fixed dynamics, that we will note E_2 , corresponding to a mass of $0.5kg$ and uniform mass distribution. The stacked RNN was first pretrained on data from that environment, which were gathered from three episodes with significant action-state coverage: one with random controls, one successful episode with neural-ODE based control, and another episode using the same neural-ODE but with noise injected into the actions. It was then compared during 30 model-predictive trials to the specialist neural-ODE model that was learned in the previous section on the E_1 environment⁶. Note that while the dynamic system

⁵The number of timesteps necessary for an episode based on a specialist network to succeed was computed by sampling $N_{env} = 30$ environments and training N_{env} corresponding neural ODEs on each of them to obtain specialist networks. Then, each neural ODE was used in 10 control episodes on the environment it was trained on. Except for a few outliers, the average number of timesteps fell in between 40 and 50.

⁶We note that both models were trained on approximately the same amount of data, i.e. ~ 400 observations (with train/test split sizes of 300 – 400) and their associated ~ 1100 commands.

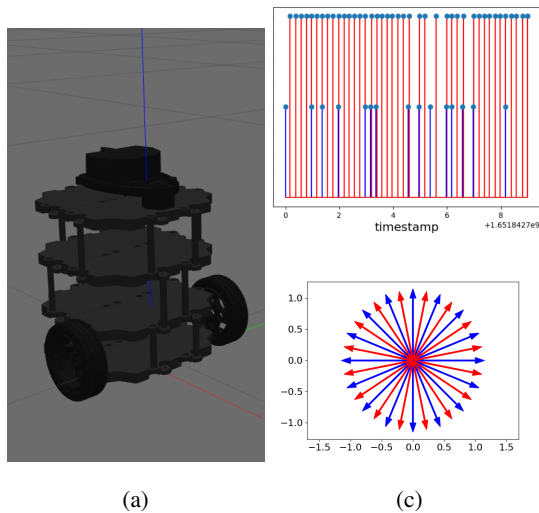


Fig. 7: (a) The turtlebot simulation (b) Stem-plot showing an example of observation (in blue) and action (in red) occurrence. In this particular example, observations are dropped with probability $P_{drop} = 0.5$, further accentuating the asynchronous/irregular nature of actions and observations. (c) The wind directions used to define the dynamics distributions. Blue arrows indicate directions that are used for training and meta-updates, and red arrows show the directions on which the test distribution is based.

is fixed, the observations and actions change from one episode to the other, as they are the result of interpolations at random timestamps, as previously specified in section §V-A1. The results are reported in table II.

It can be seen that the ODE-based network, specialized for the E_1 environment, performs significantly better than the recurrent model on the E_2 model, for which the latter is a specialist. Figure 6(a) shows two example box pose trajectories from successful episodes, one obtained using the neural ODE (in green), and the other from using the recurrent model (blue curve). Each curve represents the distance from the current pose to the target state. It can be seen that the trajectory obtained with the neural ODE is much smoother than the one resulting from the stacked RNN.

Regarding model accuracy, we also observed that when trained from scratch on the same data from E_2 as the RNN, a neural ODE converged to lower loss values on both train/test splits (Figure 6(b)).

We conjecture that the observations above are principally due to 1) the coarser linearization that is implied by the RNN in comparison to the neural ODE and 2) the fact that the RNN has to learn a more complex function. Indeed, in the neural ODE formulation, integration is decoupled from the learned model which only needs to approximate the time derivatives.

B. Gazebo Turtlebot3 simulation

We describe the simulation and dynamics distributions in section §V-B1. In section §V-B2, we define different levels of irregularity in the data, and compare the use of neural ODEs and RNNs for controlling the turtlebot in each of those settings. The complete system’s ability to adapt to unseen environment physical properties is then evaluated in section §V-B3.

1) *Gazebo turtlebot3 simulation: description:* This simulation [20] uses the ROS Gazebo [45] package and is therefore closer to a real robotic system than the pybullet simulation discussed in the previous section. Furthermore, policies trained in this simulation or more generally with Gazebo simulation can be successfully transferred to real systems, although strategies such as domain randomization are often necessary to avoid overfitting to the simulator [46], [47], [48]. A screenshot of the simulation is given in figure 7 (a).

The command space of the robot is given by $[-0.22, 0.22] \times [-0.22, 0.22]$ for linear and angular velocity control⁷. We use the pose estimation of the robot—as published to the `/odom` topic by the Gazebo turtlebot3 simulation—as the observation, but reduce the publication frequency of that information to ~ 6 Hz. As in the box rotation experiments, we approximate the state to propagate as $\hat{z} = \xi(\omega_t, \omega_{t-1}) \triangleq [\omega_t, \frac{\omega_t - \omega_{t-1}}{\Delta t}]$ where ω_{t-1}, ω_t are the two last pose observations.

In all experiments, the robot starts at the origin and its objective is to navigate to an arbitrarily set target. We will first compare the effectiveness of RNNs and neural ODEs when faced with different levels of irregularity, that will be defined by the probability P_{drop} of dropping observations. Note that even $P_{drop} = 0$ will result in irregularity in the data as a result of the asynchronous nature of ROS. An example of action and observation occurrences during a randomly selected time interval with value $P_{drop} = 0.5$ is shown in figure 7.

The comparisons between neural ODEs and recurrent models which are presented in sections V-B2 use the default settings of the simulator, where except for gravity, no external forces are applied to the robot. In order to evaluate the ability of the complete system to adapt to unseen environment physics (section §V-B3), we additionally define a distribution over physical properties of the simulation which are given by a choice of constant “wind” that is implemented as a constant acceleration applied to the robot. For simplicity, we consider a discrete set of directions, uniformly spaced on the unit circle (figure 7(c)), and a discrete set of magnitudes. Details on the different splits used for training, meta-updates and tests will be given in section §V-B3.

Note that the particular instance of algorithm 2 which is used in this section is equivalent to a simple random shooting (appendix B). While this sacrifices some precision, it allows the algorithm to publish decisions at a higher rate. The same motivation led us to replace the adaptive-step solver of the previous experiments by a fixed step ODE solver (RK4) which proved sufficient for controlling the turtlebot.

2) *Gazebo turtlebot3 simulation: neural ODEs vs stacked RNNs:* Here we concentrate on control using an environment with fixed physical properties in order to investigate the advantages of neural ODEs for control in the case of irregular/asynchronous observations. The physical properties are set to the defaults and no wind is applied. As in the previous section, we use a stacked RNN—the hyperparameters of which are specified in appendix D—as a baseline. Note that both the RNN and the N-ODE are pre-trained until convergence with $P_{drop} = 0$ and using continuous actions that are sampled according to a uniform distribution over $[-0.22, 0.22]$ for both the linear and angular velocity controls. However, as detailed in appendix B, during control, actions are selected among a discretized set of velocity controls.

In many real world robotic systems⁸, different components often publish asynchronously to different topics and at different frequencies based on multiple hardware and software related constraints. This can result in significant irregularities. As controlling the simulated turtlebot is done via those same publication, subscription and service mechanisms, irregularities in actions and observations also naturally occur. However, in these simplified settings, those irregularities are rather small. Therefore, we introduce an additional source of irregularity by dropping each in-coming observation with a probability of P_{drop} . In what follows, we compare the RNN and the N-ODE for values of $P_{drop} = 0, 0.2, 0.5$. Figure 7(c) shows a stemplot of command/observation occurrences for $P_{drop} = 0.5$.

Figure 8 shows a few example trajectories for different values of P_{drop} and arbitrary set target positions. It can be seen that the performance of the RNN drops significantly as P_{drop} increases, while the impact on the neural ODE is limited. Indeed, while the trajectories obtained with the neural ODE remain short and smooth for different

⁷The maximum absolute value of the angular velocity that can be sent to the turtlebot is 2.84, but we use a lower value in order to simplify the dynamics.

