

# UC Berkeley

## UC Berkeley Electronic Theses and Dissertations

### Title

From Simulation to Reality: An End-to-End Pipeline for Reproducible Robot Learning

### Permalink

<https://escholarship.org/uc/item/5p94x2md>

### ISBN

9798189271571

### Author

Zakka, Kevin

### Publication Date

2026-05-01

Peer reviewed|Thesis/dissertation

From Simulation to Reality: An End-to-End Pipeline for Reproducible Robot Learning

By

Kevin Zakka

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Pieter Abbeel, Chair

Professor Sergey Levine

Professor Koushil Sreenath

Spring 2026

From Simulation to Reality: An End-to-End Pipeline for Reproducible Robot Learning

Copyright 2026  
by  
Kevin Zakka

## Abstract

From Simulation to Reality: An End-to-End Pipeline for Reproducible Robot Learning

by

Kevin Zakka

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Pieter Abbeel, Chair

Simulation-based reinforcement learning has produced many of the most capable robot controllers reported in recent years, with policies trained in simulation and deployed zero-shot on legged, dexterous, and aerial systems. Reproducing this methodology on a new robot, however, is hard: the work spans an end-to-end pipeline whose stages depend on infrastructure of varying maturity and openness, and the closed or fragmented pieces of that infrastructure are the bottleneck more often than the algorithms running on top.

This thesis develops the open infrastructure that each stage of the pipeline needs. For *grounding*, where a faithful simulated model of the hardware must exist before training is meaningful, we build a curated library of robot descriptions for an open physics engine and a system identification toolbox that calibrates those models against specific hardware. For *training*, where the iteration loop between rewards and resulting policies sets project velocity, we contribute open, GPU-accelerated reinforcement learning infrastructure ranging from a minimal-abstraction library for rapid prototyping to a manager-based framework for codebases that span multiple robots and task families, on which a single sim-to-real recipe transfers across six robot platforms within weeks. For *task priors*, the human-derived guidance needed when reward design alone is insufficient, we show that expert fingering annotations make dexterous bimanual piano playing learnable from sheet music, and we develop the inverse-kinematics infrastructure that converts human motion-capture, teleoperation, and video data into robot reference trajectories.

The artifacts produced across these stages have been taken up by external research groups, robot manufacturers, and other simulator projects, and now serve as the substrate for downstream work on humanoid motion imitation, video-based piano playing, and teleoperated whole-body control. Across the chapters, the open, end-to-end infrastructure documented here is what allows the sim-to-real methodology to be reproduced across labs.



# Contents

|                                                     |            |
|-----------------------------------------------------|------------|
| <b>Contents</b>                                     | <b>i</b>   |
| <b>List of Figures</b>                              | <b>iii</b> |
| <b>List of Tables</b>                               | <b>vi</b>  |
| <b>1 Introduction</b>                               | <b>1</b>   |
| <b>I Grounding</b>                                  | <b>5</b>   |
| <b>2 MuJoCo Menagerie</b>                           | <b>6</b>   |
| 2.1 The Digital Twin Problem . . . . .              | 6          |
| 2.2 Design Principles . . . . .                     | 7          |
| 2.3 Breadth of Coverage . . . . .                   | 8          |
| 2.4 Adoption as Evidence . . . . .                  | 9          |
| 2.5 Discussion . . . . .                            | 11         |
| <b>3 mujoco.sysid</b>                               | <b>12</b>  |
| 3.1 Introduction . . . . .                          | 12         |
| 3.2 Preliminaries . . . . .                         | 13         |
| 3.3 The <code>mujoco.sysid</code> Toolbox . . . . . | 14         |
| 3.4 Related Work . . . . .                          | 16         |
| 3.5 Experimental Validation . . . . .               | 18         |
| 3.6 Limitations . . . . .                           | 22         |
| 3.7 Recommendations . . . . .                       | 23         |
| 3.8 Conclusion . . . . .                            | 24         |
| <b>II Training Infrastructure</b>                   | <b>25</b>  |
| <b>4 MuJoCo Playground</b>                          | <b>26</b>  |
| 4.1 Introduction . . . . .                          | 26         |

|          |                                        |           |
|----------|----------------------------------------|-----------|
| 4.2      | Environments . . . . .                 | 27        |
| 4.3      | Batch Rendering with Madrona . . . . . | 28        |
| 4.4      | Results . . . . .                      | 29        |
| 4.5      | Related Work . . . . .                 | 37        |
| 4.6      | Limitations . . . . .                  | 38        |
| 4.7      | Conclusion . . . . .                   | 39        |
| <b>5</b> | <b>mjlab</b>                           | <b>40</b> |
| 5.1      | Introduction . . . . .                 | 40        |
| 5.2      | Design Philosophy and Scope . . . . .  | 41        |
| 5.3      | Architecture . . . . .                 | 41        |
| 5.4      | Components . . . . .                   | 42        |
| 5.5      | Manager-Based API . . . . .            | 44        |
| 5.6      | Tasks . . . . .                        | 46        |
| 5.7      | Software Design . . . . .              | 47        |
| 5.8      | Adoption . . . . .                     | 48        |
|          | <b>III Task Priors</b>                 | <b>49</b> |
| <b>6</b> | <b>RoboPianist</b>                     | <b>50</b> |
| 6.1      | Introduction . . . . .                 | 50        |
| 6.2      | Related Work . . . . .                 | 51        |
| 6.3      | Experimental Setup . . . . .           | 52        |
| 6.4      | ROBOPIANIST System Design . . . . .    | 54        |
| 6.5      | Results . . . . .                      | 56        |
| 6.6      | Discussion . . . . .                   | 60        |
| <b>7</b> | <b>mink</b>                            | <b>61</b> |
| 7.1      | Motivation . . . . .                   | 61        |
| 7.2      | The QP . . . . .                       | 62        |
| 7.3      | Tasks . . . . .                        | 62        |
| 7.4      | Limits . . . . .                       | 63        |
| 7.5      | Implementation . . . . .               | 64        |
| 7.6      | Applications and Adoption . . . . .    | 65        |
| 7.7      | Related Work . . . . .                 | 65        |
| 7.8      | Limitations and Discussion . . . . .   | 66        |
| <b>8</b> | <b>Conclusion</b>                      | <b>67</b> |
|          | <b>Bibliography</b>                    | <b>70</b> |

# List of Figures

|     |                                                                                                                                                                                                                                                                                                                                                                                                      |    |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1 | A selection of robot models from MuJoCo Menagerie, spanning the morphological categories listed in Table 2.1: manipulator arms, dexterous hands and grippers, humanoids, quadrupeds and bipeds, and mobile manipulators, aerial vehicles and biomechanical models. . . . .                                                                                                                           | 10 |
| 3.1 | The YAM six-DoF manipulator: a low-cost, fixed-base arm with quasi-direct-drive proprioceptive actuators. Joint torques are estimated from motor currents rather than measured by dedicated torque sensors. . . . .                                                                                                                                                                                  | 18 |
| 3.2 | Position predictions (rad) on a held-out trajectory generated by random Cartesian waypoints. The identified model (green) tracks the recorded data (black) substantially better than the nominal model (blue) across all six joints. . . . .                                                                                                                                                         | 19 |
| 3.3 | The FPV quadrotor and identification fit: the optimizer aligns simulated angular velocities and thrust-induced accelerations with the motion-capture recording from a 6 s free-flight segment. . . . .                                                                                                                                                                                               | 21 |
| 3.4 | Position and angular velocity during an open-loop 360-degree flip: simulation vs. real. Both use the same control sequence; the open-loop trajectory was designed in simulation using the identified model. . . . .                                                                                                                                                                                  | 22 |
| 4.1 | A preview of locomotion and manipulation environments available in MuJoCo Playground.                                                                                                                                                                                                                                                                                                                | 27 |
| 4.2 | Sample renders from the Madrona batch renderer for the Panda and Aloha environments. Left-most images are the original environments. The remaining images highlight the support for lighting, shadows, textures, and colors, including the ability to domain randomize these parameters during training. . . . .                                                                                     | 29 |
| 4.3 | Several DM Control Suite environments. . . . .                                                                                                                                                                                                                                                                                                                                                       | 30 |
| 4.4 | Footage from four of our deployed policies. a) Go1 joystick policy recovering from a kick while travelling at $\sim 2\text{m/s}$ , b) Berkeley humanoid joystick policy tracking an angular velocity command on a slippery surface. c) In-Hand Cube Reorientation transitioning between two target poses. d) Non-prehensile policy issuing torque commands to rotate a block by 180 degrees. . . . . | 31 |
| 4.5 | Training wallclock time for LeapCubeReorient on different GPU device topologies. 1x 4090 takes $\sim 2080$ (s) to train and 8x H100 takes $\sim 670$ (s) to train. All runs use the <i>same hyperparams</i> (e.g. 8192 num envs); we leave tuning hyperparams per topology as a future exercise. . . . .                                                                                             | 35 |

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |    |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 4.6 | Environment steps per second on the single-camera CartpoleBalance and PandaPick-CubeCartesian environments with pixel-based observations from our on-device renderer.                                                                                                                                                                                                                                                                                                                                                                                                                                                    | 36 |
| 4.7 | Reward curves for PPO trained with RSL-RL and brax on an RTX-4090 GPU for 3 seeds each on the Unitree Go1. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | 37 |
| 5.1 | Entities are composed into an MjSpec, compiled, and transferred to MuJoCo Warp for GPU simulation. The ManagerBasedRLEnv orchestrates the MDP; RSL-RL handles training. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                          | 42 |
| 5.2 | A composite terrain grid generated by <code>mjlab</code> . Sub-terrain patches include flat ground, stairs, sloped pyramids, heightfield noise, and wave patterns. Difficulty increases along one axis for curriculum-based training. . . . .                                                                                                                                                                                                                                                                                                                                                                            | 44 |
| 5.3 | The three robot morphologies shipped with <code>mjlab</code> : Unitree G1 humanoid (left), Unitree Go1 quadruped (center), and YAM robot arm (right). All models are adapted from MuJoCo Menagerie [12]. . . . .                                                                                                                                                                                                                                                                                                                                                                                                         | 46 |
| 6.1 | <b>RoboPianist</b> simulation featuring a full-size digital keyboard (A) with 88 piano keys modeled as linear springs (B). In the piano playing task, two (left and right) anthropomorphic Shadow hands (C) are tasked with playing a musical piece encoded as a trajectory of key presses (D). . . . .                                                                                                                                                                                                                                                                                                                  | 51 |
| 6.2 | <b>RoboPianist Design Considerations.</b> The F1 performance for 3 songs of increasing difficulty. From left to right as specified by the legend, each curve <b>inherits all MDP attributes of the curve before it</b> but makes one additional modification as described by its label. . . . .                                                                                                                                                                                                                                                                                                                          | 54 |
| 6.3 | Policies in the ROBOPIANIST environment displaying skilled piano behaviors such as (left) simultaneously controlling both hands to reach for notes on opposite ends of the keyboard, (middle) playing a chord with the left hand by precisely and simultaneously hitting a note triplet, and (right) playing a trill with the right hand, which involves rapidly alternating between two adjacent notes. . . . .                                                                                                                                                                                                         | 56 |
| 6.4 | <b>Multi-song training with RL.</b> We test the ability to learn multiple songs simultaneously. The training performance on the shared song, Für Elise, is shown on the left. We see adding more songs degrades performance on individual songs. On the right, we evaluate on the held out ROBOPIANIST-ETUDE-12 songs, suggesting the difficulty of generalization with multitask RL training. . . . .                                                                                                                                                                                                                   | 56 |
| 6.6 | Here we show the learning curves of BC, showing the interplay between the attempted number of training songs and model size. The song Für Elise, is in the training dataset for each experiment. We show the F1 score across Für Elise and ROBOPIANIST-ETUDE-12. We see that for a small number of songs, the model quickly overfits, suggesting that for the given model size the amount of data and diversity is lacking. We see that adding more songs results in a higher test F1, suggesting that the model is beginning to generalize. However, zero-shot test performance is far from that of the expert. . . . . | 58 |

|     |                                                                                                                                                                                                                                                                                                                                                                                     |    |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 6.5 | The F1 scores achieved by ROBOPIANIST (blue) and MPC (green) for each of the ROBOPIANIST-ETUDE-12 tasks. . . . .                                                                                                                                                                                                                                                                    | 58 |
| 6.7 | <b>Lookahead &amp; Control Timestep.</b> These parameters jointly modulate the foresight and control granularity of the agent. We find a control timestep of 0.05 offers optimal control granularity, and performance generally improves with the lookahead up to a point. . . . .                                                                                                  | 59 |
| 6.8 | <b>Discount factor.</b> We find the discount factor to have a high impact on F1, especially as the task complexity increases. While on simpler songs (Twinkle Twinkle) the discount factor does not have any visible effect, it is critical for faster-paced songs (Für Elise).The curves show that any discount in the range of 0.84-0.92 can be used with similar effect. . . . . | 59 |

# List of Tables

|     |                                                                                                                                                                                                                                                                                                              |    |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1 | Model counts by category in Menagerie. Some platforms span multiple categories (for example, a mobile base with two arms), in which case they are counted under their primary morphology. . . . .                                                                                                            | 9  |
| 3.1 | Maximum torque prediction errors (Nm) on a held-out trajectory: nominal versus identified model. Joint 6 is omitted from the torque comparison. . . . .                                                                                                                                                      | 20 |
| 3.2 | Gravity compensation performance: joint-space RMS error (rad) and end-effector position error (m) averaged over 50 configurations. Joints 1 and 5 have minimal gravitational loading due to axis alignment. . . . .                                                                                          | 20 |
| 4.1 | In-hand reorientation results on the LEAP hand over 10 trials, reporting the number of consecutive successful rotations before failure. The final two columns show the median and mean of the #Rotations metric. . . . .                                                                                     | 33 |
| 4.2 | Sim-to-real reorientation performance on the Franka Emika Panda robot, evaluated across 35 hardware trials. Each metric is reported as the median and mean (with a 95% confidence interval). The success rate is bolded to highlight final task performance. The training was done on 16x A100 GPUs. . . . . | 34 |
| 6.1 | The agent’s observation space. $L$ corresponds to the lookahead horizon. . . . .                                                                                                                                                                                                                             | 53 |
| 7.1 | Per-step IK timing on Apple M1 Max (median / p95, in $\mu s$ ). . . . .                                                                                                                                                                                                                                      | 64 |

## Acknowledgments

There are many people I'd like to thank for making my time at Berkeley so much fun.

