Designing Simulators for Robot Learning

van der Heijden, D.S.

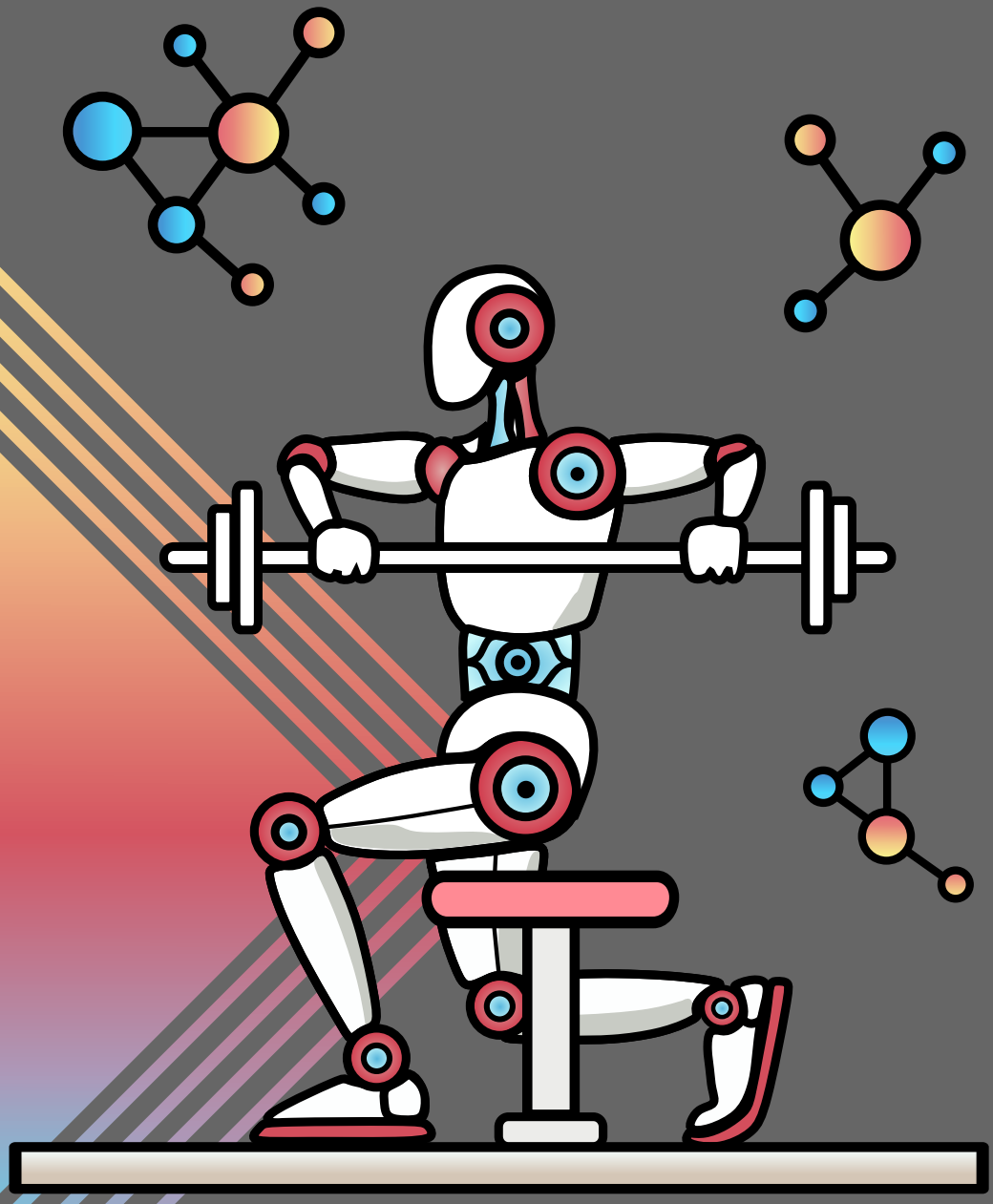
DOI
10.4233/uuid:d41281cc-d8cc-450d-b863-2a41d9d4a203

Publication date
2025

Document Version
Final published version

Citation (APA)
van der Heijden, D. S. (2025). *Designing Simulators for Robot Learning*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:d41281cc-d8cc-450d-b863-2a41d9d4a203

Designing Simulators for Robot Learning



Douwe Sebastiaan van der Heijden

Designing Simulators for Robot Learning

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof.dr.ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates
to be defended publicly on
Tuesday 29 April 2025 at 15.00 o'clock

by

Douwe Sebastiaan VAN DER HEIJDEN

Master of Science in Systems and Control, Delft University of Technology

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	voorzitter
Prof.dr. R. Babuska,	Delft University of Technology, promotor
Dr.-Ing. J. Kober,	Delft University of Technology, promotor
Dr. L. Ferranti,	Delft University of Technology, copromotor
Independent members:	
Prof.dr.ir. B. De Schutter,	Delft University of Technology
Prof.dr. A. Plaat,	Leiden University
Prof.dr. A. Valada,	University of Freiburg, Germany
Dr. H.C. van Hoof,	University of Amsterdam
Prof.dr.ir. J. Hellendoorn,	Delft University of Technology, reserve member

This work was supported by the European Union's H2020 project Open Deep Learning Toolkit for Robotics (OpenDR) under grant agreement No 871449.

Email: b.heijden@hotmail.com

Printed by: Gildeprint

Cover: Douwe Sebastiaan van der Heijden

ISBN 978-94-6518-019-9

An electronic version of this dissertation is available at
`https://repository.tudelft.nl/`.

All models are wrong, but some are useful.

George Box

Contents

Summary

Reinforcement learning has emerged as a promising approach for enabling robots to learn from interactions with their environments, without relying on predefined behaviors. However, robots face significant challenges when learning directly from real-world interactions. Real-world learning is time-consuming and resource-intensive, often requiring extensive data collection over long periods. Additionally, the risks involved in trial-and-error learning in physical settings are high, as faulty policies can lead to safety issues or system damage. Simulations offer a safer and more efficient alternative, allowing robots to learn in simulated environments at faster-than-real-time speeds. Despite these benefits, simulations often serve as imperfect approximations of reality. As a result, robots may learn behaviors that exploit simulation-specific quirks, which may not perform well in real-world settings, creating difficulties in transferring learned behaviors from simulation to real environments—a challenge known as the *sim-to-real gap*. Several factors contribute to the sim-to-real gap, such as unmodeled physical phenomena like friction and deformation, and the asynchronous nature of real-world systems that simulations often fail to capture accurately. Additionally, using separate software stacks for simulation and deployment can unintentionally lead to discrepancies. Finally, simulating at faster-than-real-time speeds with asynchronous frameworks that distribute computation across multiple cores may also introduce inaccuracies without proper synchronization.

This thesis focuses on improving simulation tools and methodologies to enhance the efficiency and effectiveness of learning-based approaches in robotics. The work addresses key trade-offs between flexibility, speed, and accuracy in robotic simulations, which are critical for successfully transferring learned policies from simulation to real-world environments. Additionally, it introduces a strategy to improve resilience, ensuring that learned behaviors are robust to irrelevant and unknown dynamics. By tackling these challenges, this thesis provides insights into the design of effective robotic simulators and presents contributions that help bridge the gap between simulated and real-world robotic learning.

A flexible simulation environment is essential to circumvent and abstract away inaccuracies, while concentrating the learning process on the relevant and realistic parts of the environment. Chapter 2 introduces EAGERx (Engine Agnostic Graph Environments for Robotics), a graph-based sim-to-real framework designed to enhance flexibility in robotic simulations. EAGERx allows for modular representation of tasks, enabling users to configure systems more flexibly that accomdates various state, action, and time-scale abstractions. A key component of EAGERx is a novel synchronization algorithm that supports the simulation of asynchronous, hierarchical systems at faster-than-real-time speeds. Additionally, the framework integrates delay simulation and domain randomization, which further enhances its ability to reduce the sim-to-real gap. The flexibility offered by EAGERx improves learning efficiency, as robots can more effectively focus on task-relevant dynamics. This is demonstrated in two robotic benchmark tasks, where EAGERx successfully reduces the sim-to-real gap.

Chapter 3 builds on the flexibility introduced in Chapter 2, focusing on simulation speed, which is critical for scaling learning efficiency in complex tasks. Graph-based simulations, while flexible, present challenges for parallelization, particularly when handling asynchronous interactions and delays. This chapter introduces a novel parallelization technique that constructs a *supergraph*, capturing all possible execution paths across parallel simulations. By minimizing redundant computations, this method enables more efficient parallel execution on accelerator hardware, significantly reducing training times without compromising simulation accuracy. The proposed approach extends the flexibility of graph-based simulations with efficient parallelization capabilities without sacrificing accuracy, providing an effective solution for graph-based simulators used in reinforcement learning tasks.

While flexibility and speed are crucial, the accuracy of simulations remains a key factor in closing the sim-to-real gap. Chapter 4 addresses this by focusing on improving simulation fidelity through better modeling of system dynamics with the addition of an explicit delay model. The chapter introduces a framework, REX (Robotic Environments with jaX), for the simultaneous estimation of dynamics and delays from real-world data, building on the graph-based architecture discussed in earlier chapters. The key innovation in this framework is the simulation of sensing, computation, and actuation delays, while also demonstrating delay compensation strategies to minimize their impact on the learned policies. Accurate modeling of real-world delays and dynamics significantly improves the quality of sim-to-real transfer, ensuring that robots perform more reliably when deployed in real environments.

The final core technical contribution of this thesis, detailed in Chapter 5, is the introduction of the DeepKoCo algorithm, which addresses the need for resilience in learned policies. From self-driving cars to vision-based robotic manipulation, emerging technologies are characterized by visual measurements of highly nonlinear physical systems. Unlike in highly controlled lab environments where any measured change is likely relevant, cameras in real-world settings are notorious for mainly capturing task-irrelevant information, such as, the movement of other robots outside of a manipulator's workspace or cloud movements captured by the cameras of self-driving cars. While flexibility, speed, and accuracy are essential for effective simulated learning, these are effects that may never be fully captured in simulations. Resilience to task-irrelevant dynamics is therefore crucial when deploying learned policies in real-world environments. DeepKoCo uses a lossy autoencoder to filter out irrelevant dynamics, allowing the robot to focus on task-relevant information. By learning a latent representation that prioritizes important dynamics, DeepKoCo enhances the robustness of learned behaviors. This ensures that the learned policies are not only accurate but also resilient to the distractions and uncertainties present in real-world applications.

In conclusion, this thesis presents a structured approach to balancing flexibility, speed, accuracy, and resilience in robotic simulators. The contributions, including the graph-based framework, parallelization techniques, and improved modeling of real-world dynamics with delays, provide a comprehensive toolkit for advancing reinforcement learning in robotics. The work demonstrates that achieving this balance is essential for closing the sim-to-real gap and enabling more efficient, robust robotic learning. Looking forward, future research could explore co-designing learning algorithms and simulation environments to further optimize the learning process. Additionally, decoupling estimation and

control tasks, or integrating parallelized simulations into online planning, hold promise for further improving both learning efficiency and resilience. The open-source tools developed as part of this thesis are made available to the robotics community, supporting further advancements in the field of reinforcement learning in robotics.

Samenvatting

Reinforcement learning lijkt een veelbelovende aanpak om robots zelfstandig te laten leren van interacties met hun omgeving, zonder gebruik te maken van voorgeprogrammeerd gedrag. Echter, zijn er aanzienlijke uitdagingen wanneer robots leren door middel van interacties in de echte wereld. Leren in de echte wereld is tijdrovend en vergt veel middelen, zoals het verzamelen van data over lange periodes. Bovendien zijn de risico's verbonden aan trial-and-error leren in de fysieke wereld hoog, aangezien foutief gedrag kan leiden tot gevaarlijke situaties en schade. Simulaties bieden een veiliger en efficiënter alternatief, waardoor robots kunnen leren in gesimuleerde omgevingen op snelheden die hoger zijn dan real-time. Ondanks deze voordelen zijn simulaties vaak onvolmaakte benaderingen van de realiteit. Als gevolg hiervan kunnen robots gedrag leren dat misbruik maakt van simulatiespecifieke eigenaardigheden, die niet goed werken in de echte wereld, wat moeilijkheden creëert bij de overdracht van geleerd gedrag van simulatie naar de echte wereld—een uitdaging die bekendstaat als de *sim-to-real gap*. Verschillende factoren dragen bij aan de sim-to-real gap, zoals niet-gemodelleerde fysieke fenomenen zoals wrijving en vervorming, en de asynchrone aard van echte systemen die simulaties vaak niet nauwkeurig vastleggen. Daarnaast kan het gebruik van aparte softwarestacks voor simulatie en de echte wereld onbedoeld leiden tot discrepanties. Ten slotte kan het simuleren op snelheden hoger dan real-time met asynchrone software die berekeningen over meerdere cores verdelen, ook onnauwkeurigheden introduceren zonder correcte synchronisatie.

Dit proefschrift richt zich op het verbeteren van simulaties en methodologieën om de efficiëntie en effectiviteit van op leren gebaseerde benaderingen in robotica te vergroten. Het werk behandelt belangrijke afwegingen tussen flexibiliteit, snelheid en nauwkeurigheid in robotsimulaties, die cruciaal zijn voor de succesvolle overdracht van geleerd gedrag van simulatie naar de echte wereld. Daarnaast introduceert het een algoritme om veerkracht te verbeteren, zodat geleerd gedrag robuust is tegen irrelevante en onbekende dynamica. Door deze uitdagingen aan te pakken, biedt dit proefschrift inzichten in het ontwerp van effectieve robotsimulatoren en presenteert het bijdragen die helpen de kloof tussen de gesimuleerde en echte wereld in robotleren te overbruggen.

Een flexibele simulatiewereld is essentieel om onnauwkeurigheden te omzeilen en te abstraheren, terwijl het leerproces wordt geconcentreerd op de relevante en realistische delen van de simulatie. Chapter 2 introduceert EAGERx (Engine Agnostic Graph Environments for Robotics), een graafgebaseerd sim-to-real applicatie ontworpen om de flexibiliteit in robotsimulaties te vergroten. EAGERx maakt een modulaire representatie van taken mogelijk, waardoor gebruikers systemen flexibeler kunnen configureren die verschillende toestand-, actie- en tijdschaal abstracties ondersteunen. Een belangrijk onderdeel van EAGERx is een nieuw synchronisatie-algoritme dat de simulatie van asynchrone, hiërarchische systemen ondersteunt op snelheden hoger dan real-time. Daarnaast integreert de applicatie tijdsvertraging simulatie en domein randomisatie, wat het vermogen om de sim-to-real gap te verkleinen verder vergroot. De flexibiliteit die EAGERx biedt, verbetert

de leerefficiëntie, omdat robots effectiever kunnen focussen op taakrelevante dynamica. Dit wordt gedemonstreerd in twee robotbenchmarktaken, waar EAGERx met succes de sim-to-real gap vermindert.

Chapter 3 bouwt voort op de in Chapter 2 geïntroduceerde flexibiliteit, met een focus op simulatiesnelheid, wat cruciaal is voor het opschalen van leerefficiëntie in complexe taken. Graafgebaseerde simulaties, hoewel flexibel, bemoeilijken parallellisatie, met name bij het simuleren van asynchrone interacties en tijdsvertragingen. Dit hoofdstuk introduceert een nieuwe parallellisatietechniek die een *supergraaf* construeert, die alle mogelijke uitvoeringspaden over parallelle simulaties omvat. Door redundante berekeningen te minimaliseren, maakt deze methode efficiëntere parallelle uitvoering op acceleratorhardware mogelijk, waardoor trainingstijden aanzienlijk worden verminderd zonder afbreuk te doen aan de simulatienauwkeurigheid. De voorgestelde aanpak breidt de flexibiliteit van graafgebaseerde simulaties uit met efficiënte parallellisatie mogelijkheden zonder nauwkeurigheid op te offeren, en biedt een effectieve oplossing voor graafgebaseerde simulatoren die worden gebruikt in reinforcement learning.

Hoewel flexibiliteit en snelheid cruciaal zijn, blijft de nauwkeurigheid van simulaties essentieel bij het dichten van de sim-to-real gap. Chapter 4 adresseert dit door zich te richten op het verbeteren van de simulatiegetrouwheid door betere modellering van systeemdynamica met de toevoeging van een expliciet tijdsvertragingsmodel. Het hoofdstuk introduceert een applicatie, REX (Robotic Environments with jaX), voor de gelijktijdige afschatting van dynamica en tijdsvertragingen op basis van echte data, voortbouwend op de graafgebaseerde architectuur die in eerdere hoofdstukken is besproken. De belangrijkste innovatie in deze applicatie is de simulatie van sensor, berekenings- en actuator tijdsvertragingen, terwijl ook tijdsvertragingcompensatie algoritmes worden gepresenteerd om hun impact op het geleerde gedrag van de robot te minimaliseren. Nauwkeurige modellering van realistische tijdsvertragingen en dynamica verbetert de kwaliteit van sim-to-real overdracht aanzienlijk, waardoor robots betrouwbaarder presteren wanneer ze in de echte wereld worden ingezet.

De laatste technische bijdrage van dit proefschrift, besproken in Chapter 5, is de introductie van het DeepKoCo-algoritme, dat inspeelt op de behoefte aan veerkracht in geleerd robot gedrag. Van zelfrijdende auto's tot het gebruik van camera's door robotarmen, worden opkomende technologieën gekenmerkt door visuele metingen van sterk niet-lineaire fysieke systemen. In tegenstelling tot sterk gecontroleerde labomgevingen waar elke gemeten verandering waarschijnlijk relevant is, staan camera's in de echte wereld erom bekend voornamelijk taakirrelevante informatie vast te leggen, zoals de beweging van andere robots in de omgeving of wolkenbewegingen vastgelegd door de camera's van zelfrijdende auto's. Hoewel flexibiliteit, snelheid en nauwkeurigheid essentieel zijn voor effectief gesimuleerd leren, zijn dit effecten die mogelijk nooit volledig kunnen worden vastgelegd in simulaties. Veerkracht tegen taakirrelevante dynamica is daarom cruciaal bij het inzetten van geleerd gedrag in de echte wereld. DeepKoCo gebruikt een verlieslatende autoencoder om irrelevante dynamica eruit te filteren, waardoor de robot zich kan concentreren op taakrelevante informatie. Door een latente representatie te leren die belangrijke dynamica prioriteert, verbetert DeepKoCo de robuustheid van geleerd gedrag. Dit zorgt ervoor dat het geleerde gedrag niet alleen nauwkeurig is, maar ook veerkrachtig is tegen de afleidingen en onzekerheden die aanwezig zijn in de echte wereld.

Concluderend presenteert dit proefschrift een gestructureerde aanpak voor het balanceren van flexibiliteit, snelheid, nauwkeurigheid en veerkracht in robotsimulatoren. De bijdragen, waaronder graafgebaseerde applicaties, parallellisatietechnieken en verbeterde modellering van dynamica met tijdsvertragingen, bevorderen het gebruik van reinforcement learning in robotica. Het werk demonstreert dat het bereiken van deze balans essentieel is voor het dichten van de sim-to-real gap en het mogelijk maken van efficiënter, robuuster robotleren. Vooruitkijkend zou toekomstig onderzoek kunnen verkennen hoe leeralgoritmen en simulatiewerelden kunnen worden co-ontworpen om het leerproces verder te optimaliseren. Daarnaast bieden het ontkoppelen van schattings- en controletaken, of het integreren van geparallelliseerde simulaties in online planning, perspectieven voor verdere verbetering van zowel leerefficiëntie als veerkracht. De open-source applicaties die als onderdeel van dit proefschrift zijn ontwikkeld, worden beschikbaar gesteld aan de robotica-gemeenschap ter ondersteuning van verdere vooruitgang op het gebied van reinforcement learning in robotica.

1

Introduction

The evolution of technology has centered on automating tasks that were once performed manually. Early examples of automation include devices like the water mill and windmill, which converted natural forces into mechanical energy to reduce human labor in tasks such as grinding grain or pumping water. With the onset of the Industrial Revolution, more complex systems like the steam engine-powered assembly lines further advanced automation, significantly increasing efficiency in manufacturing processes [1]. With advances in computing and sensor technologies, automation has progressed from mechanical systems like the water mill to sophisticated autonomous robots. These modern robots are capable of not only performing predefined tasks but also gathering sensory data, processing it, and making decisions autonomously in real time. This shift from manually controlled machines to intelligent systems marks a key development in automation, where the primary challenge is now designing mechanisms that enable robots to effectively map sensory inputs to real-time actions in dynamic environments.

1.1 Paradigm of Learning-Based Robotics

Traditionally, robots were programmed using detailed knowledge of physics and control theory, allowing designers to pre-program specific behaviors to close the action loop. This approach relies heavily on the designer's ability to anticipate every potential scenario the robot might encounter, and in practice, engineers often simplify problems by linearizing dynamics [2] or assuming specific noise distributions [3] to make them mathematically tractable. While effective in controlled and predictable environments, these assumptions can significantly limit performance, particularly in complex or highly nonlinear settings [4].

A useful analogy can be found with early chess computers. Chess, despite having relatively simple rules, is extremely difficult to play at a competitive level. Initially, chess computers relied on predefined strategies developed by grandmasters to guide their search algorithms under constrained computational resources [5]. Similarly, robots were programmed with control strategies based on simplified physical models, such as Model Predictive Control using linearized dynamics, to enable fast, real-time responses given the robot's limited processing capabilities [6]. In both cases, robots and chess computers

were limited by their reliance on human-designed heuristics, overconstraining the type of solutions that could be discovered.

This limitation was overcome in chess with the development of learning-based approaches, exemplified by AlphaZero [7], a program that fundamentally changed how chess computers operate. Instead of relying on predefined human strategies, AlphaZero learned to play by playing against itself, requiring minimal assumptions about strategy[1]. Designers only needed to encode the rules of chess, and through repeated self-play, AlphaZero discovered strategies far beyond the capabilities of traditional, human-designed systems, eventually achieving superhuman performance[2].

This paradigm shift in chess mirrors the ongoing shift in robotics [10]. Rather than depending on predefined models and strategies derived from extensive expert knowledge, learning-based methods—particularly reinforcement learning [11]—allow robots to discover optimal control policies through trial-and-error interactions with their environment. This bypasses the need for simplifying assumptions about system dynamics, making learning-based approaches particularly powerful in scenarios where environments are highly nonlinear.

If we could transfer this learning capability to the physical world, it could revolutionize robotics. Robots, without relying on predefined strategies or simplifying assumptions, could autonomously discover optimal ways to perform tasks, even in highly dynamic and complex environments. This would enable them to continuously improve their performance and adapt in real time, unlocking new possibilities in areas such as manufacturing [12], healthcare [13], and exploration [14].

1.2 Hurdles of Real-World Application

Transferring the learning-based approach from a game like chess to the physical world poses significant challenges, primarily due to the intrinsic differences between digital simulations and real-world interactions. The rate at which a system like AlphaZero can accrue experiences in chess through simulations vastly exceeds the pace of real-world learning. In practical terms, a robot's learning in the physical world is bound by real-time constraints—akin to the difference between the two hours it might take to play a single chess game versus the millions of games that can be simulated in parallel within the same timeframe. Moreover, the stakes in real-world applications are vastly higher. In chess, experimenting with different strategies merely risks losing a game, providing valuable experience without real-world consequences. In contrast, applying a trial-and-error learning approach in physical environments, especially those involving robots, can have serious safety and cost implications.

1.3 Simulating Success

In this regard, robotic simulators offer significant advantages by addressing both the safety and time constraints inherent to physical world experimentation. They enable

[1] Self-play was not a novel concept introduced by AlphaZero. For example, TD-Gammon [8] utilized self-play and achieved master-level performance in backgammon.

[2] The approach of pure self-play, without human knowledge, was first shown to reach superhuman performance in Go by AlphaGo Zero [9].

rapid, parallel simulations analogous to AlphaZero's chess games, accelerating the learning process without the risks associated with real-world trials [15–18]. However, while chess can be perfectly simulated due to its defined rules and deterministic outcomes, the real world is far more complex and unpredictable, making perfect simulation impossible. These inaccuracies in simulation can lead robots to develop strategies that effectively exploit the specific limitations of the simulated environment rather than addressing the underlying task in a universally applicable manner [19]. As a result, these learned behaviors, while successful within the context of the simulation, may prove ineffective or counterproductive when the behavior is transferred to the real world. The robot, having optimized its actions for the quirks and artifacts of the simulation, might be ill-prepared for the nuances and unforeseen variables of real environments, undermining the efficacy and safety of its real-world operations.

Acknowledging the limitations and discrepancies inherent in simulations, it is crucial to recognize the flexibility available in designing robotic tasks and learning environments. Unlike the fixed setting of chess, the real world allows roboticists the freedom to modify the learning context. This adaptability allows roboticists to "change the game" by modifying learning environments more closely to realistic conditions or even adjust the tasks to circumvent certain complexities, thereby mitigating the risk of learning unrealistic or ineffective behaviors [20]. By considering the design of the simulator as an integral part of the problem-solving process, roboticists can tailor simulated learning experiences to align more closely with the real-world. This holistic approach, viewing both the learning algorithm and the simulator design as malleable and interdependent components, indeed provides a powerful tool for overcoming challenges in robot learning. In discussing simulator design, three key considerations are highlighted: the flexibility to customize learning scenarios, the accuracy of the simulation, and the speed of the simulation.

Navigating the interplay between flexibility, accuracy, and speed is pivotal in designing effective robotic simulators. While flexibility allows for tailoring learning scenarios to emphasize relevant aspects of the environment [20], too much abstraction risks limiting the agent's ability to discover optimal strategies and undermines the accuracy required for real-world applicability [19]. Hence, accuracy, in turn, is crucial for applying learned behaviors in real-world scenarios, but it should not unduly slow down simulations [21], especially when using parallelization to accelerate learning [22]. These considerations underscore a critical challenge: *designing a simulator structure that effectively balances flexibility, accuracy, and speed.* Such a balance is essential to optimize learning outcomes without incurring prohibitive computational costs or sacrificing the simulation fidelity necessary for effective real-world transfer.

1.4 This Thesis

This sets the stage for the central research question of this work: *How can we design a simulation-based robot learning framework that balances the trade-offs between flexibility, speed, and accuracy, while ensuring resilience to the complexities and unpredictability of real-world environments?*

The design is built around balancing three key trade-offs: flexibility, accuracy, and speed. These elements are fundamental to creating an efficient and scalable learning framework that can be broadly applied across different robotic tasks. The focus on simulator design

1

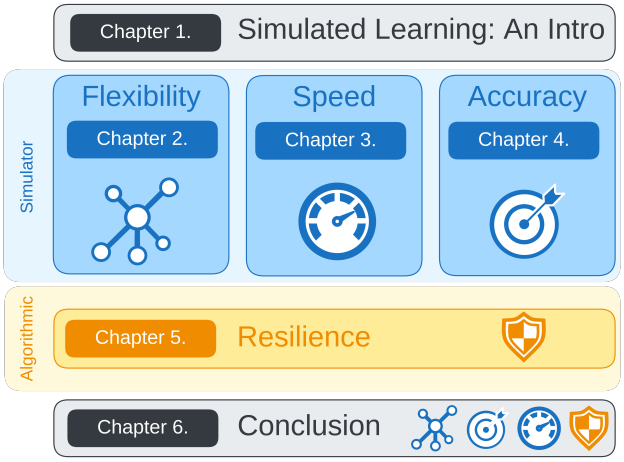


Figure 1.1: Thesis overview. Chapter 1 introduces the challenges in simulated robot learning and emphasizes the need for a holistic approach that views both the algorithm and simulator design as interdependent components for effective robot learning. Chapters 2, 3, and 4 focus on key trade-offs in simulation design—flexibility, speed, and accuracy—each addressing a fundamental aspect of creating scalable and efficient simulators. Chapter 5 explores algorithmic strategies to improve resilience, ensuring learned behaviors are robust to irrelevant dynamics and unknown factors in real-world environments. Chapter 6 concludes with a summary of findings and discusses future directions for enhancing both simulator design and resilience strategies.

addresses many of the core challenges in robot learning by allowing for adaptable task scenarios (*flexibility*), rapid learning (*speed*), and accurate modeling of dynamics (*accuracy*). However, real-world environments are often filled with noise and irrelevant dynamics that simulations may never fully capture. To address these challenges, it is critical not only to optimize simulators but also to consider strategies that make the learning process resilient to such unknown and irrelevant factors (*resilience*). By integrating approaches that expose the learning process to such irrelevance, the system can better adapt to unstructured environments. This final component ties the simulator's design to real-world robustness, ensuring that learned behaviors remain effective even when faced with the unpredictable elements of real-world scenarios. A visual summary of the topics discussed in this thesis is given in Fig. 1.1.

Flexibility Flexibility is crucial in the design of robot simulators meant to accommodate diverse robotic systems and tasks. Firstly, robotic tasks often involve evolving requirements, necessitating a framework that can adapt to changing needs. Secondly, finding the right level of abstraction for effective learning requires a flexible design. State, action and temporal abstractions are crucial for concentrating the learning process to the parts that cannot be solved by simpler methods, enhancing the efficiency of learning. Moreover, these abstractions can help mitigate inaccuracies in simulations by abstracting away irrelevant details that may not be accurately modeled.

To address these challenges, Chapter 2 introduces EAGERx (Engine Agnostic Graph

Environments for Robotics), a graph-based sim2real framework designed to provide the flexibility necessary to handle a wide range of robotic tasks. Graphs are particularly well-suited for accommodating flexibility, as they allow for modular representation of tasks and systems. Each component, such as sensors, actuators, and controllers, is represented as a node in the graph, and the relationships between these components can be easily reconfigured by adjusting the graph's structure. Nodes can be added, removed, or modified to adapt to different task requirements, system setups, and learning objectives. The graph-based framework also enables strict grouping and swapping of simulation and real-world components, enabling a unified software pipeline for both real and simulated robot learning. It also allows users to switch between different simulation engines, depending on the specific needs of the task at hand. A novel synchronization algorithm is proposed to ensure that the simulator accurately models the asynchronous, hierarchical nature of real-world systems, ensuring accurate simulation at faster than real-time simulation speeds. We demonstrate the efficacy of EAGERx in accommodating diverse robotic systems and maintaining consistent simulation behavior across five different systems: a pendulum, a manipulator, a quadrotor, a quadruped, and a swimming pool environment.

Speed Speed in simulated learning brings critical benefits such as faster training times and the ability to conduct extensive hyperparameter tuning, making parallelization a key strategy for achieving these advantages. Traditional reinforcement learning setups assume a single, synchronized environment that interacts with the agent in a step-by-step manner, allowing for straightforward parallelization [23]. However, the hierarchical and asynchronous nature of real-world systems presents a significant challenge to this approach [24]. When employing a graph-based simulation method to enhance flexibility and accuracy, each simulation step becomes a diverse mix of computation blocks from various nodes operating at different time scales. This leads to irregular execution paths which complicate the parallelization process. Running multiple instances of a graph-based simulation in parallel can become inefficient as these irregular paths often require serialization, reducing the effectiveness of GPU utilization [25].

In Chapter 3, we introduce a solution that efficiently parallelizes these graph-based simulations on accelerator hardware. Our method extends existing accelerated physics simulations by integrating them as nodes in a graph-based simulation framework that enables latency simulation capabilities through the construction of a *supergraph*. This supergraph captures all data dependencies across the parallelized simulation steps, ensuring both the accuracy and efficiency of the simulations. By optimizing for the smallest possible supergraph, we reduce redundant computations, maintaining high simulation speeds without compromising the fidelity of real-world dynamics. We validate our approach on two real-world robotic systems and demonstrate superior performance over baseline methods. Additionally, we conduct a scalability analysis on two large-scale system topologies: vehicle-to-vehicle platooning [26] and unmanned aerial vehicle swarm control [27].

Accuracy Achieving high fidelity in simulations is crucial for effective sim2real transfers, yet discrepancies between simulated and real environments often hinder this. Inaccuracies from inaccurate dynamics and latencies can cause robots to learn policies that perform well in simulation but fail to transfer to the real world.

1

In Chapter 4, we addresss this by introducing a framework, REX (Robotic Environments with jaX), that enhances simulation fidelity by estimating system dynamics and delays from real-world data. The framework builds on the graph-based model from Chapter 2 and leverages the parallelization strategy from Chapter 3. The framework's innovation lies in its ability to simulate asynchronous, hierarchical systems by explicitly modeling computation, communication, actuation, and sensing delays, while incorporating delay compensation strategies for improved sim2real transfer. We validate our framework on two real-world systems, demonstrating its effectiveness in improving sim-to-real performance by accurately modeling both system dynamics and delays. A pendulum swing-up task illustrates how neglecting delay simulation can impair policy transfer even in seemingly simple scenarios, highlighting the need for delay-aware approaches. The quadrotor task further demonstrates the framework's scalability to more complex robotic systems.

Resilience From self-driving cars to vision-based robotic manipulation, emerging technologies are characterized by visual measurements of highly nonlinear physical systems. Unlike in highly controlled lab environments where any measured change is likely relevant, cameras in real-world settings are notorious for mainly capturing task-irrelevant information, such as, the movement of other robots outside of a manipulator's workspace or cloud movements captured by the cameras of self-driving cars. While flexibility, speed, and accuracy are essential for effective simulated learning, these are effects that may never be fully captured in simulations. Resilience to task-irrelevant dynamics is therefore crucial when deploying learned policies in real-world environments. To bridge the gap between simulation and reality, it is important to incorporate resilience into the learning process, enabling agents to handle unknown or irrelevant factors without sacrificing performance. By preparing the system to adapt to such uncertainties, we can improve the robustness of learned behaviors, ensuring their effectiveness in real-world scenarios.

In Chapter 5, we introduce an algorithmic approach to enhance resilience in learning-based robotics. We present DeepKoCo, a model-based reinforcement learning agent designed to enhance resilience by focusing on task-relevant dynamics. Using a lossy autoencoder [28], DeepKoCo learns a latent representation that prioritizes dynamics critical to the task, while disregarding irrelevant information. This enables the agent to plan in the latent space using efficient linear control methods, such as model predictive control [6]. By filtering out distractions and focusing on essential dynamics, DeepKoCo improves the robustness of learned policies, making them robust against irrelevant factors that may appear in real-world scenarios. We demonstrate the success of our approach on two simulated control tasks, showing that it is more robust to irrelevant dynamics than baseline methods.

Summary of Contributions This thesis advances the state-of-the-art in simulation-based robot learning through the following contributions:

- **Flexibility:** Novel graph-based sim2real frameworks (*EAGERx* and *REX*) are introduced, integrating with multiple simulation engines and enabling state, action, and time abstractions. These frameworks offer greater flexibility compared to more rigid, single-engine solutions.

- **Accuracy:** Methods are developed to accurately model, estimate, and compensate for real-world latencies and asynchronous dynamics, improving sim2real transfer beyond the capabilities of existing delay-agnostic simulators.
- **Speed:** A parallelization approach (*supergraph* method) for graph-based simulations is proposed, increasing GPU efficiency without sacrificing fidelity in simulating asynchronous and hierarchical systems. This improves upon state-of-the-art parallelization schemes that assume strictly synchronized environments.
- **Resilience:** A model-based reinforcement learning approach (*DeepKoCo*) is introduced, employing a lossy autoencoder to learn task-relevant latent representations. By filtering out irrelevant dynamics, this approach enhances robustness in real-world deployments compared to conventional reinforcement learning approaches.

These contributions address the central research question, demonstrating how graph-based, data-driven, and accelerated simulator frameworks can bridge the gap between simulation and reality. In doing so, this thesis provides novel insights and tools that expand the state of the art in learning-based robotics, paving the way for safer, more adaptable, and more effective robotic systems. Chapter 6 summarizes this thesis with an overview of the key findings and discusses potential directions for future research in simulation-based robot learning.

2

Flexibility: A Graph-Based Simulator

One of the key limitations in robotic learning is the need for flexibility in simulation environments to accommodate diverse robotic systems and tasks. Traditional frameworks often lack the adaptability required for evolving requirements, constraining their effectiveness in more dynamic or varied scenarios.

This chapter addresses this gap by introducing EAGERx, a graph-based framework designed to provide the flexibility necessary for sim2real transfers. The framework allows users to modularly represent robotic components, integrate multiple simulation engines, and handle asynchronous, hierarchical systems, thereby facilitating a seamless transition between simulated and real-world environments.

This chapter is partly based on *B. van der Heijden*[*], *J. Luijkx*[*], *L. Ferranti, J. Kober, and R. Babuska, (2024). "Engine Agnostic Graph Environments for Robotics (EAGERx): A Graph-Based Framework for Sim2real Robot Learning", IEEE Robotics and Automation Magazine (RAM)* [29].

[*]Equal contribution.

2.1 Introduction

Transferring control policies trained in simulation to the real world, known as sim2real, has gained considerable interest in the field of robotics due to its potential to address complex tasks with remarkable efficiency [22, 30, 31]. Simulations offer a safe, cost-effective, and controlled environment for training and testing robotic algorithms, allowing roboticists to refine their models and controllers without the risks and expenses associated with real-world experimentation. The sim2real approach, however, faces challenges due to the *sim2real gap*, that is, unaccounted discrepancies between simulation and reality. These disparities may stem from *inaccurate modeling of physical phenomena* (e.g., friction, deformations, and collisions) or from the use of *separate software implementations* for reality and simulation, which may lead to unintended mismatches as depicted in Fig. 2.1. Another subtle but significant source of discrepancy is the *asynchronous nature* of robotic systems [32]. While robotic systems are typically simulated sequentially [23], sensing, computation, and acting happen concurrently in reality. Disregarding these differences can be detrimental to the real-world performance of a policy trained in simulation.

Inaccurate modeling of physical phenomena in simulation is typically mitigated by domain randomization [31]. However, this approach can make the simulation more challenging, which may lead to longer training times and suboptimal policies. Reformulating the task to the right level of abstraction may be more effective to alleviate the sim2real gap if the abstraction captures the task and can be extracted accurately both from simulated and real data [19]. Abstractions can take various forms, such as action abstraction that simplifies control issues using high-level actions [33], time-scale abstraction that uses macro-actions for multi-scale planning and learning [34], and state abstraction that condenses raw sensor data into key features [19]. Therefore, existing sim2real frameworks [35–37] have exploited the multi-rate graph-based design of ROS [24] to obtain a unified software pipeline that allows for the integration of various kinds of abstractions. However, these frameworks restrict users to the Gazebo simulator [38], which can be limiting as different tasks may require specific types of simulators. Additionally, these frameworks fall short in synchronizing components that operate in parallel within the simulation. At faster-than-real-time simulation speeds, this can exacerbate communication and processing delays, leading to inconsistencies, inaccuracies, and potential system instability. Such amplified delays can compromise the proper functioning of the simulated system, rendering learned policies ineffective when transferred to real-world environments. Conversely, naive synchronization may also widen the sim2real gap if it overlooks the concurrent nature of sensing, computation, and acting in reality.