⁸We exclude real-time Operating Systems from this discussion.

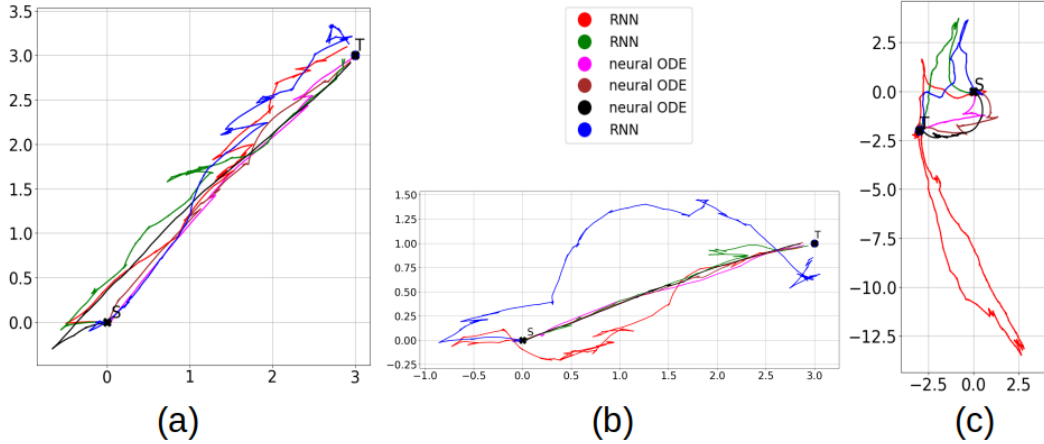


Fig. 8: Examples of trajectories obtained with Neural ODEs and RNNs for different values of P_{drop} . In all figures, the initial position of the robot and the target pose are respectively noted S, T . (a) Trajectories obtained when $P_{drop} = 0$. Note that in this case, irregularities in the input data are still present due to the asynchronous nature of ROS. (b) Example trajectories with $P_{drop} = 0.2$. (c) Trajectories obtained with $P_{drop} = 0.5$.

P_{drop} values, the trajectories produced when using the stacked RNN become longer and less regular with the increase of P_{drop} .

Quantitative results. We ran a total of 150 control experiments, in which the initial position of the robot was set to the origin and its orientation to the identity matrix. Each experiment was fully determined by the choice of

- 1) a pretrained model: RNN or neural ODE,
- 2) a target out of five possible positions⁹,
- 3) Observation drop probability P_{drop} , chosen in the set $\{0.0, 0.2, 0.5\}$.

resulting in 30 possible combinations, and ran each of them 5 times. The trajectories were compared according to three characteristics:

- 1) Trajectory length. This is just the length of the curve.
- 2) Irregularity, which we define as $S_\kappa \triangleq \int_a^b \kappa(s) ds$, where κ is the unsigned curvature and ds is the length element.
- 3) Episode length in terms of number of applied actions.

Indeed, while shorter and smoother trajectories are desirable, a control process that would only chose conservative, small actions might indicate poor predictions from the learned model, hence the consideration of the number of actions as the third characteristic.

The results are reported figure 9. Note that in figures 9(a, b, c), markers of the same color correspond to episodes that had the same positional target for the robot. As it can be seen in figure 9(a), the small irregularities that are present due to the asynchronous nature of ROS in general seem to result in trajectory lengths and irregularities that are slightly larger for the recurrent models. As we increase P_{drop} to 0.2 (figure 9(b)), the trajectories become longer and less regular for both models. However, the effect is more pronounced for the RNNs, in particular in terms of trajectory irregularity: neural ODEs mostly remain in the same interval ($S_\kappa < 1000$) as in the previous experiment ($P_{drop} = 0$, figure 9(a)), but the majority of RNNs produce trajectories with irregularities $S_\kappa > 1000$ while that was not the case in the previous experiment with $P_{drop} = 0$. As for trajectory lengths, RNNs more frequently produce trajectories with lengths greater than the upper bounds on the results of the previous experiment (dashed lines in figure 9(b)).

The gap between the performances of neural ODEs and RNNs is widened further as P_{drop} is increased to 0.5 (figure 9(c)). The

majority of lengths and irregularities of the trajectories generated by RNNs takes values greater than the upper bounds on previous experiments (dashed line in figure 9(c)). The lengths in particular are increased by factors of up to 5, while irregularities are increased by factors of up to 3. On the other hand, length-wise, the trajectories generated by neural ODEs remain in the same interval as in the previous experiments. The increase in irregularity is also much less pronounced than with the recurrent models.

Neural ODEs also outperform the considered RNNs in terms of number of required actions. As shown in figure 9(d), the average number of actions necessary for success using neural ODEs is upper bounded by the corresponding value for RNNs which significantly increases with P_{drop} .

3) *Gazebo turtlebot3 simulation: Adaptation to novel dynamics:* As mentioned in section §V-B1, a distribution of physical properties based on applying "wind" as a constant acceleration vector defines the dynamics distribution $P(\mathcal{T}_{turtle})$ that we consider. More formally, noting $S_{mag} \triangleq \{0.1, 0.2, 0.4, 0.5\}$ the set of possible magnitudes, the train and test distributions are given by

$$\begin{aligned}
 P(\mathcal{T}_{turtle})^{train} &\triangleq \mathcal{U}(\{[\cos(u), \sin(u)]^T | u = n \frac{2\pi}{n_d} \\
 &\quad \forall n = 2k \quad , k \in \mathbb{Z}\} \times S_{mag}) \\
 P(\mathcal{T}_{turtle})^{test} &\triangleq \mathcal{U}(\{[\cos(u), \sin(u)]^T | u = n \frac{2\pi}{n_d} \\
 &\quad \forall n = 2k + 1 \quad , k \in \mathbb{Z}\} \times S_{mag})
 \end{aligned} \tag{7}$$

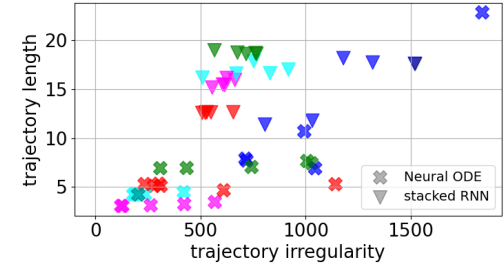
where \mathcal{U} denotes the uniform distribution and n_d is the number of wind directions. In our implementation, we chose $n_d = 32$, resulting in 16 directions for each of the train and test splits (figure 7(c)).

Setting $P_{drop} = 0.5$ and starting with the pretrained neural ODE of section V-B2, we evaluated the effect of meta-updates, each based on $N = 25$ train environments. As the asynchronous nature of the publishing/subscription/service mechanisms can lead to different outcomes on each execution, we averaged the results of three distinct experiments, each making 5 updates to the initial model. The results, displayed in figure 10 (a,b), indicate that just a few iterations result in significant improvements in both the number of solved environments (figure 10(a)) and the number of necessary steps (10(b)).