First and foremost, I am deeply grateful to my advisor, Pieter. Pieter has given me complete freedom to pursue any research direction I was excited about and has been unwaveringly supportive at every step. He has believed in me and advocated for me whenever it mattered. I also want to warmly thank the rest of my committee, Professors Sergey Levine and Koushil Sreenath, for their support, advice, and help throughout the years.

I started my research journey at Google with Andy Zeng and Shuran Song. They took a chance on a kid who hadn't done any research, taught me much of the robotics I know today, and gave me a real sense of what it looks like to do good research and work hard at it. I'm grateful for all of it: the late nights and the dinners we shared together. During my PhD, I had the rare luck of close mentorship from Yuval Tassa at DeepMind. It's rare to find someone as passionate about their work as Yuval. He taught me to care about the details and to hold presentation and quality to a high standard; he was patient with all of my questions, always available to chat despite the time zone difference, and always excited to hear progress updates. Thank you, Yuval. You were a second advisor to me in every real sense.

I have been fortunate to work with an extraordinary group of collaborators on the projects in this thesis. I spent much of years three through five working closely with Qiayuan Liao, who taught me an enormous amount about hardware and whose work ethic was genuinely infectious. Levi Burner is one of the most knowledgeable and kind collaborators I've had, and I always felt lucky to be working alongside him. To the rest, Arthur Allshire, Carmelo Sferrazza, Mustafa Haiderbhai, Pete Florence, Sumeet Singh, Taylor Howell, and Jason Peng: thank you for your time and ideas.

During my research internships at Google, I had the pleasure of working with the simulation team: Nimrod Gileadi, Saran Tunyasuvunakool, Erik Frey, and Baruch Tabanpour. At Boston Dynamics, I learned a lot from Robin Deits, Lucas Manuelli, Alphonsus Adu-Bredu, Pat Marion, and Scott Kuindersma, who deepened my appreciation for what rigorous robotics engineering looks like in practice.

I am so lucky to have made the friendships I did during my time at Berkeley. To Laura Smith and Philipp Wu, my closest friends from these years, for doing everything alongside me, sharing the highs and lows, and making the whole thing immeasurably better. To Yen-Chen Lin and Jimmy Wu, who I met during my summers at Google and who made those internships some of the most fun times of my PhD. To Ameesh Shah, Aya Mouallem, Catherine Glossop, Cheng Chi, Chuer Pan, Colin Li, Dibya Ghosh, Gerard Touma, Homer Walke, Huy Ha, Ilya Kostrikov, Jessy Lin, Jonathan Tompson, Katie Kang, Kyle Stachowitz, William Chen, Seohong Park, Xiaomeng Xu, and Brent Yi, for sharing this stretch of life with me and being a constant source of joy and perspective.

To Berkeley, my home for these years. I came to love Artis on Fourth Street with Joe and the wonderful baristas, Peet's on Vine St, Yali's Cafe near my office, and the walks along the Marina as much as anything else from this time.

Finally, and most importantly, to my family. Laura, thank you for your love and support through all of this. My parents, Jacky and Roy Zakka, have given me everything and never asked for anything in return; I owe more than I can ever say. To my siblings, Sasha, Kim, Cyril, and Chloe, who I can always count on and who bring me more joy than they know.



# Chapter 1

## Introduction

One of the most successful patterns for developing robot controllers has been to train a policy in a physics simulator with reinforcement learning and deploy it zero-shot on hardware. The methodology has produced controllers for legged locomotion [1, 2], dexterous manipulation [3, 4], humanoid walking [5], and acrobatic drone flight [6], often with policies trained in minutes on a single GPU. The common element behind these results is not a single algorithm but a *pipeline*. It begins with a simulated model of the robot, calibrates that model against the specific physical platform, trains a policy in a suitable environment, optionally supplies priors that constrain the learning problem, and ends with a transfer step to hardware.

Despite the consistency of this methodology across published results, the pipeline itself is fragmented in practice. The dominant GPU-accelerated training stack, Isaac Gym [7] and its successor Isaac Lab [8], depended for years on the proprietary GPU implementation of the PhysX physics engine [9]; the lowest layer could not be inspected when a training run misbehaved. NVIDIA released the PhysX GPU kernel sources in April 2025, but the surrounding stack remains heavyweight and tied to a single vendor’s runtime.

The fragmentation extends beyond the training stack. Most laboratories construct robot models ad hoc, with quality and conventions that vary between groups. Researchers treat calibration against real hardware as a research problem of its own, with little shared tooling. Each new project rebuilds reward functions, training loops, and task definitions from scratch. Reproducing a published sim-to-real result, or porting one to a new robot, therefore becomes a substantial engineering exercise rather than a code-level adjustment.

This thesis develops the components of an open, end-to-end sim-to-real pipeline anchored on MuJoCo [10], a widely adopted open-source physics engine used across the robotics and reinforcement-learning communities. Each contribution is released as a standalone open-source library that any lab can adopt without re-engineering from scratch. The central question of the thesis is: *what does it take to make every stage of the sim-to-real pipeline a shared, reusable foundation rather than a per-project engineering effort?*

The thesis is organized in three parts, one per stage of the pipeline. For a lab or a student starting from a new robot, the goal is that the components developed across these parts compose into a deployed controller in days rather than the months that re-engineering

each stage from scratch demands.

## I. GROUNDING.

A policy trained against an inaccurate model can fail to transfer for reasons that have nothing to do with the training procedure itself. Part I addresses the two components of this stage: obtaining a high-quality nominal model of the robot, and calibrating that model against the specific hardware in the laboratory.

- i. Chapter 2 introduces *MuJoCo Menagerie*, a curated, open library of robot models for **MuJoCo** spanning 65 platforms across manipulators, dexterous hands, humanoids, quadrupeds, mobile manipulators, and drones. Menagerie supplies the starting models that every subsequent chapter draws on, and now functions as the default model source for **MuJoCo**-based robotics research, with manufacturers including Franka, Flexiv, I2RT, and PAL Robotics contributing and maintaining their own robots directly within the repository.
- ii. Chapter 3 introduces `mujoco.sysid`, a system identification toolbox that calibrates a Menagerie model against measurements from a specific physical robot. The toolbox ships natively with **MuJoCo**, casts identification as a box-constrained nonlinear least-squares problem, and reports confidence intervals that surface identifiability failures. In a representative experiment, identification reduces gravity-compensation drift by roughly three orders of magnitude relative to the nominal CAD-based model.

## II. TRAINING INFRASTRUCTURE.

With a grounded model in hand, the pipeline next requires an environment in which a policy can be trained. The dominant constraint at this stage is wallclock time: the loop from changing a reward function to evaluating the resulting policy governs how productively a sim-to-real project can be developed. Reducing this loop time while keeping the entire stack open and inspectable is the focus of Part II.

- i. Chapter 4 introduces *MuJoCo Playground*, a GPU-accelerated reinforcement learning library built on MJX, the JAX-based branch of **MuJoCo**. Playground favors minimal abstractions and monolithic environment definitions for rapid prototyping. A single deliberately simple sim-to-real recipe, applied consistently across the library, transfers across six robotic platforms within minutes to hours of training on commodity hardware.
- ii. Chapter 5 introduces `mjlab`, a composable reinforcement learning framework built on **MuJoCo Warp**, a more recent GPU implementation of **MuJoCo** built on NVIDIA Warp rather than JAX. Where Playground favors quick, monolithic prototyping, `mjlab` provides a manager-based, declarative API for codebases that span multiple robots and task families. Within six months of public release the

framework has accumulated over 2,000 GitHub stars and is in use at Boston Dynamics, Google, and PAL Robotics; the official Unitree reinforcement-learning environments for the Go2, G1, and H1 are built on it.

### III. TASK PRIORS.

Grounded simulation and fast training infrastructure are not always enough. When the action space is high-dimensional, or when many distinct behaviors score similarly under the reward, unconstrained reinforcement learning fails to discover useful behavior at all. The remedy is to inject human knowledge into the learning problem in a form that biases exploration toward the relevant region of behavior space. Part III studies two such priors.

- i. Chapter 6 presents *RoboPianist*, a benchmark and training system for dexterous bimanual piano playing. A task-specific prior in the form of fingering annotations from sheet music turns an otherwise intractable exploration problem into a tractable one. Follow-on work [11] extends the system to learn from internet piano video.
- ii. Chapter 7 generalizes from task-specific to broader human-motion priors. We introduce *mink*, a Python library for differential inverse kinematics built natively on *MuJoCo*, bridging human-derived motion data and robot reference trajectories for downstream learning pipelines. It is increasingly adopted for humanoid motion retargeting and teleoperation-based data collection for imitation learning.

## Contributions

This thesis contributes six open-source artifacts spanning the three stages of the sim-to-real pipeline:

- **MuJoCo Menagerie** (Chapter 2): 65 high-quality robot models for *MuJoCo* spanning six platform categories; now the default model source for *MuJoCo*-based research, with manufacturers contributing and maintaining models directly.
- **mujoco.sysid** (Chapter 3): calibrates a robot model’s physical parameters against real hardware measurements; ships natively with *MuJoCo*.
- **MuJoCo Playground** (Chapter 4): a single sim-to-real recipe that transfers across six platforms within minutes to hours of training on commodity hardware, with the full stack open and inspectable.
- **mjlab** (Chapter 5): 2,000+ GitHub stars in six months; in production at Boston Dynamics, Google, and PAL Robotics; the official Unitree RL environments for the Go2, G1, and H1 are built on it.

- **RoboPianist** (Chapter 6): fingering annotations from sheet music turn an otherwise intractable dexterous exploration problem into a tractable one.
- **mink** (Chapter 7): differential IK bridging human-derived motion data and robot reference trajectories; increasingly adopted for humanoid motion retargeting and teleoperation-based data collection for imitation learning.

## Publications

This thesis comprises research conducted in collaboration with many colleagues. The following lists the publications and releases that support each chapter:

Chapter 2: Kevin Zakka, Yuval Tassa, and the **MuJoCo** Menagerie Contributors. *MuJoCo Menagerie: A Collection of High-Quality Simulation Models for MuJoCo*. Software release, 2022 [12].

Chapter 4: Kevin Zakka, Baruch Tabanpour, Qiayuan Liao, Mustafa Haiderbhai, Samuel Holt, Jing Yuan Luo, Arthur Allshire, Erik Frey, Koushil Sreenath, Lueder A. Kahrs, Carmelo Sferrazza, Yuval Tassa, Pieter Abbeel. *MuJoCo Playground*. Robotics: Science and Systems (RSS) 2025, Outstanding Demo Paper Award [13].

Chapter 5: Kevin Zakka, Qiayuan Liao, Brent Yi, Louis Le Lay, Koushil Sreenath, Pieter Abbeel. *mjlab: A Lightweight Framework for GPU-Accelerated Robot Learning*. arXiv preprint, 2026 [14].

Chapter 6: Kevin Zakka, Philipp Wu, Laura Smith, Nimrod Gileadi, Taylor Howell, Xue Bin Peng, Sumeet Singh, Yuval Tassa, Pete Florence, Andy Zeng, Pieter Abbeel. *RoboPianist: Dexterous Piano Playing with Deep Reinforcement Learning*. Conference on Robot Learning (CoRL) 2023 [15].

Chapter 7: Kevin Zakka. *Mink: Python Inverse Kinematics Based on MuJoCo*. Software release, 2024 [16].

# Part I

## Grounding

## Chapter 2

# MuJoCo Menagerie: A Curated Library of Robot Models

Every stage of the sim-to-real pipeline runs against a simulated model of the robot, yet before 2022 no curated, high-quality source of MJCF models existed for MuJoCo. Groups working on different platforms each ported whatever robots they needed independently and inconsistently, treating models as per-project appendices rather than shared infrastructure. This chapter introduces *MuJoCo Menagerie* [12], a curated library of robot models for MuJoCo [10] that fills this gap; it is the source of robot models that every subsequent chapter draws on.

## 2.1 The Digital Twin Problem

A simulated model whose physical and geometric properties correspond to a specific piece of hardware is sometimes referred to as a *digital twin*. In MuJoCo, that model is an MJCF file, MuJoCo’s XML-based model format, specifying the robot’s kinematic tree, its dynamic properties (link inertias, joint and actuator parameters), and its collision geometry. The MJCF file is therefore a primary artifact of the sim-to-real workflow: an error in any of these terms changes the dynamics the policy learns to exploit, and a policy trained against such an error can fail to transfer for reasons that have nothing to do with the training algorithm itself.

Before Menagerie, the standard approach to obtaining robot models was ad hoc. A researcher who had not built the robot in-house would typically search GitHub for whatever model someone else had assembled, and would then evaluate, often only cursorily, whether it reflected the actual hardware, whether the license permitted use, and whether a collaborator trying to reproduce the results would find the same file. Models were treated as appendices to individual projects rather than as shared infrastructure: necessary for the work, but not worth engineering carefully or sharing as first-class artifacts. The downstream cost of this practice is that inconsistent models make results difficult to reproduce, and an inaccurate

model that propagates through a lab’s codebase contaminates everything that depends on it without leaving obvious traces.

Curated robot-description corpora did exist on the URDF side, primarily as per-manufacturer `*_description` ROS packages and the assets bundled with PyBullet [17], but they were federated across hundreds of repositories, varied in modeling quality and convention, and had no MJCF analogue.

## 2.2 Design Principles

The library is organized around four design principles that apply uniformly to every model in the collection.

**Provenance.** Geometry traces to an official manufacturer source: either CAD or URDF provided directly by the manufacturer, or descriptions hosted in the manufacturer’s own GitHub repository. Several manufacturers, including Franka, Flexiv, I2RT, and PAL Robotics, now contribute models of their own robots directly to the repository, in some cases releasing meshes under permissive licenses specifically for inclusion. The traceability is deliberate. When a model is later found to be wrong, the error can be located and corrected at its source rather than buried in ad hoc tuning by downstream users.