In addition to ROS-based frameworks, existing robot learning frameworks provide integration of abstractions through a modular design and unified framework, often coupled with a specific simulator. Notable examples include Isaac Orbit [39] and Drake [40]. Isaac Orbit is a modular robot learning framework built on top of the Isaac Sim simulator [41], offering benchmarks and readily available robot models for convenient experimentation. On the other hand, Drake is a model-based framework combining a multibody dynamics engine with a systems approach and optimization framework [40]. However, these frameworks are tied to a single simulator, while various robot simulators are available, each with its own strengths and weaknesses. Existing robot learning frameworks lack the flexibility to choose a simulator or leverage various simulators' strengths.

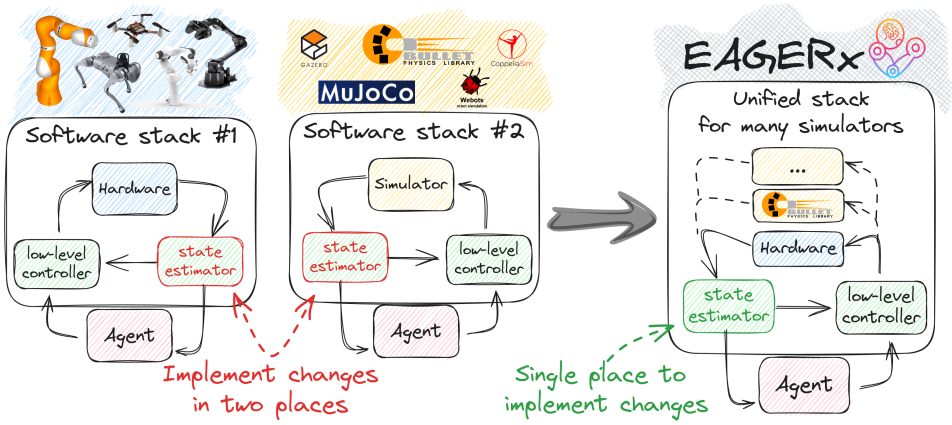


Figure 2.1: Our framework offers a unified software pipeline for both simulated and real robot learning. It can support various simulators and aids in integrating state, action and time-scale abstractions.

The main contribution of this chapter is EAGERx (Engine Agnostic Graph Environments for Robotics), that is, a robot learning framework with a unified software pipeline compatible with both simulated and real robots that supports the integration of various abstractions and simulators as depicted in Fig. 2.1. EAGERx introduces a novel synchronization protocol that coordinates inter-node communication based on node rates and anticipated delays. By simulating delays, our protocol maintains asynchronous robotic system relationships *synchronously*, preserving the benefits of modular and synchronous simulation. Contrasting with sequential simulation, the protocol permits nodes to transmit messages asynchronously and perform tasks without waiting for immediate responses, thereby accelerating the simulation and allowing nodes to progress based on their processing capabilities and data availability. EAGERx is Python-based and offers high simulation accuracy without compromising speed, native support for domain randomization and delay simulation, and a modular structure for easy manual reset procedures and prior knowledge integration. Our framework features a consistent interface, an interactive GUI, continuous integration with tests covering 94% of the code, and comprehensive documentation, including interactive tutorials, easing new user adoption. The documentation, tutorials, and our open-source code can be found at https://eagerx.readthedocs.io. A motivational video for our approach is included as supplementary material [42].

In summary, we make four key contributions:

C1 A synchronization protocol that ensures consistent simulation behavior even beyond real-time speeds.

C2 A modular design that can support various robotic systems and state, action, and time-scale abstractions.

C3 An agnostic design that allows compatibility with multiple engines.

C4 The integrated delay simulation and domain randomization features in EAGERx can narrow the sim2real gap.

The remainder of the paper is structured as follows. Sec. 2.2 provides a high-level introduction to the framework. Sec. 2.3 elaborates on a key low-level component of the framework, i.e., its novel synchronization algorithm. Sec. 2.4 provides an extensive experimental evaluation to show the applicability of EAGERx for sim2real robot learning. Sec. 2.5 shows the framework's utility beyond sim2real robot learning in two real-world robotic use cases. Sec. 2.6 compares EAGERx and existing frameworks, and Sec. 2.7 concludes the paper.

2.2 Framework

This section provides an overview of EAGERx. Sec. 2.2.1 outlines the framework's main components. Then, Sec. 2.2.2 discusses the package management system promoting modularity and versioned compatibility. Finally, Sec. 2.2.3 discusses the framework's capabilities for domain randomization, simulator augmentation, and delay simulation, which are essential to minimize the sim2real gap.

2.2.1 Agnostic Framework

First, we provide a brief overview of the main components, followed by a code example.

Graph

EAGERx processes are represented as nodes within a graph structure, linked by directed edges from a node's output to one or multiple node inputs. Nodes communicate via edges by exchanging messages. This versatile decentralized architecture, ideal for networked hardware and off-board computer interactions, is especially useful for robotics.

Node

Nodes are central to EAGERx, representing individual processes that execute concurrently. Each node begins a new episode with a user-defined reset that sets its initial state, followed by the execution of user-defined code, termed a *callback*, at a predetermined rate. These callbacks determine the node's functionality and define how inputs from other nodes are transformed into outputs that are, in turn, sent as output to subsequent nodes. A typical robotic system usually consists of many such interconnected nodes. For instance, one node may be responsible for capturing camera images, another for localization using these images, and yet another for directing the robot's movement based on the localization data.

Nodes can be launched in various ways, according to their operational needs. For example, *CPU-bound nodes*, which are computationally intensive, benefit from being launched as subprocesses. This approach leverages multi-processing to bypass the limitations imposed by Python's Global Interpreter Lock (GIL), thus enhancing computational efficiency. In contrast, *I/O-bound nodes*, which primarily handle input/output operations, are more efficiently launched as separate threads. This minimizes the overhead associated with message serialization, streamlining communication. Furthermore, EAGERx facilitates *distributed computing* by enabling nodes to be launched as external processes on different machines. This feature allows for the distribution of computational loads across a network, optimizing the overall performance of the robotic control system.

Object

EAGERx objects enable flexible node replacement when transitioning a robotic system from simulation to reality. For instance, in reality, nodes for extracting sensor data from a physics engine become obsolete, requiring replacement with nodes interfacing robot hardware. EAGERx objects accommodate this adaptability.

Objects define abstract inputs and outputs, as well as subgraphs for each supported physics engine. Users can add objects to graphs (Fig. 2.2a), and establish connections between nodes and objects. Upon selecting a physics engine, abstract objects are replaced by corresponding subgraphs (Fig. 2.2b, Fig. 2.2c), rendering the node and object graph *engine-agnostic* (Fig. 2.2a), as it supports multiple physics-engines. Notice how the framework treats reality as just another physics engine. Practically, objects represent entities interacting directly with the physical environment. For instance, a robot may have an abstract input and output for its motors and encoders, respectively. Depending on the chosen physics engine, the robot's subgraph comprises nodes interfacing with real hardware or nodes communicating with a simulator.

The Object's design also accommodates the difference in available data between simulators and real-world hardware. They enable the definition of simulation-specific outputs, such as data exclusive to simulators, and inputs like randomized external disturbances, that can be used to enhance policy robustness. Users can easily configure these elements, selecting or deselecting them as needed, to ensure compatibility across different physics engines, thereby adapting the node and object graph for diverse simulation and real-world scenarios.

Engine

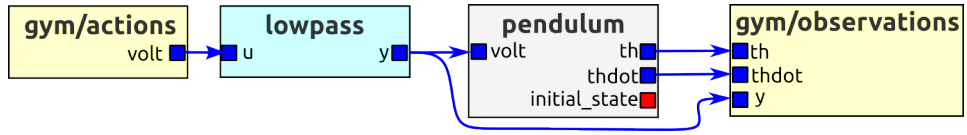
Physics-engines (e.g., PyBullet [43], Gazebo [38]) are interfaced by a special node called the *engine*. The engine initiates the physics engine, adds 3D meshes, and sets dynamic parameters (e.g., friction coefficients). It controls time passage and its rate defines the simulation step size.

Backend

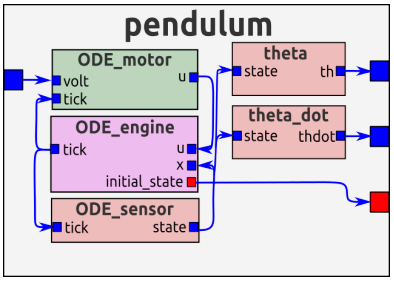
Node processes, launched in various ways (i.e., subprocess, multi-threaded, distributed), communicate through edges and interact with a collective database called the parameter server. The backend facilitates low-level node-to-node communication (i.e., establishing connections and the serialization of messages) for every edge and controls the parameter server. EAGERx supports two backends (i.e., ROS1, SingleProcess), with an abstract backend API allowing users to implement custom backends. Defined graphs can be initialized as distributed networks of subprocesses or run in a single process. EAGERx provides an abstraction layer over ROS, adding key features for robot learning such as synchronized faster-than-real-time simulation, domain randomization, and delay simulation.

BaseEnv

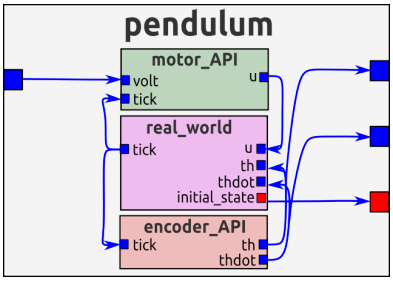
EAGERx favors composition over inheritance as a design principle because robotic systems are more naturally constructed from various components than by finding commonalities and using inheritance. EAGERx environments consist of an engine, backend, and graph, which is composed of nodes and objects. This design promotes code reuse and handles future requirement changes better than an inheritance-based environment. Nodes operating



(a) Agnostic graph



(b) ODE subgraph



(c) Real-world subgraph

Figure 2.2: (a) Displays the engine-agnostic graph of the pendulum environment from Fig. 2.3 as generated by the GUI. The engine-specific subgraphs for replacing the object (i.e., pendulum) are depicted for the ODE (b) and real-world (c) engines. The yellow nodes, split for visualization clarity, symbolize the agent's actions and observations. Blue squares represent I/O channels, while red squares indicate node states and/or parameters that can be randomized at the start of an episode.

within the graph of an EAGERx environment support multi-processing, thus enabling efficient parallel operations. Additionally, EAGERx facilitates vectorization across multiple environments, thereby enhancing the system's scalability and performance capabilities. BaseEnv conforms to the OpenAI Gym interface [23]. The reset method initializes episodes by setting the aggregate initial state of all graph nodes, enabling domain randomization over any registered node state, and returns the first observation. Users then determine actions, which are relayed to connected nodes through the *step* method.

Fig. 2.3 showcases the steps to create an environment using EAGERx for the pendulum swing-up problem, a classic problem in reinforcement learning [23]. It begins with the creation of a *pendulum* object and a *lowpass* node to filter the agent's actions, thereby reducing wear and tear on the system (lines 1-4). Subsequently, an *agnostic* graph is constructed in which the various components are connected, anticipated delays are specified for simulation, and cyclical connections are handled (lines 6-16). The environment is set up with the *OdeEngine* physics engine and a *SingleProcess* backend (lines 18-23). Equally, the *RealEngine* could be used to switch to real-world scenarios. Following initialization, an interaction is implemented by sampling an action and applying it to the environment (lines 25-30), with the environment being cleanly shut down at the end (line 31).

2.2.2 Support

Robotic system design often involves multiple cycles of design, implementation, evaluation, and refinement. EAGERx supports the users as follows:

```python
from .tutorials.pendulum import Pendulum # Make object
o = Pendulum.make(name="pendulum")
from .tutorials.low_pass import LowPass # Make node
n = LowPass.make(name="lowpass", rate=15, cutoff=7)

from eagerx import Graph # Make `agnostic` graph
g = Graph.create([o, n])
g.connect(action="volt",      target=n.inputs.u)
g.connect(source=n.outputs.y, target=o.actuators.volt,
          delay=0.1)  # Simulates actuator delay
g.connect(source=n.outputs.y,      observation="y",
          skip=True) # Resolves cyclic dependency
g.connect(source=o.sensors.th,     observation="th",
          window=2)  # Use last 2 sensor readings
g.connect(source=o.sensors.thdot, observation="thdot",
          window=2)  # Use last 2 sensor readings

from eagerx_ode.engine import OdeEngine # Select engine
e = OdeEngine.make(rate=30,
                   real_time_factor=0, # 0 -> unlimited
                   sync=True) # toggles synchronization
from eagerx.backends.single_process import SingleProcess
b = SingleProcess.make()  # Make backend

from .tutorials.env import CustomEnv # Make env
env = CustomEnv(g, ode, b, name="env_id", rate=30)

obs, info = env.reset()           # Start a new episode
a    = env.action_space.sample()    # Select an action
obs, reward, terminated, truncated, info = env.step(a)
env.shutdown() # Release resources
```

Figure 2.3: Environment creation for the swing-up problem.

Visualization Tools

EAGERx offers interactive visualization tools that aid in understanding and debugging robotic systems. Users can visualize the graph of nodes and inspect the parameter specifications of individual nodes with EAGERx's interactive GUI. The ability to visualize a complex robotic system is a powerful tool for debugging and understanding the system's behavior. Example visualizations of the GUI are shown in Figures 2.2a, 2.8a, and 2.8b.

Package Management

EAGERx incorporates a package management system that fosters modularity, versioned compatibility, and automated unit tests covering 94% of the code. This system allows users to easily share, import, and reuse code modules in different projects. By promoting modular design, EAGERx enables users to build complex robotic systems by combining smaller, well-tested components.

Onboarding Resources

EAGERx provides comprehensive onboarding resources, including interactive tutorials, code samples, and documentation, to help users quickly learn and adopt the framework.

2.2.3 Mitigating the Sim2Real gap

To address the sim2real gap, EAGERx's modular design enables manual reset routines, simulator augmentation, and supports domain randomization and delay simulation.

While resetting simulations is straightforward, real-world resets demand meticulously crafted routines to revert the system to its initial state. To this end, specialized reset nodes can be integrated into the graph, simplifying the real-world reset process between episodes.

These nodes might execute procedures requiring human interaction or engage safety filters within the graph. Operational only during reset phases, they remain inactive during regular episodes.

Simulator augmentation in EAGERx enables the integration of custom models, capturing complex dynamics absent in standard simulations. For instance, in [22], augmenting the simulator with a custom actuator model was key to successful sim2real transfer. This flexibility in EAGERx enhances simulation accuracy and fidelity, thus facilitating a more effective sim2real transition.

EAGERx enables domain randomization by varying simulation parameters such as object shapes and lighting [31]. Within this framework, nodes can register any parameter as a state, enabling its randomization via the *reset* method of the environment.

Delay simulation is enabled by our synchronization protocol discussed in Sec. 2.2.2, and emulates communication latency and computational delays encountered in real-world systems, yielding a more accurate simulation. Delays can be implemented across any graph edge, encompassing edges between nodes and objects, thus simulating sensor and actuator delays as demonstrated in Fig. 2.3, line 10.

2.3 Synchronization

Parallel computation, used in robotic system simulations via ROS [24] in existing sim2real frameworks [35–37], can increase simulation speeds. When run at faster-than-real-time speeds, however, these frameworks suffer from unsynchronized parallel components, unintentionally widening the sim2real gap. Here, the individual computation delays become more pronounced relative to the accelerated simulation clock. Without suitable synchronization at high speeds, certain components may struggle to match pace and gradually fall out of sync, leading to a deviation in the simulation from its real-world counterpart. Consequently, the learned control policy's performance may deteriorate, as it could receive outdated or mismatched observations, yielding actions based on inaccurate data. This may render the learned policies ineffective when transferred to the real-world environment.

2.3.1 Protocol

We developed a synchronization protocol for each of the nodes representing the robotic system that enables parallel computation and minimizes additional message-passing overhead, thereby enhancing system efficiency and accuracy. This protocol ensures that each node's callback, a user-defined code block executed at a predetermined rate for processing inputs and generating outputs, is triggered under the right conditions. Properly constructed communication patterns and protocols can achieve global synchronization without a central coordinator, whereby each node proceeds with its tasks once necessary input data or conditions have been satisfied.

Each node is launched as a subprocess that runs a local protocol version, depicted in Alg. 1. The conditions for a node to proceed with the next callback are based on the expected ordering of events, as dictated by assumed rates and delays of the system (lines 5-9). Executed with an event loop thread and dedicated input channel threads, the protocol compares received and expected message counts for input channels before

executing subsequent callbacks (lines 10-13). This comparison informs whether a node proceeds with the next callback or awaits more messages. Nodes perform tasks based on the protocol's decision and asynchronously transmit output to connected nodes (line 14). Only upon completion of the previous callback or receipt of a new input channel message does the event-driven protocol evaluate conditions for the subsequent callback (line 17). Consequently, task execution is entirely independent of any global clock or synchronization messages, thus minimizing additional message-passing overhead.

Algorithm 1: Synchronization protocol executed by each node

Input: node rate f_n, input rates f_i, input delay τ_i, input channels $i \in \mathcal{U}$, output channels $j \in \mathcal{Y}$
Output: Processed data sent to downstream nodes

```
1  k ← Initialize callback index to 0
2  B_i ← Initialize empty buffers for every input channel i
3  Start eventLoopThread
4  Start inputChannelThread for every i ∈ U
5  eventLoopThread:
6      foreach i ∈ U do
7          if channel i is cyclical then
8              δ_i ← Expected message count (Alg. 3)
9          else
10             δ_i ← Expected message count (Alg. 2)
11     if δ_i ≤ size(B_i) for every i ∈ U then
12         foreach i ∈ U do
13             u_{i,k} ← Pop last δ_i messages from B_i
14         y_k ← Run callback with inputs u_{i,k}, ∀i ∈ U
15         Send y_k to all output channels j ∈ Y
16         k ← Increment callback index to k + 1
17         Trigger event on eventLoopThread
18     WaitForEvent
19 inputChannelThread i:
20     B_i ← Buffer received message
21     Trigger event on eventLoopThread
```

The protocol computes expected messages per input channel with node n executing its callback at rate f_n and receiving messages at rate f_i delayed by τ_i over input channels $i \in \mathcal{U}$ as summarized by Alg. 2. Assuming nodes maintain their rates, callbacks occur every $\Delta t_n = \frac{1}{f_n}$ seconds, and messages are received every $\Delta t_i = \frac{1}{f_i}$ seconds. The protocol expects the kth callback after $k\Delta t_n$ seconds, anticipating $\lfloor (k\Delta t_n - \tau_i)/\Delta t_i \rfloor$ messages from each input channel i, where $\lfloor a/b \rfloor$ denotes the integer division operator.

Algorithm 2: Expected number of messages to receive between the $k-$ 1th and kth callback

Input: callback index k, node rate f_n, input rate f_i, input delay τ_i
Output: Expected number of messages δ to receive between the $k-1$th and kth callback

1 **if** $k = 0$ **then**
2 $\delta \leftarrow 1$ // Init count to 1
3 **else**
  /* Expected count between k-1 and k */
4 $N_{k-1} \leftarrow \lfloor (f_i(k-1) - f_n f_i \tau_i)/f_n \rfloor$
5 $N_k \leftarrow \lfloor (f_i(k) - f_n f_i \tau_i)/f_n \rfloor$
6 $\Delta \leftarrow N_k - N_{k-1}$
  /* Correct expected count with delay */
7 $c \leftarrow \lfloor (f_i k - f_n \Delta - f_n f_i \tau_i)/f_n \rfloor$
8 $\delta \leftarrow \Delta - \min(\Delta, \max(0, -c))$

Algorithm 3: Expected number of messages to receive between the $k-1$th and kth callback to resolve a cyclical dependency

Input: callback index k, node rate f_n, input rate f_i, input delay τ_i, fudge factor $\epsilon \approx 10^{-9}$
Output: Expected number of messages δ to receive between the $k-1$th and kth callback

1 **if** $k = 0$ **then**
2 $\delta \leftarrow 0$ // Set initial count to 0
3 **else**
  /* Calculate count as if k is shifted */
4 **if** $f_n > f_i$ **then**
5 $o \leftarrow \lfloor (f_n - \epsilon)/f_i \rfloor$ // Forward
6 **else**
7 $o \leftarrow -1$ // Backward
  /* Expected count between k-1 and k */
8 $N_{k-1} \leftarrow \lfloor (f_i(k-1+o) - f_n f_i \tau_i)/f_n \rfloor$
9 $N_k \leftarrow \lfloor (f_i(k+o) - f_n f_i \tau_i)/f_n \rfloor$
10 $\Delta \leftarrow N_k - N_{k-1}$
  /* Correct expected count with delay */
11 $c \leftarrow \lfloor (f_i k - f_n(\Delta - 1) - f_n f_i \tau_i)/f_n \rfloor$
12 $\delta \leftarrow \Delta - \min(\Delta, \max(0, -c))$

While this intuition underpins the synchronization protocol, the implementation in Alg. 2 is more complex. Computations are recast in rates to improve numerical stability by minimizing floating-point imprecision in case of high rates (small time intervals). The protocol sets every input channel's initial expected message count to 1, irrespective of τ_i, simplifying callback implementations.

The protocol also handles the special case of cyclical dependencies-—common in robotics systems interacting with a physic-engine and can cause deadlocks otherwise—with Alg. 3. In EAGERx, users can designate input channels as cyclical, postponing dependency to the next callback. This strategy allows one node to execute first in a cycle, while others await this node's output.

2.3.2 Limitations

The protocol's limitations should be considered in the context of the underlying communication protocol, which must ensure the preservation of message order and be lossless (e.g., TCP instead of UDP [44]). The protocol assumes that the robotic system can be represented by nodes with fixed rates and at least one input. Although the protocol can be easily toggled between synchronous and asynchronous modes, it does not allow for a hybrid mode, where some nodes are synchronized and others are not. Finally, the protocol does not account for jitter and assumes deterministic delay; however, this limitation can be mitigated by varying the delay across episodes if needed.

2.4 Experimental Evaluation

This section presents experiments to show the capabilities of our framework and to support the four key contributions discussed at the beginning of the paper and repeated below for the reader's convenience:

C1 A synchronization protocol that ensures consistent simulation behavior even beyond real-time speeds.

C2 A modular design that can support various robotic systems and state, action, and time-scale abstractions.

C3 An agnostic design that allows compatibility with multiple engines.

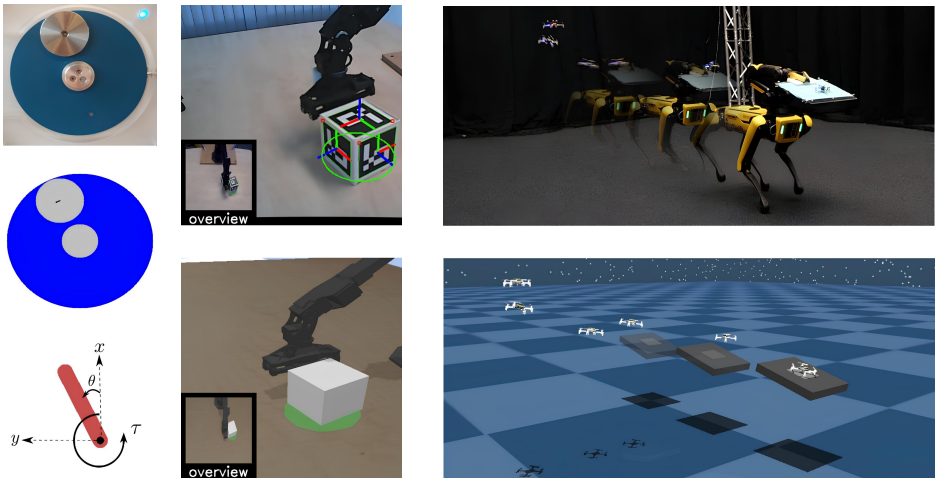
C4 The integrated delay simulation and domain randomization features in EAGERx can narrow the sim2real gap.

2.4.1 Experimental Setup

EAGERx is validated with a pendulum swing-up task, a vision-based box-pushing task, and an inclined landing experiment for a quadrotor. The simulated and real-world setups of all three tasks are depicted in Fig. 2.4. To validate **C1**, we experimentally assess Alg. 1 and employ it in accelerated, parallelized training for all tasks. Claims **C2-C3** are validated by the tasks involving different *engines* and distinct types of systems like pendulums, manipulators, quadrotors, and quadrupeds. Claim **C4** is validated by demonstrating the detrimental effect of delays and model mismatch on sim2real performance and showing how simulating delays and domain randomization can restore sim2real performance. All policies are trained in simulation, and zero-shot evaluated on their real-world counterparts.

Swing Up The inverted pendulum task addresses the classic control problem of swinging up and stabilizing an underactuated pendulum. The choice of this task is intentional; it emphasizes the critical challenge of delay compensation in reinforcement learning. By showing how ignoring delay simulation can hinder policy transfer even in straightforward scenarios, we highlight the significant consequences for more complex systems where delays are inevitable and complexity is higher. The simplicity of the task underscores the fundamental importance of addressing delays in sim2real approaches. We conduct zero-shot evaluations using a real-world pendulum setup comprising a mass on a disk driven by a DC motor. To train policies, we utilize two simulators. The first simulator's dynamics model aligns with the physical system, representing the pendulum as a disk [45]. In contrast, the second simulator adopts the OpenAI Gym Classic Control's *Pendulum* environment, modeling the pendulum as a rod [23], inadvertently introducing a sim2real gap that requires mitigation. All three systems are depicted in Fig. 2.4a. In all experiments, the pendulum is controlled at a rate of 20 Hz, while sensor measurements are obtained at a rate of 60 Hz. The agent observes the last two received sensor measurements. Users can specify such a rolling window length when connecting nodes in EAGERx, as shown in lines 14 and 16 of Fig. 2.3). Policies are trained using the soft actor-critic (SAC) [46] implementation from [47].

Box Pushing In the box-pushing experiment, a Viper 300x robotic manipulator moves a box to a target based on streaming webcam images. To emphasize the importance of



(a) Swing up (b) Box pushing (c) Inclined landing

Figure 2.4: Diverse robotic system tasks illustrating the EAGERx framework's flexibility. (a) Swing-up task with an inverted pendulum, highlighting delay compensation in reinforcement learning. The task involves zero-shot evaluations on a real-world pendulum setup, comparing a disk-based simulator with the OpenAI Gym rod-based environment. (b) Box-pushing experiment using a Viper 300x robotic manipulator, emphasizing the need for domain randomization with a low-resolution Logitech C170 webcam for box localization tracking. (c) Inclined landing task where a quadrotor lands on a moving and inclined deck, showcasing the integration of multiple mobile robots into a dynamic task.

domain randomization, we use a consumer-grade Logitech C170 webcam, selected for its low resolution, modest frame rate, and high latency, to track the box's position and orientation. For evaluation, we selected six unique initial configurations (three positions approximately 30 cm from the goal for both a yaw angle of 0 and $\frac{\pi}{2}$ rad) and repeated them thrice per policy. Policies are trained in PyBullet using the SAC [46] implementation from [47] with hindsight experience replay [48]. The simulation and real-world setups are shown in Fig. 2.4b.

Inclined Landing To demonstrate the framework's ability to facilitate control in highly dynamic environments, we trained an agent to perform the challenging maneuver of landing a quadrotor on an inclined and moving landing deck. Due to the configuration of its rotors, standard quadrotors can only exert thrust upwards, as rotor spinning directions cannot be reversed mid-flight. This under-actuation complicates landing on an incline, as the agent can only decelerate when approaching the deck. Therefore, if the agent initiates the landing procedure with insufficient momentum, it cannot accelerate, resulting in a crash. In [30], PPO [49] was used to learn a policy for landing on a stationary landing deck in 2D (xz-plane) with a fixed inclination (25°). In this chapter, we follow a similar approach using the PPO implementation from CleanRL [50]. However, we extend the policy's capability to land in 3D (xyz-plane), at various inclinations (0 – 25°), and on a moving landing deck (0 – 1m/s).

As in [30], the quadrotor dynamics are prescribed by ODEs identified with real-world

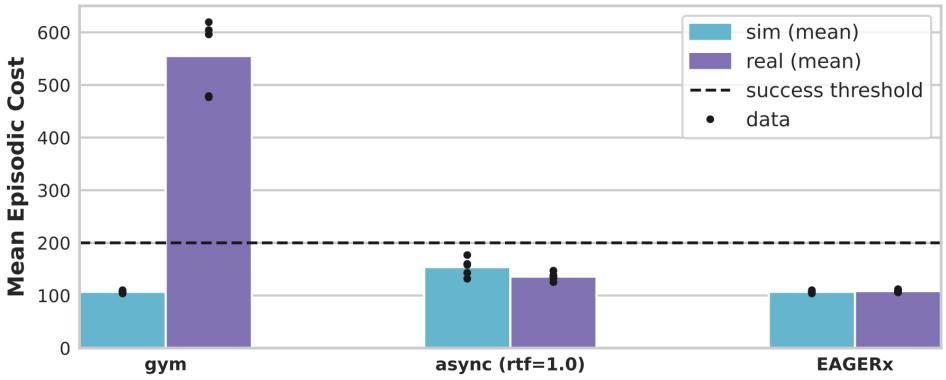


Figure 2.5: Comparison of mean episodic cost between simulations and real-world pendulum performance. The success threshold denotes the level below which a 100% success rate is achieved. Performance drops notably in the real-world scenario with a conventional *gym* approach, illustrating the sim2real gap. Asynchronous simulation (*async*) at real-time speeds mitigates the gap but leads to excessively long training times. Synchronized training under our protocol (*EAGERx*) facilitates consistent performance at faster-than-real-time simulation speeds.

data. In simulation, the landing deck moves in a straight line at a fixed inclination, varying speed, inclination, and direction across episodes to learn multi-goal behavior. To model the interaction between the landing deck and quadrotor, we extend the ODE dynamics with MuJoCo's [18] collision detection capabilities to detect successful landings and crashes. During real-world evaluation, we move the landing deck around with a quadruped and track the pose of both the deck and quadrotor with an accurate motion capture system. The simulation and real-world setups are shown in Fig. 2.4c.

2.4.2 Analysis

C1 We tested Alg. 1's ability to maintain consistent simulation behaviors at speeds surpassing real-time, using experiments with the disk pendulum. Initially, we utilized the disk simulator within a standard OpenAI Gym environment, trained a policy, and then conducted a zero-shot evaluation on the actual system. As depicted in Fig. 2.5, the performance significantly declines in the real world, indicating a substantial simulation-to-reality (sim2real) gap. This discrepancy results from the sequential communication in simulations contrasted with the asynchronous sensor and actuator commands in the real system via ROS topics [24], forcing the agent to sometimes rely on outdated information in real-world scenarios. To mimic this asynchronous nature, we adapted the gym environment to use asynchronous communication in simulation. This adaptation enabled the policy to handle occasional delays, enhancing its real-world applicability. However, this required limiting simulation speed to a real-time factor of 1, considerably prolonging training duration. The real-time factor, the ratio of simulation to real-world time, at 1 signifies running the simulation in real-time. Fig. 2.6 demonstrates that increasing this factor degrades performance, underscoring the challenges in accelerating simulation beyond real-time while ensuring effective real-world transfer.

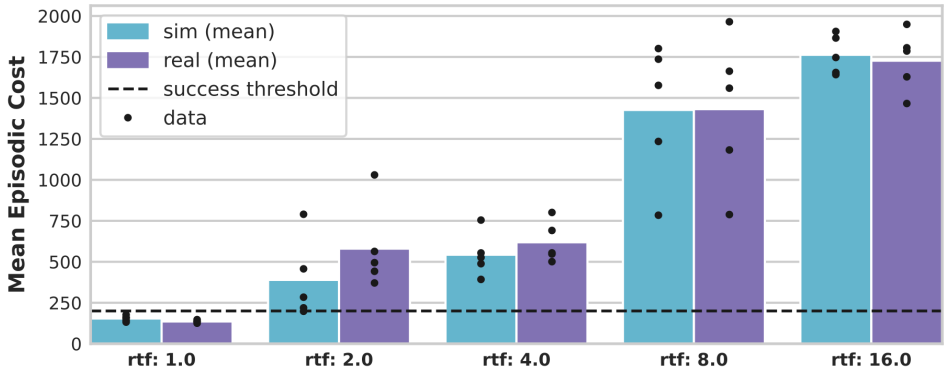
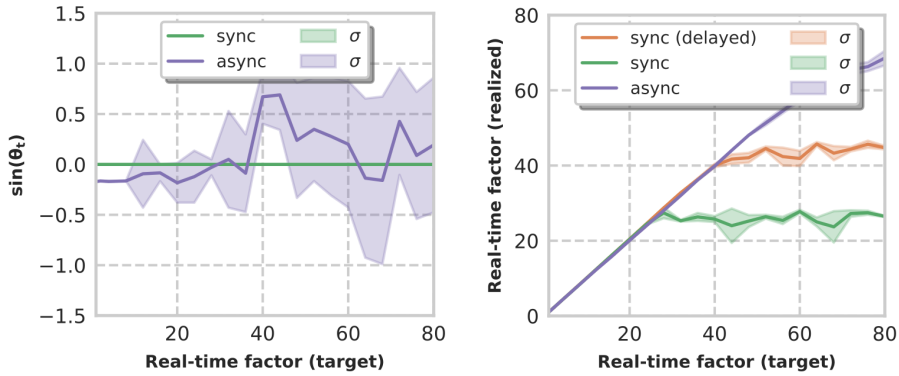


Figure 2.6: The impact of varying real-time factors (rtf) on the mean episodic cost in a simulated pendulum environment. Performance declines as the rtf increases, indicating the challenges of maintaining fidelity in faster-than-real-time simulations when components operate asynchronously.

Setting an excessively high target real-time factor may cause parallel components in the simulation to desynchronize and lag. Figures 2.7a and 2.7b demonstrate the consequences of this lag in asynchronous simulations. Specifically, Fig. 2.7a displays the variation in the simulated pendulum's angle, $\sin(\theta)$, at $t = 2.0$ s across five runs with identical input sequences, highlighting the increasing discrepancy in angle measurements as the real-time factor rises. In contrast, simulations synchronized via our protocol remain deterministic while still allowing parallel operations, enhancing speed without sacrificing accuracy. Synchronous simulations naturally cap the real-time factor to preserve synchronization, whereas asynchronous ones might show a misleadingly high factor, as evidenced in Fig. 2.7a, where increased speed incurs greater variability and component desynchronization. This illustrates the adverse effects of unsynchronized, accelerated simulations. Adapting the disk simulator into an EAGERx environment for synchronized training under our protocol facilitated faster-than-real-time speeds while ensuring consistency between simulated and real-world behaviors, as depicted in Fig. 2.5.

Our protocol, designed for robotic system synchronization, does not necessitate synchronous operation within the simulation. In fact, asynchronous communication still permits nodes to transmit messages and perform tasks without waiting for immediate responses, thereby accelerating the simulation and allowing nodes to progress based on their processing capabilities and data availability. This is illustrated in 2.7b, where we introduced a simulated delay between the pendulum actuator and the physics engine. Consequently, the pendulum's callback and the physics engine's callback can be executed concurrently, as the physics engine's callback relies on the pendulum's output from the previous timestep rather than the current one. Since each node's protocol operates independently, this parallelization occurs naturally, resulting in approximately 50% increase in the realized real-time factor for the synchronized simulation compared to the case without delay.

C2 To support the claimed contribution that the framework accommodates various robotic systems, the tasks involve distinct robot systems such as pendulums, manipulators,



(a) Variation in angle sin(θ)

(b) Real-time factor

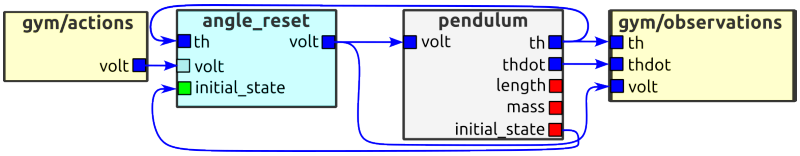
Figure 2.7: A comparison between asynchronous (*async*) and synchronous (*sync*) simulations of a pendulum at faster-than-real-time speeds. Fig. 2.7a shows the variation in angle sin(θ) at t = 2.0 s over 5 runs of a simulated pendulum as a function of the real-time factor. Fig. 2.7b shows the realized real-time factor of the simulation for both synchronous and asynchronous cases.

quadrotors, and quadrupeds. EAGERx's graph-based design, enabling diverse abstractions, is demonstrated in the vision-based box-pushing task. Rather than end-to-end training on raw images, an *aruco detector* is used for state abstraction as depicted in Fig. 2.8e, negating the need for photorealistic rendering. Action abstractions, visible in Fig. 2.8b, include an *inverse kinematics* node for task-space learning and a *safety filter* correcting hazardous commands. Nodes set at optimal rates ensure efficient resource use and learning. The pendulum task underlines the framework's modularity using an *angle reset* node, visible in Fig. 2.8a, to position the pendulum at the initial angle via PID control before a new episode. Finally, we demonstrate EAGERx's capability to coordinate diverse systems, such as a quadruped and quadrotor, in a delay-sensitive and dynamic task with the inclined landing experiment.

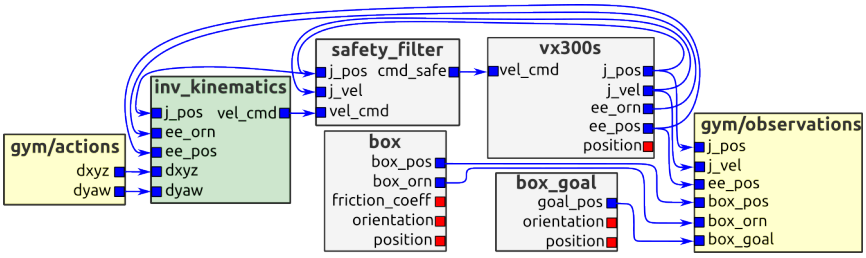
C3 To support the claimed contribution that EAGERx is compatible with a variety of physics engines and the real world, we conducted experiments with four different *engines*—PyBullet [43], OpenAI Gym Classic Control [23], real-world, and simulations with sets of ODEs—showing the ability to switch between real and simulated counterparts. The box-pushing task demonstrates how a division of the graph into engine-specific and engine-agnostic subgraphs resulted in a unified pipeline between PyBullet and reality. The *inverse kinematics* and *safety filter* nodes work with any simulator, as seen in the agnostic graph (Fig. 2.8b), while the *aruco detector* and *webcam* nodes are swapped with PyBullet-specific nodes in Figures 2.8d and 2.8e. Likewise, the agnostic graph in Fig. 2.8a was used in all pendulum experiments to display sim2real transfer across physics engines. The inclined landing experiment further illustrates the framework's flexibility by combining the collision detection capabilities of MuJoCo [18] with the accurately identified ODE dynamics of the quadrotor. This task highlights how different physics engines can be integrated seamlessly within EAGERx. The collision detection in MuJoCo is used to detect successful landings and

crashes, while the ODE dynamics ensure realistic quadrotor behavior. Collision detection is used both in simulation and reality, so it is therefore placed in the agnostic graph Fig. 2.8c. During real-world evaluation, the landing deck is moved by a quadruped and the poses of both the deck and the quadrotor are tracked using a motion capture system. Since simulating the full dynamics of a quadruped during policy learning is unnecessary and would only slow down training with redundant computation, the quadruped control nodes are placed in the real-world engine-specific graph. We can simulate just a moving landing platform without the quadruped, as actuation is not required to move objects in simulation. This approach focuses computational resources on what truly matters for training.