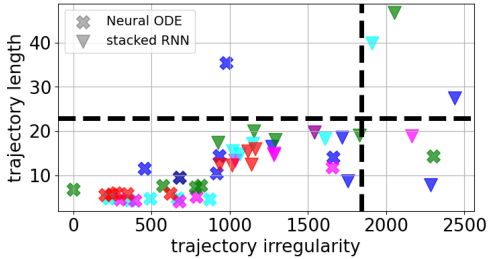
VI. EXPERIMENTS ON THE SOTO2 ROBOT

In this section, we first show the feasibility of controlling an industrial robot using the proposed neural ODE based model predictive

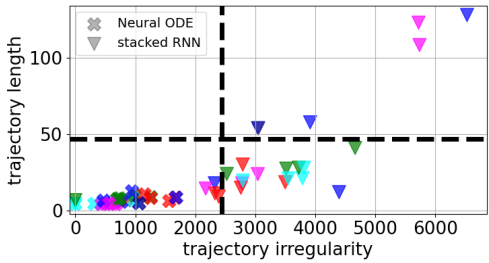
⁹The target position were set over all quadrants of the xy plane in an arbitrarily manner to $\{(3.0, -2.0, 0.0), (-3.0, -2.0, 0.0), (3.0, 1.0, 0.0), (3.0, 3.0, 0.0), (2.5, 6.3, 0.0)\}$



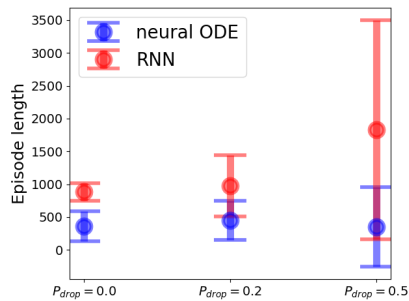
(a) Trajectory length and irregularity for $P_{drop} = 0.0$



(b) Trajectory length and irregularity for $P_{drop} = 0.2$. The dashed lines mark the maximum irregularity and length that were reached in the previous experiment ($P_{drop} = 0$, figure 9(a)).



(c) Trajectory length and irregularity for $P_{drop} = 0.5$. The dashed lines mark the maximum irregularity and length that were reached in the previous experiment ($P_{drop} = 0.2$, figure 9(b)).



(d) Mean and standard deviation of episode lengths (in terms of required actions for success).

Fig. 9: (a), (b), (c) each correspond to 50 experiments corresponding to a value of P_{drop} , such that half experiments use a neural ODEs and the other half uses a recurrent model. The experiments in which the positional target of the robot was the same are indicated using the same color. (d) compares the length of episodes in terms of number of commands necessary to reach the goal.

approach. However, due to limited access to the robot, we present meta-learning results only based on offline logs gathered with robots already operating at customer sites.

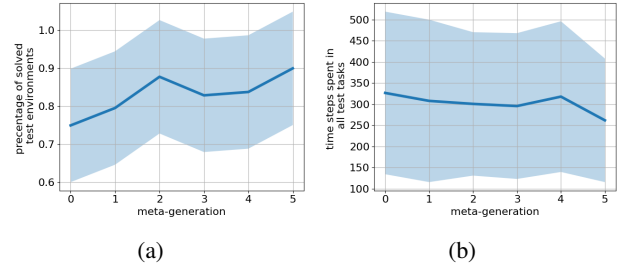


Fig. 10: The effect of a few meta-updates to the pretrained neural ODE, averaged over three distinct execution. Both figures show average values and one single standard deviation.

A. Controlling the robot with neural ODEs

We focus on the gripper of the SOTO2 robot (figure 1) and similar to the simulated experiments of section V-A, consider rotating a given box by 90° as the objective to reach during an episode. A notable difference between this experimental setup and the environment considered in the box rotation simulation of section V-A is that the distance between the two conveyor belts can vary from an episode to the next: once the SOTO2 has approached a box, it adjusts the distance between the two conveyor belts according to the box’s dimensions before loading it onto them.

The observation and command spaces in this experiment are similar to those of the simulated box rotation experiments (V-A1). More precisely, the action space, which corresponds to the independent velocity control of the two conveyor belts is given by $[0.05, 0.05] \times [0.05, 0.05]$ (in m/s). The state is given by $\hat{z} = \xi(\omega_t, \omega_{t-1}) \triangleq [\omega_t, \frac{\omega_t - \omega_{t-1}}{\Delta t}]$ with ω_t denoting the pose of the box at time t ¹⁰. As illustrated in figure 11 (a), when controlling the gripper, both actions and observations can be applied/received irregularly.

We pre-trained¹¹ a neural ODE model on nine episodes where and empty box of dimensions $12cm \times 30cm \times 40cm$ (figure 11 (b)) was rotated through manual publication of commands, on conveyor belts separated by a distance of roughly $22cms$. This amounts to a total of 891 recorded commands issued over a time interval of about $5.5min$ and the 1605 observations that were received during that time. For testing, we considered three different box mass distributions, and paired them with different distances between the conveyor belts, which were fixed at the beginning of each episode. The box mass distributions that were used are shown in figure 11 (c, d, e, f). In all experiments, the initial pose of the box up to some variation¹² is similar to the one showed in figures 11(c, e, f). Note that figure 11(d) shows the target position achieved after a successful rotation.

The action selection mechanism of (algorithm 2) was guided by a dense reward defined using the distance between the current and target poses (appendix B-C). Success and failure were defined in a binary manner: an episode was considered successful if the box reached the target orientation in less than 150 commands, and its outcome was recorded as failure otherwise. Information on hyper-parameters can be found in appendix D.

Large box with centered shoe. In this experiment, a shoe was placed in the center of the large box of size $12cm \times 30cm \times 40cm$ which was used during training. This places the center of gravity near the center of the box, so the mass distribution of the box is relatively similar to that of the empty box that was used for

¹⁰The planar pose of the box, comprised of $2d$ position and yaw in the plane defined by the conveyor belts, is estimated by a pose-tracking pipeline which takes as input a stream from an RGB-D camera and uses standard computer vision tools. The camera is positioned above at the top of the SOTO2, and is looking down on the gripper.

¹¹As in the previous experiments, the pretrained model is still refined during each episode, as specified in algorithm 1.

¹²A few centimeters ($< 5cms$) in the direction parallel to the conveyor belts, and less than $3cms$ in the direction orthogonal to them.

training. A Total of 60 experiments were performed, such that each group of 10 successive experiments was associated with a fixed distance between the conveyor belts from the set $C_{dist} \triangleq \{15.08, 16.45, 18.78, 21.06, 23.17, 25.81\}$ (in cms). The results are reported in figure 12 (in blue). As expected, while the solution seems to adapt reasonably well to all different environments, the larger success rates as well as the shortest episode lengths happen when the physical conditions are close to those of the training dataset.

Small box with centered shoe. In this second set of experiments, we used a smaller box of size $12cm \times 20cm \times 30cm$, and again placed a single shoe in it as in figure 11 (c). We performed 10 experiments for each of the conveyor belt distances from C_{dist} that were applicable for this smaller box, *i.e.* we considered distances of 15.08 and 16.45. The results are reported in figure 12 (in red). While success rate remained at 90%, the number of commands needed to reach the target pose was significantly higher, mainly due to the larger number of gradient updates applied to the model. This is not surprising as the difference between the mass distributions of the test environments and the mass distribution the neural ODE was pretrained on is more pronounced than in the previous experiments with the larger box.

Skewed mass distribution. We also considered a highly skewed mass distribution by placing a weight ($\sim 300gr$) inside a shoe which in turn was placed in on side of the box. We only performed 10 experiment with a distance of 18.02cms between the two conveyor belts. Among those episodes, 70% led to success, and the average and standard deviations of successful episode lengths were respectively of 57.71 and 32. Those results are considerably worse than the results obtained with similar distances between conveyor belts in the previous two experiments. Such difficulties in adapting to unseen mass distributions was part of the motivation for the meta-learning components of ACUMEN, which we evaluated in simulation in sections V-A and V-B.