**Conventions.** Every model follows the same modeling conventions. Visual and collision geometry are split across distinct rendering groups: visual geoms in group 2 carry the manufacturer’s original meshes and are excluded from contact computation (`contype="0"`, `conaffinity="0"`), while collision geoms in group 3 are the only ones the contact solver sees. Wherever a link’s geometry permits, collision geoms are primitives (boxes, spheres, capsules), which generate a small, well-defined contact manifold and are inexpensive to evaluate; for links whose geometry does not permit this, the collision mesh is convex-decomposed in advance using V-HACD [18] or CoACD [19]. The separation matters because MuJoCo’s contact solver requires convex geometries: at compile time, MuJoCo silently replaces any non-convex mesh geom with its convex hull, which can be a poor approximation of the intended collision surface. Listing 2.1 illustrates the convention on the Franka Panda. Beyond the geometry split, every model declares its parameters under a model-scoped `<default class="...">` block so that multiple Menagerie robots compose into a single scene without identifier collisions, follows shared compiler and integrator settings (angles in radians, meshes in an `assets/` subdirectory, the `implicitfast` integrator), and specifies joint and actuator parameters explicitly (reflected motor inertia, joint friction or damping, actuator control and force ranges) rather than relying on MuJoCo’s zero defaults. A pipeline written against one model therefore loads another without per-robot special-casing.

**Documentation and versioning.** Each model is documented and versioned. A `README` describes how the model was generated, which hardware revision it targets, and what known limitations exist; a `LICENSE` file accompanies every model; and a central `CHANGELOG` records modifications. A specific git commit can be pinned to load exactly the model that a prior result depended on, and a reader can reproduce that load without guesswork. Versioning

Listing 2.1: Menagerie’s visual/collision convention, illustrated on the Franka Panda. Visual geoms (group 2) carry detailed manufacturer meshes and are invisible to the contact solver; collision geoms (group 3) are convex-decomposed meshes where geometry requires it (link 5, three convex pieces) and primitives where it does not (a fingertip pad).

```
<default class="visual">
  <geom group="2" contype="0" conaffinity="0"/>
</default>
<default class="collision">
  <geom group="3"/>
</default>

<!-- Geometrically complex link: V-HACD / CoACD convex decomposition -->
<geom class="collision" mesh="link5_c0"/>
<geom class="collision" mesh="link5_c1"/>
<geom class="collision" mesh="link5_c2"/>

<!-- Simple region: primitive box, no mesh required -->
<geom class="collision" type="box"
  size="0.0085 0.004 0.0085" pos="0 0.0055 0.0445"/>
```

of this kind is what keeps a shared model library a reliable common ground rather than a moving target.

**Installation.** Installation is intentionally low-friction (Listing 2.2): clone the repository and load the MJCF file directly, or load Menagerie models by name through `robot_descriptions.py` [20], a companion Python registry that indexes robot descriptions across the open-source ecosystem. There is no manual directory management, no proprietary toolchain, and no per-platform configuration. Nothing about acquiring or installing a model should contribute to the difficulty of a sim-to-real project.

## 2.3 Breadth of Coverage

Menagerie currently comprises 65 robot model entries spanning the major morphological categories used in robotics research. Table 2.1 reports model counts by category, and Figure 2.1 shows representative models from each.

Manipulator arms range from consumer-grade research platforms (Franka Panda, Universal Robots UR5e, Kinova Gen3, KUKA iiwa14) to recent low-cost arms (YAM, ufactory Lite6, ARX L5). Dexterous hands include the Shadow Hand, Allegro, and LEAP. Humanoids span the Unitree G1 and H1, the Berkeley Humanoid, Apptronik Apollo, Booster T1, and PAL Robotics’ Talos, among others. Quadrupeds include the Unitree Go1/Go2, Boston Dynamics’ Spot, Anybotics’ ANYmal series, and Google’s Barkour



Listing 2.2: Loading the Franka Panda from Menagerie. Either path returns the same compiled `MjModel`.

```
import mujoco

# Option 1: clone the repository, load the MJCF directly.
model = mujoco.MjModel.from_xml_path(
    "mujoco_menagerie/franka_emika_panda/panda.xml"
)

# Option 2: pip install robot_descriptions, load by name.
from robot_descriptions.loaders.mujoco import load_robot_description
model = load_robot_description("panda_mj_description")
```

Table 2.1: Model counts by category in Menagerie. Some platforms span multiple categories (for example, a mobile base with two arms), in which case they are counted under their primary morphology.

| Category                        | Count |
|---------------------------------|-------|
| Manipulator arms                | 22    |
| Dexterous hands and grippers    | 8     |
| Humanoids                       | 12    |
| Quadrupeds and bipeds           | 9     |
| Mobile manipulators             | 6     |
| Aerial vehicles                 | 2     |
| Bimanual platforms              | 1     |
| Other (educational, biological) | 5     |
| Total                           | 65    |

platforms. Mobile manipulators include the Hello Robot Stretch and Stanford TidyBot. Aerial vehicles are represented by the Bitcraze Crazyflie and Skydio X2. The coverage tracks the range of hardware the sim-to-real community actively works with, so the library can serve as a common starting point regardless of which platform a project targets.

## 2.4 Adoption as Evidence

If the design principles of Section 2.2 had not addressed a real gap, Menagerie would have remained an internal artifact used by a single group. Three observations suggest that they did. First, as of early 2026 the public repository has accumulated approximately 3,400 GitHub stars and has been cited in 66 published works since its release; it now functions as the default model source for the majority of MuJoCo-based robotics research. Second, manufacturers including Franka, Flexiv, I2RT, and PAL Robotics contribute and maintain their own robot models directly within the repository, a shift from the earlier dynamic

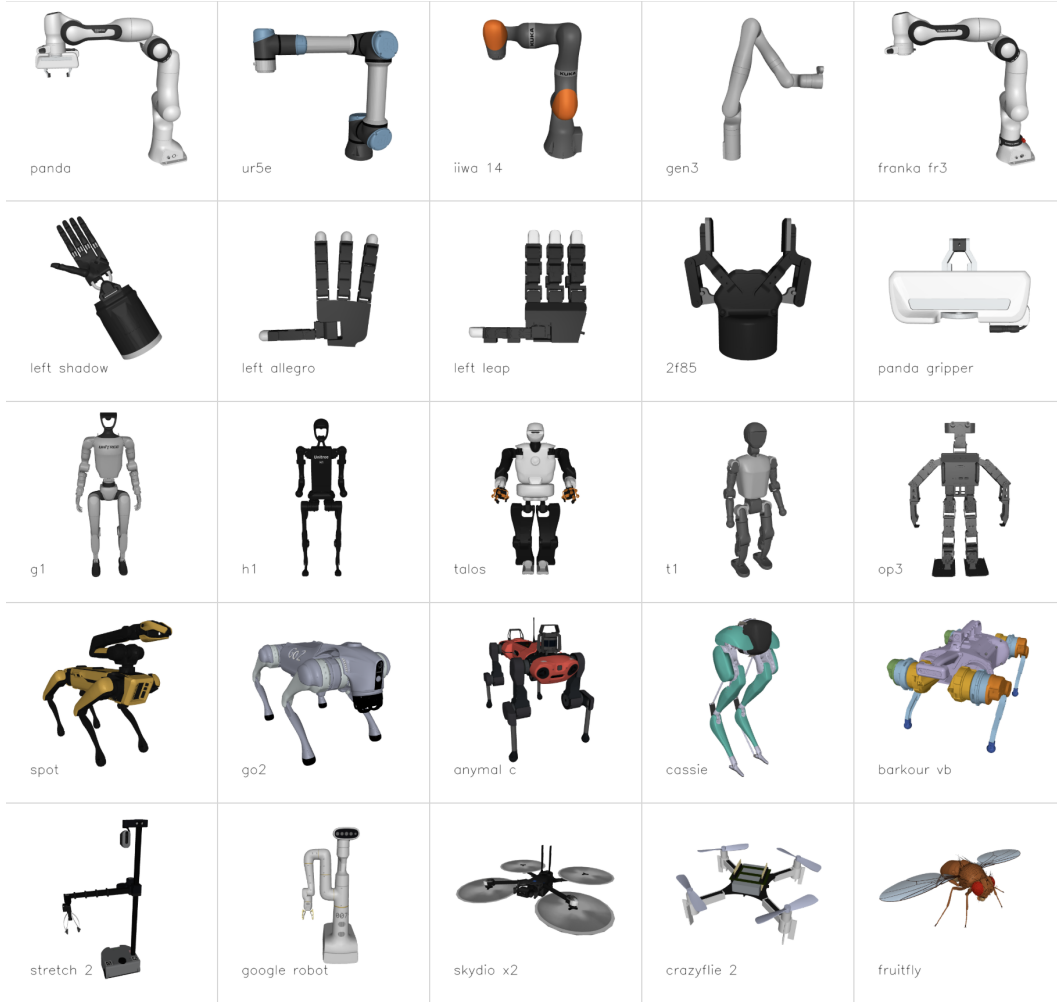


Figure 2.1: A selection of robot models from MuJoCo Menagerie, spanning the morphological categories listed in Table 2.1: manipulator arms, dexterous hands and grippers, humanoids, quadrupeds and bipeds, and mobile manipulators, aerial vehicles and biomechanical models.

in which model quality depended on whoever had most recently worked on a community port to one in which the entity with ground truth about the hardware is also the entity maintaining the corresponding simulation model. Third, in the years following Menagerie’s release, Drake, Newton, and Isaac Lab each added native MJCF loading support to their simulators; the presence of a large, curated MJCF corpus made the format worth supporting outside MuJoCo, even though MJCF remains MuJoCo’s native description format. None of these observations is causal proof on its own, but together they indicate that the conventions and tooling described above mapped onto an existing need rather than an internal one.

Within this thesis, Menagerie provides the starting models for every subsequent chapter. The system identification work in Chapter 3 uses Menagerie models as the prior that identification refines. MuJoCo Playground (Chapter 4), mjlab (Chapter 5), RoboPianist

(Chapter 6), and `mink` (Chapter 7) all load robot configurations directly from Menagerie. The library functions as the shared substrate on which the remaining contributions of this thesis are built.

## 2.5 Discussion

Menagerie provides correct geometry, consistent conventions, and models that load and simulate without modification. This covers a substantial fraction of what simulation-based robotics research requires: motion planning, visualization, inverse kinematics, and early-stage reinforcement learning experimentation each rely on no more than what Menagerie provides, and a wide range of published results have been produced from Menagerie models without further modification.

What Menagerie does not provide is an exact digital twin of a specific robot. The link masses in a Menagerie model are computed from manufacturer CAD and reflect the nominal design rather than the particular unit on the lab bench with its accumulated wear, replaced components, and manufacturing variation. Joint friction, actuator dynamics, and sensor delays depend on the specific hardware instance and are not captured by a shared nominal model. Menagerie therefore addresses one half of the grounding stage: it removes the failure mode of simulations that are wrong because the underlying model was reconstructed carelessly, but not the failure mode in which the nominal model does not match the specific instance of the hardware. Closing that gap, by bringing the simulated model into quantitative agreement with a specific physical robot, requires system identification, which is the subject of Chapter 3.

## Chapter 3

# mujoco.sysid: A System Identification Toolbox for MuJoCo

A Menagerie model is built from manufacturer CAD data and nominal specifications; the robot in the laboratory is a specific physical instance, worn by use and deviating from those specifications in ways that matter for control. This chapter addresses the calibration step that closes that gap, introducing `mujoco.sysid`, a system identification toolbox built directly into MuJoCo, along with practical recommendations drawn from applying it across several robot platforms.

### 3.1 Introduction

System identification, sometimes called *characterization* or *calibration*, is the practice of improving the model of a dynamical system using data collected from the real system. Modern robotics relies heavily on simulation for training, planning, and evaluation, but the usefulness of simulation is bounded by the so-called *sim-to-real gap*: the difference between the real robot and its simulated counterpart. Closing that gap reduces the need for domain randomization and real-world fine-tuning, and is in many cases the deciding factor between a controller that works in simulation and one that works on hardware.

Despite its centrality to real-world robotics, system identification remains under-documented in practice. The literature is fragmented and often focuses on narrow theoretical settings whose assumptions rarely hold on physical hardware. Many existing methods rely on handcrafted excitation strategies or bespoke optimization procedures that do not generalize across robots. What is missing is a practical, general-purpose methodology that works across robot types and modeling frameworks without requiring deep theoretical background to use effectively.

At its core, system identification is an optimization problem: tune model parameters so that the simulator best explains the observed real-world data. In practice, the process is highly sensitive to the choice of identified parameters, excitation trajectories, and the

temporal alignment of simulated and real signals.

This chapter presents a practical, open-source toolbox for robotics system identification, along with a set of best-practice recommendations distilled from experience.

The headline result, presented in detail in Section 3.5, is on the YAM arm [21] (Figure 3.1). Under gravity compensation, the nominal model drifts the end-effector by approximately 44 centimeters from the commanded pose. Identification on a single chirp-excitation sequence brings that drift below half a millimeter, a reduction of roughly three orders of magnitude over the nominal model and a factor of four over a hand-tuned constant torque scaling characterized by the robot’s designers using external measurement equipment.

## 3.2 Preliminaries

A controlled dynamical system with state  $x_t$  and control  $u_t$  evolves according to  $\dot{x}_t = f(x_t, u_t)$ . For most robotics purposes, the macroscopic evolution of the dynamics can be considered deterministic. The state contains the positions  $q$  and velocities  $\dot{q}$ , which evolve according to the equations of motion, plus optional auxiliary states associated with the controller. The state is observed via noisy, delayed sensor readings  $y_t \sim p(y \mid x_{t-t_d})$ , where  $t_d$  is the delay associated with a sensor.

The corresponding discrete-time simulated system, with simulated quantities denoted by overbars, is  $\bar{x}_{i+1} = \bar{f}(\bar{x}_i, u_i; \theta)$ . The simulated state  $\bar{x}_i$ , driven by the same controls as the real system, is evolved by the simulator  $\bar{f}$ , which also produces simulated sensor readings  $\bar{y}_i = \bar{g}(\bar{x}_i; \theta)$ . The vector  $\theta$  contains the parameters to identify.

### System identification as parameter estimation

We focus on *structured* system identification for rigid-body robotic systems, where the equations of motion are known and implemented by the simulator. The goal is to recover physical parameters such as link masses or friction coefficients within a fixed model class.