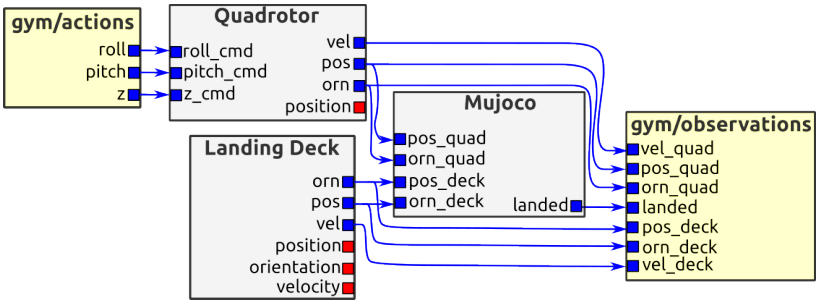
C4 We show that the integrated delay simulation and domain randomization features can reduce the sim2real gap by demonstrating that the negative impacts of actuator delay can be counteracted using the delay simulation feature during training for two different simulated versions of the pendulum. In this task, we supported **C4** by evaluating policies on the real system with an actuator delay set at the smallest value that led to a breakdown in baseline performance. When we progressively increased the actuator delay, it resulted in baseline policy failure for delays of 0.025 s and 0.035 s for the rod and disk pendulum, respectively. Our experiments studied the potential of training with domain randomization and/or delay simulation to mitigate the adverse effects of the actuator delay. For the disk pendulum, we applied randomization within ±10% of the mean values (0.033 kg for mass and 0.1 m for length). For the rod pendulum, randomization was limited to ±5%, considering the higher accuracy of this model. Delay simulation involved randomization within ±0.005 s around the set actuator delay. The results shown in Fig. 2.9 suggest that delay simulation can mitigate the adverse effects of actuator delay for zero-shot transfer from both the rod and disk simulator to the real pendulum system. In the disk scenario, adding domain randomization to delay simulation further improved performance and resulted in successful transfer with the smallest performance gap between simulation and reality. The effectiveness of domain randomization is further highlighted in the box-pushing task (Fig. 2.10). We examined its impact by altering the box's friction coefficient between 0.1 and 0.4. Fig. 2.10 shows that, compared to the baseline, friction randomization reduces the performance gap between simulation and reality despite lowering overall performance, thereby illustrating that relying solely on domain randomization can increase task difficulty. Conversely, incorporating the inverse kinematics node combined with friction randomization enhances performance while reducing the gap between simulation and real-world execution. We used a delay simulation of 0.02 seconds for the inclined landing task and randomized the mass within ±5%, leading to the results in Fig. 2.11. However, we refrain from conducting an extensive ablation study on the effects of delay simulation and domain randomization to avoid unnecessary hardware damage.



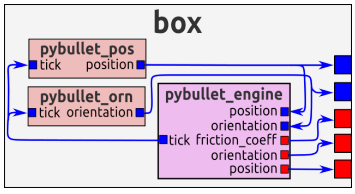
(a) Agnostic graph - swing up



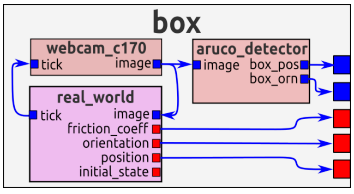
(b) Agnostic graph - box pushing



(c) Agnostic graph - inclined landing



(d) PyBullet subgraph



(e) Real world subgraph

Figure 2.8: Diverse robotic system tasks demonstrating the versatility of EAGERx's graph-based design. (a) The pendulum swing-up task uses an agnostic graph with an *angle reset* node for initializing the pendulum's position. (b) The vision-based box-pushing task utilizes an *inverse kinematics* node for task-space learning and a *safety filter* for correcting hazardous commands. The engine-specific subgraphs for replacing the *box* object in (b) are depicted for the PyBullet (d) and real-world (e) engines. (c) The inclined landing task illustrates how EAGERx integrates collision detection in MuJoCo with ODE dynamics to get the best of both simulators.

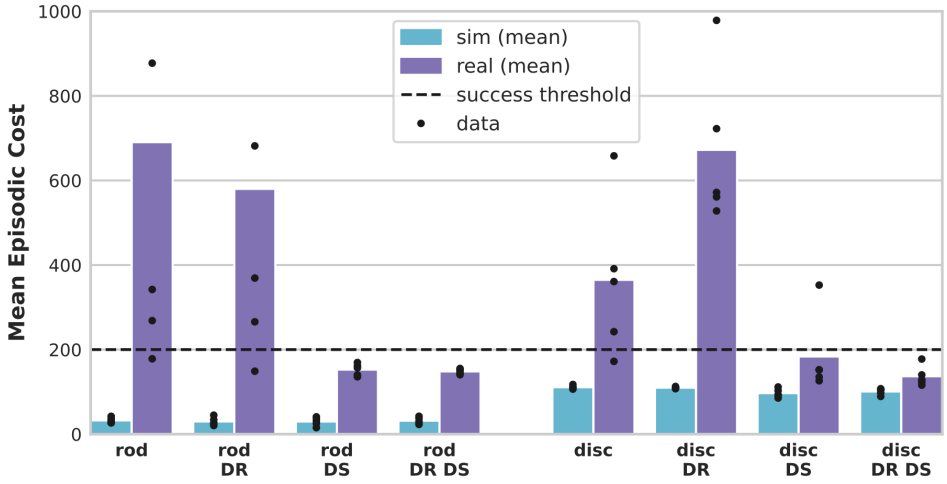


Figure 2.9: Results for the pendulum swing-up task show the mean episodic cost for 5 policies (10 episodes per policy) and the impacts of domain randomization (DR) and delay simulation (DS). Here *rod* and *disk* refer to the engine used during training, as depicted in Fig. 2.4a. For clarity, the y-axis is capped at 1000, though note this truncates some data points. The *success threshold* indicates 100% success rate, meaning successful pendulum swing up and stabilization each episode for all evaluations below this threshold.

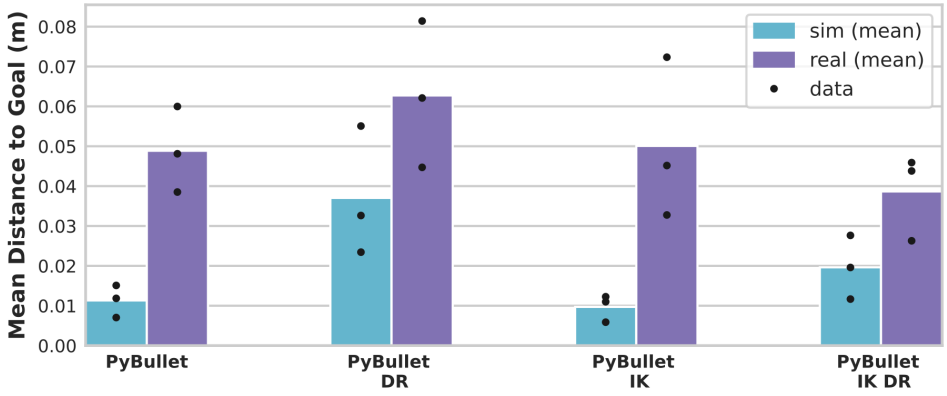


Figure 2.10: Results for the box-pushing task show the mean distance from the goal at the end of 16 episodes for 3 policies and evaluate the benefits of an inverse kinematics (IK) node (facilitating task space control) and domain randomization (DR) of the friction coefficient.

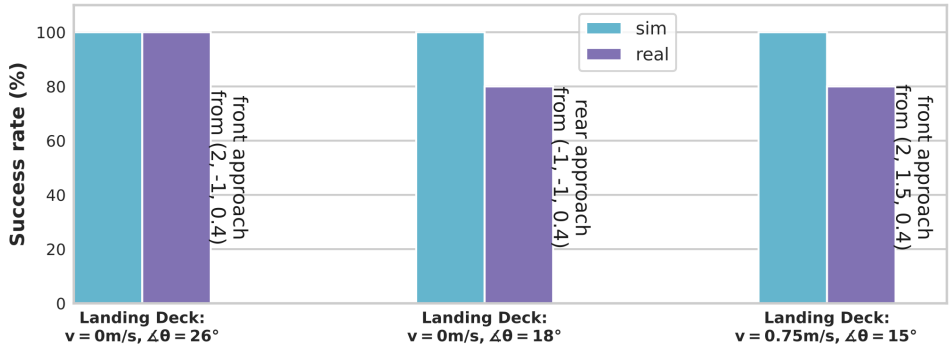


Figure 2.11: Results for the inclined landing experiment show the success rate for landing on a stationary and moving deck at various inclinations in simulation and real-world settings. The experiments evaluate the performance of the policy in terms of successful landings across 10 episodes, demonstrating the framework's capability to handle dynamic and delay-sensitive tasks involving diverse robotic systems like quadrupeds and quadrotors.

2.5 Applications beyond Reinforcement Learning

The modular design and unified software pipeline of the framework have utility in various other domains. This section explores two such instances: interactive imitation learning and Machine Learning (ML) enhanced classical control, showcasing EAGERx's utility beyond reinforcement learning.

2.5.1 Interactive Imitation Learning

This application shows how EAGERx is suitable for (interactive) imitation learning. Here, the task involves assembling a mock-up Diesel engine by following voice commands from a human operator. The parts used in this task are 3D-printed versions of the parts from an actual Diesel engine assembly setup [51]. To solve this task, we apply a learning from demonstration approach based on CLIPort [52]. However, we utilize an interactive imitation learning approach instead of gathering offline demonstrations only. Collecting on-policy data helps us to, for example, learn to recover from failures. Learning recovery behavior is often not possible using demonstrations collected offline by experts since they are unlikely to visit failure states. We apply an active learning method based on uncertainty quantification [53]. This method actively queries the human teacher for a demonstration in case there is high prediction uncertainty. EAGERx offers three main advantages in this scenario. First of all, we can easily create a digital twin of the real-world environment in simulation. This allows one to debug a large portion of the pipeline in simulation, which is safe and time-efficient. Moreover, the simulated environment facilitates the cost-effective collection of synthetic demonstrations. These can be used to pre-train the policy in simulation and speed up learning. Lastly, EAGERx's modular graph structure enables the simple connection of various components. In this case, the graph includes a speech-to-text transcriber based on Whisper [54], the policy node, as well as the RGB-D camera, and the manipulator. An overview of the task is shown in Fig. 2.12. A video demonstration of this application is available at `https://eagerx.readthedocs.io`.

2.5.2 ML-Enhanced Classical Control

This application illustrates EAGERx's integration of pre-trained ML models with classical control in a custom simulator, addressing a practical challenge. EAGERx was applied to an adaptive swimming pool environment, showcased in Fig. 2.13. This environment enhances traditional counter-current pools by dynamically adjusting the current based on the swimmer's position. Normally, it is the swimmer's task to stay centered in the pool, a difficult task for beginners. Our approach, however, modifies the pool's counter-current in line with the swimmer's location, maintaining central positioning regardless of swim speed. This adaptability makes the pool more user-friendly for novice swimmers.

Variable transport delays complicate the control problems [55]. Specifically, alterations in motor power do not instantaneously translate into flow changes; this delay results from the gradual response of the water pump's first-order dynamics, as well as the variable time it takes for a change in water flow to impact the swimmer, contingent on their position in the pool. When the swimmer is towards the front, they feel the effects of flow velocity changes more rapidly than when positioned at the rear. The absence of a readily available off-the-shelf simulator for this specific scenario underscored the utility of EAGERx, which

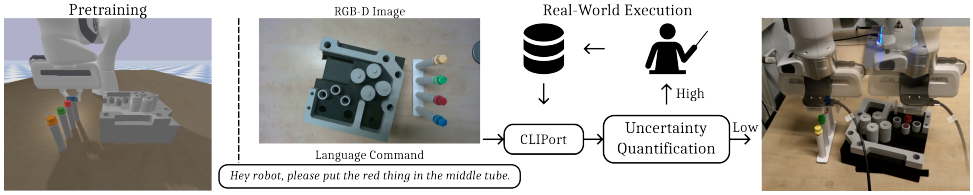
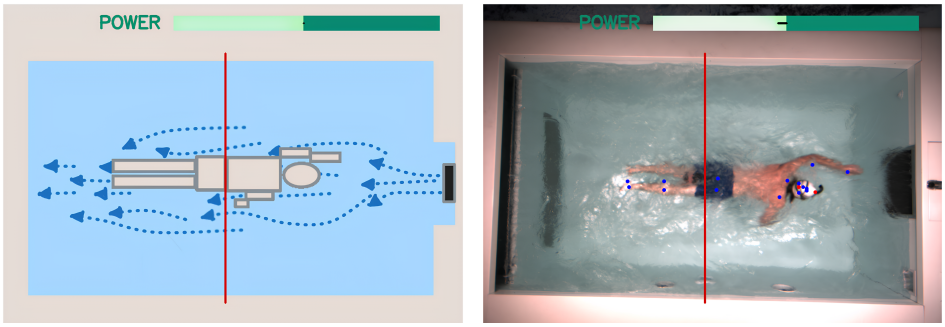


Figure 2.12: In this application of EAGERx, a CLIPort [52] model is trained using an active learning approach that queries the human teacher for a demonstration in case of high prediction uncertainty. Also, the model is pre-trained using demonstrations gathered in simulation.



(a) Custom simulator (b) Real

Figure 2.13: Application of EAGERx in an adaptive swimming pool environment. The system modulates the pool's counter-current in response to the swimmer's hip position relative to a preset boundary (red line), utilizing a pose detector and Kalman filter for position estimation and a PID controller for current adjustment.

facilitated the creation of a custom simulator, proving invaluable in the development of the control pipeline.

The modular architecture of EAGERx facilitated the integration of a pose detector [56] with a Kalman filter, resulting in estimates of the swimmer's position and velocity from solely top-view camera imagery. Subsequently, a PID controller was employed to modulate the pool current in alignment with these estimates. A video demonstration is available in the supplemental material.

2.6 Discussion

Comparing EAGERx with ROS [24] might seem natural due to their modular structures and asynchronous communication. Nonetheless, such a comparison risks being misleading since EAGERx represents an abstraction based on the actor model [57] and can operate atop a backend like ROS. This abstraction layer offers vital functionality for robot learning, including synchronized faster-than-real-time simulation, domain randomization, and delay simulation, not inherently supported by ROS. Recent research [58] presented a reactive solution to ROS's asynchronous programming challenges via an event-driven API, inspiring EAGERx's synchronization approach. However, this API didn't specifically aim to

synchronize simulations using expected rates and delays, as demonstrated in our work. Importantly, EAGERx's protocol extends beyond ROS to other backends as well.

The proposed synchronization protocol can be seen as an application of the actor model for computation [57]. It is a powerful and flexible model of concurrent computation where actors, the primary units, execute tasks concurrently and communicate by exchanging messages. The actor model is well-suited for synchronizing robotic systems represented as graphs of nodes, where various nodes need to operate concurrently. Our protocol operates on an event-driven basis and circumvents dependence on a global/local clock, a central coordinator [59], or extra synchronization messages [60]. Instead, it assesses conditions for subsequent callbacks exclusively after finalizing the preceding one or obtaining a new input channel message. This can outperform busy-waiting techniques (or spinlock) [61] that continuously evaluate conditions at a fixed time interval.

Ptolemy II [62] constitutes a software framework for designing, modeling, and simulating heterogeneous systems. Like EAGERx, it applies the actor model of computation, enabling concurrency and asynchronous communication. Both frameworks offer graphical user interfaces for visualizing complex systems. Ptolemy II holds an advantage over EAGERx in its support for a wider range of computation models and the ability to combine them within a single system. Nevertheless, Ptolemy II serves as a general-purpose framework, while EAGERx specifically targets robot learning. Furthermore, Ptolemy II employs a Java-based structure, in contrast with EAGERx's exclusive use of Python.

In comparison to Gym [23] — which offers a flexible API but lacks a unified sim2real framework — EAGERx addresses this deficiency. Unlike Gym's default sequential simulation, EAGERx supports concurrent, distributed operations across devices within environments, enhancing its applicability to robot learning. Gym environments use object-oriented classes, frequently constructed via inheritance and extended with wrapper patterns. However, this approach in Gym, particularly with extensive use of wrappers, tends to create overly complex and difficult-to-manage class structures in robotic systems, leading to maintenance challenges and reduced clarity in system design. Additionally, incorporating time abstraction within Gym environments is challenging, often confining it to multiples of the environment's step size. Conversely, EAGERx allows each node within the graph environment to operate at separate frequencies.

Various robot learning frameworks with connections to EAGERx have been introduced in the field. Among these, Isaac Orbit [39] and Drake [40] stand out as recent frameworks with shared design principles. In line with EAGERx, Orbit and Drake adopt a modular approach to constructing robot environments, enabling the execution of different nodes at varying rates to support both lower and higher-level control for effective robot learning. However, these frameworks exhibit three critical differences with EAGERx. Firstly, EAGERx is designed to be engine-agnostic, whereas Orbit relies on a proprietary simulator, and Drake incorporates an integrated multi-body dynamics simulator, hence restricting them to a single simulation platform. Secondly, EAGERx features dedicated reset procedures in the form of reset nodes. These nodes can be added to the *graph* and are only activated during environment resets. Thirdly, EAGERx offers a unified pipeline for both simulation and reality. Although Orbit and Drake promote component reusability in both simulation and reality, EAGERx enforces this more rigorously through *engine-agnostic* and *engine-specific graphs*. This effectively isolates the engine-agnostic code and minimizes the risk of

	EAGERx	Orbit [39]	Drake [40]	robo-gym [35]	gym-gazebo2[36]
Engine Agnostic	✓	✗	✗	−	✗
Specialized Reset Procedures	✓	✗	✗	✗	✗
Unified Pipeline Sim/Real	✓	−	−	−	−
Synchronized Simulation	✓	✓	✓	✗	✗
Distributed Computing	✓	✓	✓	✓	✓
GPU Accelerated	✗	✓	✗	✗	✗
Gradient Information Available	✗	✗	✓	✗	✗
Domain/Delay Randomization	✓ / ✓	✓ / ✗	✓ / ✓	✗ / ✗	✗ / ✗
Environment Visualization	✓	✓	✓	−	−
Open Source / License-free	✓ / ✓	− / −	✓ / ✓	✓ / ✓	✓ / ✓
Documentation / Tutorials	✓ / ✓	✓ / ✓	✓ / ✓	− / ✗	✗ / ✗
Last commit (age)	< 1 week	2 months	< 1 week	1 year	4 years

Table 2.1: A comparison of various modular sim-to-real robot learning frameworks, where − indicates partial feature presence.

discrepancies. Additional frameworks such as Robo-Gym [35] and Gym-Gazebo(2) [36] aimed to exploit the node structure of ROS for robot learning and were primarily centered around the Gazebo simulator without synchronization. To speed up training, EAGERx uses multi-processing instead of complete GPU acceleration for parallelization across multiple environments. While GPU parallelization can significantly speed up learning [22], its practicality can sometimes be limited for simulations requiring CPU-bound computations or non-GPU-adaptable libraries. In such cases, the latency from data transfer between GPU and CPU can become the dominant factor in simulation speed [17]. Among the frameworks discussed, only Orbit currently enables parallel training on a GPU. A comparative summary of the discussed robot learning frameworks is presented in Tab. 2.1.

2.7 Conclusion

This chapter presented EAGERx, a novel framework to facilitate the transfer of robot learning policies from simulation to the real world. Our unified framework is compatible with simulated and real robots. Its design can accommodate various abstractions and simulators. The presented synchronization protocol simulates delays without sacrificing simulation speed or accuracy, enabling effective policy training in simulation and subsequent transfer to real robots. We evaluated our framework on two benchmark robotic tasks, demonstrating its effectiveness in reducing the sim2real gap. Finally, we demonstrated the utility of the framework beyond sim2real robot learning in two real-world robotic use cases.

We plan to extend the open-source code base with more code examples for future work. Also, training can be sped up using GPU acceleration, and gradient information can be provided to facilitate optimization through nodes. Lastly, it will be valuable to provide real2sim functionalities to reduce the sim2real gap further using real-world data.

3

Speed: Parallelizing Graph-Based Simulations

Following the introduction of the EAGERx framework, another critical limitation in simulator design is the need for high-speed simulations to support efficient robotic learning. While flexibility is important, slow simulation speeds can hinder the learning process, especially for reinforcement learning where many iterations are required.

In this chapter, we tackle the issue of simulation speed by introducing a method for efficiently parallelizing graph-based simulations on accelerator hardware. This method optimizes execution paths to enable higher simulations speeds while maintaining simulation fidelity, thereby enabling scalable and efficient learning in large and complex robotic tasks.

This chapter is partly based on *B. van der Heijden, L. Ferranti, J. Kober, and R. Babuska, (2024). "Efficient Parallelized Simulation of Cyber-Physical Systems", Transactions on Machine Learning Research (TMLR)* [63].

3.1 Introduction

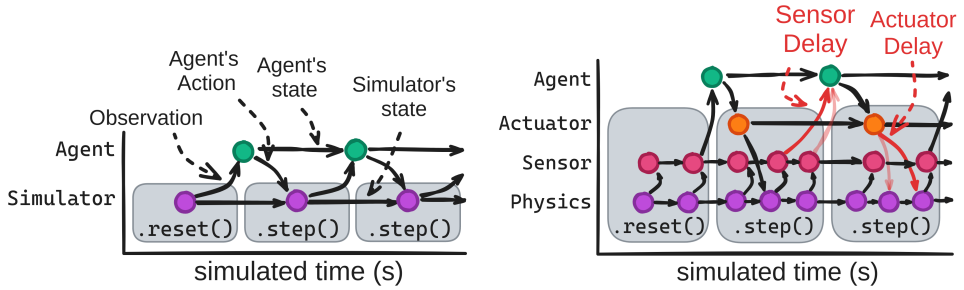
Physics simulations on accelerator hardware [15–18] have significantly reduced training times for reinforcement learning policies that conform to traditional, sequentially-structured agent-simulator interactions [22]. Such interactions lead to clear-cut and predictable execution paths, allowing for efficient parallelization, as shown in Fig. 3.1a. However, this sequential approach fails to capture the concurrent and dynamic nature of the real world.

Accounting for latency is crucial in the simulation of cyber-physical systems (CPS), which integrate computational algorithms with physical processes. In CPS, embedded computers and networks both monitor and control these processes, typically through feedback loops where physical processes impact computations and vice versa via sensors and actuators [64]. A critical application of CPS is vehicular platooning, involving multiple vehicles that operate in close proximity, coordinating their actions based on shared sensor data in real-time. This coordination is highly sensitive to time delays, making the accurate simulation of these delays critical for developing robust platooning algorithms [26].

Delayed sensor data causes an agent to choose actions based on outdated information. Similarly, slow policy evaluation can unavoidably extend the effect of previous actions beyond their planned duration. Moreover, the focus has traditionally been on single agents trained end-to-end [65]. In practice, however, AI systems deployed in real-world settings often rely on a pipeline of models. Accounting for the latency between these models will become crucial as tasks grow in complexity [66]. Finally, physics simulators often bundle physics, sensor, and actuator simulations into a single unit running at a single rate. However, in reality, there are vital asynchronous effects within this block that need to be accounted for. Overlooking these asynchronous effects in simulation widens the sim2real gap and can lead to policies that do not perform well in the real world.

To represent the asynchronous interactions between components, we advocate the division of the simulator into separate parts. This matches the typical design in robotics, where systems consist of interconnected nodes operating asynchronously at various rates [24]. This division enables the creation of computation graphs that accurately represent data flow in real-world situations, including latency effects. Consequently, each simulation step turns into a diverse mix of computation units from various components that run at different time scales, as illustrated in Fig. 3.1b. These must be executed in a sequence that respects the data dependencies outlined by the graph's edges. Simulating with these diverse partitions improves accuracy but complicates parallelization (i.e., simulating multiple copies of the simulation in parallel), as distinct partitions may need to execute simultaneously across GPU threads, hindering GPU efficiency. Such misalignment can happen with independent episodic resets, often initiated based on variable reset criteria. One parallel simulation might reset because the agent reached its goal, while another continues because the agent is still far away. Diverging execution paths can significantly reduce kernel efficiency [25]. When GPU threads take different paths, they must be serialized, leading to more instructions and reduced performance.

The main contribution of this chapter is an approach to parallelize cyber-physical system simulations that emulates asynchronicity and delays with minimized computational overhead on accelerator hardware. This allows existing accelerated physics simulations to be extended with efficient latency simulation capabilities. We achieve this by identifying a



(a) Sequential interaction

(b) Asynchronous interaction

Figure 3.1: Comparative illustration of computation graphs with and without simulated delays. Vertices represent periodic computations, and edges represent data dependencies. (a) The absence of delay simulation creates consistent blocks of computation, enabling efficient parallelization across simulation steps yet failing to capture the inherent asynchrony of the real world. (b) While improving simulation fidelity, simulated delays between various components turn every simulation step into a diverse mix of computation, challenging parallelization efficiency.

graph—ideally the smallest one possible—that encodes all the data dependencies outlined by every simulation step's edges. This universal graph, referred to as a *supergraph*, is determined prior to simulation. Sorting the supergraph topologically yields a static execution order for parallel processing of simulation steps without violating data dependencies. By targeting the smallest supergraph, we minimize redundant computation. Finding the smallest supergraph is generally a complex, NP-hard problem [67]; however, our greedy algorithm efficiently approximates this supergraph by leveraging the inherent periodicity in cyber-physical systems.

In sum, we make three key claims: Our approach (i) emulates asynchronicity leading to more accurate simulation, (ii) efficiently handles time-scale differences and asynchronicity, resulting in higher parallelized simulation speeds than baseline approaches, and (iii) scales to complex system topologies. These claims are supported by an experimental evaluation on two real-world robotic systems, followed by a scalability analysis on two cyber-physical system topologies: vehicle-to-vehicle (V2V) platooning [26] and unmanned aerial vehicle (UAV) swarm control [27]. An ablation study on the effects of the algorithmic simplifications was also conducted and included as an appendix. Finally, a motivational video for our approach is included as supplementary material [68].

3.2 Preliminaries

Before diving into the details of our approach, we first lay down some basic definitions and notation that will aid in the formalization of our problem and the description of our approach. We consider graphs $\mathcal{G} = (V, E)$ consisting of a set of vertices $V(\mathcal{G})$ and a set of directed edges $E(\mathcal{G})$. Edge $(u, v) \in E(\mathcal{G})$ denotes an edge from vertex u to vertex v. The notation $|V(\mathcal{G})|$ denotes the number of vertices in $\mathcal{G}$. Any subset of vertices $V' \subseteq V(\mathcal{G})$ induces a unique *subgraph* $\mathcal{G}' \subseteq \mathcal{G}$. The *difference* $\mathcal{G}_2 - \mathcal{G}_1$, where $\mathcal{G}_1 \subseteq \mathcal{G}_2$, yields a graph $\mathcal{G}$ with $V(\mathcal{G}) = V(\mathcal{G}_2) \setminus V(\mathcal{G}_1)$ and $E(\mathcal{G}) = E(\mathcal{G}_2) \setminus E(\mathcal{G}_1)$. The edges that connect $\mathcal{G}_1$ and $\mathcal{G}_2 - \mathcal{G}_1$

are defined as the *cut-set* $C_{\mathcal{G}_2}(\mathcal{G}_1)$, which is a subset of $E(\mathcal{G}_2)$. The *union* of graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with respect to a set of edges E_{12} is denoted as $\mathcal{G} = \mathcal{G}_1 \cup_{E_{12}} \mathcal{G}_2$, where $V(\mathcal{G}) = V(\mathcal{G}_1) \cup V(\mathcal{G}_2)$ and $E(\mathcal{G}) = E(\mathcal{G}_1) \cup E(\mathcal{G}_2) \cup E_{12}$. The *addition* $\mathcal{G}_1 + \mathcal{G}_2$, where $\mathcal{G}_1, \mathcal{G}_2 \subseteq \mathcal{G}_3$, yields a subgraph $\mathcal{G}_{12} \subseteq \mathcal{G}_3$, by unifying $\mathcal{G}_1 \cup_{E_{12}} \mathcal{G}_2$ where $E_{12} = C_{\mathcal{G}_3}(\mathcal{G}_1) \cap C_{\mathcal{G}_3}(\mathcal{G}_2)$. An *edge contraction* on an edge $(u, v) \in E(\mathcal{G})$ yields a new graph $\mathcal{G}'$ such that $V(\mathcal{G}') = V(\mathcal{G}) \setminus \{u, v\} \cup \{w\}$ and

$$\begin{aligned} E(\mathcal{G}') = (E(\mathcal{G}) \setminus \{(u, v), (v, u)\}) \\ \cup \{(w, x) \mid (u, x) \in E(\mathcal{G}) \text{ or } (v, x) \in E(\mathcal{G})\} \\ \cup \{(x, w) \mid (x, u) \in E(\mathcal{G}) \text{ or } (x, v) \in E(\mathcal{G})\}. \end{aligned}$$

The *ancestors* of a vertex $A_{\mathcal{G}}(u)$ are all vertices $V'(\mathcal{G}) \subseteq V(\mathcal{G})$ that can reach u via a directed path in $\mathcal{G}$. The *roots* of a graph $\mathcal{G}$ are the set of vertices that have no incoming edges, formally $R(\mathcal{G}) = \{u \in V(\mathcal{G}) \mid \forall v \in V(\mathcal{G}), (v, u) \notin E(\mathcal{G})\}$. Similarly, the *leafs* of a graph $\mathcal{G}$ are the set of vertices that have no outgoing edges. A *Directed Acyclic Graph (DAG)* is a directed graph that contains no cycles. A *topological sort* τ of a directed acyclic graph $\mathcal{G}$ is a linear ordering of its vertices such that for every directed edge $(u, v) \in E(\mathcal{G})$, vertex u comes before v in the ordering. Multiple topological sorts may exist for a given graph $\mathcal{G}$, and the set of all possible topological sorts is denoted by $\mathcal{T}(\mathcal{G})$. A *labeling function* $L : V \longrightarrow l$ is a function that assigns a label to each vertex. The set of all vertices with label l is denoted by $V_l(\mathcal{G})$ and is arranged as a sorted list consistent with a topological sort of $\mathcal{G}$. We denote the set of topological sorts where the final vertex is of label l in $\mathcal{G}$ as $\mathcal{T}_l^{-1}(\mathcal{G})$. Formally, this is defined as:

$$\mathcal{T}_l^{-1}(\mathcal{G}) = \{\tau \in \mathcal{T}(\mathcal{G}) \mid I(\tau, u) = |\mathcal{G}|, u \in V_l(\mathcal{G})\},$$

where $I(\tau, u)$ gives the position of vertex u in the sorted set τ. A *matching function* $f_{\text{m}} : V \times V \longrightarrow \{\text{True}, \text{False}\}$ is defined as follows:

$$f_{\text{m}}(u, v) = \begin{cases} \text{True} & \text{if } L(u) = L(v), \\ \text{False} & \text{otherwise.} \end{cases}$$

A *mapping* between two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a bijective function $M : V'(\mathcal{G}_1) \longrightarrow V'(\mathcal{G}_2)$ where V' represent a subset of the vertices. Its domain $\text{dom}(M)$ is $V'(\mathcal{G}_1)$ and its range $\text{rng}(M)$ is $V'(\mathcal{G}_2)$. Operations like union $\cup$, intersection $\cap$, and difference $\setminus$ can be applied to both $\text{dom}(M)$ and $\text{rng}(M)$. A mapping M can extend to M' by adding a new vertex pair (u, v) with $M' = M \cup \{(u, v)\}$ where $u \in V(\mathcal{G}_1) \setminus \text{dom}(M)$ and $v \in V(\mathcal{G}_2) \setminus \text{rng}(M)$. A *subgraph monomorphism* $M : V(\mathcal{G}_1) \longrightarrow V'(\mathcal{G}_2)$ is a specialized mapping that maps each vertex u to v such that $L(u) = L(v)$ and each edge (u, v) corresponds to an edge $(M(u), M(v))$ in $\mathcal{G}_2$. If such M exists, $\mathcal{G}_2$ is a *supergraph* of $\mathcal{G}_1$ and can be reduced to $\mathcal{G}_1$ by removing vertices and edges in $\mathcal{G}_2$. The transformed set of edges $E_M(\mathcal{G}_1)$ under the mapping M is defined as follows:

$$\begin{aligned} E_M(\mathcal{G}_1) = \{(u', v') \mid (u, v) \in E(\mathcal{G}_1), \\ u' = M(u) \text{ if } u \in \text{dom}(M), u' = u \text{ otherwise}, \\ v' = M(v) \text{ if } v \in \text{dom}(M), v' = v \text{ otherwise}\} \end{aligned}$$

This set includes edges (u', v') where u' and v' are either mapped vertices of u and v under M if they are in the domain of M, or are u and v themselves otherwise.

3.3 Our Approach

Consider the set of computation graphs generated by multiple episodes of an asynchronous system, as illustrated in Fig. 3.2, where vertices of the same color represent the same periodic computation unit, and edges represent data dependencies. These graphs might be partially recorded from real-world executions or synthetically created to reflect expected computation and communication delays. Variations in these graphs across episodes lead to distinct execution paths. However, managing these variations with conventional if-else branching for parallel execution on GPUs is inefficient, as highlighted in [25]. Predication [69] is a technique that sidesteps the need for if-else branching by executing all possible paths and masking out the computations that are not needed. This approach, while eliminating branching, can be inefficient due to the execution of all vertices in the paths, making it crucial to minimize the number of vertices.

To achieve this, we introduce an approach to identify a *minimum common supergraph* (mcs) that is acyclic and encapsulates all potential execution paths from a collection of computation graphs, optimizing for the fewest vertices (i.e., computational overhead). Topologically sorting the supergraph yields an execution order that, via predication masking, is transformed into a valid order for any given graph, as the supergraph encodes all data dependencies.

In aligning with standard simulator interfaces [23], illustrated in Fig. 3.1, we first *partition* these computation graphs into disjoint subgraphs, each corresponding to a simulation step. Crucially, we designate a *supervisor* node in each partition, a pivotal element that dictates the boundaries of these subgraphs. In the context of reinforcement learning, the supervisor node is akin to the agent, while all other nodes within the partition form the environment, providing observations to and receiving actions from the supervisor node. The supervisor node's operating rate sets the simulation time step, ensuring that each partition accurately reflects a step of the simulation process. We then find a supergraph that accommodates all possible paths in every partition with a minimum number of vertices. This supergraph serves as a template that can be reduced to match any of the partitions (i.e., simulation steps) by masking (i.e., removing) specific vertices and edges. This setup enables parallel execution of any partition on accelerated hardware.

3.3.1 Problem Definition

Consider a set of observed computation graphs denoted by $\{\mathcal{G}_0, \mathcal{G}_1, \dots\}$, where each $\mathcal{G}_i$ is a DAG. For a given supervisor label s, our goal is to partition each $\mathcal{G}_i$ into disjoint subgraphs $\mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \dots$. Each subgraph corresponds to a discrete simulation step and contains exactly one leaf vertex labeled as s. The objective is to determine these valid partitions along with the smallest DAG, $\mathcal{S}$, that serves as a common supergraph for all partitions. Similar to each partition, a single instance of the designated supervisor vertex in $\mathcal{S}$ must be a leaf vertex. Here, 'smallest' is defined by the number of vertices to minimize computational overhead. We aim to find a subgraph monomorphism $M_{i,j} : V(\mathcal{P}_{i,j}) \longrightarrow V'(\mathcal{S})$ for each partition $\mathcal{P}_{i,j}$. This mapping allows us to reduce $\mathcal{S}$ into $\mathcal{P}_{i,j}$ using a predication mask. The predication mask is a binary mask applied to $\mathcal{S}$ to selectively remove vertices and edges not present in $\mathcal{P}_{i,j}$. Specifically, the mask is false for vertices and edges not in $\text{rng}(M_{i,j})$ and $E_{M_{i,j}}(\mathcal{P}_{i,j})$, respectively, and true otherwise.

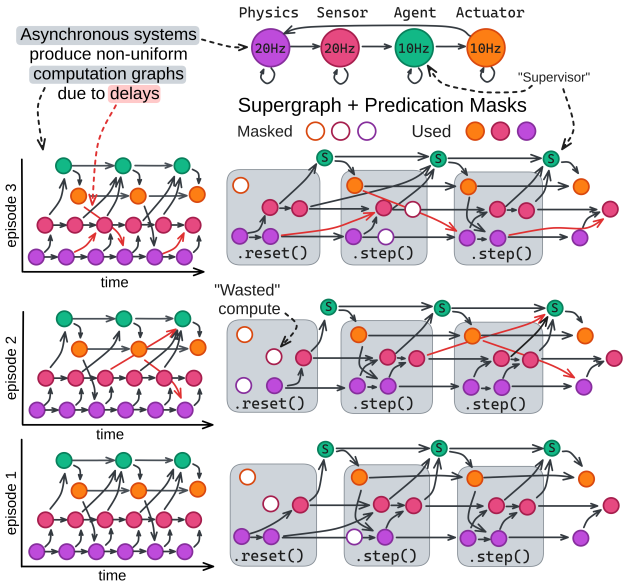


Figure 3.2: This figure illustrates our approach to efficiently simulating multi-rate asynchronous systems. Given variable delays, computation graphs can differ across episodes (left). We find a supergraph and predication masks, illustrated by the grey shaded blocks, for every computation graph that enables parallel execution across partitions (right). This mask, randomized during simulation, allows us to efficiently emulate asynchronicity and time-scale differences with minimal computational waste.