Before closing this section, we remark that the rewards (appendix B) as well as the binary success/failure criterion used in this section do not take additional constraints, necessary for safe operations in industrial environments into account. While the trajectories during each episode were reasonably smooth, a few failures were due to the box getting too close to the sensors at the extreme end of the gripper (seen in the right side of the bottom image of figure 1), where the corner of the box was stuck. Such failure cases could have been avoided with more careful reward shaping, hyperparameter tuning, or by a more complex environment model. Furthermore, we observed that during successful experiments with the highly skewed distribution (figure 11(f)), even though the desired box orientation was reached, the final position of the box was asymmetrical with respect to the conveyor belts. One potential solution to that problem would be to endow the agent with the ability to issue an additional command for adjusting the distance between the two conveyor belts. However, this is beyond the scope of this paper.

B. Meta-learning from off-line logs

Due to limited access to Magazino’s robots, we only present meta-learning experiments based on $1k$ offline logs from robot operating at customer sites gathered while manipulating boxes with various unknown mass distributions. We randomly split this dataset into train, validation and test splits of size 600, 200, 200, and meta-learned a model by setting an inner optimization budget of $N_{it} = 5$ (*i.e.* the same budget as in the control experiments of the previous section) for each sampled trajectory. The data was *not* collected using our control method, but via a scripted policy which aligns and translates the boxes after they are loaded on the conveyor belts. While this results in lack of diversity in terms of box poses and joint actions that are present in the dataset (figure 13(a,b,c)), it should be noted that the boxes handled by the system span a wide array of mass distributions with total masses in the range $[0.1, 30.0]kg$.

We ran 5 meta-learning experiments that each had a fixed inner optimization budget of $N_{it} = 5$ and a duration of 35 meta iterations.

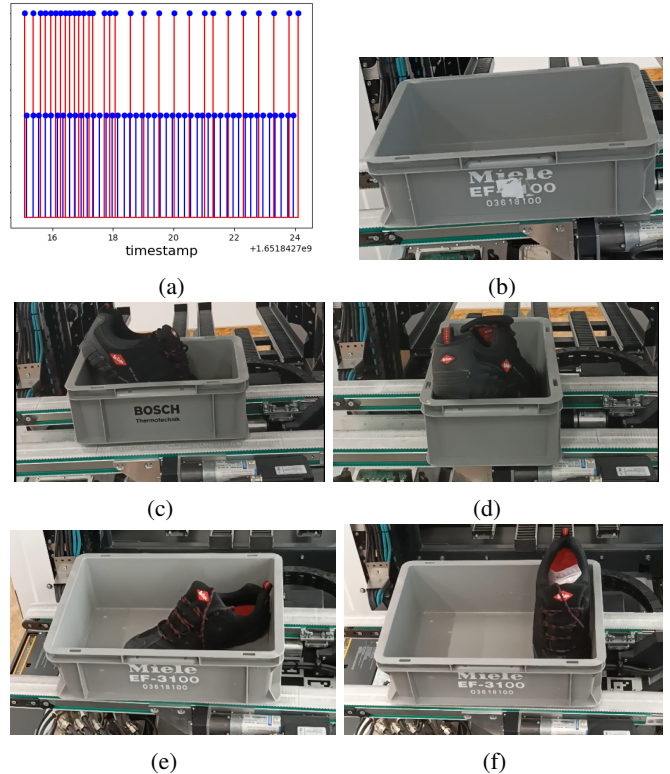


Fig. 11: (a) Stemplot indicating the occurrence of actions (in red) and observations (in blue) in an experiment with the SOTO2 robot, over an arbitrary time interval. It can be seen that both actions and observations are subject to irregularity. (b) The box used for training the neural ODE. (c, d, e, f) Different box volumes and mass distributions used for testing. Examples (c, d) indicate the initial state and the target state for the same box, obtained after a successful rotation. While the center of gravity of the boxes shown in (c, e) is close to the center, the distribution displayed in (f) where a small weight is also added to the shoe is highly skewed.

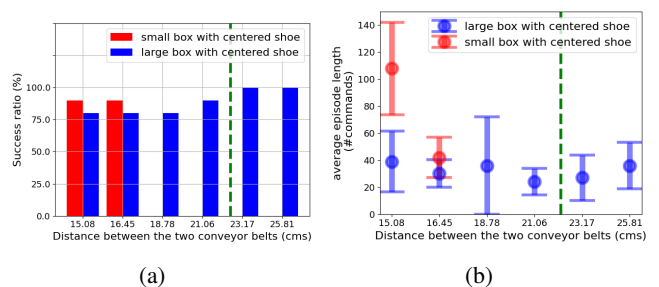


Fig. 12: The outcome resulting from using the proposed control method initialized with the pre-trained model. The green dashed line in both figures indicates the approximate distance between the conveyor belts when the training data was collected. (a) The percentage of successful rotations. (b) The average and standard deviation of episodes (in terms of number of commands) for successful rotations.

The averaged results are reported in figure 13. Note that as there is

little to no rotational motion in the dataset, the orientation approximation error is close to zero and can be ignored. Therefore, figure 13 only reports translational errors in *cms*. It can be observed that the average error given the budget limit $N_{it} = 5$ decreases from about 55mm to 46mm during the meta optimization.

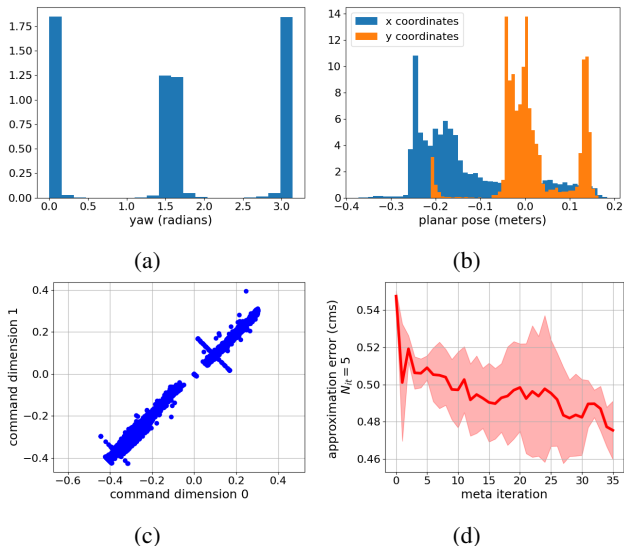


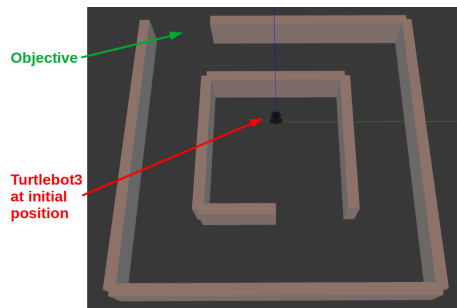
Fig. 13: A description of the contents of the offline logs (a,b,c) and the results of meta-learning experiments with $N_{it} = 5$. **(a)** In the logs, the box is either approximately aligned with the conveyors (~ 0 and ~ 3.14 radians) or rotated by around $\frac{\pi}{2}$. Therefore, there is almost no rotational motion in that dataset. **(b)** The planar position mostly varies along the x axis. This corresponds to the translational motion of the scripted policy. **(c)** Action dimensions have a correlation of ~ 1.0 , which also corresponds to the translational behavior of the scripted policy. **(d)** Average translational error from 5 meta-learning experiments with $N_{it} = 5$, each with a duration of 35 meta iterations. As there is little to no orientation variation in the dataset, orientation errors, which are close to zero, have not been reported in that plot.

VII. DISCUSSION

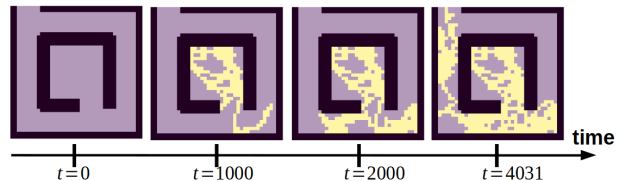
Before discussing the limitations of the proposed approach (§VII-B), we highlight some of its advantages compared to MPC approaches that learn the dynamics in a more task-oriented manner.