Using non-subscripted symbols for concatenated time series, let  $y = \{y_i\}_{i=1}^N$  denote a trajectory of sensor observations and  $\bar{y}(\theta) = \{\bar{y}_i(\theta)\}_{i=1}^N$  the corresponding simulated values. The trajectory-matching objective is the nonlinear least-squares problem

$$\min_{\theta \in [\theta_{\min}, \theta_{\max}]} \frac{1}{2} (\bar{y}(\theta) - y)^\top (\bar{y}(\theta) - y) \quad \text{subject to} \quad \bar{x}_{i+1} = \bar{f}(\bar{x}_i, u_i; \theta), \bar{y}_i = \bar{g}(\bar{x}_i; \theta). \quad (3.1)$$

The difference  $r(\theta) = \bar{y}(\theta) - y$  is the *residual*. Equation (3.1) is simplified in several ways: it does not account for delays, it uses the quadratic norm (corresponding to Gaussian noise) but other norms are possible, it weights all sensor types identically, and it does not address how the initial simulated state  $\bar{x}_1$  is matched with the real initial state  $x_1$ . The remainder of this section addresses each of these points.

**Bounds on  $\theta$ .** Lower and upper bounds for the parameters  $\theta \in [\theta_{\min}, \theta_{\max}]$  enforce physical plausibility (preventing negative mass or a center of mass outside the corresponding link, for

example) and inject prior knowledge by confining the search to a reasonable range. Convergence to a bound is itself diagnostic: it often signals model misspecification or identifiability issues.

**Identification of delays.** Simulating arbitrary delays is awkward, requiring data buffers and restricting delays to discrete multiples of the timestep. In the identification setting, continuous-valued delays can instead be modeled by treating simulated measurements as continuous signals via interpolation and introducing a real-valued temporal offset  $t_d$  that is itself included as a parameter in  $\theta$ . Unmatched data at trajectory boundaries is prevented by truncating the time series on each end by the maximum possible delay, so the overlapping window has fixed length and no signal is extrapolated.

**Choice of norm.** The toolbox supports any differentiable convex norm. In the case studies below we use the quadratic, corresponding to a Gaussian noise profile. For sensors with distinctly non-Gaussian noise, such as touch sensors or those prone to sporadic outliers, robust norms (for example the Huber norm) are expected to offer better performance.

**Sensor weighting.** System identification is often interpreted as a maximum-likelihood problem, in which case each sensor’s residuals are weighted according to its noise. Estimating that noise can be difficult. By default, the toolbox makes the pragmatic assumption that a sensor’s noise is proportional to the magnitude of its recording:  $\text{Var}(y^j) = \alpha \|y^j\|^2$  for some positive  $\alpha$ , where  $y^j$  is the residual vector for sensor  $j$ . Since the minimum is invariant to scaling, this amounts to weighting sensor  $j$ ’s residuals by  $1/\|y^j\|$ . Per-sensor relative weights can be adjusted by the user. When combined with bounds on  $\theta$ , the result is a maximum-a-posteriori estimator under a uniform prior on  $\theta$ .

**Initial states and drift.** For the optimization in Equation (3.1) to be well posed, the real and simulated initial states  $x_1$  and  $\bar{x}_1$  must match reasonably well. For most rigid-body systems this is straightforward, using motionless poses. Some systems have intrinsic drift, however: chaotic pendulums and quadrotors are canonical examples, and over long horizons their simulated and real trajectories will diverge regardless of parameter accuracy. Two strategies help. Where possible, position control reduces drift and allows long trajectories to be matched as a single unit (provided gains are not so high that they obscure the underlying dynamics). Otherwise, an estimator can be used to obtain initial states at multiple points along the recorded trajectory; the simulated state is then reset to these, effectively chopping one long trajectory into many shorter ones. We call this *chunking* and support it natively. Even in the position-controlled setting, chunking is useful because it parallelizes across threads.

### 3.3 The mujoco.sysid Toolbox

The toolbox is an open-source Python package distributed as an optional extra of MuJoCo, installed via `pip install mujoco[sysid]`. It is designed to be reusable across robots and workflows: rather than prescribing a single end-to-end pipeline, it provides a small set of composable primitives that can be assembled into a problem-specific identification setup. A

typical workflow begins with designing excitation trajectories in simulation, where they can be vetted for safety against the nominal model. The same trajectories are then executed on the real robot and sensor data is recorded. With both simulated and real data in hand, the user defines the parameter vector and its bounds and runs the optimizer. The remainder of this section describes the design choices and the features that support each step.

**Gray-box framing.** The toolbox implements gray-box identification: the user supplies the model structure as MuJoCo XML, designates which parameters are unknown, and the optimizer adjusts only those parameters while leaving the structure fixed. This is in contrast to *black-box* approaches that learn dynamics entirely from data with no assumed structure. The benefits of the gray-box view are interpretability, physical plausibility, and a small parameter count that is identifiable from comparatively little data; the cost is that the toolbox cannot recover effects that the model class does not represent.

**MuJoCo as the backbone.** The toolbox is built on MuJoCo [10] because MuJoCo is fast, widely used in the robotics and reinforcement-learning communities, and exposes a rich model-specification format (MJCF) that can be edited programmatically. Procedural model modification through MuJoCo’s model-editing API means that the set of identifiable parameters is not hard-coded: it is whatever the user chooses to vary, and includes any floating-point quantity in the MJCF description.

**What can be identified.** The parameters fall into four categories, organized by where in the model they appear:

- **Link parameters:** the inertial properties of each body, namely mass, center of mass, and the full inertia tensor. These can be parameterized in several ways, ranging from a single scalar mass to the full 10-dimensional pseudo-inertia of Rucker and Wensing [22], which guarantees a positive-definite inertia matrix without singularities.
- **Joint parameters:** viscous and Coulomb friction at each joint.
- **Actuator parameters:** P and D gains, armature (the reflected inertia of the rotor), and torque-scaling factors for actuators that estimate torque from motor current rather than measuring it directly.
- **Sensing parameters:** per-sensor delay, gain, and bias, applied as a post-processing step on the predicted residual rather than inside the rollout.

The parameter vector  $\theta$  is constructed by the user from these categories. A single parameter can be assigned to multiple elements (a single friction coefficient shared across three joints, for example), or each element can have its own. The semantics of  $\theta$  need not match MuJoCo’s native representation; the pseudo-inertia parameterization above is one such case, converted to MuJoCo’s internal form during simulation.

**Finite-difference Jacobians.** The residual Jacobian  $J = \partial r / \partial \theta$  is computed by finite differences rather than through a differentiable simulator. This provides generality: any parameter that appears in the MJCF model, from masses and inertias to solver tolerances, can enter the parameter vector without an analytical derivative. Each column of  $J$  requires

one perturbed rollout, and the rollouts are independent and trivially parallel across threads. In practice this is fast enough that identification problems with tens of parameters converge in seconds.

**Optimization.** The optimizer uses the Gauss-Newton method. Given the residual  $r(\theta)$  and its Jacobian  $J$ , the approximate Hessian is  $H \approx J^\top J$ , the gradient is  $g = J^\top r$ , and the update is  $\Delta\theta = -H^{-1}g$ . This achieves second-order convergence from first-order derivatives alone. Box constraints are handled by projected steps. Both SciPy’s and MuJoCo’s least-squares solvers are supported as backends.

**Confidence intervals.** For each identified parameter, the toolbox reports a confidence interval derived from the Gauss-Newton Hessian at the optimum. This is the toolbox’s principal technical contribution beyond the surrounding engineering: it is easy to construct an identification objective without a unique solution, in which case the Hessian is ill-conditioned and a naive covariance estimate from  $H^{-1}$  produces misleadingly tight intervals. The toolbox instead uses the eigendecomposition of  $H$  to detect directions in parameter space that the data does not constrain, and assigns unreasonably large or infinite intervals to parameters that project onto those directions. The pseudo-inverse  $H^+$ , which a reader might propose as an alternative, drastically underestimates the intervals and should not be used. The practical effect is that identifiability failures are immediately visible: a parameter whose confidence interval is infinite is one the data simply did not constrain.

**HTML reports.** At the end of each run, the toolbox emits an HTML report containing parameter values and confidence intervals, sim-versus-real residual plots, identified-versus-nominal trajectory overlays, parameters that hit their bounds, and the optimization trace. The report is the unit of audit: rather than a fitted vector of numbers, the user receives the full record of which parameters moved, which stayed at their bounds, and where the residual is largest.

## 3.4 Related Work

System identification is a fundamental technique in robotics for improving simulation accuracy, enabling model-based control, and facilitating sim-to-real transfer. We situate our contribution within several relevant research threads.

**System identification and parameter estimation.** Classical system identification methods estimate physical model parameters to align simulation with real-world behavior [23, 24], often targeting inertial, frictional, and actuator properties [25]. To ensure physical plausibility, structured parameterizations such as the pseudo-inertia matrix are used [22, 26]. Two persistent challenges in this literature are *identifiability*, whether a given dataset constrains the parameters uniquely, and *excitation design*, the question of which control inputs to apply in order to maximize parameter sensitivity [27–29]. A complementary practitioner tactic, sometimes referred to as *disambiguation*, is to collect data under known modifications to the system (a rigidly attached known mass, a controlled change in gravity direction) so that otherwise confounded parameter groups become distinguishable. We treat excitation design



and disambiguation as open problems that practitioners must navigate in conjunction with the toolbox, rather than as problems the toolbox itself solves. For a broader overview, see the survey of Lee et al. [30].

**Software peers.** The closest software peer in scope is the regressor-based identification supported by Pinocchio [31], which exploits the linearity of rigid-body dynamics in inertial parameters ( $\tau = Y(q, \dot{q}, \ddot{q}) \pi$ ) to cast inertial identification as a linear least-squares problem. That linear setup is faster and better-conditioned than the nonlinear procedure used here, but is restricted to inertial parameters. `mujoco.sysid` addresses a more general class of identification problems in which the parameter vector can include joint friction, sensor delays, actuator gains, and any other floating-point quantity that appears in the MJCF description, at the cost of requiring finite-difference Jacobians and Gauss-Newton iteration. Confidence intervals derived from the Gauss-Newton Hessian are reported as a first-class output, which we have not seen in this combination elsewhere.

**Learning-based dynamics models and sim-to-real transfer.** Recent work has explored learning dynamics models directly from data, using neural networks or hybrid models that learn residuals over physics engines [32]. These methods bypass explicit system identification, favoring expressiveness over interpretability, but often struggle in out-of-distribution settings. In contrast to classical identification, which targets physically grounded parameters through structured modeling and informative excitation, learning-based approaches embed dynamics implicitly, making them harder to generalize from or diagnose.

**Differentiable simulation.** Our software uses gradients computed by finite differences. These computations take the bulk of the optimization time, and differentiable simulators [33] offer a shortcut that can speed them up, with some works [34, 35] demonstrating this directly. Our finite-difference approach differentiates through any floating-point number in the model specification, whereas differentiable simulators typically support only a subset.

**Robust policies and domain randomization.** In sim-to-real applications, domain randomization [36, 37] and system identification through policy adaptation [38, 39] are widely used to bridge the reality gap. RMA [40] exemplifies a recent trend of learning adaptive policies conditioned on a latent representation of dynamics, effectively amortizing identification into the policy itself. Deciding what to randomize and how much remains manually intensive, and excessive randomization yields overly conservative policies.

**Improved actuators and friction modeling.** Realistic modeling of joint friction and actuator dynamics substantially improves sim-to-real performance. Recent work on extended friction models [41] introduces flexible formulations for servo actuators, enabling more accurate force production and contact interactions. Hwangbo et al. [42] introduced neural actuator networks to capture series-elastic actuator dynamics, enabling high-fidelity simulation of the ANYmal robot. Our method is complementary: rather than replacing physics, it improves the estimation of physically interpretable parameters, and can be used alongside learned actuator models.

### 3.5 Experimental Validation

We validate the toolbox on two platforms chosen for the different identification regimes they exercise. The YAM manipulator (Section 3.5) is a fixed-base, six-degree-of-freedom arm whose torques are estimated from motor currents rather than measured by dedicated joint torque sensors. It tests whether identification can recover a usable feedforward dynamics model from a single chirp-excitation sequence on a low-cost platform, with downstream performance measured by gravity-compensation drift. The custom FPV quadrotor (Section 3.5) is a floating-base, free-flying system, with parameters that include inner-loop PID gains and a polynomial throttle-to-thrust map. It tests the toolbox in a considerably more constrained data regime (six seconds of motion-capture data from a single free-flight segment) and exercises its flexibility, since the case study required accommodating sensors not natively supported by MuJoCo and optimization through a Python replication of the on-board flight controller. Downstream performance is measured by the transferability of an open-loop maneuver designed in simulation using the identified model.

#### Manipulator: YAM



Figure 3.1: The YAM six-DoF manipulator: a low-cost, fixed-base arm with quasi-direct-drive proprioceptive actuators. Joint torques are estimated from motor currents rather than measured by dedicated torque sensors.

YAM [21] is a low-cost, fixed-base, six-DoF arm with quasi-direct-drive proprioceptive actuators [43] (Figure 3.1). Unlike higher-end manipulators with dedicated joint torque sensors, YAM estimates joint torques from motor currents and derives its inertial properties from CAD. It is representative of a growing class of affordable research arms.

The primary evaluation task is gravity compensation: applying feedforward torques that hold the arm stationary at arbitrary configurations. The nominal CAD-based model performs poorly, with the arm drifting substantially once feedback is removed. The robot’s designers addressed this by manually scaling all commanded torques by a constant factor of 1.3 after characterization with external measurement equipment.

We identified link masses  $m_i$ , center-of-mass offsets  $\mathbf{r}_{c_i}$ , viscous and Coulomb joint friction, sensor delays, and reflected inertias. Excitation was provided by a joint-space chirp signal sweeping from 0.01 Hz to 3 Hz, applied independently to each joint under position

control. Data were sampled at 500 Hz and included joint positions and current-based torque estimates.