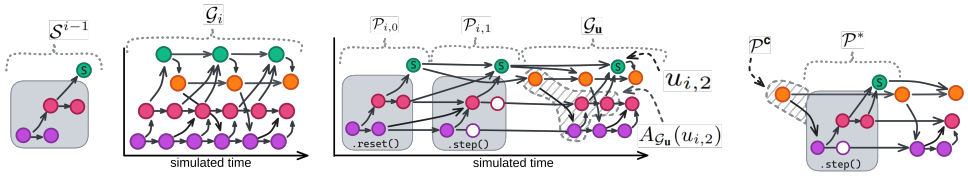
3.3.2 Supergraph Search

Algorithm 4: Minimum Common Supergraph Search (mcs)

Input: Designated supervisor label s
Input: Number of steps to backtrack β
Input: A set of observed computation graphs $\{\mathcal{G}_0, \mathcal{G}_1, \dots\}$
Output: A set of partitions $\{\mathcal{P}_{0,1}, \mathcal{P}_{0,2}, \dots \mathcal{P}_{i,j} \dots\}$
Output: A supergraph $\mathcal{S}$ and mapping $M_{i,j}$ for all partitions $\mathcal{P}_{i,j}$

$\mathcal{S} \leftarrow$ Initialize with $V(\mathcal{S}) = \{u | L(u) = s\}$ and $E(\mathcal{S}) = \emptyset$
for $\mathcal{G}_i \in \{\mathcal{G}_0, \mathcal{G}_1, \dots\}$ **do**
  $\mathcal{G}_\text{u} \leftarrow$ Initialize unmatched graph as $\mathcal{G}_i$
  `/* Until all supervisor vertices are matched */`
  while $V_s(\mathcal{G}_u) \neq \emptyset$ **do**
    $u_{i,j} \leftarrow$ Get next supervisor u_i from sorted set $V_s(\mathcal{G}_\text{u})$ with index $j = I(V_s(\mathcal{G}_i), u_i)$
    $A_\text{u} \leftarrow u_{i,j}$ and its ancestors: $A_{\mathcal{G}_\text{u}}(u_{i,j}) \cup \{u_{i,j}\}$
    $M^* \leftarrow$ Get largest map: Alg. 5 with $(s, \mathcal{S}, \mathcal{G}_\text{u}, A_\text{u})$
    $\mathcal{P}^* \leftarrow$ Partition subgraph: $\text{dom}(M^*) \subseteq V(\mathcal{G})$
    $\mathcal{P}^- \leftarrow$ Missing subgraph: $A_{\mathcal{G}_\text{u}}(u_{i,j}) \setminus V(\mathcal{P}^*) \subseteq V(\mathcal{G})$
    if $V(\mathcal{P}^-) = \emptyset$ **then**
      `/* All ancestors were matched */`
      $M_{i,j} \leftarrow$ Store subgraph monomorphism M^*
      $\mathcal{P}_{i,j} \leftarrow$ Store partition $\mathcal{P}^*$
      $\mathcal{G}_\text{u} \leftarrow$ Remove matched partition: $\mathcal{G}_\text{u} - \mathcal{P}_{i,j}$
    else
      `/* Partial match */`
      $\mathcal{G}_\text{u} \leftarrow$ Restore β partitions in $\mathcal{G}_\text{u} \subseteq \mathcal{G}_i$ with $\mathcal{G}_\text{u} + \mathcal{P}_{i,j-\beta} + \mathcal{P}_{i,j-\beta+1} + \dots + \mathcal{P}_{i,j-1}$
    $\mathcal{S} \leftarrow$ Update to $\mathcal{S}'$ with missing vertices and edges: $\left(V(\mathcal{S}) \cup V(\mathcal{P}^-), E(\mathcal{S}) \cup E_{M^*}(\mathcal{P}^* + \mathcal{P}^-)\right)$

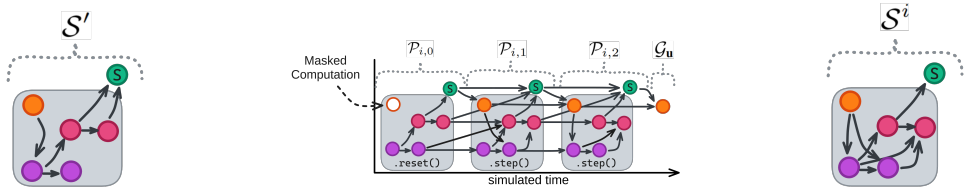
Our approach, as outlined in Alg. 4 and illustrated in Fig. 3.3, aims to simultaneously achieve three main objectives: identifying the supergraph $\mathcal{S}$, determining the partitionings $\mathcal{P}_{i,j}$, and discovering the associated mappings $M_{i,j}$. For each computation graph $\mathcal{G}_i$, the algorithm iterates until all supervisor vertices are matched, as specified in Line 4. In every iteration, the largest partition $\mathcal{P}^*$ and its associated mapping M^* are sought (Line 5-7), following the method detailed in Alg. 5 and explained later on in Sec. 3.3.3.



(a) Process next computation graph starting with current supergraph $\mathcal{S}^{i-1}$ (line 2)

(b) After two partitions, match the next supervisor vertex (lines 5-6)

(c) Find the largest match (lines 7-9)



(d) Update $\mathcal{S}^{i-1}$ with unmatched yellow vertex (line 16)

(e) Backtrack $\beta = 2$ with newly updated supergraph (line 15)

(f) Resulting final supergraph after processing $\mathcal{G}_i$

Figure 3.3: Illustration of the Minimum Common Supergraph Search process (Alg. 4) at a midway point. A partial match is found for $u_{i,2}$, leading to an updated $\mathcal{S}'$ with missing ancestors $\mathcal{P}^-$ and initiating a backtrack to re-evaluate previous partitions.

Depending on whether all ancestors are matched, the algorithm finds either a complete or a partial match corresponding to the supervisor vertex $u_{i,j}$. In the case of a complete match, both $\mathcal{P}^*$ and M^* are stored (Line 10-13). For partial matches, the algorithm backtracks β iterations to reconsider previously matched partitions (Line 15). In either case, the supergraph $\mathcal{S}$ is updated using Eq. (3.1) to ensure it remains a supergraph of its previous version and incorporates all necessary ancestors $V(\mathcal{P}^-)$ for future matches, as follows:

$$\mathcal{S}' = \Big(V(\mathcal{S}) \cup V(\mathcal{P}^-), E(\mathcal{S}) \cup E_{M^*}(\mathcal{P}^* + \mathcal{P}^-)\Big), \tag{3.1}$$

where $\mathcal{P}^* + \mathcal{P}^- \subseteq \mathcal{G}_i$. More edges in the updated supergraph $\mathcal{S}'$ effectively constrain the number of possible mappings for subsequent partitions by reducing the number of topological sorts available in the supergraph. Conversely, more vertices in $\mathcal{S}'$ increase its expressiveness by increasing the number of vertices that can be mapped to a vertex in subsequent partitions, but also increase the computational overhead of the simulation. In the next section, we detail the algorithm for finding the largest match, Alg. 5, which is a critical component of the supergraph search algorithm. It may only result in mappings M^*

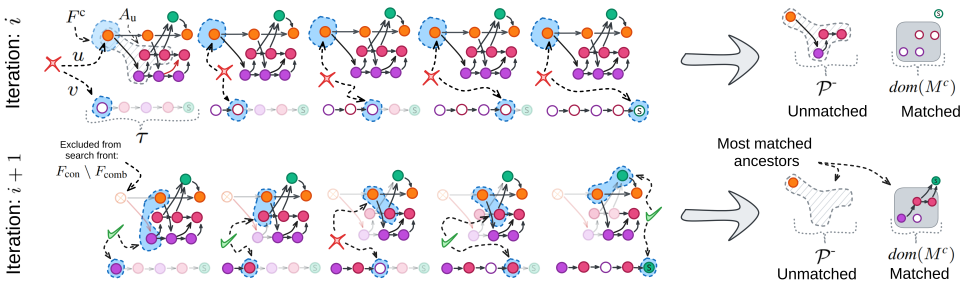


Figure 3.4: Midway illustration of Line 9-16 in the Largest Match Search (Alg. 5) linked to Fig. 3.3c. The ✓ and ✗ symbols indicate whether a vertex in the candidate front F^c is matched in Line 12. The first iteration yields an empty mapping due to the absence of a yellow vertex in the supergraph. In the second iteration, the vertex is excluded from the front, enabling a large partial match. The search is only displayed for a single topological sort.

that ensure that collectively contracting all edges $(u, v) \in E(\mathcal{P}^*)$ in $\mathcal{G}_u$ will not introduce any cycle in the resulting graph $\mathcal{G}'_u$. This constraint is critical to ensure that the updated supergraph $\mathcal{S}'$, as governed by equation Eq. (3.1), remains acyclic. This sets us apart from previous work [70, 71], which do not consider this constraint.

3.3.3 Largest Match Search

Our approach to identifying the largest valid mapping M^* for each supervisor vertex $u_{i,j}$ in $\mathcal{S}$ is outlined in Alg. 5 and illustrated in Fig. 3.4. Initially, it tries to match all ancestor vertices in candidate subgraphs $\mathcal{G}^c$, extracted from the unmatched graph $\mathcal{G}_{excl}$ (Line 2). The search is refined by iteratively excluding ancestors in a breadth-first manner from $\mathcal{G}^c$ (Line 8, and 21).

To this end, a refined search front F^c is formed for each iteration (Line 4-6, Line 9). The algorithm explores all k-combinations of F_{con} in descending order of k (Line 6). For each combination, the largest candidate mapping M^c is sought by traversing all valid topologies of $\mathcal{S}$, starting from the refined search front F^c (Line 7-10). Every time a match is found, the mapping is extended and the search front is updated (Line 11-16). Our approach assumes that vertices are stateful, i.e., vertices of similar labels are connected with one another, ensuring the uniqueness of the match (Line 12-13). If a larger mapping is found, M^* is updated (Line 17-18). The algorithm halts the search if no larger match can be found (Line 19-20). To lower computational complexity, we apply crucial greedy approximations in lines 6 and 7, as further motivated in Sec. 3.3.4.

We may only consider mappings M^* that ensure that the updated supergraph $\mathcal{S}'$ remains acyclic after updating with Eq. (3.1). To ensure this, we initiate each candidate search at the roots of $\mathcal{G}_{excl}$ and $\mathcal{S}$, as specified in Line 4 and 7, adopting a search strategy aligned with the topological sort of $\mathcal{S}$ and a breadth-first search of $\mathcal{G}_{excl}$. This approach guarantees that edges between matched vertices in dom(M^*), represented by $\mathcal{P}^*$, cannot create cycles in $\mathcal{S}'$. For vertices not matched in dom(M^*) (designated as $\mathcal{P}^-$), their positioning is either strictly prior to or following $\mathcal{P}^*$ in the topological sense, thus also ensuring acyclicity in $\mathcal{S}'$. This strict placement is achieved by initially removing only root vertices from consideration

Algorithm 5: Largest Match Search

Input: Designated supervisor label s
Input: Supergraph $\mathcal{S}$
Input: Unmatched computation graph $\mathcal{G}_{\text{u}}$
Input: Vertices to be matched A_{u}
Output: Largest mapping M^*

$M^* \leftarrow$ Initialize an empty mapping
$\mathcal{G}_{\text{excl}} \leftarrow$ Initialize search graph as $\mathcal{G}_{\text{u}}$
while *True* **do**
  $F_{\text{excl}} \leftarrow$ Initialize search front as roots $R(\mathcal{G}_{\text{excl}})$
  $F_{\text{con}} \leftarrow$ Determine constrained front: $F_{\text{excl}} \cap A_{\text{u}}$
  forall $F_{com} \in k\text{-}comb(F_{con})$ **do** // Greedy: one F_{com} per k
    forall $\tau \in \mathcal{T}_s^{-1}(\mathcal{S})$ **do** // Greedy: only a single τ
      $\mathcal{G}^{\text{c}} \leftarrow$ Remove $u \in F_{\text{con}} \setminus F_{\text{com}}$ from $V(\mathcal{G}_{\text{excl}})$
      $F^{\text{c}} \leftarrow$ Initialize front: $F_{\text{excl}} \setminus (F_{\text{con}} \setminus F_{\text{com}})$
      $M^{\text{c}} \leftarrow$ Initialize an empty candidate mapping
      forall $v \in \tau$ **do**
        if $\exists u \in F^c : f_m(u, v) = True$ **then**
          $u \leftarrow \{u \in F^{\text{c}} : f_{\text{m}}(u, v) = \text{True}\}$
          $M^{\text{c}} \leftarrow$ Extend mapping: $M^{\text{c}} \cup \{(u, v)\}$
          $\mathcal{G}^{\text{c}} \leftarrow$ Remove matched u from $V(\mathcal{G}^{\text{c}})$
          $F^{\text{c}} \leftarrow$ Update front: $F^{\text{c}} \setminus \{u\} \cup R(\mathcal{G}^{\text{c}})$
      if $|dom(M^c) \cap A_u| > |dom(M^*) \cap A_u|$ **then**
        $M^* \leftarrow M^{\text{c}}$ /* Store largest mapping */
      $s_{\text{max}} \leftarrow |A_{\text{u}}| - |A_{\text{u}} \setminus V(\mathcal{G}_{\text{excl}})| - (|F_{\text{con}}| - |F_{\text{com}}|)$
      if $(|dom(M^*) \cap A_u|) \geq s_{max}$ *or* $|dom(M^*)| = |\mathcal{S}|$ **then return** M^*
  $\mathcal{G}_{\text{excl}} \leftarrow$ Exclude vertices from search graph: $V(\mathcal{G}_{\text{excl}}) \setminus F_{\text{con}}$

(Line 4) and subsequently extending the search frontier solely upon removing a newly found match that subsequently leads to new root vertices (Line 15-16).

3.3.4 Limitations and Approximations

The efficacy of our approach is contingent on a set of assumptions. Firstly, the best performance is achieved when the computation graphs exhibit a recurring topological structure. Secondly, the model assumes substantial time-scale differences between what we term the *supervisor vertex* and other vertices. Finally, our approach assumes that vertices are stateful, i.e., vertices of similar labels are connected with one another. These assumptions are particularly well-suited for cyber-physical systems where components are stateful and run at fixed target frequencies, and where the supervisor vertex often takes the form of a slower, learning agent or an outer-loop controller. Moreover, the algorithm assumes that the computation graphs' structure does not depend on the data processed by the vertices. Specifically, we assume delays in the system are not a function of the internal states, outputs, or incoming inputs.

Identifying the minimal common supergraph is an NP-hard problem [67]. To manage this complexity, we make several approximations to Alg. 5. If all vertices are assumed to be stateful, then the constrained front F_{con} can contain at most one vertex for each label, i.e. $|F_{\text{con}}| = |\text{rng}(L)|$. Then, the worst-case time complexity for considering all topological sorts of the supergraph $\mathcal{S}$ and all combinations of F_{con} is $\mathcal{O}(2^{|\text{rng}(L)|} + |V|!)$ (Line 6-7 in Alg. 5). We alleviate this by considering only a single topological sort of $\mathcal{S}$ and a single combination per combination size k, reducing the worst-case time complexity to $\mathcal{O}(|\text{rng}(L) + 1| + |V|)$. We have found that these approximations do not significantly impact the resultant supergraph in our evaluations, as detailed in the ablation study in Appendix 3.B. Lastly, the sequence in which computation graphs are processed can affect the resultant supergraph. Similar to [71], this has not proven to have a significant impact in our evaluations.

3.4 Experimental Evaluation

The main focus of this work is an efficient approach to simulate delays in parallelized simulation on accelerator hardware. We present our experiments to show the capabilities of our approach and to support our key claims that our approach (i) emulates asynchronicity leading to more accurate simulation, (ii) efficiently handles time-scale differences and asynchronicity, resulting in higher parallelized simulation speeds than baseline approaches, (iii) scales to complex system topologies. In the remainder of this section, we will use *mcs* to refer to our proposed method.

3.4.1 Baselines

We outline three baseline methods for our experimental evaluation. The *sequential* baseline (*seq*) assumes no delays in computation graph processing, illustrating a conventional approach as shown in Fig. 3.1a. This baseline serves as a reference for evaluating the impact of realistic delays in simulations.

We then introduce two baselines that incorporate delays by randomizing predication masks in parallelized simulations, but differ in supergraph construction. Given the absence of existing methods that can handle the DAG constraint and partitioning requirements

for our supergraph (as discussed in Sec. 3.3.3), these baselines represent straightforward strategies for supergraph construction. Both baselines sequentially stack K layers in the supergraph, with each layer containing a vertex for every non-supervisor label and concluding with a final layer of a single supervisor vertex. This structure ensures the supergraph is a DAG and with its size as $|S| = K \times (|\text{rng}(L)| - 1) + 1$, thereby ensuring subgraph monomorphisms across partitions with an adequate number of layers. The *topological* baseline (*top*) sets K equal to the number of vertices in the largest partition. While this method guarantees a subgraph monomorphism with each partition, it can lead to disproportionately large supergraphs with sparse layer utilization. The *generational* baseline (*gen*), on the other hand, sets K as the maximum path distance across partitions. This approach is more space-efficient but also tends to over-include vertices, as it does not account for time-scale differences between vertices. Consequently, each layer incorporates every vertex label, even those infrequently used.

To evaluate these methods, we introduce the *supergraph efficiency* metric (η):

$$\eta = 100 \times \frac{1}{N} \sum_{i,j} \frac{|\mathcal{P}_{i,j}|}{|S|}$$

Here, N denotes the total number of partitions, with η indicating the mean partition size relative to the total supergraph size. This metric effectively quantifies the proportion of vertices actively utilized (unmasked) in emulating the computation graphs across episodes. Note that a 100% efficiency may not be achievable in practice, as it would imply that all partitions have an equal number of vertices.

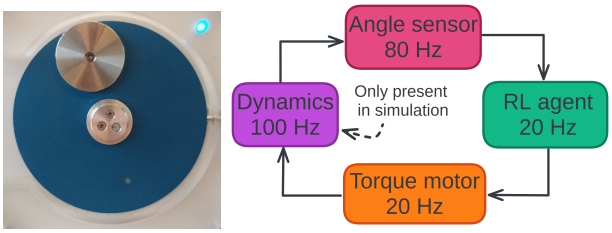
3.4.2 Performance

In this set of experiments, we aim to validate that randomizing predication masks during training enhances the fidelity of robotic simulations and our approach to identifying the supergraph leads to more efficient parallelized simulations. We validate the performance on two real-world systems: a pendulum swing-up task and a vision-based robotic manipulation task. We use two different control strategies, reinforcement learning (RL) and model predictive control (MPC), to demonstrate the utility of our approach in different real-world settings.

Pendulum swing-up task

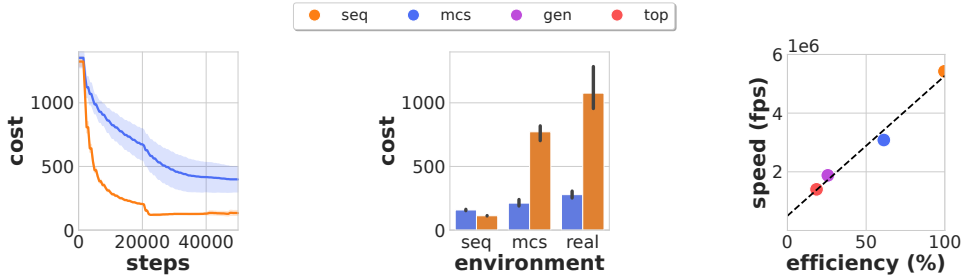
The pendulum swing-up task is a well-known RL benchmark with nonlinear, unstable, and underactuated dynamics sensitive to delays [45]. The choice for this task is deliberate; it highlights the core challenge of delay compensation in reinforcement learning. By demonstrating how neglecting delay simulation can impair policy transfer even in seemingly simple scenarios, we underscore the greater consequences for complex systems where delays are unavoidable and complexity is greater, as discussed in prior work [27, 72–74]. The simplicity of the task serves to clarify the fundamental importance of accounting for delays in sim2real approaches.

The experimental setup and control diagram are depicted in Fig. 3.5. A failure to emulate the asynchronous real-world interactions between components makes a simulation-trained policy ineffective when transferred to a real-world setting. Policies were trained using soft actor-critic (SAC) [75] in two simulators: one emulating delays (our approach: *mcs*) and



(a) Pendulum (b) Control diagram

Figure 3.5: Experimental setup and control diagram for the pendulum swing-up task. Panel (a) depicts the experimental setup, and panel (b) shows the control diagram.

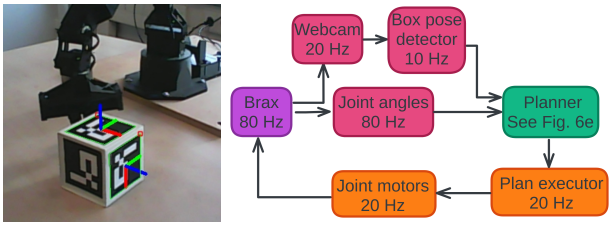


(a) Training curve (b) Sim2real (c) Speed

Figure 3.6: Sim2real evaluation of an RL policy trained to swing up a pendulum with (*mcs*) and without delays (*seq*). Panels (a) and (b) show the training curve and sim2real evaluation, respectively, while panel (c) shows the speed performance.

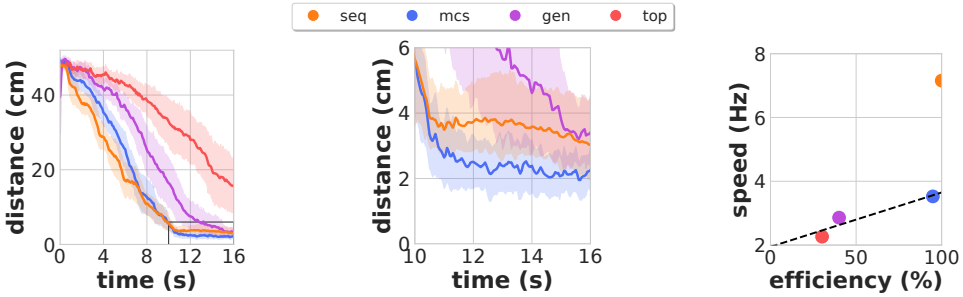
another without delays (sequential approach: *seq*). Note that the *gen* and *top* baselines are not included in the sim2real evaluation. This exclusion is due to their replication of the same effective computation graphs as *mcs*, leading to identical policy outcomes. Hence, we only consider these baselines later on in the simulation speed evaluation within this section. We record 10 computation graphs from the real-world system to identify a supergraph, partitioning and corresponding predication masks that were randomized during training. Each experiment was replicated five times with different random seeds and the results are presented in Fig. 3.6. Though the sequential (*seq*) approach exhibits quicker convergence and superior simulated performance, it underperforms in real-world tests compared to our approach that includes latency simulation during training. A smaller performance gap between simulation and reality suggests that our approach leads to more accurate simulation, yielding more effective real-world policies. This is further supported by cross-evaluations of the trained policies in each other's training environment, where *mcs* proved effective in both environments, unlike *seq*.

On average, it took 0.54 seconds to identify the supergraph and predication masks for the 10 recorded computation graphs, which is a one-time startup cost that is small compared to the total training and compilation time of 100 seconds. To establish the link between efficiency and simulation speed, we carried out a parallelized performance evaluation



(a) Box pushing (b) Control diagram

Figure 3.7: Experimental setup and control diagram for the box pushing task. Panel (a) depicts the experimental setup, and panel (b) shows the control diagram.



(a) Performance (b) Final accuracy (c) Planning rate

Figure 3.8: A comparison of four MPC strategies for a task where a manipulator moves a box to a target: three consider delays (*mcs*, *gen*, *top*) and one does not (*seq*). Panels (a) and (b) depict the mean convergence rate and final accuracy over 10 episodes with 95% confidence intervals, respectively, while panel (c) correlates these with the achieved replanning rate. The *seq* strategy, although faster initially, leads to less accurate movements due to ignoring delays. The *mcs* method, while replanning less frequently, achieves approximately 40% higher accuracy. Moreover, *mcs* exhibits the highest replanning rate with a smaller supergraph (*mcs*: $|\mathcal{G}| = 54$) compared to *gen* ($|\mathcal{G}| = 139$) and *top* ($|\mathcal{G}| = 223$) that also consider delays.

of the swing up-task on an RTX 3070 GPU. We deliberately measure simulation speed during policy evaluation rather than measuring the overall training time to clearly separate simulation speed improvements from any learning algorithm and training-related overhead. We compiled the supergraph with JAX [76] and randomized the predication masks across 1000 parallelized episodes. We used the supergraphs produced by our approach with backtracking $\beta = 5$ and both baseline methods and recorded the simulation frames per second (fps). As indicated in Fig. 3.6, our method notably outperforms other baselines that include delays, achieving an approximate simulation speed of 3 million fps. This improvement is largely attributed to a more compact supergraph. We observed a clear linear relationship between η and simulation fps, which is consistent with the inverse proportionality between simulation fps and supergraph size.

Manipulation task

In the manipulation task, a Viper 300x robotic manipulator moves a box to a target based on streaming webcam images. The goal is to minimize the distance between the box and a goal position. Our experimental setup and control diagram are shown in Fig. 3.7. Emphasizing the importance of delay simulation, we use a consumer-grade Logitech C170 webcam, chosen for its low resolution, modest frame rate, and high latency, to track the box's position and orientation.

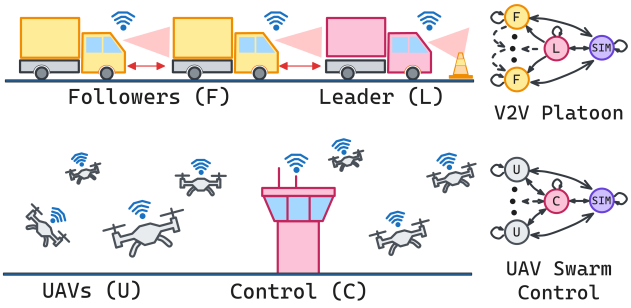
We adopt the MPC approach from [77], planning actions based on the most recent robot observations using the Cross Entropy Method (CEM) [78]. CEM, known for its efficient, derivative-free optimization, is particularly advantageous due to its parallelizability. Considering the contact-rich nature of box pushing, we opt for Brax [17] as our dynamics model within the MPC framework, instead of learning complex contact dynamics. Brax, a differentiable physics simulator, is optimized for GPU acceleration and effectively handles contact-rich tasks. In a similar approach, [79] recently used PhysX [15] to solve a box-pushing task. Our implementation employs CEM for three iterations, involving 75 samples per iteration and a planning horizon of two control steps, each lasting 0.15 seconds. We implement our approach using JAX [76] and execute it on an RTX 3070 GPU.

We evaluate four MPC strategies: three accounting for delays (*mcs*, *gen*, *top*) and one ignoring them (*seq*). Delay-inclusive strategies, following [77], use past plans to predict future box positions and orientations at action time. This prediction is based on the 10 recorded computation graphs of the system that are used to identify a supergraph, partitioning, and corresponding predication masks. On average, it took 1.53 seconds to identify the supergraph and predication masks for the 10 recorded computation graphs, which is a one-time startup cost that is small compared to the total evaluation time of 160 seconds. Due to their computational load, these strategies have a lower replanning rate compared to the delay-agnostic *seq*. The slower the replanning rate, the further into the future the planner must predict, increasing the likelihood of inaccurate predictions. As Fig. 3.8b shows, *mcs* achieves 40% higher accuracy than *seq*, despite less frequent replanning. Moreover, the *mcs* method also results in smoother operations than *seq*. The larger supergraphs in *gen* and *top* result in excessively slow replanning, significantly reducing convergence rates, and final accuracy. This illustrates the trade-off between accuracy and efficiency, where the improved accuracy must justify the additional computational load.

3.4.3 Scalability

The next set of experiments support the claim that our approach scales to complex system topologies. In Sec. 3.4.2, we showed that employing a supergraph with randomized predication masks can effectively emulate direct delay simulation. We also identified an approximate linear correlation between graph efficiency η and simulation speed. Next, we assess our method's scalability, analyzing various system topologies and modifying node counts and asynchronicity degrees to ascertain their effects on identifying efficient supergraphs. In this section, we consider two cyber-physical systems for which delay simulation is crucial: vehicle-to-vehicle (V2V) platooning [26, 74, 80] and unmanned aerial vehicle (UAV) swarm control [27]. Furthermore, a detailed analysis of the impact of different abstract topological characteristics on supergraph efficiency is provided in Appendix 3.A.

Fig. 3.9a illustrates the V2V platooning and UAV swarm control systems. In V2V



(a) Schematic (b) Topology

Figure 3.9: Panel (a) shows the V2V platooning and UAV swarm control systems, with the former comprising a leader and followers, and the latter a central controller and UAVs. Panel (b) depicts their respective topologies, where every component communicates at 20 Hz with each other, while the simulator runs at 200 Hz. The leader and controller are chosen as the supervisor nodes, respectively.

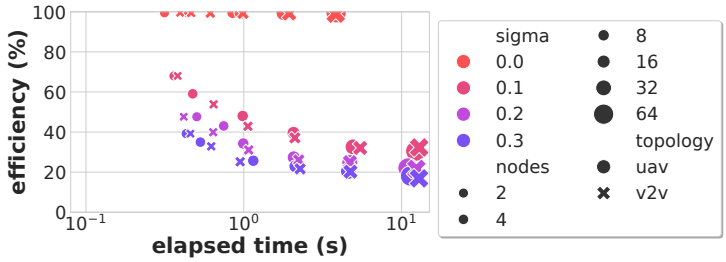
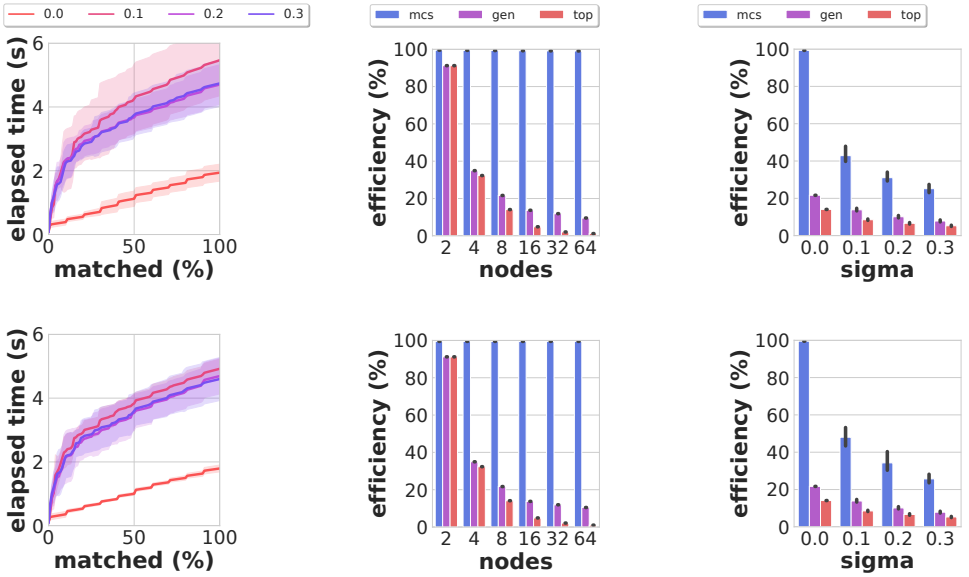


Figure 3.10: The computational complexity versus efficiency for different topologies, asynchronicity levels, and node counts, highlighting their impact on performance.

platooning, vehicles maintain a set distance and speed, following a leader. This requires each vehicle to respond to the leader, highlighting the necessity for delay-aware simulation. Vehicles communicate with the leader and the vehicle ahead. For UAV swarm control, a central entity directs the UAVs to prevent collisions and achieve formation, with UAVs communicating solely with this controller. Additionally, each component connects to the simulator to enable physics simulation. Accurate simulations require delays simulation in both systems, as discussed in [26] and [27]. Within systems encompassing N nodes, there is a single simulator and one leader or controller designated as the *supervisor*, alongside $N-2$ Followers and UAVs, respectively, as illustrated in Fig. 3.9b. The simulator runs at 200Hz, while all other nodes communicate with each other at a target rate of 20Hz. The effective sampling time of every node i is computed as $\Delta t_{i,k} = \Delta t_i + \max(0, x_k \Delta t_i)$, where $x_{i,k}$ is the delay of node i, experience during sequence number k, scaled with the node's nominal sampling time. An Ornstein-Uhlenbeck (OU) process [81] is used to model every node's delay to reflect the temporal correlation of delays, defined as follows:

$$x_k = \theta x_{k-1} + \sigma v, \tag{3.2}$$



(a) Elapsed time (b) Size (c) Asynchronicity

Figure 3.11: Top row shows V2V plots, while the bottom row shows UAV plots. Panel (a) shows the elapsed time for completion with $N = 32$ nodes across various asynchronicity. Initial episodes are time-intensive due to numerous preliminary partial matches, followed by a consistent linear time scaling in processing time. Panels (b) and (c) compare the efficiency of *mcs* (our approach), *top*, and *gen*. In panel (b) the number of nodes is varied with no asynchronicity ($\sigma = 0$), while in panel (c) the asynchronicity levels are varied with a fixed size of $N = 8$ nodes.

where θ is a correlation coefficient, σ is the standard deviation, and ν is a Gaussian random variable with zero mean and unit variance. The standard deviation of an OU process is related to the standard deviation of a Gaussian distribution with $\sigma_g = \sqrt{\frac{\sigma^2}{2\theta}}$. We artificially generate computation graphs for the topologies depicted in Fig. 3.9b, varying the asynchronicity level $\sigma \in \{0, 0.1, 0.2, 0.3\}$ and the number of nodes $N \in \{2, 4, 8, 16, 32, 64\}$. We replicate each experiment 5 times using different random seeds. For each configuration, we generate 10 computation graphs, each running for a duration of 10 seconds. Example computation graphs are presented in Appendix 3.C. We employ Alg. 4 to identify a supergraph on a single core of an Intel Core i9-10980HK and compare its performance with two baseline approaches.

Fig. 3.10 presents an analysis of our method's computational complexity in constructing the supergraph, considering both the computation graph's characteristics (N, σ) and topology (*v2v*, *uav*). We observe that efficiency is inversely related to the asynchronicity level and, to a lesser extent, to the number of nodes. Moreover, a decrease in efficiency correlates with an increase in computation time, primarily because fewer complete matches are found. Nevertheless, the one-time upfront cost of identifying the supergraph is usually minor when compared to the overall simulation time, substantiating our claim that our approach scales effectively to complex system topologies. Fig. 3.11a details the required computation

time under varying levels of asynchronicity, as it processes all recorded computation graphs. The initial episodes incur higher computational costs due to the increased computational overhead of handling numerous partial matches (Line 6-7 in Alg. 5), while subsequent episodes demonstrate linear scaling in time.

Fig. 3.11b compares the performance of our algorithm with baseline approaches for different numbers of nodes when there is no asynchronicity ($\sigma = 0$). Our approach achieves a 100% efficiency, whereas the efficiency of baseline approaches declines rapidly as the number of nodes increases. Fig. 3.11c demonstrates the performance of our algorithm compared with baseline approaches for different levels of asynchronicity when the topologies comprise $N = 8$ nodes. As asynchronicity increases, partitions become more dissimilar, and the efficiency of our approach does decline, yet it remains multiples higher than that of the baseline approaches. The supergraphs generated by our method, along with those from the baseline approaches, are depicted in Appendix 3.C.

In summary, our evaluation suggests that our method successfully emulates asynchronicity, offering more accurate and faster parallelized simulations compared to baseline approaches. At the same time, our method scales well to larger system topologies by finding more efficient supergraphs than baseline methods. Thus, we have substantiated all our key claims through this experimental evaluation.

3.5 Related Work

Accelerated Physics Simulation Accelerated physics simulators like Brax [17], MJX [18], and PhysX [15] are designed for GPU execution. However, they lack features for simulating delays between their physics engine and other components, such as sensors and actuators. Moreover, to mimic complete systems, these simulators must be extended with controllers and perception modules. Yet, these extensions typically interact with simulators sequentially, ignoring the concurrent and asynchronous nature of real-world systems. Our approach builds on this by dividing these simulators into separate components, facilitating the simulation of asynchronous interactions between them.

Adressing Asynchronicity and Delays The ORBIT framework [39] and research by [82] have explored integrating delays into robotic simulations. While ORBIT introduces actuator delays to PhysX, it overlooks the asynchronicity between other system components. [82]'s work centers on compensating for system delays in the learning algorithm, not addressing the dynamic interactions among delayed components. In contrast, our method extends beyond actuator delays, encompassing asynchrony across all components.

Minimum Common Supergraph Our approach addresses a variant of the *minimal universal supergraph* (MUG) problem, which seeks the smallest supergraph, i.e., the mcs, containing all graphs in a given set as a subgraph [83]. Unlike the brute-force exact algorithm presented in [83], which is suitable only for small graph sets, our approximate greedy algorithm is capable of handling graphs with more than 2000 vertices. In [71], an iterative update strategy, based on [70], is utilized to approximate the mcs. Our method shares similarities but satisfies an additional constraint: the resulting mcs must remain acyclic post-merge. Furthermore, our extended objective is to efficiently partition a provided

set of larger graphs into smaller subgraphs before finding the mcs for these partitioned subgraphs. In contrast, [83] and [71] start from a given and static collection of graphs and focus strictly on the identification of the mcs, meaning the partitioning we perform together with the supergraph identification is already a given in their scenario.

Both [83] and [71], and our method, (approximately) solve the *maximum common subgraph* problem as a subroutine to find the *minimum common supergraph* (mcs) [84, 85]. However, our focus is on subgraph monomorphisms, which allow for additional edges in the subgraph, rather than induced subgraph isomorphisms, which require a one-to-one correspondence between every node and edge in the subgraph and target graph. To efficiently identify the largest mapping, we introduce an algorithm that leverages the acyclic nature of our mcs that accelerates the search for a large approximate mapping. Note that our algorithm restricts the largest mapping to connected subgraphs, potentially overlooking larger disconnected mapping candidates.

3.6 Conclusion

In this chapter, we introduced a method for efficiently simulating inherently asynchronous systems on accelerator hardware. Our approach leverages recorded computation graphs from real-world operations to accurately model asynchronicity and time-scale differences. The experiments suggest that our approach provides a scalable, efficient, and accurate means for simulating cyber-physical systems. We evaluated our method in two real-world scenarios against baselines and confirmed its efficacy in emulating asynchronicity and handling time-scale differences efficiently. Our work opens avenues for developing fast and accurate cyber-physical system simulations. Finally, our approach holds promise for enhancing the integration of other machine learning algorithms that generate dynamic graphs into frameworks like Jax [76], by aligning dynamic computation graphs with static ones.

Appendix 3.A: Scalability Analysis

In this scalability study, we focus on artificially generated computation graphs, as they allow us to systematically vary the number of nodes, the level of asynchronicity, and the topology of the graph. We consider three different topologies: *unidirectional*, *bidirectional*, and *unirandom*, depicted in Fig. 3.12.

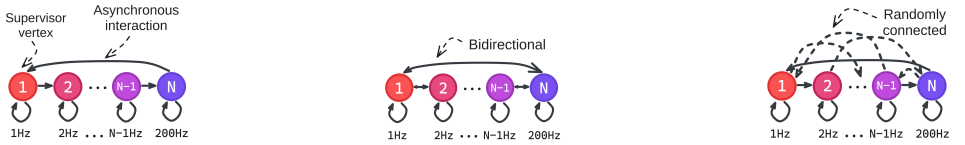
The nominal sampling time of each node is set according to the node's index i as $\Delta t_i = \frac{1}{i}$ s, except for the last node's sampling time which is set to $\Delta t_N = \frac{1}{200}$ s. These topologies resemble cascaded control schemes that are common in robotic systems, with slower learning-based nodes and faster simulator nodes with intermediate controllers, estimators, sensors and actuators. The effective sampling time is computed with Eq. (3.2) as further detailed in Sec. 3.4.3.

As in Sec. 3.4.3, we consider a different number of nodes $N \in \{2, 4, 8, 16, 32, 64\}$, and a varying levels of asynchronicity $\sigma \in \{0, 0.1, 0.2, 0.3\}$, and replicate every experiment 5 times using different random seeds. For each configuration, we generate 20 computation graphs, each running for a duration of 100 seconds.