A. Flexibility

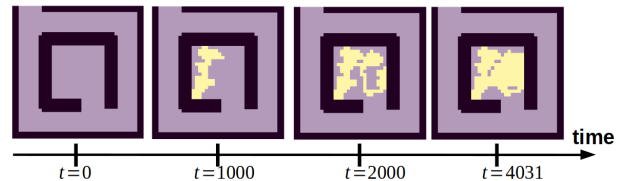
A desirable property for many robotics systems, especially in open-ended learning settings, is to be able to adapt to both new dynamics and tasks. MPC methods in general trade off flexibility with respect to task changes for accuracy and efficiency in trajectory-tracking in varying dynamics conditions. Indeed, adaptive MPC approaches [17], [49], [27], [16] minimize a function that is dependent on a state-space path (e.g. the classical $x^T Q x$ term in LQR). However, not all tasks can be specified as trajectories in a straight-forward manner, and it can be preferable to infer such trajectories without explicit planning. This is the case for tasks that require exploration and precise interactions with the environment, including other agents. An example maze navigation task is given in figure 14(a), where the simulated turtlebot3 used in the previous sections is now required to explore a maze. As it can be seen from 14(b), simply swapping the previous goal-based reward function for an exploration reward (appendix B-D) allows our proposed pipeline to reuse the previously learned neural-ODE model to solve this task, while random exploration (figure 14 c) fails. We note that several RL algorithms indeed do achieve similar task-agnostic behavior. However, as discussed in previous sections (§I, §III), unlike our approach and related MPC methods that leverage neural-ODEs, they do not handle irregular inputs.



(a) A maze navigation task specified in the turtlebot3 gazebo simulation



(b) Exploration progress using the proposed approach with an exploration-based reward.



(c) Progress of random exploration, which fails to cover enough states to solve the task. This highlights the need for exploration rewards, and thus the advantages of methods, such as ours, that can seamlessly switch between exploration and trajectory-based rewards.

Fig. 14: A maze navigation task which requires exploration. As the proposed approach is task-agnostic, it can be adapted to new tasks by simply changing the reward function. See appendix B-D for details on the reward function used for this task.

B. System Limitations

The main limitation of the system lies in its computational complexity (appendix C). While publishing commands at rates of 4–5Hz proved sufficient for controlling the SOTO2 and the turtlebot, we reached those frequencies by switching to a fixed-step solver and by using a simpler action sampling mechanism, sacrificing precision for speed. While this was possible for the SOTO2 and turtlebot experiments, there are many real-world applications where precision and high-frequency control are essential. In order to enable the deployment of the proposed method on such systems, several directions seem worth exploring: 1) Parallelism: in our current implementation which has not been optimized, the robot becomes idle during model updates that are made every *train_freq* iterations. Parallelising control and model updates, e.g. in a manner similar to the one used in Jiaho *et al.*[16] can help alleviate that problem and reduce the resulting latency. 2) Using more efficient solvers/training [50], [51], [52]. 3) Training goal-conditioned policies assuming the availability of a rich joint goal-reward embedding.

VIII. CONCLUSION

Two challenges are often encountered in industrial robotic systems such as the SOTO2 robot: (i) the asynchronous/irregular publication of observations/actions which violates the assumptions made by the majority of RL algorithms and (ii) dramatic discontinuous changes

in environment dynamics from an episode to the next. Furthermore, it is desirable for many robotics systems, especially in open-ended learning settings, to be adaptive not only to changes in environment dynamics, but also to changes in tasks. Motivated by these observations, we proposed ACUMEN, a task-agnostic Model Predictive Control method that formulates the problem of asynchronous control as a particular case of continuous-time control using learned neural ODEs as environment models, and incorporates meta-learning techniques to ensure adaptivity to changes in dynamics. We evaluated our framework in two simulated environments as well as on the real SOTO2 robot. We also discussed limitations and future directions for improvement. In particular, the computational complexity of the proposed algorithm, while not prohibitive for many use-cases—such as the studied examples—remains significant and should be improved in order to make the method applicable to robotic systems requiring high-frequency control.

IX. ACKNOWLEDGMENTS

This work was supported by the European Union’s H2020-EU.1.2.2 Research and Innovation Program through FET Project VeriDream under Grant Agreement Number 951992. We wish to thank Quentin Levent for their initial work on the conveyor belt simulation. Likewise, we express our gratitude to Patrick Gallinari, Yuan Yin and Kathia Melbouci for their valuable input and criticism during the development of the presented material.