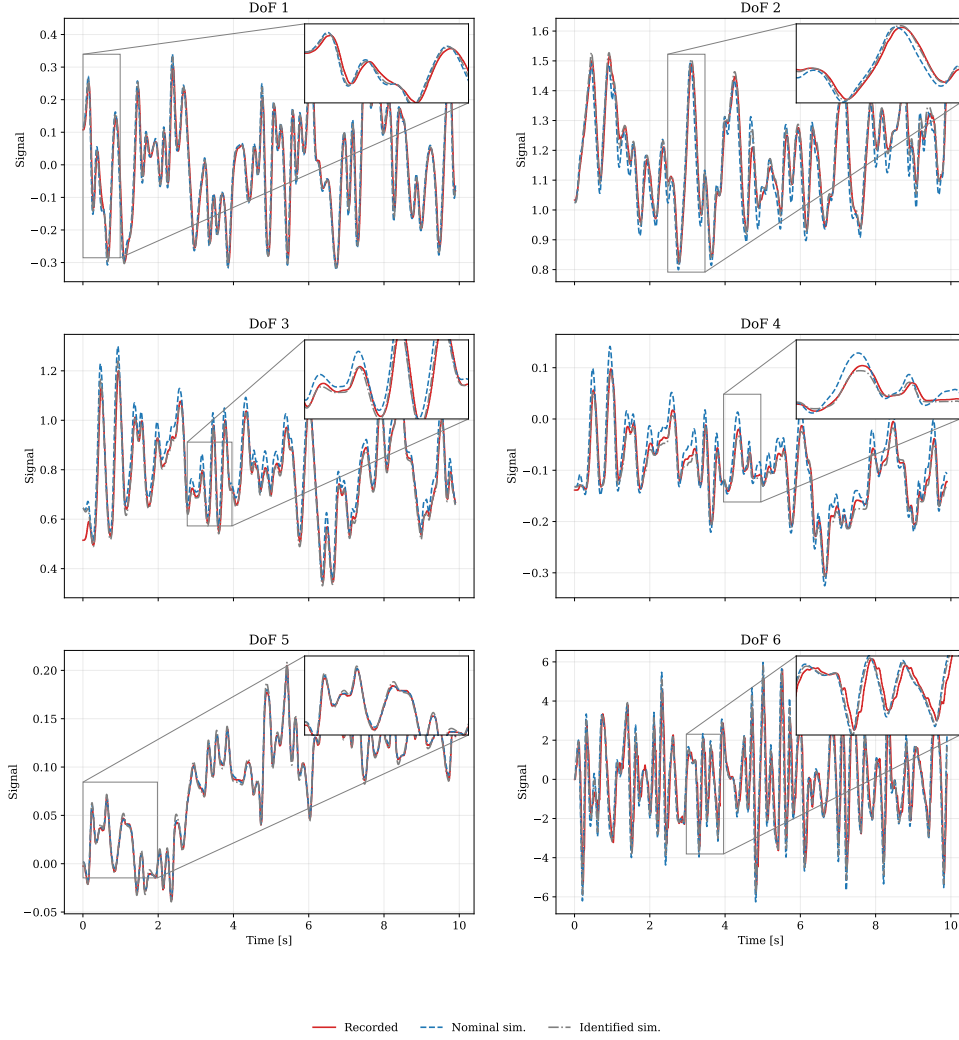


Figure 3.2: Position predictions (rad) on a held-out trajectory generated by random Cartesian waypoints. The identified model (green) tracks the recorded data (black) substantially better than the nominal model (blue) across all six joints.

Figure 3.2 compares position predictions from the nominal and identified models on a held-out trajectory generated by random Cartesian waypoints. The identified model reduces peak torque prediction error by 33.7% averaged across joints (Table 3.1).

For gravity compensation, we drove the arm to 50 random collision-free configurations, held each with PD feedback and feedforward gravity torques for two seconds, then disabled feedback and measured drift. Table 3.2 reports the results under three conditions: nominal (CAD-based model), manual ( $1.3\times$  torque scaling), and identified. The identified model reduces end-effector drift from  $1.7 \times 10^{-3}$  m to  $4.1 \times 10^{-4}$  m relative to the hand-tuned  $1.3\times$

Table 3.1: Maximum torque prediction errors (Nm) on a held-out trajectory: nominal versus identified model. Joint 6 is omitted from the torque comparison.

| Joint | Nominal | Identified |
|-------|---------|------------|
| 1     | 0.105   | 0.073      |
| 2     | 0.178   | 0.109      |
| 3     | 0.192   | 0.130      |
| 4     | 0.072   | 0.045      |
| 5     | 0.013   | 0.014      |
| Mean  | 0.112   | 0.074      |

Table 3.2: Gravity compensation performance: joint-space RMS error (rad) and end-effector position error (m) averaged over 50 configurations. Joints 1 and 5 have minimal gravitational loading due to axis alignment.

|       | Nominal           | Manual ( $1.3\times$ )                      | Identified                                  |
|-------|-------------------|---------------------------------------------|---------------------------------------------|
| $q_1$ | $0.093 \pm 0.275$ | $4.0 \times 10^{-5} \pm 1.1 \times 10^{-4}$ | $8.0 \times 10^{-5} \pm 1.5 \times 10^{-4}$ |
| $q_2$ | $0.669 \pm 0.589$ | $2.6 \times 10^{-3} \pm 6.8 \times 10^{-3}$ | $5.3 \times 10^{-4} \pm 1.4 \times 10^{-3}$ |
| $q_3$ | $0.521 \pm 0.565$ | $2.7 \times 10^{-4} \pm 4.5 \times 10^{-4}$ | $1.9 \times 10^{-4} \pm 4.6 \times 10^{-4}$ |
| $q_4$ | $0.050 \pm 0.048$ | $8.0 \times 10^{-5} \pm 1.5 \times 10^{-4}$ | $1.5 \times 10^{-4} \pm 1.9 \times 10^{-4}$ |
| $q_5$ | $0.002 \pm 0.002$ | $4.6 \times 10^{-4} \pm 2.3 \times 10^{-4}$ | $5.3 \times 10^{-4} \pm 2.5 \times 10^{-4}$ |
| EEF   | $0.444 \pm 0.364$ | $1.7 \times 10^{-3} \pm 4.3 \times 10^{-3}$ | $4.1 \times 10^{-4} \pm 9.7 \times 10^{-4}$ |

scaling, a roughly fourfold improvement, and from 0.444 m to  $4.1 \times 10^{-4}$  m relative to the nominal CAD-based model, a roughly thousandfold improvement.

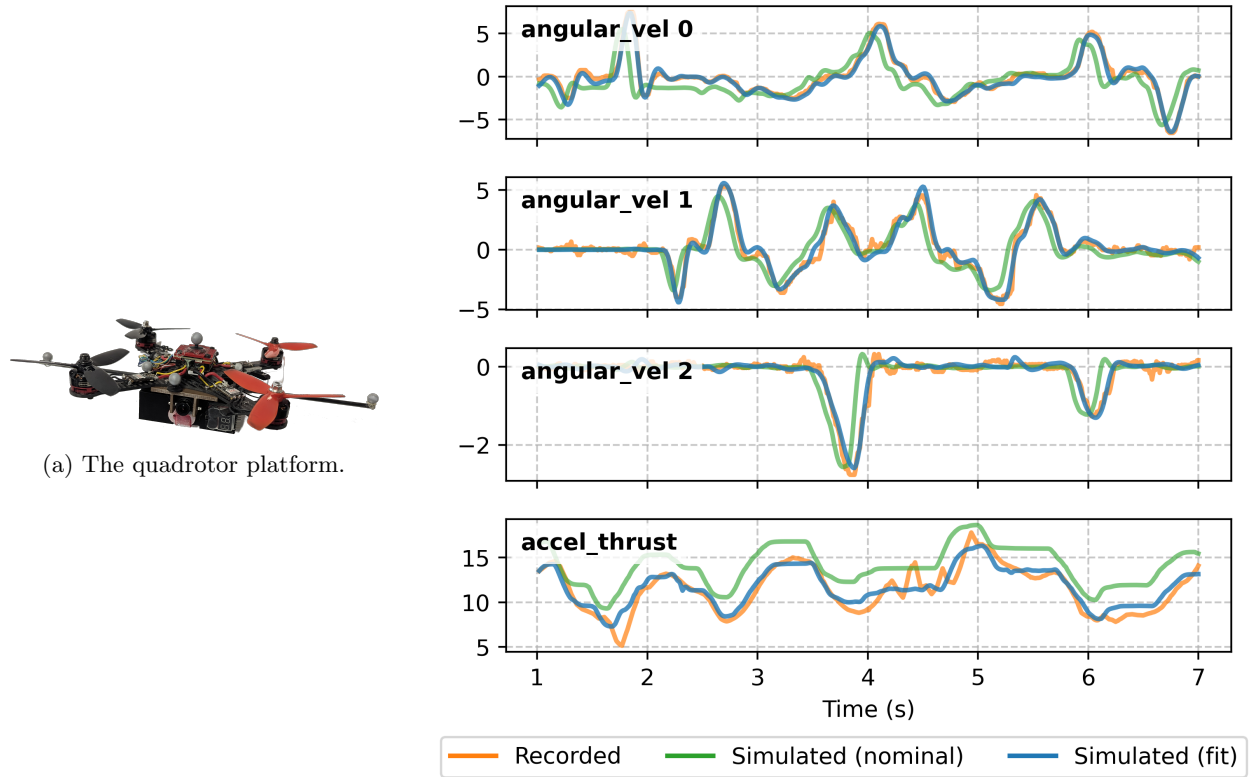
The identified model is *useful*, in that it substantially improves gravity compensation, but it is not necessarily *physically truthful*. Some identified centers of mass fall outside the physical geometry of the corresponding links, which indicates that the optimizer is absorbing unmodeled effects (most likely actuator dynamics and the current-to-torque relationship) into whichever parameters are available. The resulting model predicts torques and positions well in the excitation regime, but the individual parameter values should not be interpreted as ground truth. This is a general limitation of structured identification when the model class does not capture all relevant dynamics.

## Quadrotor

We identified the inner-loop controller gains and the throttle-to-thrust curve of a custom FPV quadrotor (Figure 3.3). The parameter vector contained PID gains for pitch, roll, and yaw angular velocity controllers (the inner loop running on the onboard Betaflight flight controller), the coefficients of a polynomial throttle-to-thrust map, mass, and a delay parameter.

Six seconds of motion-capture data were collected during free flight with a human pilot. The identification residual was the difference between predicted and recorded angular ve-

locities and thrust-induced accelerations. The case study required accommodating sensors not natively supported by MuJoCo (normed acceleration with gravity subtracted) and optimization through a Python replication of the flight controller’s inner loop, exercising the toolbox’s flexibility.



(b) Identified-model fit on angular velocities and thrust-induced acceleration.

Figure 3.3: The FPV quadrotor and identification fit: the optimizer aligns simulated angular velocities and thrust-induced accelerations with the motion-capture recording from a 6 s free-flight segment.

The identified model was used to design an open-loop 360-degree flip sequence in simulation. The sequence transferred to the real quadrotor with a minor throttle adjustment: the simulation used  $1.5g$  thrust for 0.5 s, while the real system required  $1.75g$  for 0.4 s (Figure 3.4). An LQR outer-loop controller, also designed using the identified model, stabilized the quadrotor before and after the flip.

The dominant unmodeled effect was battery-voltage variation, which changes the throttle-to-thrust relationship over the course of a flight. This was not included in the identification and required the manual throttle adjustment for transfer. Even so, the case study demonstrates that limited data (six seconds) is sufficient to produce a model that supports the design of transferable open-loop maneuvers, with the residual sim-to-real gap attributable to effects (battery state, motor temperature) that the model class does not capture.

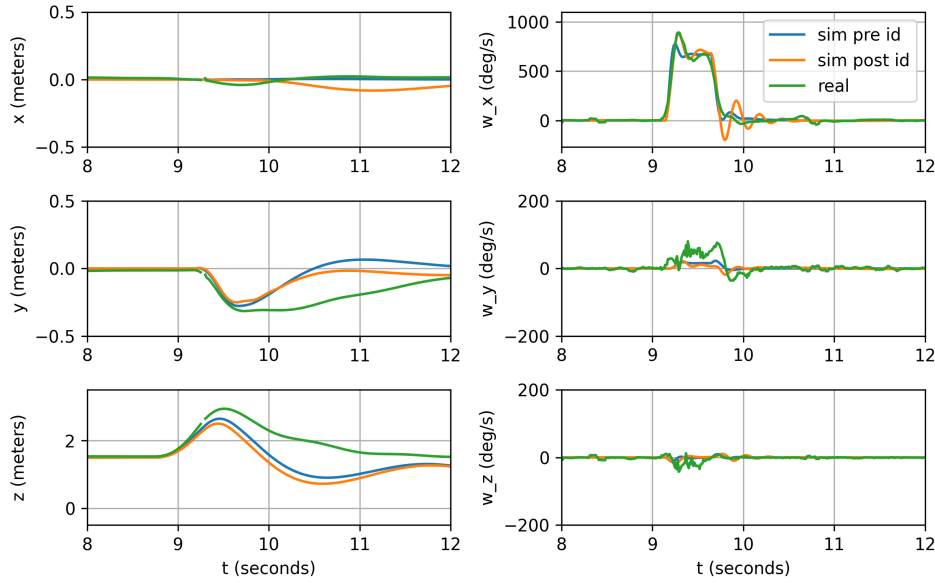


Figure 3.4: Position and angular velocity during an open-loop 360-degree flip: simulation vs. real. Both use the same control sequence; the open-loop trajectory was designed in simulation using the identified model.

## 3.6 Limitations

The methodology has several known limitations.

The automatic procedure for determining sensor noise is pragmatic but far from optimal. Future work could consider hill-climbing approaches such as that of Ting et al. [44], which jointly estimates identification parameters and sensor noise.

We have not developed a methodology for recognizing and mitigating the effect of unmodeled dynamics on parameter estimates. A robot with a substantial vibration mode or joint elasticity, for example, will often push identified parameters to their bounds, which the HTML report does flag. Recognizing such situations and devising a fix remains the user’s responsibility.

We have not addressed optimal excitation design, which has a long history in the literature [30]. The toolbox solves the optimization problem given data; it does not solve the experiment design problem.

The YAM experiments demonstrate strong improvement on gravity compensation, which is dominated by mass and center-of-mass parameters. We did not identify the full inertia tensor on this platform, which would matter for rapid motions. Tooling for determining the set of structurally identifiable inertial parameters [30] would extend the toolbox’s reach here.

The quadrotor experiments demonstrate flexibility and sim-to-real transfer of an open-loop maneuver, but they are limited in scope. Future work should consider identification of drag and motor-level dynamics, with evaluation on closed-loop trajectory-tracking performance.

The toolbox uses parallel rollouts for finite differencing, which often makes optimization converge in seconds. Extending parallel rollouts to support arbitrary user-defined functionality currently requires writing MuJoCo plugins in C++. Fully custom rollout functions implemented in Python are also supported, but they are single-threaded and substantially slower. Future work can address this by providing a framework for staging data across multiple Python processes.