Figures 3.13a, 3.13b, and 3.13c illustrate the performance of our algorithm for different numbers of nodes when there is no asynchronicity ($\sigma = 0$). Our approach achieves a 100% efficiency for the unidirectional topology, whereas the efficiency of baseline approaches declines rapidly as the number of nodes increases. The superior efficiency of our approach in the unidirectional topology is attributed to its fewer connections. Figures 3.13d, 3.13e, and 3.13f demonstrate the performance of our algorithm for different levels of asynchronicity when the network comprises $N = 8$ nodes. As asynchronicity increases, partitions become more dissimilar, and the efficiency of our approach does decline, yet it remains multiples higher than that of the baseline approaches.

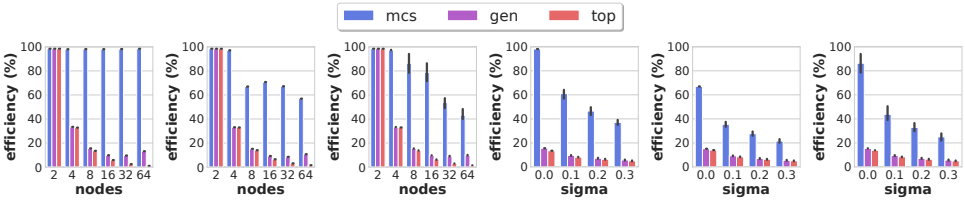
Figure 3.14 presents an analysis of our method's computational complexity in constructing the supergraph, considering both the computation graph's characteristics and topology, and the scaling of supergraph search complexity over all recorded computation graphs. Figures 3.14a, 3.14b, and 3.14c detail our algorithm's complexity under varying asynchronicity levels through time as it processes all recorded computation graphs. The initial episodes incur higher computational costs due to the increased computational overhead of handling numerous partial matches (Line 6-7 in Alg. 5), while subsequent episodes demonstrate linear scaling in time.

In Fig. 3.15, we again observe that efficiency is inversely related to both the asynchronicity level and the number of connections per node within a topology. Specifically, the unidirectional topology outperforms the bidirectional and unirandom topologies due to its fewer edges. A decrease in efficiency correlates with an increase in computation time, primarily because fewer complete matches are found, which is consistent with the results in Sec. 3.4.3. While the most substantial contributor to computation time is the number of nodes in the topology, it does not affect efficiency as similar efficiency is achieved with different numbers of nodes.



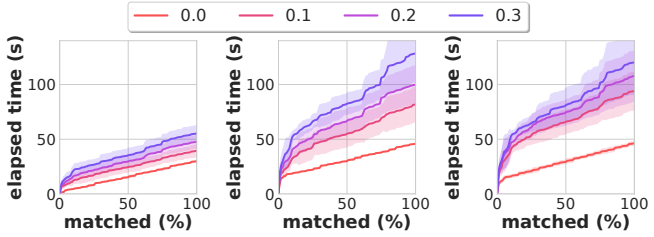
(a) Unidirectional (b) Bidirectional (c) Unirandom

Figure 3.12: Three abstract topologies to evaluate the scalability of our approach. (a) *Unidirectional*: each node has a single outgoing connection. (b) *Bidirectional*: each node has two outgoing connections. (c) *Unirandom*: akin to *Unidirectional*, but with an extra random outgoing connection per node.



(a) Unidirectional (b) Bidirectional (c) Unirandom (d) Unidirectional (e) Bidirectional (f) Unirandom

Figure 3.13: Efficiency comparison of *mcs* (our approach), *top*, and *gen*. In panels (a-c) the number of nodes is varied with no asynchronicity ($\sigma = 0$), while in panels (d-f) the asynchronicity levels are varied with a fixed size of $N = 8$ nodes.



(a) Unidirectional (b) Bidirectional (c) Unirandom

Figure 3.14: Performance analysis of computational complexity and efficiency. Panels (a-c) show the elapsed time for completion with $N = 32$ nodes across various asynchronicity levels and topologies. Initial episodes are time-intensive due to numerous preliminary partial matches, followed by a consistent linear time scaling in processing time.

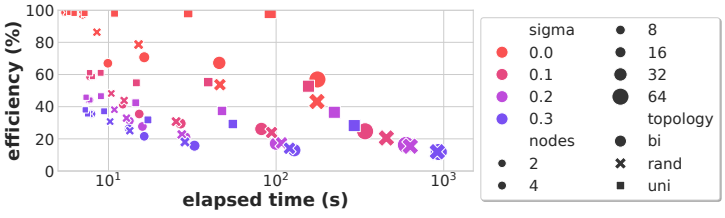
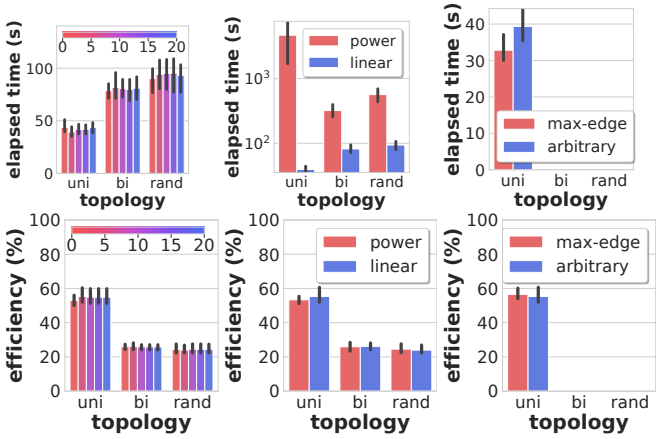


Figure 3.15: Complexity and efficiency versus node count, asynchronicity, and topology, highlighting their impact on performance.

Appendix 3.B: Ablation Study



(a) Backtrack (b) Combination (c) Sort

Figure 3.16: Ablation study on topologies with $N = 32$ and $\sigma = 0.1$, examining computational complexity and efficiency. Sub-figures show: (a) Effects of varying β; (b) Efficiency-impact of considering one (*linear*) vs. all combinations per size k (*power*); (c) Comparison of *arbitrary* and *max-edge*topological sorts.

In this study, our goal is to substantiate that our approach employs simplifications discussed in Sec. 3.3.4 that reduce computational complexity without significantly affecting performance. For this ablation study, we focus on the topologies in Fig. 3.12 with $N = 32$ and $\sigma = 0.1$, ablating the proposed simplifications.

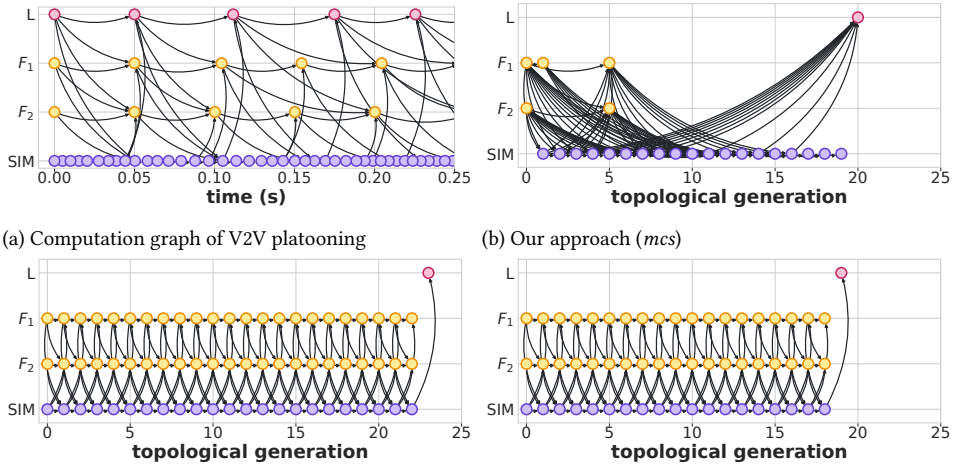
Fig. 3.16 illustrates that the benefits of *backtracking* are limited. However, it neither increases the computational complexity of our approach nor adversely affects efficiency.

We also analyzed the effect of considering only a single combination for each size k, as opposed to exploring all combinations. Fig. 3.16 demonstrates that this simplification has negligible impact on efficiency but considerably reduces the computational complexity (note the log-scale). It is worth noting that only considering a single combination even seems to perform slightly better in some cases. This outcome, while not statistically implausible, may also be caused by other factors, such as the order in which the graphs are processed. Our hypothesis centers on the specific nature of the computation graphs generated by cyber-physical systems. These graphs tend to exhibit a relatively fixed structure, meaning the variety of topological orderings is considerably constrained compared to more generic graphs. Consequently, this structural rigidity could diminish the advantages we might expect from checking all combinations.

Lastly, we explored the implications of using a single topological sort. Rather than exhaustively considering all topological sorts—an approach that would be computationally prohibitive—we compared the effects of using an arbitrary sort versus a max-edge sort. The max-edge sort of the supergraph is defined as one that accommodates the maximum number of potential edges (i.e., constraints) and therefore increases the chance of finding a

match in Alg. 5. Due to the inherent unidirectionality of the unidirectional topology, the max-edge sort arranges vertices of lower indices before those of higher indices. Since we lack max-edge sorting criteria for bidirectional and unirandom topologies, we limited this part of the study to the unidirectional topology. Fig. 3.16 shows that this simplification has negligible impact on efficiency.

Appendix 3.C: Graphs



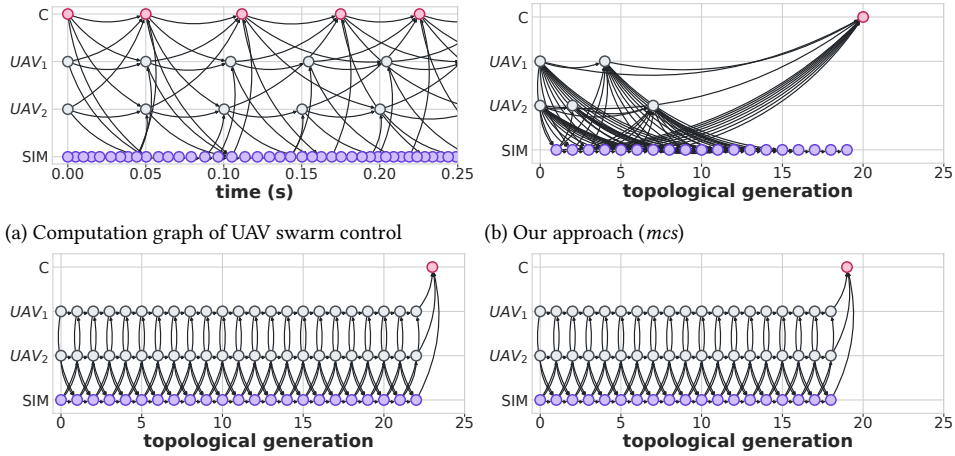
(a) Computation graph of V2V platooning

(b) Our approach (*mcs*)

(c) Topological (*top*)

(d) Generational (*gen*)

Figure 3.17: Panel (a) presents a segment of a computational graph corresponding to the platooning scenario in Fig. 3.9b with $N = 4$ and $\sigma = 0.2$. Vertices of identical color correspond to the same periodic computation unit, and edges represent data dependencies. Panels (b-d) illustrate the supergraphs generated by our method (*mcs*), as well as the topological (*top*) and generational (*gen*) methods. Our approach yields a supergraph with a reduced number of vertices, indicating enhanced efficiency in identifying commonalities across the computation graphs.



(a) Computation graph of UAV swarm control

(b) Our approach (*mcs*)

(c) Topological (*top*)

(d) Generational (*gen*)

Figure 3.18: Panel (a) presents a segment of a computational graph corresponding to the uav swarm scenario in Fig. 3.9b with $N = 4$ and $\sigma = 0.2$. Vertices of identical color correspond to the same periodic computation unit, and edges represent data dependencies. Panels (b-d) illustrate the supergraphs generated by our method (*mcs*), as well as the topological (*top*) and generational (*gen*) methods. Our approach yields a supergraph with a reduced number of vertices, indicating enhanced efficiency in identifying commonalities across the computation graphs.

4

Accuracy: Estimating Dynamics and Delays of Graph-Based Simulations

Although flexible and fast simulations are essential for robotic learning, it is often challenging to obtain accurate simulations that closely match real-world dynamics due to unmodeled effects such as delays. These inaccuracies often result in policies that work well in simulation but fail in real-world deployment.

This chapter addresses this gap by presenting a framework, REX (Robotic Environments with jaX), for improving the accuracy of simulations through the estimation of delays and system dynamics based on real-world data. The framework builds on the graph-based architecture from earlier chapters, enhancing fidelity in sim2real transfers by simulating asynchronous operations and compensating for various types of delays.

This chapter is partly based on *B. van der Heijden, J. Kober, R. Babuska, and L. Ferranti, (2025). "REX: GPU-Accelerated Sim2Real Framework with Delay and Dynamics Estimation", Transactions on Machine Learning Research (TMLR)* [86].

4.1 Introduction

Sim2real, the transfer of control policies from simulation to the real world, is crucial in robotics thanks to its ability to solve tasks efficiently without the risks associated with real-world learning [22, 31]. With recent advancements in physics simulation on accelerator hardware [15–18], parallelized simulations have greatly reduced training times for complex tasks [22, 87]. However, discrepancies between simulation and reality, such as unmodeled dynamics, often reduce the effectiveness of these policies in real-world applications. Addressing this 'sim2real' gap is essential for effective transfer of policies from simulation to the real world.

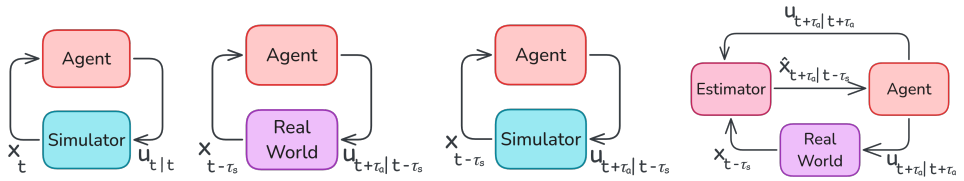
A critical yet often overlooked issue in sim2real transfer is the impact of latency in real-world systems, which can degrade performance [29, 31, 32, 88]. The real world is inherently asynchronous, with delayed sensor data causing agents to act on outdated information. Additionally, slow policy evaluations can further delay the agent's actions, compounding these latency issues and leading to suboptimal performance. To mitigate these effects, Fig. 4.1 illustrates two common compensation strategies: simulating delays during training (Fig. 4.1c) [31, 63, 88] and using an estimator to predict future states (Fig. 4.1d) [77, 89–91]. However, both strategies have limitations. Delay simulation complicates training because the agent's input must include a history of observations and actions to restore the Markov property, while an estimator requires accurately modeled system dynamics and delays, which are often difficult to identify [92].

The hierarchical and asynchronous nature of robotic systems further complicates accurate and efficient simulation on accelerator hardware. Unlike conventional RL, which assumes a single, synchronized environment [23], robotic systems consist of interconnected models operating at different rates, with asynchronous communication introducing complexities like inter-model latencies and stochastic dynamics [24, 64], leading to irregular computation patterns. Irregular execution paths require serialization, reducing GPU efficiency, and while simulating time-scale differences improves sim2real accuracy, it further exacerbates this inefficiency [25].

The main contribution of this chapter is a sim2real framework, REX (Robotic Environments with jaX), that introduces a graph-based simulation model with latency effects, optimized for parallelization on accelerator hardware. The framework's innovation lies in its ability to simulate asynchronous, hierarchical systems by explicitly modeling computation, communication, actuation, and sensing delays, while incorporating delay compensation strategies for improved sim2real transfer. Parallelization in both state and parameters allows for simultaneous estimation of system dynamics and delays from real-world data, efficiently minimizing the sim2real gap. Additionally, it supports real-world deployment by distributing computations across CPU cores and accelerators, optimizing for latency and performance.

For RL and robotics practitioners, this framework offers several advantages. It enables the modeling of both simulated and real-world systems through a unified, ROS-like graph-based pipeline [24]. The framework supports accelerated training speeds familiar to RL workflows and reduces the sim2real gap by refining models with real-world data. Integration with the JAX [76] ecosystem further supports advanced RL training and optimization [93–96].

Building on these advantages, we make four key claims: Our framework (i) enables



(a) Naiive (b) Real world (c) Delay simulation (d) Delay compensation

Figure 4.1: A policy trained in simulation (a) may perform suboptimally in the real world (b) due to delays. The notation $u_{t+\tau_a|t-\tau_s}$ denotes that an action u is applied at $t+\tau_a$ based on information up to $t-\tau_s$, where τ_a and τ_s are the actuation and sensing delays, respectively. By simulating these delays during training (c), the sim2real gap with (b) can be reduced. Alternatively, an estimator (d) can predict future states and compensate for delays, improving policy transfer from (a) to (d). The notation $x_{t+\tau_a|t-\tau_s}$ denotes that a state x is predicted at $t+\tau_a$ based on information up to $t-\tau_s$.

4

the identification of both dynamics and delays from real-world data, (ii) implements delay compensation and simulation techniques that are essential for effective sim2real transfer, (iii) facilitates efficient parallelized offline simulation on accelerator hardware, (iv) supports real-time online processing capabilities that meet the latency and performance requirements of real-world systems. These claims are supported by experiments on two real-world systems. The pendulum swing-up task clearly demonstrates how neglecting delay simulation can impair policy transfer, highlighting the need for delay-aware approaches, while the quadrotor task shows scalability to more complex robotic systems. The documentation, tutorials, and our open-source code can be found at `https://bheijden.github.io/rex/`. A video recording of the real-world experiments is available at `https://youtu.be/7j30LUjTx_I`.

4.2 Related Work

Sim2Real Frameworks Sim2real frameworks such as Orbit [39], Drake [40], and EAGERx [29] facilitate the transfer of control policies from simulation to real-world settings. However, they generally do not include direct support for delay or dynamics identification from real-world data. Our framework addresses this gap by integrating these capabilities directly into the framework. Orbit utilizes Nvidia PhysX for parallelized simulations on accelerator hardware [15]. Our framework is based on JAX [76] to support parallelized computation on accelerator hardware, while also enabling automatic differentiation. Moreover, our framework, like EAGERx [29], is not restricted to a specific simulator, as long as the simulator is compatible with JAX, such as Brax [17] or the MJX extension of MuJoCo [18]. This flexibility enables users to select and extend engines as needed within the graph-based model. Tab. 4.1 provides a feature comparison between REX and related sim2real frameworks.

Delay Estimation System identification involves estimating the system's dynamics from input-output data and is a well-established area of research [97]. Traditional methods primarily focus on linear systems, often utilizing least-squares optimization techniques

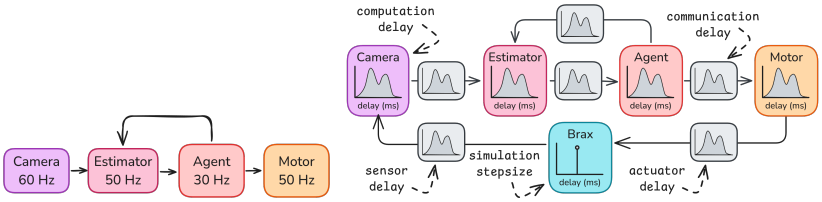
	REX	Orbit [39]	Drake [40]	EAGERx [29]
Multi-Sim Compatible	✓	✗	✗	✓
GPU Accelerated	✓	✓	✗	✗
Gradient Information	✓	✗	✓	✗
Delay Simulation	✓	✓	✓	✓
Delay Estimation	✓	✗	✗	✗
Dynamics Estimation	✓	✗	✗	✗

Table 4.1: A feature comparison between REX and related sim2real frameworks.

[98], while more recent efforts have extended to nonlinear systems [99]. Recent advances leverage the differentiability of general-purpose simulators to estimate complex system dynamics [100–102]. Our approach builds on these advancements by extending simulators with delay dynamics, allowing for the joint estimation of both system dynamics and delays. Instead of gradient-based methods, we use evolutionary strategies [103], which we found to be less susceptible to local minima and better utilize the parallelism of modern hardware [95].

Delay Simulation Frameworks like Drake, EAGERx, and Orbit provide support for fixed delay simulation [29, 39, 40]. Our framework, however, extends this capability by supporting stochastic delay simulation using Gaussian Mixture Models (GMMs). Additionally, it incorporates correlations between delays by considering the system's topology and communication structure during simulation. Although our framework allows for correlated delays, these delays are data-independent and do not change based on the simulated data. For example, even if an object detection algorithm takes longer to process when multiple objects are in view, our simulated delays remain the same regardless of the number of detected objects.

Delay Compensation Delay compensation in sim2real has been addressed through various methods. Algorithmic approaches for compensating delays have been proposed by [82, 104]. Other studies have enhanced sim2real performance by simulating delays during training and using a history of observations and actions as policy inputs [29, 31]. As part of their approach, [22] modified IsaacGym [105] to include a custom actuator model from [21], which accounts for control signal delays caused by hardware/software layers. These methods teach policies to handle delays without compensating for them directly during real-world execution. Direct compensation techniques, such as the Smith predictor [89], have long been used in robotics to manage delays from sensors, actuators [90, 91], and planning latency [77]. In our work, we demonstrate that by compensating for delays during execution, we can eliminate the need for delay simulation during training, resulting in a more efficient training process while maintaining high performance in real-world applications.



(a) Real-world system (b) Simulated system with delays

Figure 4.2: Comparison between a real-world system setup and a simulated system with integrated delays. The real-world system (a) operates with different nodes at specified rates, while the simulated system (b) incorporates various types of delays to closely mimic real-world timing behaviors, including sensor, actuator, communication, and computation delays.

4.3 Our Sim2Real Framework

In this section, we present our framework for sim2real transfer in robotics, focusing on accurately modeling and compensating for the asynchronous interactions and delays encountered in real-world systems. In the following, we will first describe the graph-based architecture that facilitates asynchronous message passing and delay modeling. We will then detail the three runtime configurations designed for simulation, accelerated training, and real-time deployment. Finally, we will cover the integration of system identification techniques and delay compensation strategies to bridge the gap between simulation and reality.

4.3.1 Overview

The central element of our framework is the node, which represents a discrete unit of computation or sensing, operating asynchronously within the system. Nodes are designed to run at specified rates, processing inputs, maintaining state, and generating outputs. In our approach, both real-world and simulated systems are implemented as networks of these nodes, where communication occurs via directed edges, as shown in Fig. 4.2a. Each node's operation is defined by a step function that determines its behavior, transforming inputs into outputs. For example, nodes can represent various components such as cameras, agents, or motors, each handling specific tasks like sensing, control, or actuation. This modular design allows for flexible state, time, and action abstractions, supporting the modeling of complex interactions in a decentralized manner. Nodes are interconnected in a directed graph, facilitating asynchronous message passing and enabling nodes to operate at different rates. This design also enables the swapping of real-world nodes with simulated nodes, resulting in a unified software pipeline that can be used for sim2real transfer.

Asynchronous operations are inherent in real-world systems due to network transmission times, processing lags, or mechanical response times, which introduces delays into the system dynamics. To address this, we introduce a delay simulation model that captures both deterministic and stochastic delays, incorporating realistic timing behavior through delay distributions for communication, computation, sensor, and actuator delays. As shown in Fig. 4.2b, our model explicitly defines these delays as non-negative distributions, ensuring that the timing characteristics of the simulated environment closely match those of the real world. While this provides the structure for delay simulation, the challenge of estimating

the correct delay parameters is addressed later in Sec. 4.3.3.

The framework supports multiple communication protocols to manage the flow of messages between nodes, allowing users to specify whether each communication channel should be blocking or non-blocking. A blocking channel ensures that a receiver node waits for the most recent message before processing, which minimizes latency in real-time systems by avoiding outdated information. However, blocking channels can introduce instability if delays cause unforeseen propagation through the graph; in such cases, non-blocking channels may be preferable. For example, an estimator node might opt for non-blocking behavior to continue predicting the system's state when sensor messages are delayed, allowing the controller to maintain responsive operation.

4.3.2 Runtimes

Our framework leverages JAX [76] for efficient computation, utilizing its ability to perform just-in-time (JIT) compilation and automatic differentiation, which are crucial for high-performance machine learning applications. Nodes are defined using a generic interface, with parameters, states, and outputs specified using data structures that can be statically analyzed, as shown in Fig. 4.3a. This approach allows for ahead-of-time (AOT) compilation of the `step` method (Fig. 4.3a, Line 12) on various architectures, including CPUs and GPUs, thereby reducing latency. By compiling nodes in this manner, they can be seamlessly employed across different runtime modes without modification, ensuring flexibility and efficiency in both real-world and simulated environments. Our framework supports three distinct runtime modes, each tailored for different stages of development, training, and deployment: `WALL_CLOCK`, `SIMULATED`, and `COMPILED`.

The **`WALL_CLOCK`** runtime is designed for real-time execution on physical hardware, operating at real-time speed with each node's step function running asynchronously at its designated rate (Fig. 4.3b, lines 1-14). Nodes can be compiled to run on dedicated hardware resources such as separate CPU cores or accelerator hardware, minimizing latency (Fig. 4.3b, Line 17). After initializing the state of the graph, which aggregates the states of all nodes, the graph can be executed for a specified number of steps while recording the outputs and their corresponding timestamps (Fig. 4.3b, lines 19-24).

The **`SIMULATED`** runtime enables faster-than-real-time simulation, allowing for accelerated testing and development without real-time constraints. Message passing is based on simulated timestamps that are generated based on the communication protocol of every connection (blocking or non-blocking) and specified delay distributions, replicating real-world asynchronous effects (Fig. 4.3c, lines 1-14).

The **`COMPILED`** runtime further leverages accelerator hardware like GPUs or TPUs for parallelized execution by enabling the compilation of entire computation graphs into a single function. This makes this runtime suitable for tasks such as training RL policies and large-scale system identification that can leverage massive parallelism. Data flows from other runtimes (e.g., (Fig. 4.3b, Line 26)) are converted into a computation graph (Fig. 4.3c, Line 21) and compiled for parallel execution (Fig. 4.3c, lines 22-26), encoding the asynchronous effects of real-world interaction or simulated delays and enabling parallel execution on accelerator hardware. By supporting these three runtime modes, our framework provides comprehensive flexibility for a wide range of applications, from real-time deployment to parallelized system identification and policy training.

```python
class Agent(BaseNode):
  def init_params(self, rng, graph_state):
    return PyTree(a=..., b=...)

  def init_state(self, rng, graph_state):
    return PyTree(x1=..., x2=...)

  def init_output(self, rng, graph_state):
    return PyTree(y1=..., y2=...)

  # AOT jit-compile with graph.warmup()
  def step(self, step_state):
    ss = step_state  # Shorten name
    # Read params, and current state
    params, state = ss.params, ss.state
    # Current episode, sequence, timestamp
    eps, seq, ts = ss.eps, ss.seq, ss.ts
    # Grab the data, and I/O timestamps
    cam = ss.inputs['cam'] # connectd node
    cam.data, cam.ts_send, cam.ts_recv
    # Compute new state, and output
    new_state = PyTree(x1=..., x2=...)
    output = PyTree(y1=..., y2=...)
    # Update step_state for next step call
    new_ss = ss.replace(state=new_state)
    return new_ss, output # Sends output
```

(a) Node definition

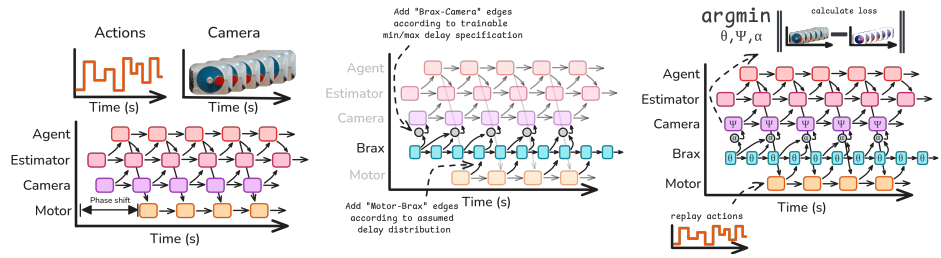
```python
# Real-world nodes
cam = Camera(rate=60)
agent = Agent(rate=30)
motor = Motor(rate=50)
nodes = [cam, agent, motor]
# Connect
agent.connect(cam) # Async msging
motor.connect(agent, # Last 2 msgs
              block=True, window=2)
# Runtime: WALL_CLOCK
# Used for real-time operation
graph = Graph(agent, nodes,
              Clock.WALL_CLOCK,
      RealTimeFactor.REAL_TIME)
# Ahead-of-time compilation of
# every node's .step() method
graph.warmup(devices=...)
# Run the graph at agent's rate
gs = graph.init() # Graph state
for i in range(100):
  gs = graph.run(gs)
graph.stop()  # Halts all nodes
# Gather data (outputs, timings)
record = graph.get_record()
# Convert to data flow
df = record.to_graph()
```

(b) Real-world runtime

```python
# Simulation nodes & connections
cam = SimCam(rate=60, delay=Gauss(0.05, 0.01))
agent = Agent(rate=30, delay=Gauss(0.02, 0.01))
motor = SimMotor(rate=50, delay=Gauss(0.04, 0.01))
brax = Brax(rate=100, delay=Deterministic(0.01))
nodes = [brax, cam, agent, motor]
brax.connect(motor, delay=Gauss(0.01, 0.01))
cam.connect(brax, delay=Gauss(0.01, 0.01))
agent.connect(cam, delay=Gauss(0.01, 0.01))
motor.connect(agent, delay=Gauss(0.01, 0.01),
              window=2, block=True)
# Runtime: SIMULATED (no throttling)
graph = Graph(agent, nodes, Clock.SIMULATED,
              RealTimeFactor.FAST_AS_POSSIBLE)
graph.warmup(devices=...)  # JIT compilation
gs = graph.init()  # Graph state
for i in range(100):  # Simulates 100 steps
    gs = graph.run(gs)
graph.stop()  # Halts all nodes
# Simulated data flow to computation graph
cg = graph.get_record().to_graph().augment(nodes)
# Runtime: COMPILED (1000 parallel rollouts)
graph = CompiledGraph(agent, nodes, cg)
rngs = jax.split(jax.random.PRNGKey(0), num=1000)
gss = jax.vmap(graph.init)(rngs)  # 1000 states
rollout = jax.vmap(graph.rollout)(gss, rngs) # run
```

(c) Simulation runtimes

Figure 4.3: Node definitions (a) use generic PyTrees that allow for compilation across different architectures for reduced latency. Examples of runtime configurations, showing different execution modes: WALL_CLOCK (b, Line 13) for real-time operation on physical hardware, SIMULATED (c, Line 13) for simulating without real-time constraints, and COMPILED (b, Line 23) for parallelized execution on accelerator hardware. Variable names and notations were slightly shortened for clarity and space.



(a) Collected data and data flow (b) Computation graph (c) Optimize

Figure 4.4: System identification example applied to the system in Fig. 4.2a. (a) Data collection from the real-world system, including sensor data and timing information. (b) Construction of a computation graph that integrates the data flow with simulated nodes for dynamics and hidden delay identification. Motor-Brax edges are added based on a specified delay distribution, while Brax-camera edges follow a trainable min-max delay specification. (c) Optimization of simulation parameters and delays to minimize discrepancies between simulated and real-world behaviors, focusing on the delay interpolation parameter α and the parameters ψ, θ for the camera and Brax nodes.

4.3.3 System Identification

System identification is crucial for minimizing the sim2real gap by ensuring that the simulated model closely mirrors the real-world system. Our framework facilitates this by identifying both the dynamics and delays inherent in real-world systems, allowing for more accurate simulation and effective delay compensation. In the following, we detail how to build and optimize a tailored computation graph from the real-world data collected to estimate system dynamics and delays (see Fig. 4.4). Data is collected from the real-world system using the `WALL_CLOCK` runtime, logging not only sensor and actuator data but also the timing information associated with message exchanges between nodes. This includes the timestamps for when a message is received, when a node begins processing, and when it sends out the output. Using this data, we construct a data flow graph that captures node interactions, including the precise timing of messages (see Fig. 4.4a, and Line 21 in Fig. 4.3c).

Dynamics The data flow graph serves as a foundation for identifying the system's dynamics. One advantage of using a data flow graph is that it inherently represents asynchronous interactions and correctly encodes time-scale differences between nodes. Accounting for these asynchronous effects is essential, as they can significantly impact the identified system dynamics [92]. Given the data flow graph, our framework builds a tailored computation graph as follows. We augment the data flow graph with a simulator that models the system dynamics by adding simulator nodes at the desired simulation rate. Edges between simulation nodes and real-world-interacting nodes are introduced to pass the simulation state to the nodes that model real-world interactions (actuators, sensors, etc.), according to the assumed delay distributions, as shown in Fig. 4.4b. These delay distributions are either trainable or prespecified, as explained later in this section. By replaying actions through the computation graph and comparing the reconstructed outputs with the collected data, we optimize the simulator parameters to minimize a reconstruction loss. During this process, all parameters within the computation graph (e.g.

simulator parameters or those in any other nodes), can be optimized. For instance, in the example shown in Fig. 4.4c, we simultaneously identify Brax's system parameters and the camera's parameters for angle-to-pixel conversion, but we could have also optimized for any other parameter in the graph, such as the motor's friction. The COMPILED runtime is particularly advantageous for this optimization process due to its ability to parallelize computations efficiently. We found evolutionary strategies effective for this task, as they leverage parallelism, constraint specification, and are less susceptible to local minima [94, 95, 103].

Measurable Delays In addition to dynamics, our framework addresses delay estimation, distinguishing between directly measurable delays and hidden delays, such as those in actuators and sensors. Using the recorded timing data, we estimate the communication and computation delays of the system by fitting a Gaussian Mixture Model (GMM) to the measurable delay data using gradient descent. Details on the GMM fitting can be found in Appendix 4.A. Typically, around a thousand samples are sufficient for fitting, which, depending on the system's rate, may require less than a minute of data collection. When sampling from the GMM, we clip the sampled values to be non-negative, as delays are inherently non-negative.

Hidden Delays With hidden delays we mean delays that are not directly observable in the data flow graph, such as delays between the real world and sensors or actuators. While we support the addition of edges between simulator and real-world-interacting nodes based on prespecified delay distributions (e.g., motor-Brax connections in Fig. 4.4b), users can also introduce trainable delays to identify hidden delays (e.g., Brax-camera connections in Fig. 4.4b). Our approach requires specifying a minimum and maximum bound for each trainable delay, which we use to introduce additional edges that accommodate all possible communication patterns between two nodes under minimum and maximum delay conditions. We then introduce a trainable parameter $\alpha \in [0, 1]$ for each connection, allowing interpolation between the minimum and maximum scenarios. Different deterministic interpolation schemes, such as linear or zero-order hold, are currently supported to model various delay characteristics.

4.3.4 Delay Compensation

Once the system dynamics and delays are identified, the framework supports various strategies for delay compensation to enhance sim2real performance.

Delay Simulation One straightforward strategy is to integrate the identified delay distributions into the simulation environment. This approach, referred to as delay simulation (Fig. 4.1c), allows the agent to learn policies that are delay aware. Notice that delays make the problem non-Markovian. To address this, a history of observations and actions can be stacked and used as input to the policy to restore the Markov property. This does make the learning problem more challenging, as the agent must learn to solve the task and handle delays simultaneously, as we will show in our experiments.

Estimator While RL approaches often treat the environment as a black box, in sim2real scenarios, we can utilize the identified system dynamics and delays to design a model-based delay compensator that predicts the system's behavior during real-world execution. Inspired by a Smith Predictor [89] and shown in Fig. 4.1d, our strategy is to predict the state we expect when the corresponding command based on this state reaches the system. By knowing all delays, we can predict when a command will arrive and estimate the system state at that future time. Specifically, when a sensor captures an observation, we timestamp it and subtract the identified hidden delay τ_s to estimate the timestamp of the world's state the observation corresponds to, $t_s - \tau_s$. When the estimator processes the observation at t_e, it can determine when the resulting command will reach the system by adding the expected estimator-to-actuator latency, τ_a, resulting in $t_e + \tau_a$. Thus, the estimator first updates the state up to $t_s - \tau_s$ and then predicts it forward to $t_e + \tau_a$ using the past control inputs and their estimated timestamps. We recommend using an Unscented Kalman Filter (UKF) [106] for this task because it effectively handles non-differentiable and non-linear dynamics, while requiring only a small number of particles that can be efficiently evaluated in parallel (see Appendix 4.B for more information). Additionally, in partially observable settings, a UKF can infer the hidden state of the system from observations and provide this state to the agent, enabling training in a fully observable, delay-free environment, which generally facilitates easier learning. In our experiments, we will evaluate the benefits of using such an estimator for delay compensation and compare the performance gains of delay compensation alone versus delay compensation with hidden state estimation.

4.3.5 Limitations

Our framework does not support running nodes on different machines; computations are restricted to different devices via JAX. This limits the ability to compile nodes for low-level controllers onboard a robot. Additionally, JAX's Just-In-Time (JIT) compilation can lead to long compilation times, although recent updates with function caching have mitigated this to some extent.

The framework estimates hidden delays as deterministic, which is a reasonable assumption for many robotics applications. Nevertheless, stochastic delays can be modeled by adding variability to the deterministically identified delays, for example, to simulate jitter in sensor readings. Also, our approach requires setting minimum and maximum bounds for trainable delays, but we have found that using large bounds often yields good results. Furthermore, our delay simulation is state independent, meaning that while it accounts for the correlation and stochastic nature of delays, it does not adapt to the specific conditions or data of each simulation step. For instance, if an algorithm takes longer to process when there are multiple simulated objects in view, our approach would not capture this increase in processing time that would occur in a real-world scenario.

As systems scale to large configurations, efficiently parallelizing full asynchronicity for every node can become a challenge. To this end, we leverage the supergraph approach in [63] to efficiently parallelize graph-based simulations. Furthermore, the graph-based framework provides flexibility by allowing users to adjust the level of detail as needed for the task. For example, users may model entire robots as single nodes, focusing on interactions between them rather than internal asynchronicity, to maintain scalability in large-scale systems. The complexity of calculations within nodes is efficiently managed

using JAX [76], which enables scalable computations through its support for parallelization and distributed computing across multiple devices.

4.4 Experimental Evaluation

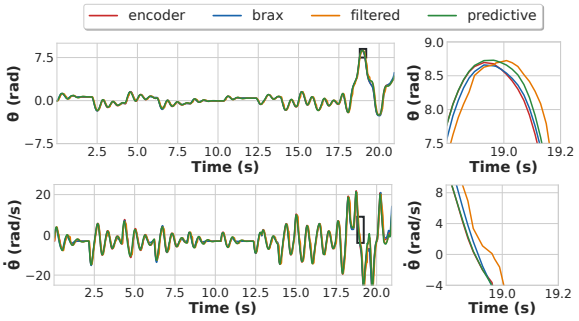
The main focus of this work is a sim2real framework that addresses asynchronous interactions in real-world systems by modeling delays and using real-world data for accurate system identification and reinforcement learning training. Our experiments are designed to validate the key claims made in Sec. 4.1 as follows. First, we identify the system dynamics and delays from real-world data, followed by a sim2real transfer evaluation using the identified system while using delay compensation techniques. We validate our approach on two distinct real-world systems: a pendulum swing-up and a quadrotor control task.