REFERENCES

- [1] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,” *The International journal of robotics research*, vol. 37, no. 4-5, pp. 421–436, 2018. 1
- [2] I. A. OpenAI, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, “Solving rubik’s cube with a robot hand,” 2019. 1
- [3] A. Kumar, Z. Fu, D. Pathak, and J. Malik, “Rma: Rapid motor adaptation for legged robots,” *arXiv preprint arXiv:2107.04034*, 2021. 1
- [4] M. Deisenroth and C. E. Rasmussen, “Pilco: A model-based and data-efficient approach to policy search,” in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, 2011, pp. 465–472. 1, 2, 14
- [5] C. Pinneri, S. Sawant, S. Blaes, J. Achterhold, J. Stueckler, M. Rolinek, and G. Martius, “Sample-efficient cross-entropy method for real-time planning,” in *Conference on Robot Learning*. PMLR, 2021, pp. 1049–1065. 1, 3, 5, 6, 14
- [6] B. Lim, L. Grillotti, L. Bernasconi, and A. Cully, “Dynamics-aware quality-diversity for efficient learning of skill repertoires,” in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 5360–5366. 1
- [7] S. Kim, A. Coninx, and S. Doncieux, “From exploration to control: learning object manipulation skills through novelty search and local adaptation,” *Robotics and Autonomous Systems*, vol. 136, p. 103710, 2021. 1
- [8] C. Finn, P. Abbeel, and S. Levine, “Model-agnostic meta-learning for fast adaptation of deep networks,” in *International conference on machine learning*. PMLR, 2017, pp. 1126–1135. 1, 3
- [9] A. Nagabandi, I. Clavera, S. Liu, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn, “Learning to adapt in dynamic, real-world environments through meta-reinforcement learning,” *arXiv preprint arXiv:1803.11347*, 2018. 1, 3
- [10] A. A. Team, J. Bauer, K. Baumli, S. Baveja, F. Behbahani, A. Bhoopchand, N. Bradley-Schmiege, M. Chang, N. Clay, A. Collister *et al.*, “Human-timescale adaptation in an open-ended task space,” *arXiv preprint arXiv:2301.07608*, 2023. 1
- [11] A. Salehi, A. Coninx, and S. Doncieux, “Few-shot quality-diversity optimization,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 1–10, 2022. 1
- [12] S. Belkhale, R. Li, G. Kahn, R. McAllister, R. Calandra, and S. Levine, “Model-based meta-reinforcement learning for flight with suspended payloads,” *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1471–1478, 2021. 1, 3
- [13] C. Yildiz, M. Heinonen, and H. Lähdesmäki, “Continuous-time model-based reinforcement learning,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 12 009–12 018. 2, 3, 6
- [14] J. Du, J. Futoma, and F. Doshi-Velez, “Model-based reinforcement learning for semi-markov decision processes with neural odes,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 19 805–19 816, 2020. 2, 3
- [15] K. Y. Chee, T. Z. Jiahao, and M. A. Hsieh, “Knode-mpc: A knowledge-based data-driven predictive control framework for aerial robots,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 2819–2826, 2022. 2, 3
- [16] T. Z. Jiahao, K. Y. Chee, and M. A. Hsieh, “Online dynamics learning for predictive control with an application to aerial robots,” in *Conference on Robot Learning*. PMLR, 2023, pp. 2251–2261. 2, 3, 12
- [17] S. M. Richards, N. Azizan, J.-J. Slotine, and M. Pavone, “Control-oriented meta-learning,” *arXiv preprint arXiv:2204.06716*, 2022. 2, 3, 12
- [18] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, “Neural ordinary differential equations,” *Advances in neural information processing systems*, vol. 31, 2018. 2, 3
- [19] E. Coumans and Y. Bai, “Pybullet, a python module for physics simulation for games, robotics and machine learning,” <http://pybullet.org>, 2016–2021. 2, 5, 6
- [20] “Gazebo turtlebot3 simulation.” [Online]. Available: <https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/> 2, 5, 8
- [21] Stanford Artificial Intelligence Laboratory *et al.*, “Robotic operating system.” [Online]. Available: <https://www.ros.org> 2
- [22] T. M. Moerland, J. Broekens, and C. M. Jonker, “Model-based reinforcement learning: A survey,” *arXiv preprint arXiv:2006.16712*, 2020. 2, 3
- [23] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine *et al.*, “Model-based reinforcement learning for atari,” *arXiv preprint arXiv:1903.00374*, 2019. 2
- [24] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, “Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 7559–7566. 2
- [25] K. Chua, R. Calandra, R. McAllister, and S. Levine, “Deep reinforcement learning in a handful of trials using probabilistic dynamics models,” *Advances in neural information processing systems*, vol. 31, 2018. 2, 3
- [26] A. Saviolo, G. Li, and G. Loiano, “Physics-inspired temporal learning of quadrotor dynamics for accurate model predictive trajectory tracking,” *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 10 256–10 263, 2022. 2
- [27] N. A. Spielberg, M. Brown, and J. C. Gerdes, “Neural network model predictive motion control applied to automated driving with unknown friction,” *IEEE Transactions on Control Systems Technology*, vol. 30, no. 5, pp. 1934–1945, 2021. 2, 12
- [28] R. Rubinstein, “The cross-entropy method for combinatorial and continuous optimization,” *Methodology and computing in applied probability*, vol. 1, no. 2, pp. 127–190, 1999. 3
- [29] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, “Learning latent dynamics for planning from pixels,” in *International conference on machine learning*. PMLR, 2019, pp. 2555–2565. 3
- [30] J. Lee and R. S. Sutton, “Policy iterations for reinforcement learning problems in continuous time and space—fundamental theory and methods,” *Automatica*, vol. 126, p. 109421, 2021. 3
- [31] M. Abu-Khalaf and F. L. Lewis, “Nearly optimal control laws for nonlinear systems with saturating actuators using a neural network hjb approach,” *Automatica*, vol. 41, no. 5, pp. 779–791, 2005. 3
- [32] H. Wang, T. Zariphopoulou, and X. Y. Zhou, “Reinforcement learning in continuous time and space: A stochastic control approach,” *J. Mach. Learn. Res.*, vol. 21, no. 198, pp. 1–34, 2020. 3
- [33] C. Tallec, L. Blier, and Y. Ollivier, “Making deep q-learning methods robust to time discretization,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 6096–6104. 3
- [34] M. Ghavamzadeh and S. Mahadevan, “Continuous-time hierarchical reinforcement learning,” in *ICML*, vol. 18, 2001, pp. 186–193. 3
- [35] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, “Meta-learning in neural networks: A survey,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 9, pp. 5149–5169, 2021. 3
- [36] T. Schaul and J. Schmidhuber, “Metalearning,” *Scholarpedia*, vol. 5, no. 6, p. 4650, 2010, revision #91489. 3
- [37] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, “Rl²: Fast reinforcement learning via slow reinforcement learning,” *arXiv preprint arXiv:1611.02779*, 2016. 3

- [38] J. Oh, M. Hessel, W. M. Czarnecki, Z. Xu, H. P. van Hasselt, S. Singh, and D. Silver, "Discovering reinforcement learning algorithms," *Advances in Neural Information Processing Systems*, vol. 33, 2020. 3
- [39] Z. Xu, H. P. van Hasselt, and D. Silver, "Meta-gradient reinforcement learning," *Advances in neural information processing systems*, vol. 31, 2018. 3
- [40] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra *et al.*, "Matching networks for one shot learning," *Advances in neural information processing systems*, vol. 29, pp. 3630–3638, 2016. 3
- [41] X. Song, W. Gao, Y. Yang, K. Choromanski, A. Pacchiano, and Y. Tang, "Es-maml: Simple hessian-free meta learning," *arXiv preprint arXiv:1910.01215*, 2019. 3, 5
- [42] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017. 5
- [43] P. M. McGuire, *Conveyors: application, selection, and integration*. CRC Press, 2009. 6
- [44] J. R. Dormand and P. J. Prince, "A family of embedded runge-kutta formulae," *Journal of computational and applied mathematics*, vol. 6, no. 1, pp. 19–26, 1980. 6
- [45] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154. 8
- [46] T. Chaffre, J. Moras, A. Chan-Hon-Tong, and J. Marzat, "Sim-to-real transfer with incremental environment complexity for reinforcement learning of depth-based robot navigation," *arXiv preprint arXiv:2004.14684*, 2020. 8
- [47] P. Nikdel, R. Vaughan, and M. Chen, "Lbgp: Learning based goal planning for autonomous following in front," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 3140–3146. 8
- [48] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2020, pp. 737–744. 8
- [49] S. M. Richards, N. Azizan, J.-J. Slotine, and M. Pavone, "Adaptive-control-oriented meta-learning for nonlinear systems," *arXiv preprint arXiv:2103.04490*, 2021. 12
- [50] M. Poli, S. Massaroli, A. Yamashita, H. Asama, and J. Park, "Hypersolvers: Toward fast continuous-depth models," *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 105–21 117, 2020. 12, 15
- [51] M. Lehtimäki, L. Paunonen, and M.-L. Linne, "Accelerating neural odes using model order reduction," *IEEE Transactions on Neural Networks and Learning Systems*, 2022. 12
- [52] J. Kelly, J. Bettencourt, M. J. Johnson, and D. K. Duvenaud, "Learning differential equations that are easy to solve," *Advances in Neural Information Processing Systems*, vol. 33, pp. 4370–4380, 2020. 12

APPENDIX A LOSS FUNCTIONS

In order to reduce model complexity and improve stability, as is usual in the literature (*e.g.* [4]), we train the dynamic model to predict changes Δz in system state. This process is encapsulated in the function `OptimizeNODE` from algorithm 1. A wide array of choices can be considered for the loss function, but as our aim is to predict pose in all of our experiments, we use the following weighted least squares formulation:

$$\mathcal{L}(\mathcal{T}_j, \theta) = \frac{1}{|\mathcal{D}_{train}|} \sum_{b \in \mathcal{D}_{train}} (E_{trans}^b(\theta) + w_l E_{rot}^b(\theta))^2 \quad (8)$$

where $E_{trans}^b(\theta)$ and $E_{rot}^b(\theta)$ respectively denote the errors in translation and orientation change predictions over example b from the training data, and where $w_l \in \mathbb{R}$ is a weighting coefficient, used values of which are given in table III. In the presented experiments where displacements are planar, the E_{rot}^b term reduces to an error over the yaw, which in order to avoid discontinuities we simply express as the Frobenius norm of $R_b^T \hat{R}_b$ with R_b, \hat{R}_b respectively the true and predicted rotation matrices.