## 3.7 Recommendations

The following recommendations are practitioner-level lessons that have recurred across our case studies. They are not specific to the toolbox, but are easier to follow with its diagnostics in hand.

**Visualize everything.** Plot ground-truth and simulated signals side by side, on every sensor channel that contributes to the residual. Heat-map visualizations of the Gauss-Newton Hessian help expose null spaces in parameter space, and visualizing rigid-body inertia ellipsoids surfaces identifications that are mathematically valid but not physically plausible.

**Treat parameters with different priors.** Manufacturer-reported parameters vary in reliability across categories. Kinematic dimensions are typically accurate; reported masses, especially of small components, less so; reported friction and damping values are rarely useful. Bounds on  $\theta$  should be set accordingly.

**Parameter isolation.** Where possible, design experiments so that subsets of parameters can be identified independently. Examples include avoiding contacts and joint limits when identifying smooth-motion parameters, moving one joint at a time during friction identification, and clamping the root body of a floating-base robot to the world.

**Timestamps.** Record measurement timestamps as precisely as the system allows, separately for each sensor, and do not assume a fixed sampling interval. Where possible, record timestamps on the robot rather than on a host machine, and when the robot has multiple onboard clocks, do not assume that they are synchronized.

**Avoid high gains.** High-gain position-controlled robots track their targets too closely, obscuring the underlying dynamics that identification depends on. Lower gains, heavier loads, or higher target accelerations all introduce the lag between commanded and actual positions that the optimizer needs.

**Cross-validation.** System identification problems are typically low-dimensional, and the parameter vector is often constrained by a small number of sequences. Running identification on a subset of the data and validating on the remainder provides one check on overfitting; for robots with repeated structure such as symmetric limbs, parameters identified on one limb can be cross-checked against another.

## 3.8 Conclusion

We presented `mujoco.sysid`, an open-source system identification toolbox that ships with `MuJoCo` and makes structured identification available via a single `pip` command. We described the underlying formulation and a set of design choices that keep the toolbox flexible across robots and workflows. We demonstrated it on the YAM manipulator and on a custom quadrotor, and reported both where it succeeds and where it falls short.

The open question that the toolbox helps investigate but does not answer is: under what conditions does improving simulation fidelity through system identification actually improve downstream task performance? We believe this question deserves systematic study, and we hope that shared tooling lowers the barrier to conducting it.



# Part II

## Training Infrastructure

## Chapter 4

# MuJoCo Playground: GPU-Accelerated Robot Learning Environments

With a grounded model in hand, the pipeline’s next bottleneck is wallclock time: how quickly a researcher can go from changing a reward function to observing the resulting behavior. This chapter presents *MuJoCo Playground*, a GPU-accelerated reinforcement learning library built on MJX, the JAX-based GPU branch of MuJoCo, and asks whether a single, deliberately simple sim-to-real recipe, applied consistently across this open stack, can transfer to a wide range of robot platforms without per-robot algorithmic modifications.

### 4.1 Introduction

The central constraint at the training stage is wallclock time. The loop from modifying a reward function to evaluating the resulting behavior determines how productively a sim-to-real project can be iterated, and that loop is slow on CPU-based simulators. Speed matters specifically for reward design because reward specification cannot be reduced to a solved problem: what the robot ought to do is an expression of human preference, and even semi-automated approaches [45] require repeated cycles of policy training and human evaluation to arrive at behavior that matches intent.

GPU-accelerated simulation shortens this loop in two ways. Keeping the entire agent-environment interaction on device avoids the CPU-GPU transfer overhead that limits throughput when physics and training run on separate processors [7, 46]. On-policy reinforcement learning [47], the dominant training paradigm for sim-to-real locomotion and manipulation, is structurally suited to wide-batch GPU operations: it requires fresh environment data at every update, which maps naturally onto massively parallel environment rollouts. Locomotion and manipulation tasks that previously required days of training on multi-host CPU setups [3, 48] can now be solved within minutes to hours on a

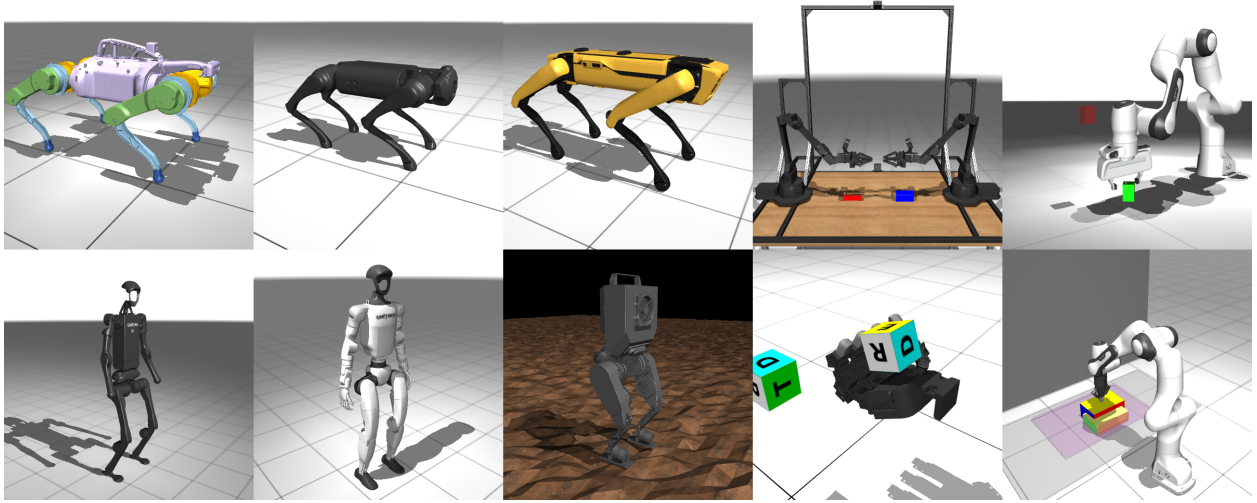


Figure 4.1: A preview of locomotion and manipulation environments available in MuJoCo Playground.

single GPU [1, 4].

MuJoCo Playground is built on MJX [49], the JAX-based GPU branch of MuJoCo, and provides three things: a suite of locomotion and manipulation environments spanning quadrupeds, humanoids, dexterous hands, and robot arms; integration of the Madrona batch renderer [50] for on-device vision-based policy training without teacher-student distillation [51]; and a reproducible training pipeline installable with a single command. The chapter presents results deploying state- and vision-based policies across six robotic platforms and examines whether a single consistent sim-to-real recipe transfers without per-platform algorithmic modification.

## 4.2 Environments

MuJoCo Playground contains environments in 3 main categories: DeepMind (DM) Control Suite, Locomotion, and Manipulation, which we briefly describe in this section. Locomotion and manipulation environments are tailored to robotic use-cases and we show zero-shot sim-to-real transfer in many of the available environments. Playground directly utilizes MuJoCo Menagerie [12] which offers a suite of robot assets and configurations tailored to run in MuJoCo.

### DM Control Suite

The majority of RL environments from [52] are re-implemented in MJX, and serve as entry-level tasks to familiarize users with MuJoCo Playground (Figure 4.3).

## Locomotion

Locomotion environments in MuJoCo Playground are implemented for multiple quadrupeds and bipeds (Figure 4.1 left). The quadrupeds include the Unitree Go1, Boston Dynamics Spot, and Google Barkour [53], while the humanoids include the Berkeley Humanoid [54], Unitree H1 and G1, Booster T1, and the Robotis OP3. For each robot embodiment, we implement a joystick environment that learns to track a velocity command consisting of base linear velocities in both the forward and lateral directions, as well as a desired yaw rate. On the Unitree Go1, we additionally implement fall recovery and handstand environments.

We demonstrate sim-to-real transfer in two main sets of experiments. First, on the Unitree Go1, we deploy joystick, fall recovery, and handstand policies. Second, we demonstrate joystick-based locomotion on the Berkeley Humanoid, the Unitree G1, and the Booster T1. More details on these sim-to-real experiments can be found in Section 4.4.

## Manipulation

Manipulation environments in MuJoCo Playground are implemented for both prehensile and non-prehensile tasks (Figure 4.1 right). With the Leap Hand [55] robot, we demonstrate contact-rich dexterous re-orientation of a block. Using the Franka Emika Panda and Robotiq gripper, we show re-orientation of a yoga block using high frequency torque control. We implement a simple vision-based pick-cube environment on a Franka arm using the Madrona batch renderer. A few additional environments, such as bi-arm peg-insertion with the Aloha robot [56], are also available.

We demonstrate sim-to-real transfer on the Leap Hand and Franka arm robots, including an environment trained from vision for the pick-cube task. More details on the sim-to-real experiments are available in Section 4.4.

## 4.3 Batch Rendering with Madrona

MuJoCo Playground enables vision-based environments through an integration of MJX with Madrona [50]. Madrona is a GPU-based entity-component-system (ECS), which contains GPU implementations of high throughput rendering [57]. Madrona provides two rendering backends: a software-based batch ray tracer written in CUDA (used for the experiments in this work) and a Vulkan-based rasterizer. The raytracing backend supports features including complex lighting scenarios, shadows, textures, and geometry materials. See Figure 4.2 for examples of rendered images using the batch ray tracer. Some features such as deformable materials, moving lights, and terrain height fields will be added in the future.

The Madrona Batch Renderer is integrated with MJX through low-level JAX [58] primitives that connect to the initialization and render functions exposed by Madrona. These JAX primitives allow for Madrona to interact seamlessly with JAX transformations such as *jit* and *vmap*. MuJoCo Playground provides two examples: (`cartpole-balance` and

PandaPickCubeCartesian) to showcase the implementation of vision-based environments and training of vision-based policies.

The Madrona MJX integration also supports customization of each environment instance, allowing for domain randomization [37] of visual properties such as geometry size, color, lighting conditions, and camera pose. These randomizations play a crucial role in the sim-to-real transfer of vision-based policies, which we discuss more in Section 4.4.

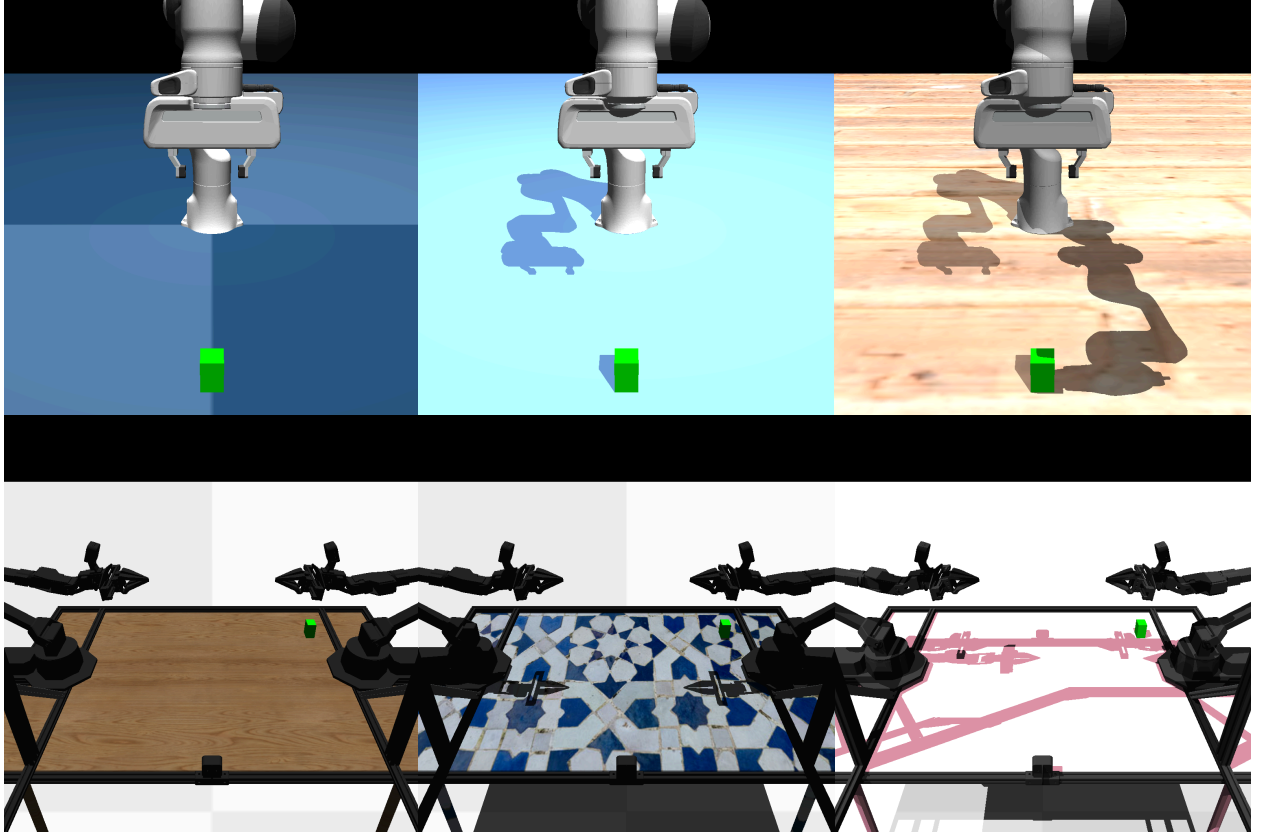


Figure 4.2: Sample renders from the Madrona batch renderer for the Panda and Aloha environments. Left-most images are the original environments. The remaining images highlight the support for lighting, shadows, textures, and colors, including the ability to domain randomize these parameters during training.

## 4.4 Results

In this section, we report RL and sim-to-real results for environments in MuJoCo Playground. Sim-to-real experiments (see some examples in Figure 4.4) are performed for locomotion and manipulation environments from both proprioceptive state and from vision. We briefly discuss RL training on different hardware devices and RL libraries.

## DM Control Suite

We train state-based policies for all available tasks, with most environments training in under 10 minutes on a single GPU device.

Using the batch renderer, we also implement pixel-based observations for the Cartpole-Balance environment. These observations are generated on the GPU, allowing us to keep physics, rendering, and training entirely on-device. Although other DM Control Suite environments can also be rendered with Madrona, we demonstrate end-to-end RL training on only one task, leaving a more comprehensive exploration for future work.