4.4.1 System Identification and Delay Estimation

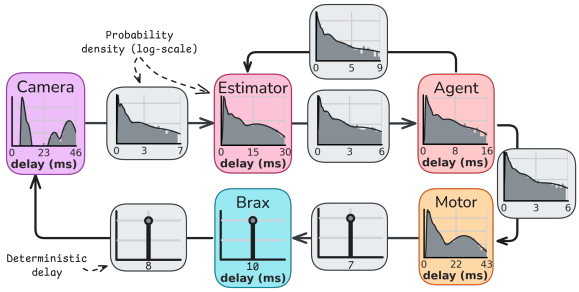
To support the claim that our approach enables the identification of both dynamics and delays from real-world data, we present system identification and delay estimation results for the two selected systems.

Pendulum In contrast to the classic swing-up task [23], which uses full state information, our setup relies solely on camera images of the pendulum. This task highlights the challenge of delay estimation and system identification from images. We apply an open-loop voltage sequence to the motor for 21 seconds while recording a stream of images from a RealSense d435i camera, in addition to the applied actions and corresponding timing information. Using this data, we construct a data flow graph that is augmented to form a computation graph, incorporating simulator nodes operating at 100 Hz. We introduce edges between the simulator nodes and the camera and motor via two trainable delays that assume a minimum and maximum delay of 0 to 50 ms, respectively. Images are first preprocessed through background subtraction and color thresholding to detect the center pixel coordinates of the red dot that marks the pendulum's mass. The actions are then reapplied to the simulator, and we optimize the parameters to minimize the reconstruction error between predicted and actual pixel coordinates. Simultaneous optimization is performed on several parameters: the physics parameters of the Brax simulator (mass, length, friction, inertia, etc.), parameters for hidden camera and motor delays, and the parameters of an ellipse model (center, axes, rotation) that maps pixel coordinates to angles using the intuition that the pendulum's motion (as pixel coordinates) will be an ellipse when projected onto the camera plane. A UKF is employed for full state estimation and delay compensation, utilizing a lightweight dynamics model (see Appendix 4.C). We use the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [103] to optimize the 27 parameters by minimizing the reconstruction error between the predicted and measured pixel coordinates. See Appendix 4.D for details on CMA-ES and the hyperparameter settings. Finally, we fit GMMs to estimate delay distributions for all measurable communication and computation delays.

The reconstructed angle and angular velocity from the simulator and estimator are shown in Fig. 4.5a, alongside the validation data obtained from the pendulum's encoder, which can be considered the ground truth with minimal delay. The open-loop reconstruction remains accurate over a 21-second time horizon. The identified delay distributions



(a) Reconstruction (b) Zoomed



(c) Estimated delays

Figure 4.5: Pendulum system identification and delay estimation. (a) Open-loop reconstruction of the angle (θ) and angular velocity ($\dot{\theta}$) with the *brax* simulator, compared to ground-truth *encoder* data. (b) A zoomed view shows that the *predictive* UKF estimate that compensates for delays, outperforms the *filtered* estimate that does not. (c) Estimated GMM delay distributions and deterministic hidden delays for the camera and motor, with the grey area indicating the measured delay distribution and the black line showing the GMM fit.

are illustrated in Fig. 4.5c, with a motor-to-Brax delay of approximately 7 ms and a Brax-to-camera delay of around 8 ms. The camera delay exhibits a multi-modal distribution, suggesting variability due to internal processing and shutter speed. The effectiveness of delay compensation is demonstrated in Fig. 4.5b by comparing the filtered and predictive estimates. The filtered estimate shows the UKF's state estimate plotted against the timestamp of when the action using the estimated state was applied to the simulator, resulting in a noticeable phase shift of around 50 ms. In contrast, the predictive estimate forecasts the filtered estimate forward, resulting in a lower mean squared error (MSE) for both the angle and angular velocity, as shown in Tab. 4.2. Finally, we use the identified system to render images from the estimated poses, as shown in Fig. 4.6. The comparison between real and rendered images from two different viewpoints qualitatively demonstrates the accuracy of the estimated system parameters.

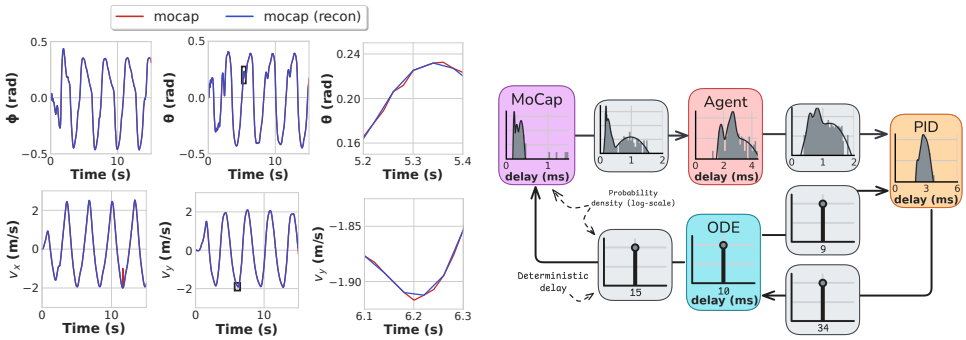
Quadrotor Next, we identify the dynamics and delays of a quadrotor system using real-world data to demonstrate the applicability of our approach to higher-dimensional

(a) Real images (b) Rendered images with Brax

Figure 4.6: Comparison of real and rendered images of the pendulum from two different viewpoints. (a) shows actual images captured from side and frontal views. (b) shows the corresponding rendered images from the estimated poses.

	MSE_θ	$\text{MSE}_{\dot\theta}$
filtered	0.069	4.256
predictive	**0.012**	**1.114**

Table 4.2: Mean squared error (MSE) with respect to ground-truth encoder data. Boldface indicates the best performance.



(a) Reconstruction (b) Zoomed (c) Estimated delays

Figure 4.7: System identification and delay estimation for quadrotor control. (a) Open-loop reconstruction of roll, pitch, and velocities in body frame over 15 seconds (*recon*), showing the accuracy of the identified model compared to the MoCap data (*mocap*). (b) Zoomed view illustrates an accurate fit. (c) Estimated GMM delay distributions, with the grey area indicating the measured delay distribution and the black line showing the GMM fit.

state-action spaces. The quadrotor's yaw is fixed, while the reference roll and pitch angles and the height setpoint are sent to a PID controller to maintain a circular flight path at a constant altitude. The PID controller converts the height setpoint to a thrust command, which, along with the roll and pitch commands, is sent to the Crazyflie. We record the actions, timing data, and state information captured by a motion capture (MoCap) system. Similar to the pendulum experiment, we construct a data flow graph that is augmented to form a computation graph with simulator nodes operating at 100 Hz. Edges are introduced between the simulator nodes and the MoCap and PID nodes, incorporating hidden delay nodes with a minimum and maximum delay of 0 to 50 ms, respectively. A dynamics model similar to that used in [30] is employed (see Appendix 4.C). Simultaneous optimization is performed on the dynamics parameters (e.g., mass, drag, gain, time constants) and interpolation parameters for hidden delays between the dynamics model, MoCap, and PID controller, using CMA-ES [103] to optimize the eight parameters by minimizing the reconstruction error between the predicted and measured quadrotor attitude and body frame velocities. See Appendix 4.D for details on CMA-ES and the hyperparameter settings. We also fit GMMs to estimate delay distributions for all measurable communication and

computation delays.

The results in Fig. 4.7a show accurate reconstruction of the quadrotor's states over 15 seconds. The identified delays are shown in Fig. 4.7c, with PID-to-ODE delay at 34 ms, ODE-to-PID delay at 9 ms, and ODE-to-MoCap delay at 15 ms. In the next section, we evaluate the advantage of delay-aware system identification for sim2real transfer by training policies with and without considering delays.

4.4.2 Sim2Real Transfer

To support the claim that our approach implements delay compensation techniques essential for effective sim2real transfer, we evaluate the sim2real performance of policies trained with and without delay compensation for the pendulum and quadrotor systems.

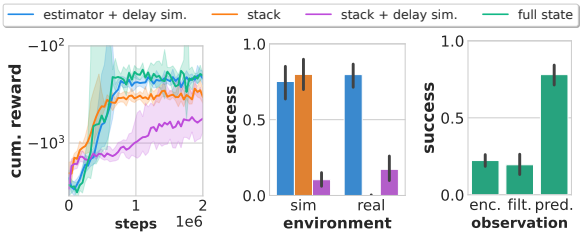
Pendulum Swing-Up This task highlights the challenge of delay compensation and partial observability in reinforcement learning. By demonstrating that neglecting delay simulation can impair policy transfer even in a seemingly simple scenario, we underscore the necessity of delay-aware approaches for more complex systems, where delays are inevitable and system dynamics are more intricate [27, 72–74]. The pendulum task's simplicity effectively clarifies the importance of addressing delays in sim2real frameworks.

To investigate the impact of delays and partial observability on task complexity, we train pendulum swing-up policies using PPO [49] under different conditions in simulation (see Appendix 4.E for more details). We evaluate policies trained with full state information, stacked observations with and without delay simulation, and estimated full state information with simulated delays. As shown in Fig. 4.8a, policies with full state information achieve higher rewards and converge faster than those relying on stacked observations, especially when delays are present. This highlights the additional challenge introduced by delays and partial observability, beyond the complexity of the task itself.

Zero-shot evaluations on the real system show that policies trained solely with stacked observations fail to consistently swing up the pendulum, while the policy trained with delay simulation, delay compensation, and full state estimation achieves reliable swing-up, as demonstrated in Fig. 4.8b. Interestingly, even a policy trained on the full state without simulated delays can achieve consistent swing-up when real-world evaluation uses an estimator that compensates for delays and estimates the full state, as indicated by *pred.* in Fig. 4.8c.

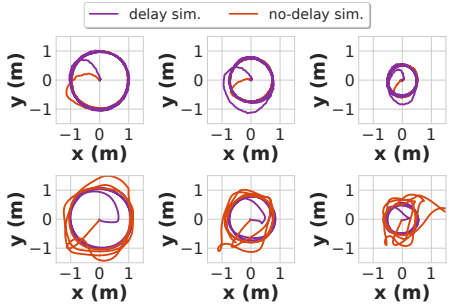
We assess the performance gains of delay compensation alone versus delay compensation with hidden state estimation, by evaluating the full state policy in two other scenarios: using the angle encoder, which provides full state information with negligible delay compared to camera images, and using the filtered state estimate from the UKF instead of the forward-predicted state. Both policies perform suboptimally, suggesting that both full state estimation and forward prediction are essential for reliable performance, as shown in Fig. 4.8c. A side-by-side comparison of the real-world swing-up performance of the different policies is available in the supplementary video.

Path Following with a Quadrotor We trained a quadrotor to fly a circular path at maximum speed with varying radii to assess the impact of delay simulation on sim2real performance. We used PPO [49] to train policies using a reward function that penalizes



(a) Training curve (b) Sim2real (c) Estimator

Figure 4.8: Sim2real evaluation of policies trained under different delay and observation conditions for the pendulum swing-up task. (a) Training curves comparing policies with full state information and stacked observations. (b) Sim2real performance showing the percentage of time the pendulum remains upright (within ±10° and ±0.5 rad/s). (c) Performance of a policy using full state estimation with delay compensation, demonstrating the importance of delay compensation for steady performance.



(a) R = 1.0m (b) R = 0.75m (c) R = 0.5m

Figure 4.9: Sim2real performance of quadrotor policies trained with and without delay simulation across different path radii. Top row: simulated path following at radii R = 1.0m, R = 0.75m, and R = 0.5m. Bottom row: real-world path following shows that delay simulation improves performance and stability, particularly at smaller radii where delays significantly impact control.

path error and rewards high speeds along the path (see Appendix 4.E for more details). In simulation, all policies achieved successful path following, with the no-delay policy reaching higher speeds and maintaining lower path errors due to its ability to fly more aggressively in the absence of simulated delays, as detailed in Tab. 4.3. However, in real-world tests, only the policy trained with delay simulation maintained stable flight; the no-delay policy exhibited oscillations around the target path. The performance gap widened at smaller radii, with the no-delay policy exhibiting highly unstable flight behavior at a radius of 0.5 m, demonstrating the critical role of delay-aware training for reliable real-world deployment, as shown in Fig. 4.9. A side-by-side comparison of the real-world flight performance of the two policies is available in the supplementary video.

Radii (m)	Simulation				Real-world			
	v_{path} (m/s)		e_{path} (m)		v_{path} (m/s)		e_{path} (m)	
	delay	no-delay	delay	no-delay	delay	no-delay	delay	no-delay
1.00	1.95	**2.23**	0.03	**0.02**	2.02	**2.05**	**0.06**	0.21
0.75	1.67	**1.92**	0.03	**0.02**	**1.64**	1.63	**0.04**	0.19
0.50	1.39	**1.61**	**0.04**	**0.04**	**1.36**	1.18	**0.04**	0.24

Table 4.3: Impact of delays on simulated vs. real-world performance across different path radii. v_{path} denotes the average speed flown along the path, and e_{path} represents the average error between the quadrotor's position and the target path. Boldface indicates the best performance in each category.

4.4.3 Computational Runtime Analysis

To support our claim that the framework enables efficient parallelized simulation on accelerator hardware, we evaluated simulation speeds using the COMPILED runtime on an NVIDIA RTX 3070 Laptop GPU. The data flow was augmented with simulator nodes and subsequently parallelized to simulate delays according to real-world settings.

We measured the computation time for CMA-ES [103] to converge during system identification for the pendulum and quadrotor tasks. For the pendulum, optimizing 27 parameters with a population size of 200 and a 21-second rollout per fitness evaluation (1,050 steps) led to convergence after 38 generations in 22.07 seconds, achieving 380k steps/s with a compilation time of 19.97 seconds. For the quadrotor, optimizing eight parameters under similar conditions but with a 15-second rollout (375 steps) resulted in convergence after 31 generations in 5.81 seconds, reaching 400k steps/s with a compilation time of 10.16 seconds. We also evaluated PPO training time using the implementation from [96]: for the pendulum, training five policies in parallel with 64 environments reached 5 million steps in 77.1 seconds (325k steps/s), while for the quadrotor, training with 128 environments for 10 million steps completed in 29.8 seconds (336k steps/s), demonstrating the framework's efficiency in supporting rapid training on real-world tasks.

To isolate simulation speed from training overhead, we performed a parallelized rollout speed analysis (Fig. 4.10). The results show a linear relationship on a logarithmic scale, indicating that as the number of parallel environments doubles, the simulation speed also roughly doubles. An initial superlinear increase is observed, likely due to constant overheads being amortized over a larger number of parallel environments, resulting in more efficient resource utilization.

The simulation speed in our framework is determined by the computational workload of each node and the ability to parallelize their interactions. Our framework extends beyond standard simulations by modeling the asynchronous interaction between components, which are inherently challenging to parallelize efficiently [63]. By demonstrating fast simulation speeds for the pendulum and quadrotor, we show that our framework achieves efficient runtime performance without introducing significant overhead beyond the computations within each node. If the simulation speed were slow, even with the simple dynamics of these systems, it would indicate a substantial fixed overhead from the framework.

To compare runtime performance with other sim2real frameworks, we evaluate the runtime of a common system across these frameworks. Specifically, we compare the

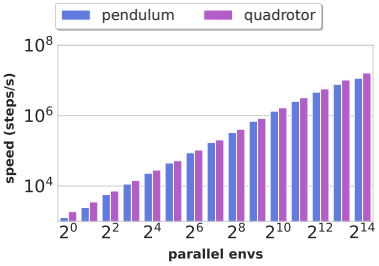


Figure 4.10: Simulation steps per second vs. number of parallel environments for a 200-step rollout on a GPU. The pendulum (20 ms/step) and quadrotor (25 ms/step) systems both demonstrate a linear scaling in simulation speed with increasing parallel environments.

Node	Delay (ms)	Rate (Hz)	Device	Computation
		Pendulum		
camera	9.8 ± 8.0	60	CPU_1	image-to-angle conv.
estimator	3.5 ± 5.2	50	CPU_2	UKF update
agent	1.7 ± 2.3	50	GPU	Policy NN(64,64)
motor	5.6 ± 7.0	50	CPU_3	Cmd to pendulum
		Quadrotor		
mocap	0.3 ± 0.1	50	CPU_1	Read quadrotor pose
agent	2.6 ± 0.5	25	GPU	Policy NN(64,64)
pid	2.8 ± 0.3	50	CPU_2	Cmds to quadcopter

Table 4.4: Delay statistics for Quadrotor and Pendulum nodes, including delay (mean ± std.) in ms, rate in Hz, device type, and a description of each computation. Subscripts indicate dedicated CPU cores. NN denotes a neural network with layer sizes in parentheses.

4

pendulum system described in Appendix 4.C with the pendulum examples in Drake [40] and EAGERx [29], as all systems involve a simple pendulum with a two-dimensional state, ensuring a fair comparison. Since GPU parallelization is not supported in these frameworks, we measure the runtime performance of a single 20-second rollout on a CPU while applying random actions. Our results indicate that EAGERx achieves 0.28k steps/s, Drake achieves 60k steps/s, and our framework achieves 15k steps/s on a single CPU core. We attribute EAGERx's slower performance to the overhead of ROS communication between nodes, while Drake benefits from its optimized C++ backend. Our framework is significantly faster than EAGERx and, compared to Drake, can support GPU-based parallelized execution, which can further improve simulation speed, as demonstrated in Fig. 4.10.

To support the claim that our framework meets the latency and performance requirements for real-time online processing in real-world systems, we evaluate the latency of different components during real-world experiments. Our framework records timing information for each node, allowing us to estimate computation and communication delay statistics across both the pendulum and quadrotor systems, as visualized in Fig. 4.5c and Fig. 4.7c. The mean and standard deviation of delays for each node's periodic computation are calculated, providing insights into system performance. By dedicating specific CPU cores to each node, we bypass the Python Global Interpreter Lock (GIL), enabling concurrent execution. Additionally, we use the GPU to accelerate policy inference in the agent node. This approach results in low latency across the system. Unexpectedly, the motor node in the pendulum system exhibited large delays, likely due to the hardware's slow response time while servicing ROS [24] service calls. As expected, the camera node had the longest delays, attributed to the time required for image retrieval and processing to convert images to angles.

In summary, our evaluation demonstrates that our approach effectively identifies both dynamics and delays from real-world data, compensates for delays to improve sim2real transfer, and facilitates efficient parallelized simulation on accelerator hardware. At the same time, our approach meets the latency and performance requirements for real-time

online processing, supporting all four key claims.

4.5 Conclusion

In this chapter, we presented a novel framework, REX (Robotic Environments with jaX), for sim2real transfer that introduces a graph-based simulation model incorporating latency effects, optimized for parallelization on accelerator hardware. Our approach models asynchronous, hierarchical systems by explicitly representing computation, communication, actuation, and sensing delays. This enables the simultaneous estimation of system dynamics and delays using real-world data, effectively minimizing the sim2real gap. We implemented and evaluated our approach on two real-world robotic systems, demonstrating its ability to support rapid training while maintaining high fidelity to real-world conditions. The experiments suggest that our framework not only improves the accuracy of policy transfer by reducing the impact of delays and partial observability but also enhances simulation efficiency by leveraging hardware acceleration.

For future work, we aim to extend the framework to support estimating stochastic hidden delays, which could further reduce the sim2real gap by more accurately capturing real-world uncertainties. Additionally, we plan to enhance the framework's scalability and real-world applicability by enabling distributed computing across multiple machines, beyond the current capability of utilizing different devices via JAX.

Appendix 4.A: Measurable Delay Fitting with Gaussian Mixture Models

Let $\{x_i\}_{i=1}^N$ represent the observed delays as one-dimensional data points. We consider a Gaussian Mixture Model (GMM) with K components, where each component k is characterized by three parameters: (π_k, μ_k, σ_k). Specifically,

- π_k: the mixing weight of component k, satisfying $\sum_{k=1}^K \pi_k = 1$ and $\pi_k \geq 0$,
- μ_k: the mean of the Gaussian component,
- σ_k: the standard deviation of the Gaussian component.

The full set of parameters of the GMM is denoted as $\boldsymbol{\theta} = \{\pi_k, \mu_k, \sigma_k\}_{k=1}^K$, which includes all mixing weights, means, and standard deviations. The probability density function for a single data point x_i under this model is expressed as:

$$p(x_i \mid \boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k^2),$$

where $\mathcal{N}(x_i \mid \mu_k, \sigma_k^2)$ is the probability density function of a normal distribution with mean μ_k and variance σ_k^2. To estimate the parameters of the GMM, we minimize the negative log-likelihood of the observed delays:

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log\left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k^2)\right).$$

The optimization is performed using a gradient-based approach. Specifically, the parameters $\boldsymbol{\theta}$ are updated iteratively using a standard solver such as the Adam optimizer [**?**], which adjusts the parameters to minimize $\mathcal{L}(\boldsymbol{\theta})$. To improve numerical stability during optimization, we normalize the data before fitting the GMM. The observed delays $\{x_i\}$ are transformed into normalized delays $\{x_i'\}$ as:

$$x_i' = \frac{x_i - \mu_x}{\max(\sigma_x, \epsilon)},$$

where μ_x and σ_x are the mean and standard deviation of the observed delays, and ϵ is a small constant (e.g., 10^{-7}) to avoid division by zero. The GMM is then fit to the normalized dataset $\{x_i'\}_{i=1}^N$, and the negative log-likelihood is computed accordingly. After fitting the model, the parameters are denormalized to map back to the original data range with:

$$\tilde{\mu}_k = \mu_k \cdot \sigma_x + \mu_x,$$
$$\tilde{\sigma}_k = \sigma_k \cdot \sigma_x.$$

This approach ensures numerical stability during optimization while providing parameters $\tilde{\mu}_k$ and $\tilde{\sigma}_k$ in the original scale of the observed delays.

Appendix 4.B: Unscented Kalman Filter

The Unscented Kalman Filter (UKF) is a state estimation algorithm designed for nonlinear systems. It employs a deterministic sampling technique to approximate the mean and covariance of the state distribution. This approach is more accurate than linearization-based methods, such as the Extended Kalman Filter (EKF), for highly nonlinear systems [3]. Since the UKF relies on sampling rather than linearization, it can effectively handle non-differentiable functions. The number of sigma points, $2n+1$, is determined by the state dimensionality n and is typically much smaller than the number of particles required for a particle filter. Additionally, the evaluation of sigma points can be efficiently parallelized. The sigma points are calculated to capture the mean and covariance of the state distribution and we follow the procedure in [107] to generate $2n+1$ sigma points. Given the current state mean $\mathbf{x}$ and covariance $\mathbf{P}$, the scaling parameter $\lambda = \alpha^2(n+\kappa) - n$ is computed, where α controls the spread of the sigma points, κ adjusts scaling, and β incorporates prior knowledge (e.g., $\beta = 2$ for Gaussian distributions). The sigma points are then determined as:

$$\chi_0 = \mathbf{x}, \quad \chi_i = \mathbf{x} \pm \sqrt{n+\lambda} \cdot [\mathbf{P}]_i, \quad i = 1, \ldots, n$$

where $\sqrt{n+\lambda} \cdot [\mathbf{P}]_i$ is the i-th column of the Cholesky decomposition of $(n+\lambda)\mathbf{P}$. The corresponding weights for mean and covariance are:

$$w_0^{(m)} = \frac{\lambda}{n+\lambda}, \quad w_0^{(c)} = w_0^{(m)} + (1 - \alpha^2 + \beta), \quad w_i^{(m)} = w_i^{(c)} = \frac{1}{2(n+\lambda)}, \quad i = 1, \ldots, 2n$$

The filter is initialized with an initial state mean $\mathbf{x}_0$ and covariance $\mathbf{P}_0$. The state transition function $f(\cdot)$ and observation function $h(\cdot)$ are nonlinear, describing the process dynamics and measurements, respectively. Process and measurement noise are assumed to be additive, Gaussian, and uncorrelated, with zero mean and covariance matrices $\mathbf{Q}$ and $\mathbf{R}$, respectively.

Actions commanded at time $t-1$ will affect the state at time $t+\tau_a$, where τ_a is the actuation delay. Similarly, the most recent measurement available at time t is from time $t-\tau_s$, where τ_s is the sensor delay. Hence, the indexing of the measurements and actions is shifted by the sensor and actuation delays. For simplicity, we assume that the sensor delay τ_s and actuation delay τ_a are constant and a multiple of the timestep. Furthermore, we consider time-invariant noise distributions, process models, and measurement models. Note, however, that the UKF can be extended to handle time-varying noise covariances, models, variable delays, and non-uniform timesteps, but we do not consider them here for notational clarity.

The UKF estimation step at time t is outlined in Alg. 6. In the prediction and update steps, the UKF updates its prior state estimate from $t-1-\tau_s$ to the last measurement time, $t-\tau_s$, using the most recent measurement available at $t-\tau_s$. Next, the UKF forward-predicts the state to $t+\tau_a$, incorporating the commanded actions available up to $t-1+\tau_a$ to account for the combined sensor and actuation delays.

Algorithm 6: UKF Estimation Step with Delay Compensation

Input: Process noise $\mathbf{Q}$, measurement noise $\mathbf{R}$, weights $w_i^{(m)}, w_i^{(c)}$, process model $f(\cdot)$, measurement model $h(\cdot)$, sensor delay τ_s, actuation delay τ_a

Input: Previous state mean $\mathbf{x}_{t-1-\tau_s}$, covariance $\mathbf{P}_{t-1-\tau_s}$, new measurement $\mathbf{z}_{t-\tau_s}$, action sequence $\{\mathbf{u}_k\}_{k=t-1-\tau_s}^{t-1+\tau_a}$

Output: Estimated state mean $\mathbf{x}_{t-\tau_s}$, covariance $\mathbf{P}_{t-\tau_s}$, forward-predicted state mean $\mathbf{x}_{t+\tau_a}$, covariance $\mathbf{P}_{t+\tau_a}$

Calculate Sigma Points:

Compute sigma points $\{\chi_i^{(t-1-\tau_s)}\}$ from current state mean $\mathbf{x}_{t-1-\tau_s}$ and covariance $\mathbf{P}_{t-1-\tau_s}$.

Prediction Step:

Propagate sigma points through process model with actions: $\chi_i^- = f(\chi_i^{(t-1-\tau_s)}, \mathbf{u}_{t-1-\tau_s})$.

Compute predicted mean: $\mathbf{x}_{t-\tau_s}^- = \sum_i w_i^{(m)} \chi_i^-$.

Compute predicted covariance: $\mathbf{P}_{t-\tau_s}^- = \sum_i w_i^{(c)} (\chi_i^- - \mathbf{x}_{t-\tau_s}^-)(\chi_i^- - \mathbf{x}_{t-\tau_s}^-)^T + \mathbf{Q}$.

Update Step:

Propagate sigma points through measurement model: $\mathbf{z}_i = h(\chi_i^-)$.

Compute predicted measurement mean: $\mathbf{z}_{t-\tau_s}^- = \sum_i w_i^{(m)} \mathbf{z}_i$.

Compute innovation covariance: $\mathbf{S} = \sum_i w_i^{(c)} (\mathbf{z}_i - \mathbf{z}_{t-\tau_s}^-)(\mathbf{z}_i - \mathbf{z}_{t-\tau_s}^-)^T + \mathbf{R}$.

Compute cross-covariance: $\mathbf{C} = \sum_i w_i^{(c)} (\chi_i^- - \mathbf{x}_{t-\tau_s}^-)(\mathbf{z}_i - \mathbf{z}_{t-\tau_s}^-)^T$.

Compute Kalman gain: $\mathbf{K} = \mathbf{C}\mathbf{S}^{-1}$.

Update state mean: $\mathbf{x}_{t-\tau_s} = \mathbf{x}_{t-\tau_s}^- + \mathbf{K}(\mathbf{z}_{t-\tau_s} - \mathbf{z}_{t-\tau_s}^-)$.

Update covariance: $\mathbf{P}_{t-\tau_s} = \mathbf{P}_{t-\tau_s}^- - \mathbf{K}\mathbf{S}\mathbf{K}^T$.

Forward-prediction Step:

for $k = t-\tau_s, \ldots, t-1+\tau_a$ **do**

  Compute sigma points $\{\chi_i^{(k)}\}$ from state mean $\mathbf{x}_k$ and covariance $\mathbf{P}_k$.

  Propagate sigma points through process model with actions: $\chi_i^- = f(\chi_i^{(k)}, \mathbf{u}_k)$.

  Compute forward-predicted mean: $\mathbf{x}_{k+1} = \sum_i w_i^{(m)} \chi_i^-$.

  Compute forward-predicted covariance: $\mathbf{P}_{k+1} = \sum_i w_i^{(c)} (\chi_i^- - \mathbf{x}_{k+1})(\chi_i^- - \mathbf{x}_{k+1})^T + \mathbf{Q}$.

end

return $\mathbf{x}_{t-\tau_s}, \mathbf{P}_{t-\tau_s}, \mathbf{x}_{t+\tau_a}, \mathbf{P}_{t+\tau_a}$

Appendix 4.C: Dynamics

Appendix 4.C.1 Pendulum Dynamics

The pendulum system is modeled by a second-order ordinary differential equation (ODE). The state $\mathbf{x} = (\theta, \dot{\theta})$ represents the angle θ and angular velocity $\dot{\theta}$. The control input u represents the applied voltage. The angular acceleration $\ddot{\theta}$ is given by:

$$\ddot{\theta} = \frac{u\frac{K}{R} + mgl\sin(\theta) - b\dot{\theta} - \dot{\theta}\frac{K^2}{R} - c\,\mathrm{sign}(\dot{\theta})}{J},$$

where J is the moment of inertia, m the mass, l the pendulum length, b the damping coefficient, K the motor constant, R the motor resistance, c the static friction, and $g = 9.81\,\mathrm{m/s^2}$ the gravitational acceleration. Dynamics are simulated using a fourth-order Runge-Kutta integration method with a fixed time step of 0.01 seconds. All parameters (J, m, l, b, K, R, c) are identified experimentally in Chapter 4 except for the gravitational acceleration g, together with any additional parameters required for hidden delay estimation.

Appendix 4.C.2 Quadrotor Dynamics

The dynamics model, similar to that in [30], is used to simulate the quadrotor's motion. The dynamics are divided into three components: rotational, translational, and motor dynamics. The state is represented by the position $p = (x, y, z)$, velocity $v = (\dot{x}, \dot{y}, \dot{z})$, attitude $\eta = (\phi, \theta, \psi)$, and thrust state Ω. The control inputs are the reference pulse-width modulation (PWM) motor signal Θ and two reference angles ϕ_{ref} and θ_{ref}. Yaw dynamics are neglected, with yaw angle assumed constant at $\psi = 0$. The rotational dynamics are approximated by a first-order system for the attitude angles ϕ and θ:

$$\dot{\phi} = \frac{k_c(\phi_{\mathrm{ref}} - \phi)}{\tau_c} \qquad \dot{\theta} = \frac{k_c(\theta_{\mathrm{ref}} - \theta)}{\tau_c}$$

where the same k_c and τ_c are used for the rotation in both angle directions due to the system's symmetry. These dynamics comprise the quadrotor's closed-loop onboard control of the attitude angles. The total thrust generated by the quadrotor's motors is modeled as a first-order system:

$$\dot{\Omega} = a_{\mathrm{m}}\Omega + b_{\mathrm{m}}\Theta \qquad f_{\mathrm{thrust}} = c_{\mathrm{m}}\Omega + d_{\mathrm{m}}\Theta$$

where a_{m}, b_{m}, c_{m}, and d_{m} are motor-specific constants. The drag force acting on the quadrotor is given by:

$$f_{\mathrm{drag}} = -\begin{bmatrix} \kappa_{xy}\omega & 0 & 0 \\ 0 & \kappa_{xy}\omega & 0 \\ 0 & 0 & \kappa_z\omega \end{bmatrix} v_b,$$

where κ_{xy} and κ_z are drag coefficients, ω is the rotor speed, and v_b is the body-frame velocity. The body-frame velocity is calculated as $v_b = R^\top v^\top$, where R is the rotation matrix from the body frame to the world frame. The rotor speed is approximated using the following relationships (see [108] for more details) between the effective PWM signal Θ_{eff}, rotor speed ω, and total thrust f_{thrust}:

$$f_{\mathrm{thrust}} = 4\left(a_{\mathrm{p}}\Theta_{\mathrm{eff}}^2 + b_{\mathrm{p}}\Theta_{\mathrm{eff}} + c_{\mathrm{p}}\right) \qquad \omega = 4\left(a_{\mathrm{r}}\Theta_{\mathrm{eff}} + b_{\mathrm{r}}\right)$$

where a_p, b_p, and c_p are PWM constants, a_r and b_r are rotor constants, and the factor 4 accounts for the quadrotor's four rotors. The translational dynamics are:

$$\dot{v} = \frac{1}{m} R\left([0,0,f_\mathrm{thrust}]^\top + f_\mathrm{drag}\right) - [0,0,g]^\top,$$

where m is the quadrotor's mass, and g is the gravitational constant. Dynamics are simulated using a fourth-order Runge-Kutta integration method with a fixed time step of 0.01 seconds.

The motor, PWM, and rotor constants are specific to the motor and require additional sensors for accurate identification. We use the experimentally identified values from [108] for the Crazyflie 2.0 quadrotor, as done in [30]:

$$\begin{aligned} a_\mathrm{m} &= -15.47, & b_\mathrm{m} &= 1.0, & c_\mathrm{m} &= 1.43\times10^{-4}, & d_\mathrm{m} &= 2.89\times10^{-7}, \\ a_\mathrm{p} &= 2.13\times10^{-11}, & b_\mathrm{p} &= 1.03\times10^{-6}, & c_\mathrm{p} &= 5.49\times10^{-4}, \\ a_\mathrm{r} &= 0.041, & b_\mathrm{r} &= 380.83. \end{aligned}$$

The remaining parameters (m, k_c, τ_c, κ_{xy}, κ_z) are experimentally identified in Chapter 4, along with any additional parameters required for hidden delay estimation.

Appendix 4.D: Covariance Matrix Adaptation Evolution Strategy

The Covariance matrix adaptation evolution strategy (CMA-ES) algorithm is a stochastic, derivative-free optimization method well-suited for non-linear or non-convex problems [103]. It evolves a population of solutions by sampling from a multivariate normal distribution, adapting the covariance matrix and step size to guide search directions efficiently.

The algorithm steps are outlined in Alg. 7, where the key hyperparameters are defined as follows. First, the total number of iterations or generations, denoted as G, determines how long the algorithm runs. Each iteration involves evaluating a population of candidate solutions, the size of which is specified by λ. From this population, the top-performing μ solutions are selected for recombination, with the condition $\mu \leq \lambda$. The learning process also depends on several adaptation rates: c_σ and d_σ control the adaptation of the step size, while c_c, c_1, and c_μ influence the covariance matrix updates, ensuring efficient exploration of the search space. The initial mean vector, $\mathbf{m}^{(0)} \in \mathbb{R}^n$, represents the initial estimate in the search space, while the initial step size, $\sigma^{(0)}$, scales the search distribution. To ensure isotropic sampling at the outset, the initial covariance matrix, $\mathbf{C}^{(0)}$, is typically set to the identity matrix, $\mathbf{I}$. Note that it is common practice to search over a normalized space, where the search distribution is isotropic and centered at the origin, to improve numerical stability. Hence, the initial mean vector and covariance matrix are initialized to zero and the identity

matrix, respectively.

Algorithm 7: CMA-ES (Covariance Matrix Adaptation Evolution Strategy)

Input: Population size λ, initial mean $\mathbf{m}^{(0)}$, initial step size $\sigma^{(0)}$, initial covariance matrix $\mathbf{C}^{(0)} = \mathbf{I}$, weights $w_1, \ldots, w_\mu$

Output: Optimized solution $\mathbf{m}^*$

Initialize:

Set $w_i \leftarrow \frac{\log(\mu+\frac{1}{2})-\log(i)}{\sum_{j=1}^{\mu}(\log(\mu+\frac{1}{2})-\log(j))}$ for $i = 1, \ldots, \mu$

Normalize weights: $w_i \leftarrow \frac{w_i}{\sum_{j=1}^{\mu} w_j}$

Set $\mathbf{p}_\sigma^{(0)} \leftarrow \mathbf{0}$, $\mathbf{p}_c^{(0)} \leftarrow \mathbf{0}$, $\mu_{\text{eff}} \leftarrow (\sum_{i=1}^{\mu} w_i)^2 / \sum_{i=1}^{\mu} w_i^2$

Set learning rates $c_\sigma, c_c, c_1, c_\mu$, damping factor d_σ, and chi constant $\chi_n \leftarrow \sqrt{n}(1 - \frac{1}{4n} + \frac{1}{21n^2})$

Generation loop:

for $g = 0, 1, \ldots,$ *G-1* **do**

  for $k \leftarrow 1$ ***to*** λ **do**

    Sample: $\mathbf{x}_k \sim \mathcal{N}(\mathbf{m}, \sigma^2 \mathbf{C})$

    Evaluate fitness: $f(\mathbf{x}_k)$

  end

  Sort $\mathbf{x}_1, \ldots, \mathbf{x}_\lambda$ by fitness, and select the μ best solutions

  Compute new mean: $\mathbf{m}^{(g+1)} \leftarrow \sum_{i=1}^{\mu} w_i \mathbf{x}_i$

  Update step-size evolution path:

  $\mathbf{p}_\sigma^{(g+1)} \leftarrow (1 - c_\sigma)\mathbf{p}_\sigma^{(g)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{\text{eff}}} \cdot \mathbf{C}^{-\frac{1}{2}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$

  $h_\sigma \leftarrow \|\mathbf{p}_\sigma^{(g+1)}\| / \sqrt{1 - (1 - c_\sigma)^{2(g+1)}} < (1.4 + \frac{2}{n+1}) \cdot \chi_n$

  Update covariance evolution path:

  $\mathbf{p}_c^{(g+1)} \leftarrow (1 - c_c)\mathbf{p}_c^{(g)} + h_\sigma \cdot \sqrt{c_c(2 - c_c)\mu_{\text{eff}}} \cdot \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$

  Update covariance matrix:

  $\mathbf{C}^{(g+1)} \leftarrow (1 - c_1 - c_\mu)\mathbf{C}^{(g)} + c_1 \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)^\top} + c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_i \mathbf{y}_i^\top$

  where $\mathbf{y}_i = \frac{\mathbf{x}_i - \mathbf{m}^{(g)}}{\sigma^{(g)}}$

  Update step size:

  $\sigma^{(g+1)} \leftarrow \sigma^{(g)} \cdot \exp\left(\frac{c_\sigma}{d_\sigma} \cdot \left(\frac{\|\mathbf{p}_\sigma^{(g+1)}\|}{\chi_n} - 1\right)\right)$

end

return $\mathbf{m}^* \leftarrow \mathbf{m}^{(g+1)}$

The table below summarized the hyperparameters used in the CMA-ES algorithm for the Pendulum and Quadrotor tasks in Chapter 4.