APPENDIX B COMMAND SELECTION

The reward functions that we have used for each of our experiments are given in the following subsections, where the particular instantiations of algorithm 2 are also discussed.

A. Simulated box rotation

Let us write $d_{trans}^{init} \triangleq \|z_{trans}^{init} - \hat{z}_{trans}\|_2$ the distance between the position predicted by the model and the initial pose of the box on the conveyor belt. Noting $z_{rot}^{target}, \hat{z}_{rot}$ respectively the desired and predicted rotation, the reward for the predicted pose \hat{z} is given by

$$R(\hat{z}) = -\|z_{rot}^{target} - \hat{z}_{rot}\|_2^2 - 10.0 \mathbb{I}[d_{trans}^{init} > r_t] d_{trans}^{init} \quad (9)$$

where \mathbb{I} is a predicate function that returns 1.0 if its argument evaluates to true and returns 0.0 otherwise. The threshold r_t was set to 0.3 in our experiments.

Algorithm 3: The particular instantiation of algorithm 2 that is used in the simulated box rotation experiments. It results from sampling time-correlated sequences of actions as in [5]. Differences with algorithm 2 have been highlighted in brown. The reward $R(\cdot)$ used in this experiment is given by equation 9.

Input: Neural ODE weights θ , state approximation $\hat{z}(t_0)$ at time t_0 , desired duration of state propagation Δt , previous actions c_1, \dots, c_l (augmented with their timestamp info), **mean and diagonal covariance** μ_0, Σ_0 , population size N_p , number of elites N_e , length of action sequence to sample H , reward function $R(\cdot)$, convergence criterion \mathcal{Y} , **colored noise parameter** β

Output: action sequence a_1^*, \dots, a_H^*

```

1 Function CEMStep( $\theta, \hat{z}(t_0), \{c_1, \dots, c_l\}, \Delta t,$ 
    $\mu_0, \Sigma_0, N_p, N_e, H, R(\cdot), \mathcal{Y}$ ):
2    $\mu \leftarrow \mu_0$ 
3    $\Sigma \leftarrow \Sigma_0$ 
4   // initialize elite set
   elites  $\leftarrow \emptyset$ 
5   while not  $\mathcal{Y}(\Sigma)$  do
6     // sample  $N_p$  action sequences of length  $H$  at regular
     intervals in  $[t_0, t_0 + \Delta t]$ 
7      $a_{1:H}^1, \dots, a_{1:H}^{N_p} \sim \text{SampleTimeCorrelatedSequence}(\mu, \Sigma, \beta)$ 
8     for each  $a_{1:H}^i$  do
9        $\pi_i \leftarrow [c_1, \dots, c_l].\text{concatenate}(a_{1:H}^i)$ 
10      // Let  $u_i(t)$  the function that computes actions via
      interpolating elements of  $\pi_i$ 
11       $u_i \leftarrow \mathcal{I} \circ \pi_i$ 
12      // propagate the state
13       $\hat{z}_i \leftarrow \hat{z}(t_0) + \int_{t_0}^{t_0 + \Delta t} \hat{\mathcal{F}}(\hat{z}(t), u_i(t), \theta) dt.$ 
14      // compute associated reward
15       $r_i \leftarrow R(\hat{z}_i)$ 
16    end
17    elites  $\leftarrow$  select best  $N_e$  action sequences
      according to the  $r_i$ 
18     $\mu, \Sigma \leftarrow$  fit Gaussian to elites
19  end
20  // return the best action sequence (alternatively, re-sample using
    $\mu, \Sigma$  after convergence)
21  return elites

```

The actions sampling mechanism that is used in those experiments results from defining the prior distribution P_ψ of algorithm 2 as a distribution over time-correlated sequences with noise parameter β ([5]). The complete selection algorithm is given in algorithm 3.

B. Gazebo Turtlebot3 simulation

Keeping the notation for the predicate $\mathbb{I}[\cdot]$ and this time noting $d_{trans}^t \triangleq \|z_{trans}^{target} - \hat{z}_{trans}^t\|_2$ the distance between the position

Algorithm 4: The action selection mechanism used for the turtlebot can be seen as a particular instance of algorithm 2, which results from setting $H = 1$ and choosing a uniform distribution in the latter. The reward function $R(\cdot)$ which guides actions selection in this set of experiments is defined by equation 11.

Input: Neural ODE weights θ , state approximation $\hat{z}(t_0)$ at time t_0 , desired duration of state propagation Δt , previous actions c_1, \dots, c_l (augmented with their timestamp info), discrete set of actions a_1, \dots, a_l to sample from, number of actions N_p to sample

Output: action sequence a_1^*, \dots, a_H^*

```

1 Function RandomShooting( $\theta, \hat{z}(t_0), \{c_1, \dots, c_l\}, R(\cdot)$ ):
2   // sample  $N_p$  actions to be applied at  $t + \Delta_t$ 
3    $a_1, \dots, a_{N_p} \sim \text{SampleUniform}(a_1, \dots, a_l)$ 
4   for each  $a_i$  do
5      $\pi_i \leftarrow [c_1, \dots, c_l].\text{concatenate}(a_i)$ 
6     // Let  $u_i(t)$  the function that computes actions via
7     // interpolating elements of  $\pi_i$ 
8      $u_i \leftarrow \mathcal{I} \circ \pi_i$ 
9     // propagate the state
10     $\hat{z}_i \leftarrow \hat{z}(t_0) + \int_{t_0}^{t_0 + \Delta t} \hat{\mathcal{F}}(\hat{z}(t), u_i(t), \theta) dt.$ 
11    // compute associated reward
12     $r_i \leftarrow R(\hat{z}_i)$ 
13  end
14   $a_{best} \leftarrow \text{select best action according to the } r_i$ 
15  return  $a_{best}$ 

```

predicted by the model and the target position, we write \hat{z}^t, \hat{z}^{t+1} the predicted poses at timestamps $t, t + 1$. In addition to d_{trans} , we consider the difference in angle between the desired heading and the predicted heading:

$$\begin{aligned}
 v_t &= \hat{z}_{trans}^{target} - \hat{z}_{trans}^t \\
 v_{t+1} &= \hat{z}_{trans}^{target} - \hat{z}_{trans}^{t+1} \\
 u_t^{t+1} &= \cos^{-1}\left(\left(\frac{v_t}{\|v_t\|_2}\right)^T \left(\frac{v_{t+1}}{\|v_{t+1}\|_2}\right)\right)
 \end{aligned} \tag{10}$$

and define the reward as

$$R(\hat{z}_{t+1}) = -d_{trans} - \mathbb{I}[u_t^{t+1} > r_t^{t+1}]u_t^{t+1}. \tag{11}$$

In other words, the error on the heading is ignored when it falls below the threshold r_t^{t+1} , which was fixed to 5° in our experiments.

In contrast with the simulated box rotation experiments, we found that the turtlebot required more frequent changes in controls which could be achieved by setting $\beta = 0$ (that is equivalent to sampling according to a Gaussian prior). Furthermore, we found that setting $H = 1$ and sampling from a discrete set of velocity values was sufficient to solve the navigation task. A natural additional benefit of this simpler sampling (algorithm 4) is the reduced computation time required to return a decision.

C. SOTO2 robot

The reward function used on the real robot results from setting $r_t = 0.2$ in equation 9. Regarding command selection, we observed that both cross-entropy based command selection (algorithm 3) and random shooting (algorithm 4) were able to lead to successful rotations. However, the latter, when performed over a reduced set of 81 discretized actions (see appendix D) was $\sim 5\times$ faster in terms of execution time. As in the turtlebot experiments, this faster operation time had the additional benefit of allowing faster corrections during box manipulations, leading to an overall lower number of necessary commands to complete the tasks. For these reasons, this sampling method was used throughout the experiments reported in section VI.