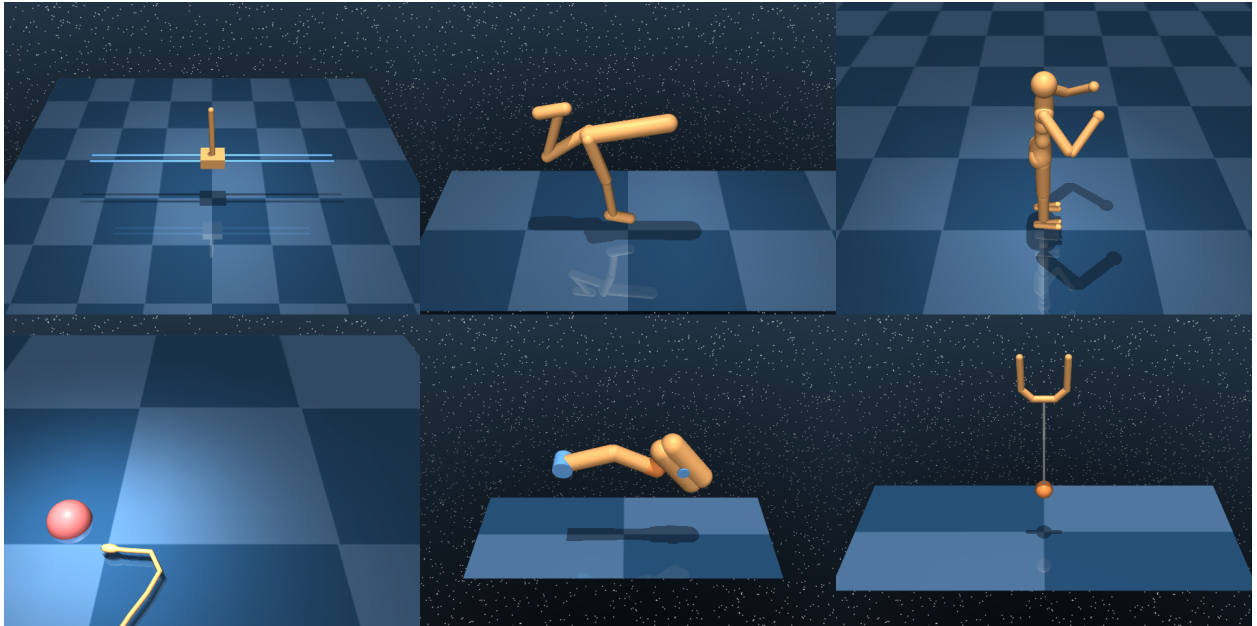


Figure 4.3: Several DM Control Suite environments.

## Locomotion

We present sim-to-real locomotion results on both a quadruped (Unitree Go1) and three humanoid platforms (Berkeley Humanoid, Unitree G1, and Booster T1).

### Quadruped Locomotion

**Task definition** We implement a joystick locomotion task as in [1, 59], where the command is specified by three values indicating the desired forward velocity, lateral velocity, and turning rate of the robot’s root body. Additionally, we design policies for handstand and footstand tasks, in which the robot balances on the front or hind legs, respectively, while minimizing actuator torque. For fall recovery, we follow [60, 61], enabling the robot to return to a stable “home” posture from arbitrary fallen configurations.



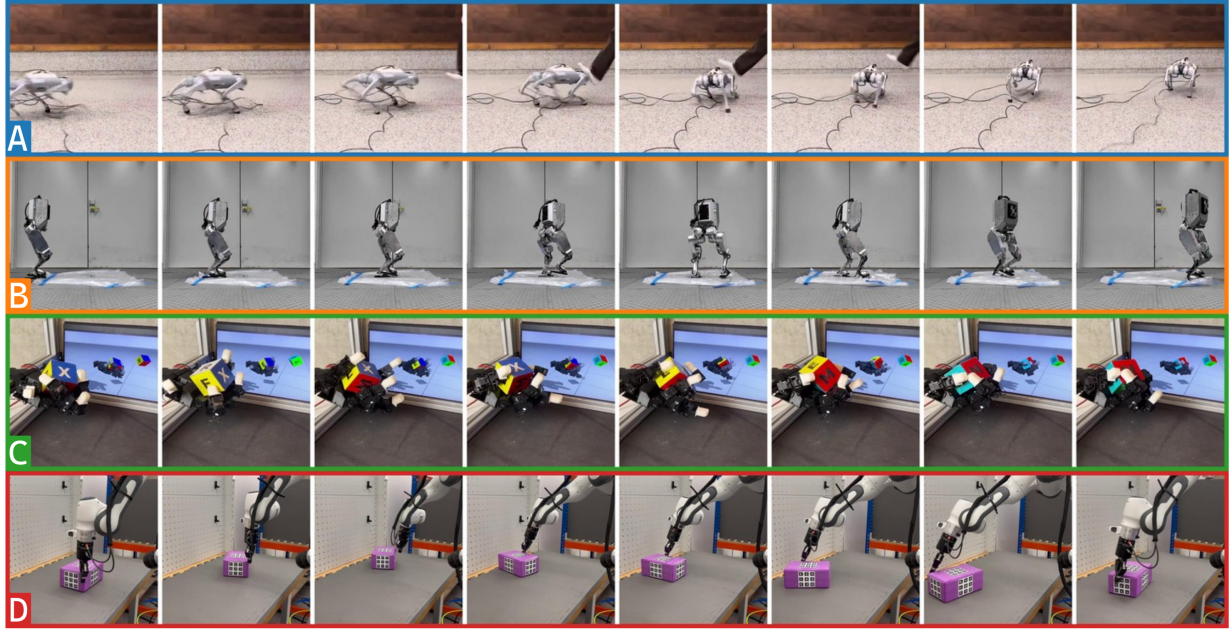


Figure 4.4: Footage from four of our deployed policies. a) Go1 joystick policy recovering from a kick while travelling at  $\sim 2\text{m/s}$ , b) Berkeley humanoid joystick policy tracking an angular velocity command on a slippery surface. c) In-Hand Cube Reorientation transitioning between two target poses. d) Non-prehensile policy issuing torque commands to rotate a block by 180 degrees.

**Hardware** We deploy on the *Unitree Go1*, which is a quadruped robot with four legs, each possessing three degrees of freedom. Trained policies run on real-world outdoor terrain (grass and concrete) and indoor surfaces with different friction properties.

**Training** We domain randomize for sensor noise, dynamics properties and task uncertainties. We first train the policy on flat ground with restricted command ranges within 5 minutes ( $2 \times \text{RTX 4090}$ ) and finetune it on rough terrain with wider ranges.

**Results** All four policies (joystick, handstand, footstand, and fall recovery) transfer robustly from simulation to reality, coping with uneven terrain and moderate external perturbations without additional fine-tuning. Videos of these deployments are provided on our project website.

## Humanoid Locomotion

**Task definition** We implement the same joystick locomotion task as shown for the quadruped environment.

**Hardware** We perform sim-to-real experiments on three different humanoid platforms: a) *Berkeley Humanoid* [54], a low-cost, lightweight bipedal robot with 6 DoF per leg, b) *Unitree G1*, a humanoid robot featuring 29 DoF in total, and c) *Booster T1*, a small-scale humanoid robot with 23 Dof. All systems are evaluated in indoor environments, with slight variations in surface friction and ground compliance.

**Training** We follow the domain randomization and finetuning strategies of the quadruped robot. Training on flat ground lasts under 15 minutes for the Berkeley Humanoid, and under 30 minutes for the Unitree G1 and the Booster T1 on two RTX 4090.

**Results** We successfully deploy joystick-based locomotion on the Berkeley Humanoid, demonstrating robust tracking of velocity commands on surfaces ranging from rigid floors to soft and slippery terrains. On the Unitree G1 and Booster T1, our zero-shot policy similarly achieves stable walking and turning on standard indoor floors. Although minor tuning for each platform’s unique dynamics may further enhance performance, these results confirm that our approach generalizes across a range of legged robot morphologies.

## Manipulation

In this section, we present sim-to-real results for a broad range of manipulation tasks, including dexterous in-hand manipulation, non-prehensile manipulation, and vision-based grasping. These tasks illustrate Playground’s ability to address a diverse segment of the manipulation spectrum and highlight its robust deployment in real-world settings.

### In-Hand Cube Reorientation

**Task definition** We implement an in-hand cube reorientation task using the low-cost, dexterous LEAP hand platform [55], closely following previous works on in-hand manipulation [3, 4]. The task involves reorienting a 7 cm cube repeatedly from random initial poses to new target orientations in  $SE(3)$  without dropping it.

**Hardware** We employ the same hardware configuration as in [62], mounting the LEAP hand on an 80/20 frame with a 3D-printed bracket that tilts the palm downward by  $20^\circ$ . A single Intel RealSense D415 camera, positioned above the workspace, provides pose estimates of the cube via a pretrained detector [4]. Although occlusions can introduce observation noise, we leave multi-camera extensions to future work. The policy operates at 20 Hz, which remains comfortably below the USB-Dynamixel control bandwidth.

**Training** To promote sim-to-real transfer, we apply domain randomization on the robot parameters as well as cube mass and friction. We also include sensor noise, and we finetune with a progressive curriculum to increase both noisy pose estimates and action regularization. The policy trains within 30 min on two RTX 4090 GPUs.



Table 4.1: In-hand reorientation results on the LEAP hand over 10 trials, reporting the number of consecutive successful rotations before failure. The final two columns show the median and mean of the #Rotations metric.

|             | Trial |    |   |   |    |   |   |   |   |    | Summary    |            |
|-------------|-------|----|---|---|----|---|---|---|---|----|------------|------------|
|             | 1     | 2  | 3 | 4 | 5  | 6 | 7 | 8 | 9 | 10 | Median     | Mean       |
| # Rotations | 3     | 27 | 8 | 2 | 15 | 3 | 4 | 1 | 3 | 5  | <b>3.5</b> | <b>7.1</b> |

**Results** As summarized in Table 4.1, our learned policy demonstrates early signs of robust in-hand reorientation with MuJoCo Playground. The most frequent failure occurs when the cube becomes wedged in the space present between the fingers and the palm of the LEAP hand, causing the policy to stall. Although less common, we also observe accidental interlocking of the index and thumb, attributed to physical flex in the low-cost hardware. Videos of real-world deployments can be found on our project page. We note that improved camera coverage and more accurate collision geometries could mitigate these edge-case failures, which we leave for future work.

### Non-Prehensile Block Reorientation

**Task definition** We present a sim-to-real setup for non-prehensile reorientation of a yoga block on a commonly available Franka Emika Panda robot arm with a Robotiq gripper, achieving high zero-shot success. The task involves moving a yoga block from a random initial pose in the robot’s workspace to a fixed goal pose. A trial is deemed successful if the agent reorients the block within 3 cm of the goal position and within  $10^\circ$  of the desired orientation.

**Hardware** The policy receives estimates of the block’s position and orientation from an open-source camera tracker [63]. We use direct high-frequency torque control at 200 Hz, where the RL policy outputs motor torques for the arm’s seven joints (with the gripper closed). By learning to control torques rather than joint positions, the agent develops smooth, compliant behavior that transfers effectively to hardware, delivering superior performance even when direct torque control at high frequencies poses learning challenges [64]. This recipe, therefore, holds broad value for practitioners.

**Training** Robust zero-shot transfer is enabled by stochastic delays and progressive curriculum learning. Each training episode injects randomization into initial poses, joint positions, and velocities, while also imposing action and observation stochastic delays to mirror practical hardware latency. A simple curriculum gradually increases the block’s displacement and orientation range upon each success, preventing overfitting to easier conditions. Training takes 10 minutes on 16x A100 devices.

Table 4.2: Sim-to-real reorientation performance on the Franka Emika Panda robot, evaluated across 35 hardware trials. Each metric is reported as the median and mean (with a 95% confidence interval). The success rate is bolded to highlight final task performance. The training was done on 16x A100 GPUs.

| Metric                                        | Median     | Mean $\pm$ 95% Confidence Interval |
|-----------------------------------------------|------------|------------------------------------|
| <b>Real Success (%) <math>\uparrow</math></b> | <b>100</b> | <b><math>85.7 \pm 12.2</math></b>  |
| Position Error (cm) $\downarrow$              | 1.95       | $5.28 \pm 3.26$                    |
| Rotation Error ( $^\circ$ ) $\downarrow$      | 1.72       | $3.32 \pm 1.59$                    |

**Results** These techniques, combined with 200 Hz direct torque control, produce a policy resilient to real-world perturbations. The agent reliably reorients the block on hardware with no additional fine-tuning. Videos of real-world deployments are provided on our project website.

### Pick-Cube from Pixels

**Task definition** We demonstrate sim-to-real transfer with pixel-based policies on a Franka Emika Panda robot. The robot must reliably grasp and lift a small  $2 \times 2 \times 3$  cm block from a random location on the table and move it 10 cm above the surface. The policy receives a  $64 \times 64$  RGB image as input and outputs a Cartesian command, which is processed by a closed-form inverse kinematics solution to yield joint commands. To simplify the task, we restrict the end-effector to a 2D Y-Z plane (while always pointing downward) and provide a binary jaw open/close action.

**Hardware** We use a Franka Emika Panda robot with a single Intel RealSense D435 camera mounted to capture top-down RGB images. The policy operates at 15 Hz, and we run inference on an RTX 3090 GPU. Our setup ensures that the block starts within the field of view over a 20 cm range along the y-axis.

**Training** To bridge the sim-to-real gap, we apply domain randomization across visual properties such as lighting, shadows, camera pose, and object colors. We also add random brightness post-processing, and introduce a stochastic gripping delay of up to 250 ms. We choose a reduced action dimension of three (Y-movement, Z-movement, and discrete jaw control) for training sample efficiency, but we have found that the task can also be solved in full Cartesian or joint space given additional camera perspectives and more training samples. Training in simulation takes ten minutes on a single RTX 4090.

**Results** Our policy achieves a 100% success rate in 12 real-world trials, robustly grasping the block and lifting it clear of the table. It demonstrates resilience to moderate variations in lighting and minor camera shaking, as shown in the videos on our project website.