Hyperparameter	Name	Pendulum	Quadrotor
G	Generations	40	100
λ	Population size	200	200
μ	Selected solutions	20	20
c_σ	Step-size learning rate	0.40	0.65
c_c	Covariance learning rate	0.14	0.68
c_1	Rank-1 update rate	0.0024	0.19
c_μ	Rank-μ update rate	0.038	0.27
d_σ	Step-size damping	1.40	2.41
$\mathbf{m}^{(0)}$	Initial mean vector	$\mathbf{0}$	$\mathbf{0}$
$\sigma^{(0)}$	Initial step size	0.4	0.4
$\mathbf{C}^{(0)}$	Initial covariance matrix	$\mathbf{I}$	$\mathbf{I}$

Appendix 4.E: Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm designed to optimize policies by maximizing the expected return [49]. However, reproducing PPO's results can be challenging due to its sensitivity to hyperparameters and implementation details [109]. For this reason, we provide the exact PPO implementation used in Chapter 4 in `https://bheijden.github.io/rex/`. Our implementation is largely based on [96], which itself builds upon [50].

The table below summarizes the hyperparameter settings used for the two tasks in Chapter 4:

Hyperparameter	Name	Pendulum	Quadrotor
T	Total timesteps	2×10^6	10×10^6
η	Learning rate	3.26×10^{-4}	9.23×10^{-4}
n_{envs}	Number of environments	128	128
n_{steps}	Number of steps per update	32	64
E	Number of epochs	8	16
$n_{\text{minibatch}}$	Number of minibatches	16	8
γ	Discount factor	0.9939	0.9844
λ_{GAE}	GAE lambda	0.971	0.939
ϵ_{clip}	Clipping epsilon	0.164	0.131
α_{ent}	Entropy coefficient	0.01	0.01
α_{vf}	Value function coefficient	0.802	0.756
g_{max}	Max gradient norm	0.963	0.76
n_{hidden}	Number of hidden layers	2	2
h_{units}	Number of hidden units	64	64
ϕ_{hidden}	Hidden activation	tanh	tanh
σ_{ind}	State-independent action noise	True	True
Squash	Action squashing	True	True
η_{anneal}	Anneal learning rate	False	False
Norm	Normalize environment	True	True

5

Resilience: Simulating Irrelevance to Enhance Task-Relevant Learning

Despite the advancements in flexibility, speed, and accuracy discussed in the preceding chapters, real-world environments are unpredictable and often introduce irrelevant dynamics that simulations cannot fully capture. These challenges can degrade the performance of learned policies when transferred to real-world applications.

In this chapter, we introduce DeepKoCo, an algorithm designed to enhance the resilience of learning-based robots by focusing on task-relevant dynamics. By using a lossy autoencoder to filter out irrelevant information, DeepKoCo improves the robustness of policies, ensuring their effectiveness in noisy real-world environments.

This chapter is partly based on *B. van der Heijden, L. Ferranti, J. Kober, and R. Babuska, (2024). "DeepKoCo: Efficient latent planning with a task-relevant Koopman representation", Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)* [110].

5.1 Introduction

From self-driving cars to vision-based robotic manipulation, emerging technologies are characterized by visual measurements of strongly nonlinear physical systems. Unlike in highly controlled lab environments where any measured change is likely relevant, cameras in real-world settings are notorious for mainly capturing task-irrelevant information, such as, the movement of other robots outside of a manipulator's workspace or cloud movements captured by the cameras of self-driving cars.

While Deep Reinforcement Learning (DRL) algorithms can learn to perform various tasks using raw images, they will require an enormous number of trials. Prior methods mitigate this by encoding the raw images into a lower-dimensional representation that allows for faster learning. However, these methods can be easily distracted by irrelevant dynamics [111]. This motivates data-driven methodologies that learn low-dimensional latent dynamics that are task-relevant and useful for control.

In the learning of latent dynamics for control, there is a trade-off between having an accurate dynamic model and one that is suitable for control. On one hand, latent dynamic models based on neural networks (NN) can provide accurate predictions over long horizons. On the other hand, their inherent nonlinearity renders them incompatible with efficient planning algorithms. Alternatively, one can choose to approximate the latent dynamics with a more restricted function approximation class to favor the use of efficient planning algorithms. In this respect, a promising strategy is represented by the Koopman framework [112]. Loosely speaking, this framework allows one to map observations with nonlinear dynamics to a latent space where the *global dynamics* of the autonomous system are approximately linear (Koopman representation). This enables the use of powerful linear optimal control techniques in the latent space [112].

While the Koopman framework is promising, existing methods have fundamental limitations that must be addressed to fully exploit the benefits of this method for control applications. First, methods that identify Koopman representations from data were designed for prediction and estimation. These methods were later adapted for control. These adaptations, however, lead to limiting assumptions on the underlying dynamics, such as assuming the Koopman representation to be linear in the states and actions [113–117]. Second, these methods are *task agnostic*, that is, the models represent all dynamics they observe, whether they are relevant to the task or not. This focuses the majority of their model capacity on potentially task-irrelevant dynamics.

Therefore, we introduce *Deep Koopman Control* (DeepKoCo), that is, a model-based reinforcement learning agent that learns a latent Koopman representation from raw pixel images and achieves its goal through planning in this latent space. The representation is *(i)* robust to task-irrelevant dynamics and *(ii)* compatible with efficient planning algorithms. We propose a lossy autoencoder network that reconstructs and predicts observed costs, rather than all observed dynamics, which leads to a representation that is task-relevant. The latent-dynamics model can represent continuously differentiable nonlinear systems and does not require knowledge of the underlying environment dynamics or cost function. We demonstrate the success of our approach on two continuous control tasks and show that our method is more robust to irrelevant dynamics than state-of-the-art approaches, that is, DeepMDP [118] and Deep Bisimulation for Control (DBC) [111].

5.2 Related Work

Koopman control Koopman theory has been used to control various nonlinear systems with linear control techniques, both in simulation [112, 116, 119] and in real-world robotic applications [114, 115]. Herein, [112, 114, 120] used a linear quadratic regulator (LQR), while [113, 115–117, 119] applied linear model predictive control (MPC). [113, 115, 117, 119] used data-driven methods that were derived from the Extended Dynamic Mode Decomposition (EDMD) [121] to find the Koopman representation. In contrast, [112, 114, 120] require prior knowledge of the system dynamics to hand-craft parts of the lifting function. Similar to [116], we rely on deep learning to derive the Koopman representation for control. However, we do not assume the Koopman representation to be (bi-)linear in the states and actions and we show how our representation can be used to control systems that violate this assumption. Compared to existing methods, we propose an agent that learns the representation online in a reinforcement learning setting using high-dimensional observations that contain irrelevant dynamics.

Latent planning Extensive work has been conducted to learn latent dynamics from images and use them to plan suitable actions [122–124]. [124] proposes a model-based agent that uses NNs for the latent dynamics and cost model. To find suitable action sequences, however, their method requires a significant computational budget to evaluate many candidate sequences. Alternatively, [122, 123] propose locally linear dynamic models, which allowed them to efficiently plan for actions using LQR. However, their cost function was defined in the latent space and required observations of the goal to be available. In contrast to our approach, all aforementioned methods are trained towards full observation reconstruction, which focuses the majority of their model capacity on potentially task-irrelevant dynamics.

Relevant representation learning [111, 118] filter task-irrelevant dynamics by minimizing an auxiliary bisimulation loss. Similar to our approach, they propose learning latent dynamics and predicting costs. Their method, however, is limited to minimizing a single-step prediction loss, while we incorporate multi-step predictions. This optimizes our model towards accurate long-term predictions. [125, 126] also proposed training a dynamics model towards predicting the future sum of costs given an action sequence. However, their method focused on discrete control variables, while we focus on continuous ones.

5.3 Preliminaries

This section briefly introduces the Koopman framework for autonomous and controlled nonlinear systems. A detailed description can be found in [112]. This framework is fundamental to the design of our latent model and control strategy.

5.3.1 Koopman eigenfunctions for autonomous systems

Consider the following autonomous nonlinear system $\dot{\boldsymbol{o}} = F(\boldsymbol{o})$, where the observations $\boldsymbol{o} \in \mathbb{R}^N$ evolve according to the smooth continuous-time dynamics $F(\boldsymbol{o})$. For such a system, there exists a lifting function $g(\cdot) : \mathbb{R}^N \to \mathbb{R}^n$ that maps the observations to a latent space

where the dynamics are linear, that is,

$$\frac{\mathrm{d}}{\mathrm{d}t}g(\boldsymbol{o}) = \mathcal{K}\circ g(\boldsymbol{o}), \tag{5.1}$$

where $\mathcal{K}$ is the infinitesimal operator generator of Koopman operators K. In theory, K is infinite dimensional (i.e., $n \to \infty$), but a finite-dimensional matrix representation can be obtained by restricting it to an invariant subspace. Any set of eigenfunctions of the Koopman operator spans such a subspace. Identifying these eigenfunctions [119, 127] provides a set of intrinsic coordinates that enable global linear representations of the underlying nonlinear system. A Koopman eigenfunction satisfies

$$\frac{\mathrm{d}}{\mathrm{d}t}\phi(\boldsymbol{o}) = K\phi(\boldsymbol{o}) = \lambda\phi(\boldsymbol{o}), \tag{5.2}$$

where $\lambda \in \mathbb{C}$ is the continuous-time eigenvalue corresponding to eigenfunction $\phi(\boldsymbol{o})$.

5

5.3.2 Koopman eigenfunctions for controlled systems

For controlled nonlinear system with action $\boldsymbol{a} \in \mathbb{R}^m$ and smooth continuous-time dynamics $\dot{\boldsymbol{o}} = \tilde{F}(\boldsymbol{o}, \boldsymbol{a})$, we follow the procedure in [112]. Given the eigenfunction $\phi(\boldsymbol{o}, \boldsymbol{a})$ augmented with $\boldsymbol{a}$ for the controlled system, we can take its time derivative and apply the chain rule with respect to $\boldsymbol{o}$ and $\boldsymbol{a}$, leading to

$$\frac{\mathrm{d}}{\mathrm{d}t}\phi(\boldsymbol{o}, \boldsymbol{a}) = \underbrace{\triangledown_{\boldsymbol{o}}\phi(\boldsymbol{o}, \boldsymbol{a})\tilde{F}(\boldsymbol{o}, \boldsymbol{a})}_{\lambda\phi(\boldsymbol{o},\boldsymbol{a})} + \triangledown_{\boldsymbol{a}}\phi(\boldsymbol{o}, \boldsymbol{a})\dot{\boldsymbol{a}}, \tag{5.3}$$

where λ is now the eigenvalue that corresponds to eigenfunction $\phi(\boldsymbol{o}, \boldsymbol{a})$. Since $\dot{\boldsymbol{a}}$ can be chosen arbitrarily, we could set it to zero and instead interpret each action as a parameter of the Koopman eigenfunctions. Thus, for any given choice of parameter $\boldsymbol{a}$ the standard relationship in Eq. (5.2) is recovered in the presence of actions. A local approximation of the Koopman representation is obtained when $\dot{\boldsymbol{a}}$ is nonzero.

5.3.3 Identifying Koopman eigenfunctions from data

To facilitate eigenfunction identification with discrete data, Eq. (5.3) can be discretized with a procedure similar to [127]. The eigenvalues $\lambda_{\pm} = \mu \pm \mathrm{i}\,\omega$ are used to parametrize block-diagonal $\Lambda = \mathrm{diag}(J^1, J^2, ..., J^P) \in \mathbb{R}^{2P\times 2P}$. For all P pairs of complex eigenvalues, the discrete-time operator Λ has a Jordan real block of the form

$$J(\mu, \omega) = e^{\mu\Delta t}\begin{bmatrix}\cos(\omega\Delta t) & -\sin(\omega\Delta t)\\ \sin(\omega\Delta t) & \cos(\omega\Delta t)\end{bmatrix}, \tag{5.4}$$

with sampling time Δt. The "forward Euler method" provides a discrete approximation of the control matrix, so that Eq. (5.3) can be discretized as

$$\varphi(\boldsymbol{o}_{k+1}, \boldsymbol{a}_{k+1}) = \Lambda\varphi(\boldsymbol{o}_k, \boldsymbol{a}_k) + \underbrace{\triangledown_{\boldsymbol{a}_k}\varphi(\boldsymbol{o}_k, \boldsymbol{a}_k)}_{B_{\varphi_k}}\underbrace{\dot{\boldsymbol{a}}_k\Delta t}_{\Delta\boldsymbol{a}_k}. \tag{5.5}$$

Herein, the stacked vector $\varphi = (\phi^1, \phi^2, ..., \phi^P)$ comprises a set of P eigenfunctions with $\phi^j \in \mathbb{R}^2$ associated with complex eigenvalue pair $\lambda_\pm^j$ and Jordan block J^j. Subscript k corresponds to discretized snapshots in time. If we view the action increment $\Delta \boldsymbol{a}_k = \boldsymbol{a}_{k+1} - \boldsymbol{a}_k$ in Eq. (5.5) as the controlled input instead, we obtain a discrete *control-affine* Koopman eigenfunction formulation with *linear autonomous* dynamics for the original *non-control-affine nonlinear* system. In the next section, we show that Eq. (5.5) plays a central role in our latent model.

5.4 Learning Relevant Koopman Eigenfunctions

For efficient planning in the latent space, we propose to learn a latent dynamics model that uses Koopman eigenfunctions as its latent state. This section describes this model and how the Koopman eigenfunctions can be identified robustly, that is, in a way that the identified eigenfunctions remain unaffected by task-irrelevant dynamics that are expected to contaminate the observations.

5.4.1 Koopman latent model

We propose a *lossy* autoencoder that builds on the deep autoencoder in [127]. Compared to [127], our autoencoder enables control. To train the latent model, we provide the training objective that is to be minimized given a buffer D that contains observed sequences $\{t_k\}_{k=0}^T$ of a Markov decision process with tuples $t_k = (\boldsymbol{o}_k, \boldsymbol{a}_k, \Delta \boldsymbol{a}_k, c_k)$, where c_k are observed scalar costs. The proposed latent model is illustrated in Fig. 5.1 with more details below on the individual components of the architecture.

Encoder The encoder φ is the approximate eigenfunction that maps an observation-action pair $(\boldsymbol{o}_k, \boldsymbol{a}_k)$ to the latent state $\boldsymbol{s}_k$. The encoder φ is parametrized by a neural network, defined as

$$\boldsymbol{s}_k = \varphi(\boldsymbol{o}_k, \boldsymbol{a}_k). \tag{5.6}$$

Latent Dynamics The latent state $\boldsymbol{s}_k$ approximates a Koopman eigenfunction, so the autonomous time evolution in the latent space is linear and dictated by Λ.

Note here that $\boldsymbol{a}_k$ is part of the *augmented* latent state and we view the action increment $\Delta \boldsymbol{a}_k$ as the controlled variable that is determined by the policy. This leads to the dynamics model in Eq. (5.7), which we derived from Eq. (5.5). The Koopman operator Λ is parametrized by P complex conjugate eigenvalue pairs $\lambda_\pm^j$. We do not assume the latent dynamics to be linear in the control. Instead, the influence of $\Delta \boldsymbol{a}_k$ on the latent state varies and depends on the partial derivative of the encoder with respect to the action, i.e., the state-dependent matrix $B_{\varphi_k} = \triangledown_{\boldsymbol{a}_k} \varphi(\boldsymbol{o}_k, \boldsymbol{a}_k) \in \mathbb{R}^{2P \times m}$.

$$\begin{bmatrix} \boldsymbol{s}_{k+1} \\ \boldsymbol{a}_{k+1} \end{bmatrix} = \begin{bmatrix} \Lambda & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_k \\ \boldsymbol{a}_k \end{bmatrix} + \begin{bmatrix} B_{\varphi_k} \\ I \end{bmatrix} \Delta \boldsymbol{a}_k. \tag{5.7}$$

Cost Model The environment contains a cost function that produces a scalar cost observation c_k at every time-step. For planning in the latent space, we require a cost model $\hat{c}_k$ as a function of the latent state. This cost approximates the observed cost (i.e., $\hat{c}_k \approx c_k$). We

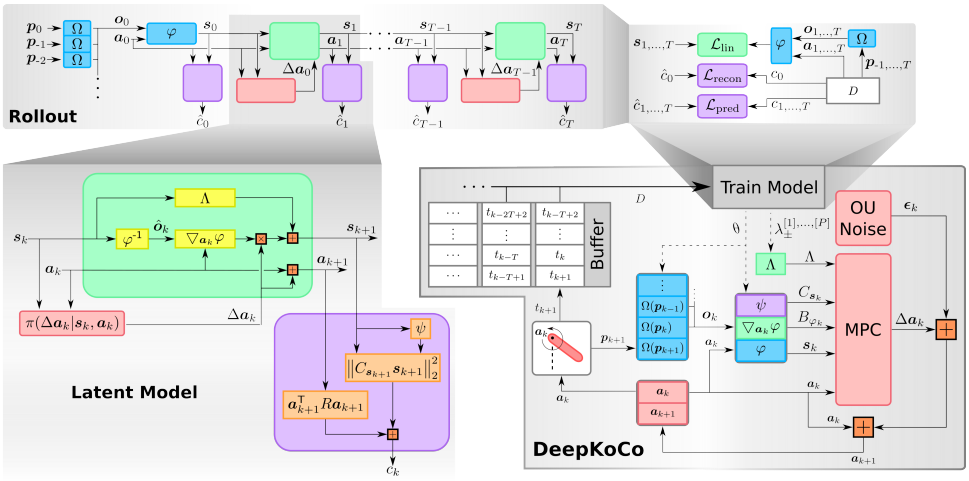


Figure 5.1: **Latent Model** The proposed network architecture of the *latent model*, consisting of the dynamic model, cost model, and policy (depicted in green, purple, and red, respectively). **Rollout** A multi-step ahead prediction with the *latent model*. Note that we only encode an observation at the first time-step (blue boxes), after which we remain in the latent space. **DeepKoCo** The training procedure that corresponds to Alg. 8.

adopt a latent state-dependent quadratic cost model to facilitate the use of fast planning algorithms (Sec. 5.5). The entries of $C_{\boldsymbol{s}_k} \in \mathbb{R}^{1\times 2P}$ are determined by a function $\psi(\boldsymbol{s}_k)$ that is parametrized by a neural network. The weights of ψ are initially unknown and must be learned together with the rest of the latent model. We assume that the cost of applying action $\boldsymbol{a}_k$ is known *a priori* and defined by matrix R. This leads to the cost model

$$\hat{c}_k = \|C_{\boldsymbol{s}_k}\boldsymbol{s}_k\|_2^2 + \boldsymbol{a}_k^\mathsf{T} R \boldsymbol{a}_k. \tag{5.8}$$

Policy The action increment $\Delta\boldsymbol{a}_k$ is the controlled variable that is sampled from a probability distribution π, conditioned on the augmented latent state (i.e., $\Delta\boldsymbol{a}_k \sim \pi(\Delta\boldsymbol{a}_k|\boldsymbol{s}_k, \boldsymbol{a}_k)$). Even though the model is deterministic, we define the policy to be stochastic to allow for stochastic exploration. The policy will be specified further in Sec. 5.5.

$$\Delta\boldsymbol{a}_k \sim \pi(\Delta\boldsymbol{a}_k|\boldsymbol{s}_k, \boldsymbol{a}_k) \tag{5.9}$$

Decoder After learning the latent model we intend to plan over it, which involves a multi-step prediction. Given only the encoder Eq. (5.6), dynamics model Eq. (5.7), policy Eq. (5.9) and the current $(\boldsymbol{o}_k, \boldsymbol{a}_k)$-pair, we would be limited to single-step predictions of $\boldsymbol{s}_{k+1}$ at run-time, because multi-step predictions $\boldsymbol{s}_{k+i}$ with $i > 1$ would require knowledge of future observations $\boldsymbol{o}_{k+i}$ to evaluate $B_{\varphi_{k+i}}$. Therefore, to make multi-step predictions, we introduce a decoder φ^{-1} (parametrized by a NN) in Eq. (5.10) that uses predicted latent states $\boldsymbol{s}_{k+i}$ to construct pseudo-observations $\hat{\boldsymbol{o}}_{k+i}$ that produce the same partial derivative as the true observation (i.e., $\triangledown_{\boldsymbol{a}_k}\varphi(\varphi^{-1}(\boldsymbol{s}_k), \boldsymbol{a}_k) \approx \triangledown_{\boldsymbol{a}_k}\varphi(\boldsymbol{o}_k, \boldsymbol{a}_k)$). Future values $\boldsymbol{a}_{k+i}$ do not

pose a problem, because they can be inferred from the policy Eq. (5.9) and dynamics model Eq. (5.7).

$$\hat{\boldsymbol{o}}_k = \varphi^{-1}(\boldsymbol{s}_k). \tag{5.10}$$

Image Processor When the observations are raw pixel images $\boldsymbol{p}_k$, not all relevant information can be inferred from a single observation. To restore the Markov property, we pass the last d consecutive pixel images through a convolutional neural network Ω in Eq. (5.11), stack the output into a single vector, and consider that to be the observation $\boldsymbol{o}_k$ instead. In that case, the observed sequences consist of tuples $t_k = (\boldsymbol{p}_k, ..., \boldsymbol{p}_{k-d+1}, \boldsymbol{a}_k, \Delta\boldsymbol{a}_k, c_k)$.

$$\boldsymbol{o}_k = \Omega(\boldsymbol{p}_k, ..., \boldsymbol{p}_{k-d+1}), \tag{5.11}$$

5.4.2 Learning the latent model

Our latent model should have linear dynamics and be predictive of observed costs. These two high-level requirements lead to the following three losses which are minimized during training.

Linear Dynamics To ensure that the latent state is a valid Koopman eigenfunction, we regularize the time evolution in the latent space to be linear by using the following loss,

$$\text{Linear loss:} \qquad \mathcal{L}_{\text{lin}} = \frac{1}{T}\sum_{k=0}^{T-1} \|\varphi(\boldsymbol{o}_{k+1}, \boldsymbol{a}_{k+1}) - \boldsymbol{s}_{k+1}\|_{\text{MSE}}, \tag{5.12}$$

where $\boldsymbol{s}_{k+1}$ is obtained by rolling out a latent trajectory as illustrated in Fig. 5.1.

Cost Prediction We want the latent representation to contain all necessary information to solve the task. If we would naively apply an autoencoder that predicts future observations, we focus the majority of the model capacity on potentially task-irrelevant dynamics contained in the observations. To learn a latent representation that *only* encodes relevant information, we propose to use a lossy autoencoder that is predictive of current and future costs instead. Such a representation would allow an agent to predict the cost evolution of various action sequences and choose the sequence that minimizes the predicted cumulative cost, which is essentially equivalent to solving the task. Because we only penalize inaccurate cost predictions, the encoder is not incentivized to encode task-irrelevant dynamics into the latent representation as they are not predictive of the cost. This leads to the task-relevant identification of the lifting function φ. Cost prediction accuracy is achieved by using the following two losses,

$$\text{Reconstruction loss:} \qquad \mathcal{L}_{\text{recon}} = \|c_0 - \hat{c}_0\|_{\text{MSE}} \tag{5.13}$$

$$\text{Prediction loss:} \qquad \mathcal{L}_{\text{pred}} = \frac{1}{T}\sum_{k=1}^{T} \|c_k - \hat{c}_k\|_{\text{MSE}} \tag{5.14}$$

Training Objective We minimize the losses in Eq. (5.12), Eq. (5.13), and Eq. (5.14), corresponding to linear dynamics regularization and cost prediction, together with an $L2$-regularization loss $\mathcal{L}_{\text{reg}}$ on the trainable variables (excluding neural network biases). This leads to the following training objective,

$$\min_{\theta, \lambda_{\pm}^{[1],\dots,[P]}} \mathcal{L}_{\text{lin}} + \alpha_1(\mathcal{L}_{\text{recon}} + \mathcal{L}_{\text{pred}}) + \alpha_2 \mathcal{L}_{\text{reg}}, \tag{5.15}$$

where θ is the collection of all the trainable variables that parametrize the encoder φ, decoder φ^{-1}, cost model ψ, and convolutional network Ω (in case of image observations). Weights α_1, α_2 are hyperparameters. The model is trained using the Adam optimizer [128] with learning rate $\tilde{\alpha}$, on batches of B sequences $\{t_k\}_{k=0}^{T}$ for E epochs.

5.5 Deep Koopman Control

This section introduces the agent that uses the Koopman latent model to find the action sequence that minimizes the predicted cumulative cost. We use linear model-predictive control (LMPC) to allow the agent to adapt its plan based on new observations, meaning the agent re-plans at each step. Re-planning at each time-step can be computationally costly. In the following, we explain how to exploit and adapt our latent model and cost model to formulate a sparse and convex MPC problem that can be solved efficiently online.

The planning algorithm should achieve competitive performance, while only using a limited amount of computational resources. This motivates choosing Koopman eigenfunctions as the latent state, because the *autonomous* dynamics are linear. The dynamics are affine in the controlled variable $\Delta \boldsymbol{a}_k$ that is multiplied in the definition of the state space by B_{φ_k}, which depends on the latent state. Similarly, $C_{\boldsymbol{s}_k}$ requires the evaluation of the nonlinear function $\psi(\boldsymbol{s}_k)$. There exist methods that can be applied in this setting, such as the State-Dependent Ricatti Equation (SDRE) method [129]. While the SDRE requires less complexity compared to sample-based nonlinear MPC (e.g. CEM [130]), it remains computationally demanding as it also requires the derivative of ψ with respect to s_k at every step of the planning horizon.

Our goal is to reduce the online complexity of our planning strategy, while also dealing with input constraints. Hence, we trade-off some prediction accuracy (due to the mismatch between the latent model and the MPC prediction model) to simplify the online planning strategy by using linear MPC. We propose to evaluate the state-dependent matrices $C_{\boldsymbol{s}_0}$ and B_{φ_0} at time-step $k = 0$ (obtained from our latent model) and keep them both fixed for the rest of the LMPC horizon. This assumes that the variation of B_{φ_k} and $C_{\boldsymbol{s}_k}$ is limited over the prediction horizon (compared to Eq. (5.7) and Eq. (5.8)). Nevertheless, thanks to this simplification we can rely on LMPC for planning that can be solved efficiently. Specifically, once we evaluate $\boldsymbol{s}_0$, $C_{\boldsymbol{s}_0}$, and B_{φ_0}, the computational cost of solving the MPC problem in the *dense form* [6] scales linearly with the latent state dimension due to the diagonal structure of Λ. As Section 5.6 details, this simplification allows our method to achieve competitive final performance, while only requiring a single evaluation of the NNs Ω, φ, and ψ. This significantly decreases the computational cost at run-time compared to sample-based nonlinear MPC (e.g., CEM [130]) that would require many evaluations of the NNs at every time-step. In contrast to LQR, LMPC can explicitly deal with actuator saturation by incorporating constraints on $\boldsymbol{a}$. The proposed planning strategy based on

Algorithm 8: Deep Koopman Control (DeepKoCo)

Input: Model parameters: P, d
Policy parameters: $\zeta = \{H, R, \tilde{R}\}$
Noise parameters: $\lambda^{\text{ou}}, \sigma^{2,\text{ou}}_{\text{init}}, N^{\text{ou}}$
Train parameters: N, L, T, E, B
Optimization parameters: $\xi = \{\alpha_1, \alpha_2, \tilde{\alpha}\}$
Output: Eigenvalues $\lambda_{\pm}^{[1],\dots,[P]}$
Trained networks $\varphi, \varphi^{-1}, \psi, \Omega$

$\theta, \lambda_{\pm}^{[1],\dots,[P]} \leftarrow$ InitializeModel(P, R)
while *not converged* **do**
  $\Lambda \leftarrow$ GetKoopmanOperator$(\lambda_{\pm}^{[1],\dots,[P]})$
  for *episode* $l = 1, \dots, N$ **do**
    $\boldsymbol{p}_0, \dots, \boldsymbol{p}_{1-d} \leftarrow$ ResetEnvironment()
    $\boldsymbol{a}_0 \leftarrow 0$
    for *time-step* $k = 0, \dots, L$ **do**
      $\boldsymbol{o}_k \leftarrow$ ProcessImages$(\boldsymbol{p}_k, \dots, \boldsymbol{p}_{k-d+1}, \theta)$
      $\boldsymbol{s}_k, B_{\varphi_k}, C_{\boldsymbol{s}_k} \leftarrow$ LatentModel$(\boldsymbol{o}_k, \boldsymbol{a}_k, \theta)$
      $\Delta \boldsymbol{a}_k \leftarrow$ LMPC$(\boldsymbol{s}_k, \boldsymbol{a}_k, B_{\varphi_k}, C_{\boldsymbol{s}_k}, \Lambda, \zeta)$
      $\boldsymbol{a}_{k+1} \leftarrow \boldsymbol{a}_k + \Delta \boldsymbol{a}_k +$ Noise$(\lambda^{\text{ou}}, \sigma^{2,\text{ou}})$
      $\boldsymbol{p}_{k+1}, c_k \leftarrow$ ApplyAction$(\boldsymbol{a}_k)$
    $D \leftarrow D \cup$ CreateSequences$(T, \{t_k\}_{k=0}^{L})$
  $\theta, \lambda_{\pm}^{[1],\dots,[P]} \leftarrow$ TrainModel$(D, \theta, \lambda_{\pm}^{[1],\dots,[P]}, \xi, E, B)$

LMPC is defined as follows:

$$\begin{aligned} \min_{\Delta \boldsymbol{a}^{0,\dots,H-1}} \quad & \sum_{k=1}^{H} \| C_{\boldsymbol{s}_0} \boldsymbol{s}_k \|_2^2 + \boldsymbol{a}_k^{\mathsf{T}} R \boldsymbol{a}_k + \Delta \boldsymbol{a}_k^{\mathsf{T}} \tilde{R} \Delta \boldsymbol{a}_k, \\ \text{s.t.} \quad & \begin{bmatrix} \boldsymbol{s}_{k+1} \\ \boldsymbol{a}_{k+1} \end{bmatrix} = \begin{bmatrix} \Lambda & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_k \\ \boldsymbol{a}_k \end{bmatrix} + \begin{bmatrix} B_{\varphi_0} \\ I \end{bmatrix} \Delta \boldsymbol{a}_k, \\ & \boldsymbol{a}^{\min} \le \boldsymbol{a}_k \le \boldsymbol{a}^{\max}, \text{ for } k = 1, \dots, H, \end{aligned} \tag{5.16}$$

where H is the prediction horizon. Positive-definite matrix $\tilde{R}$ penalizes the use of $\Delta \boldsymbol{a}_k$ and is required to make the problem well-conditioned. Its use does introduce a discrepancy between the approximate cost model Eq. (5.8) and the cumulative cost ultimately minimized by the agent Eq. (5.16). Therefore, the elements in $\tilde{R}$ are kept as low as possible.

To align the representation learning objective Eq. (5.15) with the linear MPC objective Eq. (5.16), we also fix the state-dependent terms $C_{\boldsymbol{s}_0}$ and B_{φ_0} at time-step $k = 0$ in the evaluation of the cost prediction loss Eq. (5.14) and linear loss Eq. (5.12). Note that this does not mean that $C_{\boldsymbol{s}_k}$ and B_{φ_k} are constant in the latent model Eq. (5.7), Eq. (5.8). The matrices remain state-dependent, but their variation is limited over the sequence length T. In general, we choose the sequence length to be equal to the prediction horizon. Hence, we learn a representation that provides local linear models that are particularly accurate around $\boldsymbol{s}_k$ in the direction of the (goal-directed) trajectories gathered during training.

To gather a rich set of episodes to learn the Koopman latent model, we add colored noise to the actions commanded by the agent's linear MPC policy, that is, $\boldsymbol{a}_{k+1} = \boldsymbol{a}_k + \Delta \boldsymbol{a}_k + \boldsymbol{\epsilon}_k$. This adds a stochastic exploration component to the policy. We use an Ornstein-Uhlenbeck

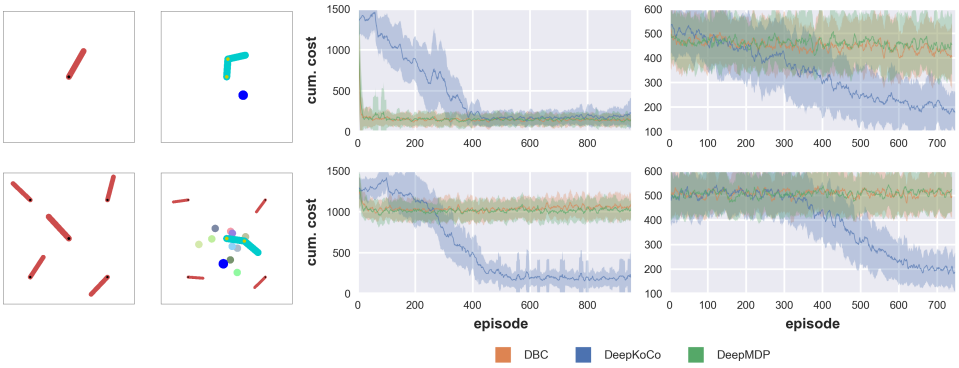


Figure 5.2: **Left** Typical setups for the two tasks in a clean scenario (first row) and distractor scenario (second row). In all setups, the center system is controlled by the agent. In the manipulator task, the moving target is a blue ball. **Right** Learning curves when using state observations. The grid-location of each figure corresponds to the grid-location of each setup on the left. The mean cumulative cost over the last 10 episodes (line) with one standard deviation (shaded area) over 5 random seed runs are shown.

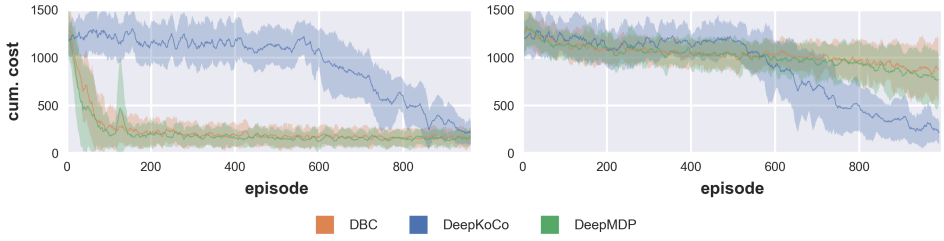


Figure 5.3: Learning curves when using images as observations in the pendulum task. The mean cumulative cost over the last 10 episodes (line) with one standard deviation (shaded area) over 5 random seed runs are shown. **Left** Clean scenario. **Right** Distractor scenario.

(OU) process to generate the additive colored noise with decay rate λ^{ou}. The variance $\sigma^{2,\text{ou}}$ is linearly annealed from $\sigma^{2,\text{ou}}_{\text{init}} \longrightarrow 0$ over $1, \ldots, N^{\text{ou}}$ episodes, that is, after N^{ou} episodes the policy becomes deterministic.

An overview of the proposed method is shown in Alg. 8. First, we initialize all model parameters. Then, we construct the Koopman operator Λ with Eq. (5.4) and gather N episodes of experience. Each time-step, we process the image observations with Eq. (5.11), evaluate the latent model Eq. (5.6), Eq. (5.7), and Eq. (5.8), and use it to find $\Delta \boldsymbol{a}_k$ with Eq. (5.16). Noise is added to the action increment before it is applied to the environment. We fill the experience buffer D with N episodes, split into sequences of length T, and train on them for E epochs with Eq. (5.15). This is repeated until convergence.

5.6 Results

We evaluate *DeepKoCo* on two continuous control tasks, namely OpenAI's pendulum swing-up task and a manipulator task. The manipulator task is similar to OpenAI's reacher

task where the two joints of a two-link arm are torque controlled and the Euclidean distance between the arm's end-point and a target must be minimized. However, we increase the difficulty by allowing the target to move at a constant angular velocity and radius around the arm's center-joint. Given that the angular velocity and radius vary randomly over episodes, the manipulator must learn to track arbitrary circular trajectories at different speeds. The dynamics for the manipulator can be formulated as $\boldsymbol{x}_{k+1} = F(\boldsymbol{x}_k) + B(\boldsymbol{x}_k)\boldsymbol{a}_k$, where $\boldsymbol{x}$ is the original nonlinear state. Such dynamics do not necessarily admit a Koopman representation that is (bi-)linear in the states and actions, as is often assumed in literature [114, 116].

To investigate the effect of distractor dynamics, we test each task in two different scenarios. In the first scenario, only relevant dynamics are observed, while in the second one we purposely contaminate the observations with distractor dynamics. The agent is either provided with a concatenated state observation containing the state measurements of both the relevant system and distractors (while not knowing which states are the relevant ones) or image observations with all systems in-frame (refer to Fig. 5.2 for the setup). The state observation dimension from the clean to the distractor scenario increases from 3 to 15 for the pendulum and from 10 to 50 for the manipulator. A video of the simulations using DeepKoCo accompanies the chapter [131].

Baselines We compare with two baselines that both combine model-free reinforcement learning with an auxiliary bisimulation loss to be robust against task-irrelevant dynamics: *(i)* Deep Bisimulation for Control (DBC) [111], and *(ii)* DeepMDP [118]. In case of state observations, we replace their convolutional encoder, with our fully connected encoder.

Hyperparameters We use the same set of hyperparameters throughout all experiments, except for the number of complex eigenvalue pairs P, that is, $P = 10$ and $P = 30$ in the swing-up and manipulator task, respectively to cope with the complexity of the scenarios. As policy parameters, we use $H = 15, R = 0.001, \tilde{R} = 0.01, \lambda^{\text{OU}} = 0.85, \sigma^{2,\text{ou}}_{\text{init}} = 0.85$. Initially, we fill the experience buffer D with $N = 90$ episodes, split into sequences of length $T = 15$, and train on them for $E = 100$ epochs. Then, we continuously add the sequences of $N = 20$ episodes to the buffer and train on the complete buffer for $E = 3$ epochs. As optimization parameters, we use $\alpha_1 = 10, \alpha_2 = 10^{-14}, \tilde{\alpha} = 0.001$. In case of image observations, we stack the last $d = 3$ images, downsample them to $3 \times 64 \times 64$ pixels before passing them through the convolutional NN defined in [132]. The networks $\varphi, \varphi^{-1}, \psi$ are 2-layered fully connected NNs with 90, 90, and 70 units per layer, respectively. The layers use ReLU activation and are followed by a linear layer. The number of complex eigenvalue pairs P, planning horizon H, and action increment cost $\tilde{R}$ are the most important parameters to tune.

Clean Scenario Concerning the pendulum task, the baselines converge more quickly to the final performance compared to DeepKoCo, as the top-left graph of Fig. 5.2 shows. Nevertheless, we do consistently achieve a similar final performance. The slower convergence can be explained by the added noise, required for exploration, that is only fully annealed after 400 episodes. We believe the convergence rate can be significantly improved by performing a parameter search together with a faster annealing rate, but we do not expect to be able to match the baselines in this ideal scenario. Note that, despite the apparent

simplicity of the application scenario, finding an accurate Koopman representation for the pendulum system is challenging, because it exhibits a continuous eigenvalue spectrum and has multiple (unstable) fixed points [127]. Concerning the manipulator task, both baselines were not able to solve the manipulator task with a moving target as the top-right graph of Fig. 5.2 shows, while they were able to learn in case the target was fixed. This shows that learning to track arbitrary circular references is significantly harder than regulating towards a fixed goal. Despite the increased difficulty, DeepKoCo learns to track the moving target. Finally, note that the manipulator task shows that the proposed method can deal with a multi-dimensional action-space and non-quadratic cost functions, that is, the Euclidean norm (the square root of an inner product).