D. Turtlebot3 exploration task

As the aim of this experiment was to highlight the flexibility of the proposed approach rather than optimal navigation, we considered a simple occupancy grid based exploration reward, that we substituted for the previous goal-oriented rewards. We divided the maze environment into 40×40 cells, and recorded the number of times each cell was visited. Then, the reward associated to each state was determined by the number of times the robot had visited the corresponding cell:

$$R(\hat{z}_t)^{\text{exploration}} = -\text{num_visits_to_cell}(\text{cell}(\hat{z}_t)) \tag{12}$$

where $\text{cell}(\cdot)$ maps each state to the grid cell it falls into.

APPENDIX C COMPUTATION TIME.

Simulated Box rotation. We used a standard desktop computer equipped with an AMD Ryzen Threadripper 1920X 12-Core Processor for benchmarking. On average, a single integration of the neural ODE with a batch size of 20 took $210ms$ while a single forward of the RNN with the same batch size took $3ms$. Similarly, training the neural ODE over a single batch of size 5 took an average of $380ms$ versus an average of $61ms$ for the recurrent model. Note that the high cost of training and inference with neural ODEs was in part due to the choice of an adaptive-step solver (`dopri-5`). Furthermore, the sampling-based control used in these experiments (appendix B) added some overhead, resulting in decision frequencies of respectively $\sim 10\text{Hz}$ and $\sim 1\text{Hz}$ for the neural ODEs.

Simulated Turtlebot3 control. Using the same hardware as in the above, a single forward pass of the RNN with batch size 21 (one example in the batch per possible discrete action), took on average $4ms$, while the integration for the same batch using the ODE solver (RK4 in this case) took on average $82ms$. In other terms, neural ODEs took about $20\times$ more time to produce a decision. Similarly, an epoch of training on 120 samples took an average of $2.63s$ for neural ODEs and an average of $0.9s$ for RNNs. Those results are hardly surprising as vanilla neural ODEs are notoriously slow [50].

That being said, as in this particular application, using neural ODEs can actually lead to lower overall computational costs for higher P_{drop} values. For example, for $P_{drop} = 0.5$, the increased length and irregularity of the trajectories produced by the RNNs resulted in execution times that exceeded $35min$ for some episodes, while the maximum length reached for neural ODE based episode was $7.1min$. This is coherent with figure 9(d).

Note that the complete decision pipeline was able to select actions at about 5Hz when using neural ODEs with discrete action selection. **SOTO2 robot.** The robot was controlled remotely, and the corresponding ros package was executed on a precision 3551 laptop with a Intel(R) Core(TM) i7-10850H @ 2.70GHz CPU. A single prediction with the RK4 solver with batch size 81 (one example in the batch per possible discretized action) took on average $132ms$. The complete decision pipeline was able to publish commands at around 4Hz .

APPENDIX D ALGORITHM HYPERPARAMETERS

In all experiments, the dynamic model \hat{F} of equation IV-A was approximated using feed-forward neural networks with \tanh activations, taking as input at each integration time-step t the vector $(t, u(t), \hat{z}_t)^T$ concatenating the time, interpolated command and the propagated state.

A. Simulations

For the simulated box rotation experiments, four hidden layers, each of dimension 48 were used. For the gazebo turtlebot3 simulation, five hidden layers of dimensions $[64, 128, 128, 64, 64]$ were used. The stacked RNNs used in each section had approximately the same capacity as the neural ODEs they were compared to. While the neural

	N	r_{split}	α	σ	N_{it}	$\tau_i.timeout$	$train.freq$	ODE solver	N-ODE learning rate	γ_{decay}	w_l
Simulated box rotation	20	0.75	5e-4	1e-2	10	300	5	dopri5	5e-4	0.9	10.0
Gazebo Turtlebot3 simulation	25	0.75	5e-5	5e-3	1	640	40	RK4	1e-4	0.99	1.0
SOTO2 robot (N-ODE based control)	N.A	N.A	N.A	N.A	5	150	5	RK4	1e-3	0.99	100.0
SOTO2 robot (meta-learning using offline logs)	20	0.75	5e-2 (decayed)	1e-3	5	N.A	5	RK4	1e-3	0.99	100.0

TABLE III: Training hyper-parameters for both the outer and lower level optimization problems.

ODE used in the simulated conveyor belts experiments had $\sim 8.5k$ parameters, the corresponding stacked RNN had 5 hidden layers of dimension 32, resulting in about $\sim 9k$ parameters. Similarly, the neural ODE used to control the turtlebot had $\sim 37k$ parameters and we used a stacked RNN with 5 hidden layers of dimension 64 to match this number of parameters.

For the simulated box rotation experiments, the CEM-based action selection method (algorithm 3) with hyperparameters $N_p = 20$, $N_e = 5$, $\mu_0 = \mathbf{0}$, $\Sigma_0 = I$, $H = 4$, $\beta = 2$ was used. In the case of the turtlebot, we found that sampling commands with no time correlation ($\beta = 0$) allowed for quicker recovery from wrong command selections. Furthermore, while the model was pretrained on continuous commands sampled from a uniform distribution over $[-0.22, 0.22]$, as mentioned in the previous section (appendix §B, algorithm 4), we found that choosing the actions from a limited discretized set produced satisfactory results during control: in the presented control experiments, linear and angular velocity control were respectively chosen from $\{-0.1, -0.05, -0.01, 0.0, 0.01, 0.05, 0.1\}$ and $\{-0.1, 0.0, 0.1\}$.

The hyperparameters for the meta update (line 30 in algorithm 1) as well as the hyper-parameters for the training of the neural ODEs are reported in table III.

B. SOTO2 robot

The feed-forward network was comprised of five hidden layers of shapes $[80, 160, 160, 80, 80]$, resulting in approximately 59.7k parameters. The empirical results presented in section VI-A were produced using the same random shooting procedure as for the turtlebot (algorithm 4), with action selection over the discrete set $[-0.05, -0.02, -0.01, -0.005, 0.0, 0.005, 0.01, 0.02, 0.05] \times [-0.05, -0.02, -0.01, -0.005, 0.0, 0.005, 0.01, 0.02, 0.05]$. The hyperparameters used for training the neural ODEs are reported in table III. Note that the meta learning rate used in section §VI-B was linearly decayed at each meta iteration by a factor of 0.9.



ticularly in open-ended contexts.

Achkan Salehi received the M.S. degree in Computer Science from Sorbonne University in 2014. They conducted their PhD research on SLAM at the French Alternative Energies and Atomic Energy Commission (CEA) and received their PhD from Université Clermont Auvergne in 2018. From 2017 to 2020, they worked on Machine Learning topics related to spatial AI at SLAMCore Ltd. They have since then been with the ISIR lab from Sorbonne University where they primarily work on problems related to generalization and adaptive learning



Steffen Rühl received his degree in Computer Science from the University of Karlsruhe in 2007. He received his PhD for his research in the area of planning and execution of manipulation tasks on bimanual robots in variable environments in 2015. The same year he joined Magazino GmbH where today he leads the development of the manipulation process for warehouse robots.



this topic is based on Quality Diversity algorithms.

Stephane Doncieux is a Professor of Computer Science at ISIR (Institute of Intelligent Systems and Robotics), Sorbonne University, CNRS, Paris, France. He serves as the deputy director of ISIR, a multidisciplinary robotics laboratory with researchers in automation, mechatronics, signal processing, computer science and cognitive sciences. His research is in cognitive robotics, with a focus on open-ended learning. He focuses on the challenges that robotics raise for learning methods, in particular on the challenge of exploration. Most of his work on