Distractor Scenario In the more realistic scenario, both baselines fail to learn anything. In contrast, our approach is able to reach the same final performance as in the clean scenario, in a comparable amount of episodes, as the bottom row of Fig. 5.2 shows. This result can be explained by noticing that our multi-step cost prediction (in our loss function Eq. (5.14)) provides a stronger learning signal for the agent to distinguish relevant from irrelevant dynamics. For the manipulator task, there is a tracking error caused by the trade-off of using fixed B_φ and C_s along the MPC prediction horizon for efficiency. While our latent model presented in Sec. 5.4 supports state-dependent matrices, we decided to keep them fixed in the control design for efficiency.

Image Observations Fig. 5.3 shows the results for the pendulum task when images are used instead of state observations. In both clean and distractor scenarios, our approach is able to reach a similar final performance compared to using state observations. As expected, the baselines struggle to learn in the distractor scenario. This supports our statement that our approach learns a task-relevant Koopman representation from high-dimensional observations. We plan to test the manipulator task with images both in simulation and in real-world experiments.

5.7 Conclusion

We presented a model-based agent that uses the Koopman framework to learn a latent Koopman representation from images. DeepKoCo can find Koopman representations that *(i)* enable efficient linear control, *(ii)* are robust to distractor dynamics. Thanks to these features, DeepKoCo outperforms the baselines (two state-of-the-art model-free agents) in the presence of distractions. As part of our future work, we will extend our deterministic latent model with stochastic components to deal with partial observability and aleatoric uncertainty. Furthermore, we will extend our cost model to deal with sparse rewards, as they are often easier to provide.

6

Conclusions and Outlook

This thesis contributes to the field of robotic learning by focusing on the balance between flexibility, accuracy, and speed in simulator design. By optimizing these factors, we address key challenges in creating simulations that are both representative of real-world scenarios and efficient for learning robotic policies. This chapter summarizes the main contributions, discuss the broader implications of the findings, and outline future research directions.

6.1 Conclusions

Given the diverse and evolving nature of robotic tasks, a flexible simulator must accommodate various systems and adapt to evolving requirements. To address this, the sim2real framework EAGERx (Engine Agnostic Graph Environments for Robotics) was introduced in Chapter 2, featuring a graph-based architecture that enables modular representation of tasks, systems, and simulators. By adding, removing, or reconfiguring nodes, EAGERx allows users to modify the architecture easily to suit different tasks. The framework was demonstrated across diverse systems such as pendulums, quadrupeds, quadrotors, and manipulators, interfacing seamlessly with multiple simulators and supporting various state, action, and time-scale abstractions. While the graph-based design enabled flexibility, it introduced challenges in simulating real-world asynchronous effects and coordinating components at faster-than-real-time speeds. To address this, EAGERx incorporated a novel synchronization protocol that manages inter-node communication based on node rates and anticipated delays, ensuring consistent simulation behavior at faster-than-real-time speeds. An ablation study confirmed that abstractions, delay simulation, and proper synchronization—key features enabled by the graph-based architecture—resulted in the smallest sim2real gap, and led to the best policy transfer performance in two real-world robotic tasks. Thus, flexibility offered by our graph-based design plays a pivotal role in integrating these features and narrowing the sim2real gap, allowing for effective policy transfer to real-world scenarios.

While flexibility is essential for accommodating diverse robotic tasks, speed is equally crucial for robot learning. Speed in simulated learning provides critical benefits, such as faster training times and the ability to conduct extensive hyperparameter tuning, which makes parallelization a key strategy to achieve these advantages. With the advent of

accelerated physics-based simulations, parallelization has become a popular approach to improving training efficiency. Traditional reinforcement learning setups typically assume a single, synchronized environment built around a physics engine that interacts with the agent step-by-step, enabling straightforward parallelization. However, robotic system simulations encompass more than just physics, involving dynamics that are asynchronous and hierarchical in nature. In Chapter 3, we showed that the hierarchical and asynchronous nature of real-world systems complicates parallelization in simulation, leading to redundant computations and longer training times. To address these challenges and unlock the potential of parallelization for hierarchical and asynchronous systems, we introduced a method that efficiently parallelizes graph-based simulations on accelerator hardware. Building on the flexibility of the graph-based architecture introduced in Chapter 2, which supports asynchronous interactions and time-scale differences, our *supergraph* method enables efficient parallelized simulation of such systems by minimizing redundant computations. We demonstrate that this method doubles computational efficiency in simulating two real-world robotic systems compared to baseline methods while maintaining superior accuracy in handling asynchronous interactions and delays, leading to improved policy transfer performance. Furthermore, our results show that efficiency gains scale significantly, increasing by up to 20 times as the system complexity grows to 64 asynchronously running components. Thus, Chapter 3 advances robot simulation for learning by extending beyond parallelized physics simulations to efficiently handle asynchronous and hierarchical dynamics of robotic systems.

While modeling asynchronous and hierarchical real-world systems is crucial for accurate simulation, and parallelizing these simulations is essential for efficient learning, these alone are not sufficient to bridge the sim2real gap. A graph-based architecture provides a parametric structure expressive enough to capture real-world complexity, but configuring it to accurately model the system remains a significant challenge. This is analogous to a neural network capable of approximating any function, but requiring data-driven training to find the right weights. Similarly, a graph-based simulator may theoretically represent complex robotic systems, but identifying the correct delays and parameters to capture system dynamics, including asynchronous interactions, demands estimation from real-world data. To address this, we introduced REX (Robotic Environments with jaX) in Chapter 4, a framework designed to estimate system dynamics and delays from real-world data, thereby enhancing simulation fidelity and policy transfer. Building on the graph-based architecture, REX incorporates asynchronous and hierarchical dynamics with efficient parallelization strategies from earlier chapters, enabling the simultaneous estimation of system dynamics and delays using evolutionary strategies that leverage accelerator hardware. We demonstrated that accurately modeling delays and asynchronous operations allows the training of delay-aware policies, significantly improving policy transfer performance compared to baseline methods that disregard delays in two real-world robotic systems. We also explored training delay-agnostic policies while compensating for delays at inference using Smith-predictor-based strategies during deployment. This approach improved performance in simulation by simplifying the learning task, since solving a delayed task is inherently harder than a delay-free version, while still achieving comparable real-world performance. This ties back to the flexibility highlighted in Chapter 2, as the graph-based architecture supports the integration of delay compensation as a state abstraction, simplifying the

overall learning process.

By demonstrating how graph-based architectures can serve as flexible models to capture the complexity of real-world systems, how the supergraph method can enhance parallelization, and how real-world data can be used to estimate delays and dynamics, we have shown the potential of graph-based simulations in balancing flexibility, speed, and accuracy for robot learning. However, as tasks grow in complexity, we inevitably face the challenge of the unknown. While policy transfer in simulated learning depends on accurately modeling all scenarios that an agent might encounter, the real world is infinitely complex, with the number of possible scenarios expanding rapidly as the observation space grows. For example, while we may be able to simulate different types of roads for a self-driving car, we cannot simulate every possible lighting condition, weather pattern, or all the objects that a car might pass. Instead of attempting to simulate every possible scenario, we should focus on accurately modeling an abstraction of all *relevant* scenarios while building resilience against the infinite number of *irrelevant* ones, thereby enhancing robustness. This was the focus of Chapter 5, where we demonstrated that learning a latent dynamics model that predicts future rewards enables the agent to focus on task-relevant dynamics while ignoring irrelevant distractions. We showed that this approach significantly improves the robustness of learned policies, maintaining performance even in the presence of distractors, achieving comparable results in two simulated benchmark tasks as in a distractor-free environment. In contrast, baseline methods struggle, failing to learn the task when distractors are present, despite achieving perfect performance in the absence of distractions. We draw parallels between the delay compensation strategies at inference time in Chapter 4 and the latent task-relevant dynamics model discussed in Chapter 5. Both approaches simplify the learning task in simulation by reducing the complexity of the environment to a delay-free and entirely task-relevant one. In doing so, the complexities of delay compensation and distractor filtering are shifted to the inference stage during real-world deployment. Thus, the resilience approach in Chapter 5 complements the delay compensation strategy in Chapter 4, both demonstrating the utility of a flexible framework by integrating state abstractions to enhance the robustness and transferability of policies learned in simulation to real-world scenarios.

In summary, this thesis contributes to the field of robotic learning by developing a structured simulator design that balances flexibility, accuracy, and speed. We have utilized graph-based architectures to enhance simulator flexibility, enabling modular representation of robotic tasks and facilitating the integration of state, action, and time-scale abstractions to improve learning efficiency and transferability. Moreover, the graph-based architectures are well-suited for creating unified software pipelines that bridge real and simulated robot learning. These graph-based simulations have proven effective in modeling and compensating for delays, thereby improving the accuracy of sim2real transfers. The supergraph method addressed the challenges of parallelizing asynchronous and hierarchical systems, reducing training times while maintaining high simulation fidelity and enabling the efficient estimation of system dynamics and delays from real-world data. Lastly, this work has resulted in the creation of two open-source sim2real frameworks (EAGERx, REX) and has contributed to existing open-source deep learning frameworks (OpenDR [133]), all made available to the research community to support advancements in robotic learning and sim2real transfer.

6.2 Discussion and Outlook

As the complexity of robotic tasks continues to increase, the field of robot learning is shifting its approach to meet these challenges. Rather than relying purely on end-to-end models, the focus is now on developing a pipeline of specialized components, each designed to address specific aspects of a larger task. This trend parallels the rise of foundation models in natural language processing [134, 135] and computer vision [136], where large, versatile models are trained on extensive datasets to provide generalized capabilities across many applications. In robotics, similar foundation models are emerging, with efforts like CLIPort [52], RT-1 [137], and SayCan [138], offering broad capabilities but requiring integration with other specialized modules (e.g., for control) to achieve complex, task-specific behaviors. Unlike NLP or computer vision, however, robotics brings unique real-time challenges. Robots must respond promptly to external stimuli to operate effectively, which places stringent demands on latency and response times. These foundation models are often computationally heavy, which means inference is either slow when performed locally or requires offloading computations to remote data centers, both of which introduce latency. This highlights the growing need for efficient modeling and management of delays between components to ensure robust real-time performance. The contributions of this thesis are well-positioned to address these needs by providing structured simulation tools that effectively balance flexibility, accuracy, and speed. By accounting for asynchronous interactions and modeling latency, this work paves the way for integrating foundation models into robotic systems in a manner that maintains real-time performance requirements and enhances system robustness.

Looking ahead, this thesis suggests a more integrated systems approach to robotic learning, where future research could focus on co-designing learning algorithms and their environments to further optimize the learning process. While this work lays the foundation for modular representations in graph-based simulations and demonstrates how, for example, delay compensation can be decoupled from the learning task, additional research is necessary to explore novel ways of decomposing robotic learning tasks into manageable subproblems. Tackling these subproblems individually may lead to more effective and scalable learning outcomes. This challenges the conventional agent-environment paradigm [23], which typically views the environment as a fixed entity and treats the agent as a solitary solver of all tasks. In contrast, future research could explore a more fluid definition of agent-environment interaction, where different subproblems necessitate tailored environmental reconfigurations. By moving beyond rigid, end-to-end training of single agents, researchers could investigate adaptive environments that evolve in response to agent progress or integrate pretrained agents or specialized learning algorithms to facilitate the learning process.

In this context, one promising direction for future work involves fundamentally splitting robotic learning tasks into separate estimation and control subtasks. This approach draws inspiration from the well-established duality between estimation and control in the Linear Quadratic Gaussian (LQG) framework [139], where the two problems are handled independently while applying a common solution tactic. A potential research avenue could be training control policies on full state information—simplifying the learning process—while separately training estimator policies that infer hidden states based on sensor observations. In theory, the estimator could be optimized with the same objective as the

control policy since accurate estimation of the relevant hidden state enables the control policy to select actions that maximize the expected return. However, introducing auxiliary objectives could further incentivize the estimator to focus on reconstructing hidden states that are most critical for the control task. By decoupling the estimation and control tasks, the challenges of partial observability and the task itself can be addressed separately, potentially making the learning easier. Moreover, this approach underscores the potential for a flexible graph-based simulation framework, where the environment, during the learning of the estimator policy, incorporates the pre-trained control policy as part of the system's overall dynamics. This flexibility facilitates redefinitions of the environment that can be tailored to specific learning phases or subtasks, further illustrating the value of a modular framework for representing robotic systems.

An exciting extension of this idea could involve jointly training the control and estimator policies under complementary objectives. For example, the control policy could be incentivized to propose actions that are easier to estimate, while the estimator policy is optimized to predict the hidden states that are most relevant to the control task. This joint optimization framework could lead to more effective policies, particularly in environments characterized by high uncertainty and partial observability. In this regard, control and estimation can be seen as two cooperating agents in a single system, potentially well-suited for a multi-agent reinforcement learning approach [140].

Recent advancements also highlight the potential of integrating parallelized simulations in online planning strategies. For instance, parallelized simulation were used in a sampling-based planning that could solve complex robotic manipulation tasks without requiring offline training [79]. In this approach, actions are sampled from a prior model, and the optimal action sequence is determined online. However, a key challenge remains in sample efficiency, as large state spaces must be explored to identify optimal actions. Moreover, the limited planning horizon inherent in these methods can restrict exploration, often leading to suboptimal performance.

To address these challenges, future work could explore the integration of off-policy reinforcement learning with GPU-accelerated simulators in an online sampling-based planner. In this approach, a policy trained offline could serve as the proposal model for the sampling-based planner, while the value function could be used to bootstrap the planner at the end of the planning horizon, improving both exploration and sample efficiency.

Additionally, using GPU-accelerated simulators as the dynamics models in sampling-based estimators, such as particle filters [3], presents another promising research direction. This approach could effectively address partial observability challenges during deployment by decoupling state estimation from task learning, much like how delay compensation was separated from task learning in Chapter 4.

Bibliography

References

[1] Shimon Y Nof. Automation: What it means to us around the world. *Springer handbook of automation*, pages 13–52, 2009.

[2] Zhongsheng Hou, Ronghu Chi, and Huijun Gao. An overview of dynamic-linearization-based data-driven control and applications. *IEEE Trans. on Industrial Electronics*, 64(5):4076–4090, 2016.

[3] Dan Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[4] Michael A Henson and Dale E Seborg. Critique of exact linearization strategies for process control. *Journal of Process Control*, 1(3):122–139, 1991.

[5] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[6] Jan M Maciejowski and Mihai Huzmezan. Predictive control. In *Robust Flight Control: A Design Challenge*, pages 125–134. Springer, 2007.

[7] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[8] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[9] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[10] Chen Tang, Ben Abbatematteo, Jiaheng Hu, Rohan Chandra, Roberto Martín-Martín, and Peter Stone. Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes. *arXiv preprint*, 2024.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[12] Hai Nguyen and Hung La. Review of deep reinforcement learning for robot manipulation. In *Proc. of the IEEE Intl. Conf. on Robotic Computing (IRC)*, pages 590–595. IEEE, 2019.

[13] Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

[14] Luiza Caetano Garaffa, Maik Basso, Andrea Aparecida Konzen, and Edison Pignaton de Freitas. Reinforcement learning for mobile robotics exploration: A survey. *IEEE Trans. on Neural Networks and Learning Systems*, 34(8):3796–3810, 2021.

[15] NVIDIA. NVIDIA PhysX, 2020.

[16] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable Programming for Physical Simulation. In *Proc. of the Int. Conf. on Learning Representations (ICLR)*, 2020.

[17] C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax-A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *Proc. of the Advances in Neural Information Processing Systems (NIPS)*, 2021.

[18] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 5026–5033, 2012.

[19] Sebastian Höfer et al. Sim2Real in Robotics and Automation: Applications and Challenges. *IEEE trans. on Automation Science and Engineering*, 18(2):398–400, 2021.

[20] Matthias Mueller, Alexey Dosovitskiy, Bernard Ghanem, and Vladlen Koltun. Driving Policy Transfer via Modularity and Abstraction. In *Proc. of the Conf. Robot Learning (CoRL)*, pages 1–15. PMLR, 2018.

[21] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[22] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning. *Proc. of the Conf. Robot Learning (CoRL)*, 164:91–100, 2022.

[23] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint*, 2016.

[24] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. ROS: an open-source Robot Operating System. *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 3:5, 2009.

[25] Naoki Shibata. Efficient Evaluation Methods of Elementary Functions Suitable for SIMD Computation. *Computer science-Research and development*, 25:25–32, 2010.

[26] Fei Zhao, Yu Liu, Jian Wang, and Li Wang. Distributed model predictive longitudinal control for a connected autonomous vehicle platoon with dynamic information flow topology. In *Actuators*, volume 10, page 204. MDPI, 2021.

[27] Godwin Asaamoning, Paulo Mendes, Denis Rosário, and Eduardo Cerqueira. Drone swarms as networked control systems by integration of networking and computing. *Sensors*, 21(8):2642, 2021.

[28] Karol Gregor, Frederic Besse, Danilo Jimenez Rezende, Ivo Danihelka, and Daan Wierstra. Towards conceptual compression. *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, 29, 2016.

[29] Bas van der Heijden, Jelle Luijkx, Laura Ferranti, Jens Kober, and Robert Babuska. Engine Agnostic Graph Environments for Robotics (EAGERx): A Graph-Based Framework for Sim2real Robot Learning. *IEEE Robotics and Automation Magazine (RAM)*, pages 2–15, 2024.

[30] Jacob E Kooi and Robert Babuska. Inclined Quadrotor Landing using Deep Reinforcement Learning. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2361–2368. IEEE, 2021.

[31] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint*, 2018.

[32] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *Intl. Journal of Robotics Research (IJRR)*, 40(4-5):698–721, 2021.

[33] David Kortenkamp, Reid Simmons, and Davide Brugali. Robotic systems architectures and programming. *Springer Verlag*, pages 283–306, 2016.

[34] Doina Precup. *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

[35] Matteo Lucchi, Friedemann Zindler, Stephan Mühlbacher-Karrer, and Horst Pichler. robo-gym–an open source toolkit for distributed deep reinforcement learning on real and simulated robots. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 5364–5371. IEEE, 2020.

[36] Nestor Gonzalez Lopez, Yue Leire Erro Nuin, Elias Barba Moral, Lander Usategui San Juan, Alejandro Solano Rueda, Víctor Mayoral Vilches, and Risto Kojcev. gym-gazebo2, a toolkit for reinforcement learning using ROS 2 and Gazebo. *arXiv preprint*, 2019.

[37] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint*, 2016.

[38] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154 vol.3, 2004.

[39] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. Orbit: A Unified Simulation Framework for Interactive Robot Learning Environments. *IEEE Robotics and Automation Letters (RA-L)*, pages 1–8, 2023.

[40] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics. `https://drake.mit.edu`, 2019.

[41] NVIDIA. NVIDIA Isaac Sim. `https://developer.nvidia.com/isaac-sim`, 2022.

[42] Bas van der Heijden, Jelle Lujikx, Laura Ferranti, Jens Kober, and Robert Babuska. Supplementary video material of "EAGERx: Graph-Based Framework for Sim2real Robot Learning". `https://youtu.be/D0CQNnTT010?si=NLNaWyJermhLqH40`, June 2024.

[43] Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

[44] George Xylomenos and George C Polyzos. TCP and UDP performance over a wireless LAN. In *Proc. of the IEEE Conf. on Computer Communications (INFOCOM)*, volume 2, pages 439–446. IEEE, 1999.

[45] Erik Derner, Jiri Kubalik, Nicola Ancona, and Robert Babuska. Constructing parsimonious analytic models for dynamic systems via symbolic regression. *Applied Soft Computing*, 94:106432, 2020.

[46] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications. *arXiv preprint*, 2019.

[47] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal on Machine Learning Research (JMLR)*, 2021.

[48] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[49] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017.

[50] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *Journal on Machine Learning Research (JMLR)*, 2022.

[51] Alexandre Angleraud, Akif Ekrekli, Kulunu Samarawickrama, Gaurang Sharma, and Roel Pieters. Sensor-based human–robot collaboration for industrial tasks. *Robotics and Computer-Integrated Manufacturing*, 86:102663, 2024.

[52] Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Cliport: What and where pathways for robotic manipulation. *Proc. of the Conf. Robot Learning (CoRL)*, pages 894–906, 2022.

[53] Jelle Luijkx, Zlatan Ajanović, Laura Ferranti, and Jens Kober. PARTNR: Pick and place Ambiguity Resolving by Trustworthy iNteractive leaRning. In *5th NeurIPS Robot Learning Workshop: Trustworthy Robotics*, 2022.

[54] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, pages 28492–28518. PMLR, 2023.

[55] Fu Zhang and Murali Yeddanapudi. Modeling and simulation of time-varying delays. In *Proc. of the Symposium on Theory of Modeling and Simulation*, pages 1–8, 2012.

[56] Rishabh Bajpai and Deepak Joshi. Movenet: A deep neural network for joint profile prediction across variable walking speeds and slopes. *IEEE Instrumentation & Measurement Magazine*, 70:1–11, 2021.

[57] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA, 1973.

[58] Henrik Larsen, Gijs van der Hoorn, and Andrzej Wąsowski. *Reactive Programming of Robots with RxROS*, volume 6 of *Studies in Computational Intelligence*, pages 55–83. Springer Verlag, 2021.

[59] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Verlag, 2013.

[60] David G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, 1990.

[61] Thomas E Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE trans. on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[62] Claudius Ptolemaeus. *System design, modeling, and simulation: using Ptolemy II*, volume 1. Ptolemy. org Berkeley, 2014.

[63] Bas van der Heijden, Laura Ferranti, Jens Kober, and Robert Babuska. Efficient Parallelized Simulation of Cyber-Physical Systems. *Trans. on Machine Learning Research (TMLR)*, 2024. Reproducibility Certification.

[64] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The Impact of Control Technology*, 12(1):161–166, 2011.

[65] Avi Singh, Larry Yang, Kristian Hartikainen, Chelsea Finn, and Sergey Levine. End-to-end robotic reinforcement learning without reward engineering. *arXiv preprint*, 2019.

[66] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I Jordan, and Ion Stoica. Real-time machine learning: The missing pieces. In *Proc. of the Workshop on Hot Topics n Operating Systems (HTOS)*, pages 106–110, 2017.

[67] James Trimble. *Partitioning algorithms for induced subgraph problems*. PhD thesis, University of Glasgow, 2023.

[68] Bas van der Heijden, Laura Ferranti, Jens Kober, and Robert Babuska. Supplementary video material of "Efficient Parallelized Simulation of Cyber-Physical Systems". `https://youtu.be/I-1asHKBX6o?si=HtASYaj1ttlSxj59`, May 2024.

[69] Ryan Taylor and Xiaoming Li. Software-Based Branch Predication for AMD GPUs. *SIGARCH Comput. Archit. News*, 38(4):66–72, 2011.

[70] Horst Bunke, Xiaoyi Jiang, and Abraham Kandel. On the minimum common supergraph of two graphs. *Computing*, 65:13–25, 2000.

[71] Horst Bunke, Pasquale Foggia, Corrado Guidobaldi, and Mario Vento. Graph clustering using the weighted minimum common supergraph. *Graph Based Representations in Pattern Recognition*, pages 235–246, 2003.

[72] Xing Liu, Hansong Xu, Weixian Liao, and Wei Yu. Reinforcement learning for cyber-physical systems. In *2019 IEEE International Conference on Industrial Internet (ICII)*, pages 318–327. IEEE, 2019.

[73] Xin Lou, Cuong Tran, David KY Yau, Rui Tan, Hongwei Ng, Tom Zhengjia Fu, and Marianne Winslett. Learning-based time delay attack characterization for cyber-physical systems. In *2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, pages 1–6. IEEE, 2019.

[74] Andrés A Peters, Richard H Middleton, and Oliver Mason. Leader tracking in homogeneous vehicle platoons with broadcast delays. *Automatica*, 50(1):64–74, 2014.

[75] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, pages 1861–1870. PMLR, 2018.

[76] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[77] Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Tingnan Zhang, Jie Tan, and Vikas Sindhwani. Data efficient reinforcement learning for legged robots. *Proc. of the Conf. Robot Learning (CoRL)*, pages 1–10, 2020.

[78] Reuven Y Rubinstein and Dirk P Kroese. The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning. *Springer*, 133, 2004.

[79] Corrado Pezzato, Chadi Salmi, Max Spahn, Elia Trevisan, Javier Alonso-Mora, and Carlos Hernandez Corbato. Sampling-based model predictive control leveraging parallelizable physics simulations. *arXiv preprint*, 2023.

[80] Dongyao Jia, Kejie Lu, Jianping Wang, Xiang Zhang, and Xuemin Shen. A survey on platoon-based vehicular cyber-physical systems. *IEEE Communications Surveys & tutorials*, 18(1):263–284, 2015.

[81] Enrico Bibbona, Gianna Panfilo, and Patrizia Tavella. The Ornstein–Uhlenbeck process as a model of a low pass filtered white noise. *Metrologia*, 45(6):S117, 2008.

[82] Yann Bouteiller, Simon Ramstedt, Giovanni Beltrame, Christopher Pal, and Jonathan Binas. Reinforcement learning with random delays. In *Proc. of the Int. Conf. on Learning Representations (ICLR)*, 2021.

[83] James Trimble. Induced universal graphs for families of small graphs. *arXiv preprint*, 2021.

[84] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. *Proc. of the Intl. Conf. on Artificial Intelligence (IJCAI)*, pages 712–719, 2017.

[85] James J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12, 1982.

[86] Bas van der Heijden, Jens Kober, Robert Babuska, and Laura Ferranti. REX: GPU-accelerated sim2real framework with delay and dynamics estimation. *Trans. on Machine Learning Research (TMLR)*, 2024.

[87] David Hoeller, Nikita Rudin, Dhionis Sako, and Marco Hutter. Anymal parkour: Learning agile navigation for quadrupedal robots. *Science Robotics*, 9(88):eadi7566, 2024.

[88] Norhan Mohsen Elocla, Mohamad Chehadeh, Igor Boiko, Sean Swei, and Yahya Zweiri. The Role of Time Delay in Sim2real Transfer of Reinforcement Learning for Unmanned Aerial Vehicles. In *Proc. of the Int. Conf. on Advanced Robotics (ICAR)*, pages 514–519. IEEE, 2023.

[89] Otto JM Smith. Closer control of loops with dead time. *Chemical engineering progress*, 53:217–219, 1957.

[90] Michael Sherback, Oliver Purwin, and Raffaello D'Andrea. Real-time motion planning and control in the 2005 cornell robocup system. In *Robot Motion and Control: Recent Developments*, pages 245–263. Springer, 2006.

[91] Frederico Augugliaro, Sergei Lupashin, Michael Hamer, Cason Male, Markus Hehn, Mark W Mueller, Jan Sebastian Willmann, Fabio Gramazio, Matthias Kohler, and Raffaello D'Andrea. The flight assembled architecture installation: Cooperative construction with flying machines. *IEEE Control Systems Magazine*, 34(4):46–64, 2014.

[92] Heinz Unbehauen and GP Rao. Continuous-time approaches to system identification—a survey. *Automatica*, 26(1):23–35, 1990.

[93] Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022.

[94] Robert Tjarko Lange. evosax: Jax-based evolution strategies. *arXiv preprint*, 2022.

[95] Yujin Tang, Yingtao Tian, and David Ha. EvoJAX: Hardware-Accelerated Neuroevolution. *arXiv preprint*, 2022.

[96] Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation. *Proc. of the Advances in Neural Information Processing Systems (NIPS)*, 35:16455–16468, 2022.

[97] Lennart Ljung. System identification. In *Signal analysis and prediction*, pages 163–173. Springer, 1998.

[98] Peter Van Overschee and BL0888 De Moor. *Subspace identification for linear systems: Theory—Implementation—Applications*. Springer Science & Business Media, 2012.

[99] Oliver Nelles. *Nonlinear dynamic system identification*. Springer, 2020.

[100] Quentin Le Lidec, Igor Kalevatykh, Ivan Laptev, Cordelia Schmid, and Justin Carpentier. Differentiable simulation for physical system identification. *IEEE Robotics and Automation Letters (RA-L)*, 6(2):3413–3420, 2021.

[101] Eric Heiden, Christopher E Denniston, David Millard, Fabio Ramos, and Gaurav S Sukhatme. Probabilistic inference of simulation parameters via parallel differentiable simulation. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, pages 3638–3645. IEEE, 2022.

[102] Ken Caluwaerts, Atil Iscen, J Chase Kew, Wenhao Yu, Tingnan Zhang, Daniel Freeman, Kuang-Huei Lee, Lisa Lee, Stefano Saliceti, Vincent Zhuang, et al. Barkour: Benchmarking animal-level agility with quadruped robots. *arXiv preprint*, 2023.

[103] Nikolaus Hansen. The CMA evolution strategy: a comparing review. *Towards a new evolutionary computation: Advances in the estimation of distribution algorithms*, pages 75–102, 2006.

[104] Erik Schuitema, Lucian Busoniu, Robert Babuska, and Pieter Jonker. Control delay in reinforcement learning for real-time dynamic systems: A memoryless approach. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 3226–3231. IEEE, 2010.

[105] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, and A. Handa. Isaac Gym: High Performance GPU Based Physics Simulation For Robot Learning. In *Proc. of the Advances in Neural Information Processing Systems (NIPS)*, 2021.

[106] Simon J Julier and Jeffrey K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[107] Rudolph Van Der Merwe. *Sigma-point Kalman filters for probabilistic inference in dynamic state-space models*. Oregon Health & Science University, 2004.

[108] Julian Förster. System identification of the crazyflie 2.0 nano quadrocopter. *B.S. thesis at ETH Zurich*, 2015.

[109] Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. *ICLR Blog Track*, 2023.

[110] Bas van der Heijden, Laura Ferranti, Jens Kober, and Robert Babuska. DeepKoCo: Efficient latent planning with a task-relevant Koopman representation. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 183–189. IEEE, 2021.

[111] Amy Zhang, Rowan McAllister, Roberto Calandra, Yarin Gal, and Sergey Levine. Learning invariant representations for reinforcement learning without reconstruction. *arXiv preprint*, 2020.

[112] Eurika Kaiser, J Nathan Kutz, and Steven L Brunton. Data-driven discovery of Koopman eigenfunctions for control. *Machine Learning: Science and Technology*, 2(3):035023, 2021.

[113] Hassan Arbabi, Milan Korda, and Igor Mezić. A data-driven koopman model predictive control framework for nonlinear partial differential equations. In *Proc. of the IEEE Conf. on Decision and Control (CDC)*, pages 6409–6414. IEEE, 2018.

[114] Giorgos Mamakoukas, Maria Castano, Xiaobo Tan, and Todd Murphey. Local Koopman operators for data-driven control of robotic systems. In *Proc. of Robotics: Science and Systems (RSS)*, 2019.

[115] Daniel Bruder, Brent Gillespie, C David Remy, and Ram Vasudevan. Modeling and Control of Soft Robots Using the Koopman Operator and Model Predictive Control. In *Proc. of Robotics: Science and Systems (RSS)*, 2019.

[116] Yunzhu Li, Hao He, Jiajun Wu, Dina Katabi, and Antonio Torralba. Learning Compositional Koopman Operators for Model-Based Control. In *Proc. of the Int. Conf. on Learning Representations (ICLR)*, 2019.

[117] Milan Korda and Igor Mezic. Optimal construction of Koopman eigenfunctions for prediction and control. *IEEE Trans. on Automatic Control*, 65(12):5114–5129, 2020.

[118] Carles Gelada, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G Bellemare. Deepmdp: Learning continuous latent space models for representation learning. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, pages 2170–2179. PMLR, 2019.

[119] Milan Korda and Igor Mezic. Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control. *Automatica*, 93:149–160, 2018.

[120] Steven L Brunton, Bingni W Brunton, Joshua L Proctor, and J Nathan Kutz. Koopman invariant subspaces and finite linear representations of nonlinear dynamical systems for control. *PLoS one*, 11(2):e0150171, 2016.

[121] Matthew O Williams, Ioannis G Kevrekidis, and Clarence W Rowley. A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25:1307–1346, 2015.

[122] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Proc. of the Advances in Neural Information Processing Systems (NIPS)*, 2015.

[123] Ershad Banijamali, Rui Shu, Hung Bui, Ali Ghodsi, et al. Robust locally-linear controllable embedding. In *Proc. of the Intl. Conf. on Artificial Intelligence and Statistics (AISTATS)*, pages 1751–1759. PMLR, 2018.

[124] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, pages 2555–2565. PMLR, 2019.

[125] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *Proc. of the Advances in Neural Information Processing Systems (NIPS)*, volume 30, 2017.

[126] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[127] Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Deep learning for universal linear embeddings of nonlinear dynamics. *Nature Communications*, 9(1):4950, 2018.

[128] Diederik P Kingma and Jimmy Lei Ba. Adam: A method for stochastic gradient descent. In *Proc. of the Int. Conf. on Learning Representations (ICLR)*, pages 1–15. ICLR US., 2015.

[129] Insu Chang and Joseph Bentsman. Constrained discrete-time state-dependent Riccati equation technique: A model predictive control approach. In *Proc. of the IEEE Conf. on Decision and Control (CDC)*, pages 5125–5130. IEEE, 2013.

[130] Istvan Szita and Andras Lorincz. Learning Tetris using the noisy cross-entropy method. *Neural Computation*, 18(12):2936–2941, 2006.

[131] Bas van der Heijden, Laura Ferranti, Jens Kober, and Robert Babuska. Supplementary video material of "DeepKoCo: Efficient latent planning with a task-relevant Koopman representation". https://youtu.be/75lOuQyHBmQ, July 2021.

[132] David Ha and Jurgen Schmidhuber. World models. *arXiv preprint*, 2018.

[133] Nikolaos Passalis, Stefania Pedrazzi, Robert Babuska, Wolfram Burgard, Daniel Dias, Francesco Ferro, Moncef Gabbouj, Ole Green, Alexandros Iosifidis, Erdal Kayacan, et al. Opendr: An open toolkit for enabling high performance, low footprint deep learning for robotics. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 12479–12484. IEEE, 2022.

[134] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[135] Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint*, 2018.

[136] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *Proc. of the Int. Conf. on Machine Learning (ICML)*, pages 8821–8831. PMLR, 2021.

[137] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint*, 2022.

[138] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint*, 2022.

[139] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *ASME Trans. Journal of Basic Engineering*, 1960.

[140] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control*, pages 321–384, 2021.

Acknowledgments

Twelve years at TU Delft—what a ride. Choosing Delft for my studies was hands down the best decision I've made, setting me up to do what I love. A big thanks to the institute for the inspiration and the launchpad for my career.

First and foremost, I want to thank my supervisors, Laura Ferranti, Jens Kober, and Robert Babuska. Laura, your guidance, support, and unwavering enthusiasm for my work gave me confidence when I needed it. Having someone so invested in my success made all the difference. Jens, your ability to absorb complex ideas with minimal explanation and immediately spot weaknesses (which I may not have initially acknowledged) was uncanny. More often than not, you were right—annoyingly so, and the quality of this work is undeniably better because of it. Robert, our technical discussions always ran longer than planned, but I never left without fresh ideas and renewed excitement. Your Friday night MATLAB plots, following up on an afternoon chat, were next-level dedication. Not many professors still code, but mine does!

To Aske, Herke, Abhinav, and Bart, thank you for kindly agreeing to be part of my doctoral committee, taking the time to review my dissertation, and providing detailed feedback.

To all my colleagues in the research groups of Laura and Jens, and the rest of the Cognitive Robotics department—thank you for the engaging discussions. A special shout-out to Jelle, my partner in crime on the OpenDR project. Our trips to Thessaloniki and the Outer Banks won't be forgotten. Best of luck finishing your PhD!

Alexander and Gijs, supervising you was a privilege. Our discussions were some of the most enjoyable parts of my PhD—thanks for keeping things interesting.

To my parents, Lex and Hilde, for their unwavering support and for instilling in me what my grandpa always said: "kennis kunnen ze niet van je afnemen, geld wel."

Finally, to Pien, for her patience, support, and for giving me the space to finish, even when it meant weekends of me locked in work mode. Looking forward to the many weekends that are actually free—together!

Bas
Amsterdam, February 2025

About the Author

Douwe Sebastiaan van der Heijden

27 September 1994 **Born in Vleuten-De Meern, The Netherlands**.

Education

2006 – 2012 **VWO (*cum laude*), Leidsche Rijn College**.

2012 – 2016 **B.Sc., Mechanical Engineering, TU Delft**.
Thesis: *Microwave Induced Plasma Gasification of Sewage Sludge*

2017 – 2019 **M.Sc., Systems & Control (*cum laude*), TU Delft**.
Thesis: *Iterative Bias Estimation for an Ultra-Wideband Localization System.*

2020 – 2025 **Ph.D. Candidate, TU Delft**.
Thesis: *Designing Simulators for Robot Learning.*
(co-)promotors: dr. L. Ferranti, dr. J. Kober, and prof. R. Babuska.

Visiting Scholar

2018 – 2019 **Visiting Researcher, ETH Zurich** (10 months).
Research Group of prof. R. D'Andrea.

List of Publications

Journal

1. **B. van der Heijden**[*], J. Luijkx[*], L. Ferranti, J. Kober, and R. Babuska. "Engine Agnostic Graph Environments for Robotics (EAGERx): A Graph-Based Framework for Sim2real Robot Learning", IEEE Robotics and Automation Magazine (RAM), 2024.

2. **B. van der Heijden**, L. Ferranti, J. Kober, and R. Babuska. "Efficient Parallelized Simulation of Cyber-Physical Systems", Transactions on Machine Learning Research (TMLR), 2024.

3. **B. van der Heijden**, J. Kober, R. Babuska, and L. Ferranti. "REX: GPU-Accelerated Sim2Real Framework with Delay and Dynamics Estimation", Transactions on Machine Learning Research (TMLR), 2025.

Conference

4. F. Sibona, J. Luijkx, **B. van der Heijden**, L. Ferranti, and M. Indri. "EValueAction: a proposal for policy evaluation in simulation to support interactive imitation learning", Proc. of the IEEE Intl. Conf. on Industrial Informatics (INDIN), 2023.

5. **B. van der Heijden**, L. Ferranti, J. Kober, and R. Babuska. "DeepKoCo: Efficient latent planning with a task-relevant Koopman representation", Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2021.

6. **B. van der Heijden**, A. Ledergerber, R. Gill, and R. D'Andrea. "Iterative Bias Estimation for an Ultra-Wideband Localization System", IFAC World Congress, 2021.

Workshop

7. A. Keijzer[*], **B. van der Heijden**[*], and J. Kober. "Prioritizing States with Action Sensitive Return in Experience Replay", European Workshop on Reinforcement Learning (EWRL), 2023.

Included in this thesis.
* Equal contribution.