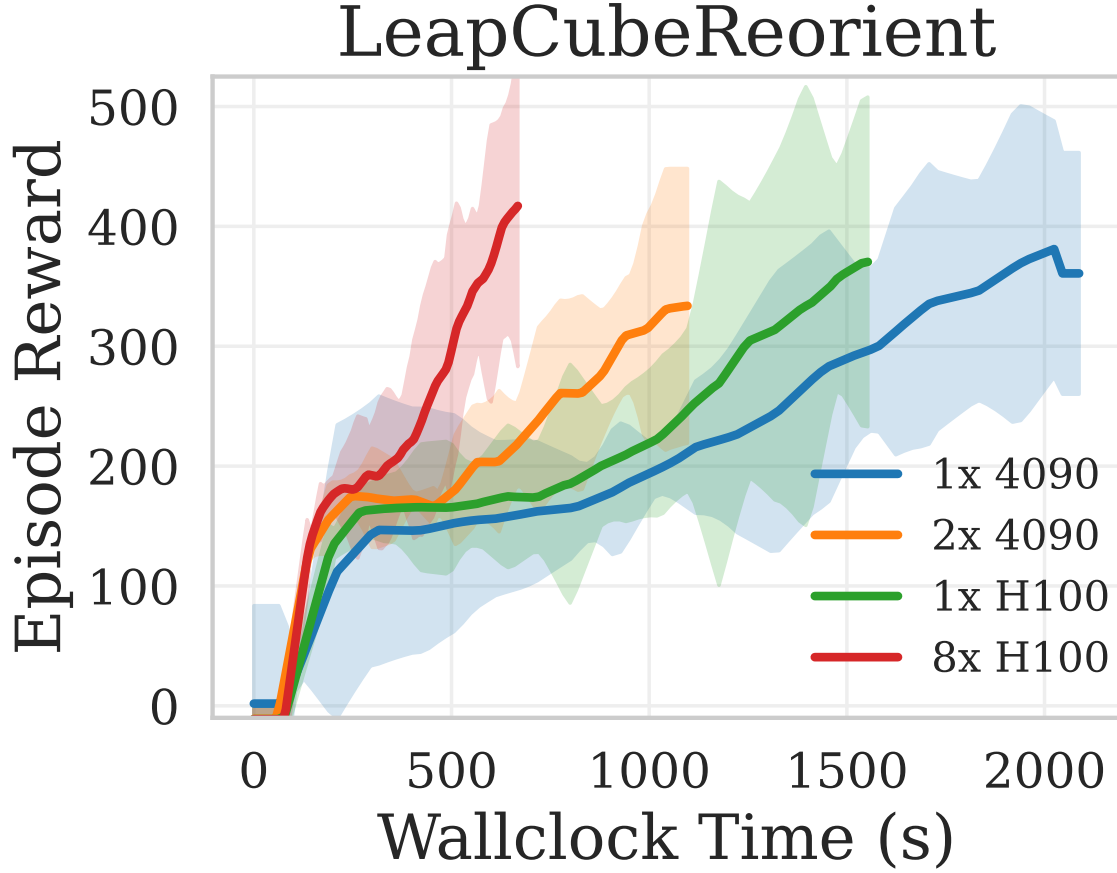


Figure 4.5: Training wallclock time for LeapCubeReorient on different GPU device topologies. 1x 4090 takes  $\sim 2080$  (s) to train and 8x H100 takes  $\sim 670$  (s) to train. All runs use the *same* hyperparams (e.g. 8192 num envs); we leave tuning hyperparams per topology as a future exercise.

These findings highlight MuJoCo Playground’s capacity for training pixel-based policies that transfer reliably to real hardware in a zero-shot manner.

## Training Throughput

Across our sim-to-real studies, we used several GPU hardware setups and topologies, including NVIDIA RTX 4090, A100, and H100 GPUs. In Figure 4.5, we break down the training performance of the LeapCubeReorient environment on different configurations for a fixed set of RL hyper-parameters, demonstrating that MJX is effective on both consumer-grade and datacenter graphics cards. We see that GPUs with higher theoretical performance and larger topologies can reduce training time by a factor of 3x on a contact-rich task like in-

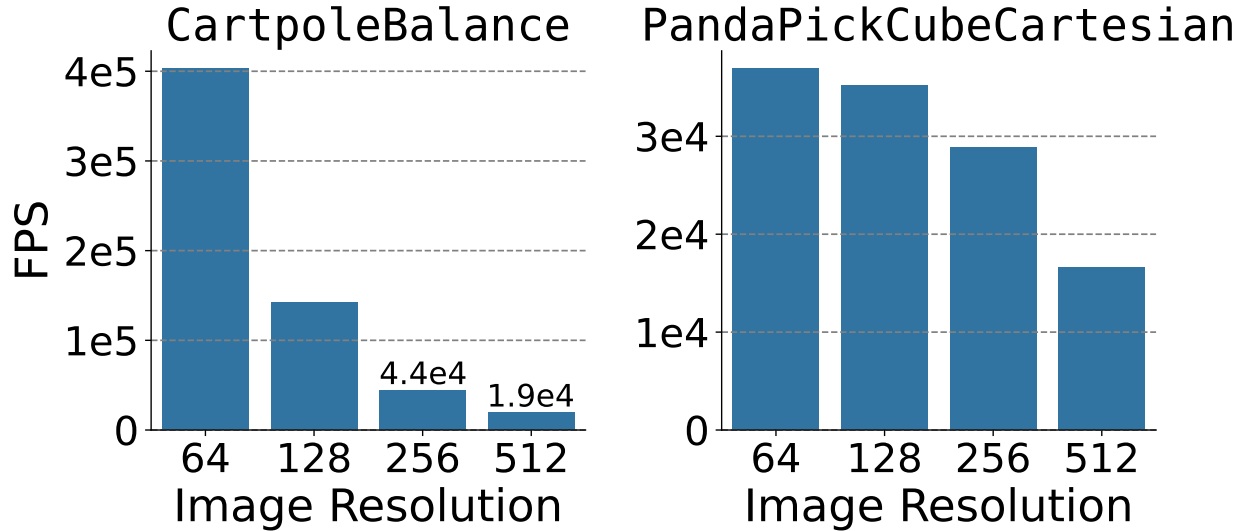


Figure 4.6: Environment steps per second on the single-camera CartpoleBalance and PandaPickCubeCartesian environments with pixel-based observations from our on-device renderer.

hand reorientation. We leave optimization of topology-specific hyper-parameters as future work (for example, the number of environments should ideally increase for larger topologies to maximize throughput, as long as the RL algorithm can utilize the increase in data per epoch).

### Training Throughput with Batch Rendering

Figure 4.6 highlights the throughput of stepping two of our environments with pixel observations at different resolutions. By pairing MJX physics with Madrona batch rendering, our Cartpole and Franka environments unroll at roughly 403,000 and 37,000 steps per second respectively. Note that our Franka physics are over 20x more costly than Cartpole’s, resulting in the lower sensitivity of FPS to image resolution.

Computationally, pixel-based policy training generally involves four main components: physics simulation, observation rendering, policy inference and policy updates. Figure 4.6 only encapsulates the former two and is not fully indicative of overall training throughput.

We find that in the context of a PPO training loop, physics, rendering, and inference together only comprise 9% and 43% of the Cartpole and Franka total training times, respectively, with most of the time spent updating the expensive CNN-based networks. Hence, compared to traditional on-policy training pipelines, we have shifted our bottleneck from collecting data to processing it.

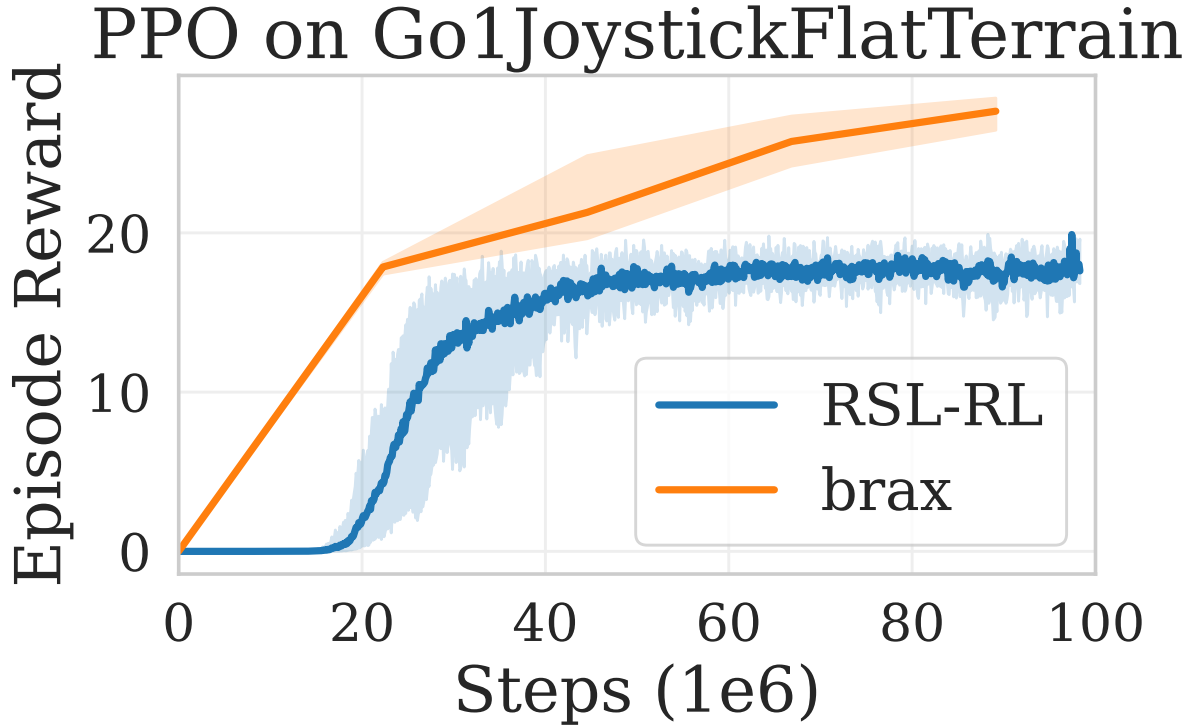


Figure 4.7: Reward curves for PPO trained with RSL-RL and brax on an RTX-4090 GPU for 3 seeds each on the Unitree Go1.

### RL Libraries

While MuJoCo Playground primarily uses a JAX-based physics simulator, practitioners are able to use both JAX and torch-based RL libraries for training RL agents. In Figure 4.7, we show reward curves for PPO agents trained using both Brax [46] and RSL-RL [65] implementations. Each corresponding RL library is trained with custom hyperparameters tailored to the corresponding PPO implementation. Both libraries are able to achieve successful rewards and gaits within similar wallclock times. All other results in this chapter were obtained using the Brax PPO and SAC implementations.

## 4.5 Related Work

**Physics simulation on GPU** The PhysX GPU implementation [9] has been heavily relied on for robotic sim-to-real workloads via IsaacGym [7] and more recently Isaac Lab [8]. The PhysX GPU implementation, however, is closed-source [9] and researchers lack the ability to extend the simulator for their specific tasks or workloads. Several GPU-based

physics engines are open-source, such as MJX [10, 49], Brax [46], Warp [66], and Taichi [67]. Only a limited set of robot environments [68, 69] leverage these open-source counterparts, in contrast to the wide range of robotic sim-to-real results that were achieved with IsaacGym and Isaac Lab. Most recently, Genesis [70] provides a rigid-body implementation similar to MJX implemented using Taichi, that allows for dynamic constraints/contacts. However, sim-to-real results are still limited to a few locomotion policies.

**Sim-to-real RL** A variety of locomotion and manipulation policies have successfully been deployed in the real world zero-shot [5, 71–76]. We complement these results by demonstrating zero-shot sim-to-real on the Leap Hand, Unitree Go1, Berkeley Humanoid, Unitree G1, Booster T1, and Franka arm using MuJoCo rather than closed-source simulators. Similar to [7, 8], we provide code for environments and training.

**Vision-based RL** State-of-the-art algorithms such as DrQ [77], RL from Augmented Data (RAD) [78], Dreamerv3 [79], TD-MPC2 [80], and EfficientZeroV2 [81] have pushed pixel-based RL performance over the years. Transferring these advances to the real world is appealing, as visual control loops offer precise positioning and robust behaviour in uncontrolled in-the-wild scenarios [82]. The limitation of training directly from pixel data is the large visual sim-to-real gap between simulation and reality, which is often overcome using domain randomization [37]. However, such training methods require exponentially more training samples. As a result, policies are typically trained with proprioceptive observations in simulation and subsequently distilled into vision-based policies offline [71, 83, 84], or trained with smaller exteroceptive observations [2, 51]. With Madrona, we are able to train vision-based policies directly in simulation without a distillation step using high-throughput batch rendering, similar to [8] and [85].

## 4.6 Limitations

MuJoCo Playground inherits the [limitations of MJX](#) due to constraints imposed by JAX. First, just-in-time (JIT) compilation can be slow (1-3 minutes on Playground’s tasks). Second, computation time related to contacts does not scale like the number of *active* contacts in the scene, but like the number of *possible* contacts in the scene. This is due to JAX’s requirement of static shapes at compile time. This limitation can be overcome by using more flexible frameworks like Warp [66] and Taichi [70]. This upgrade is an active area of development. Finally we should note that the vision-based training using Madrona is still at an early stage.

## 4.7 Conclusion

MuJoCo Playground is a library built upon the open-source MuJoCo simulator and Madrona batch renderer with implementations across several reinforcement learning and robotics environments. We demonstrate policy training on various GPU topologies using JAX and pytorch-based reinforcement learning libraries. We also demonstrate sim-to-real deployment on several robotic tasks and embodiments, from locomotion to both dexterous and non-prehensile manipulation from proprioceptive state and from pixels. We look forward to seeing the community put this resource to use in advancing robotics research and its applications.

# Chapter 5

## mjlab: A Composable Reinforcement Learning Framework

MuJoCo Playground, introduced in Chapter 4, demonstrated that a single sim-to-real recipe transfers across six robotic platforms on an open, inspectable stack. Two limitations of its design constrain how far this extends: MJX’s static tensor shapes penalize contact-heavy scenes, and monolithic environment definitions accumulate duplication across robots and task families. This chapter presents `mjlab`, a manager-based reinforcement learning framework built on MuJoCo Warp that addresses both.

### 5.1 Introduction

At the physics layer, `mjlab` is built on MuJoCo Warp [86], a GPU implementation of MuJoCo built on NVIDIA Warp [66] rather than JAX. Warp’s CUDA-backed kernels remove the static-shape constraint behind MJX’s contact-scaling penalty, and `mjlab` additionally captures the simulation step as a CUDA graph so that the kernel sequence replays without per-step dispatch overhead. At the API layer, `mjlab` adopts the manager-based design introduced by Isaac Lab [87], in which environments are composed from self-contained terms for observations, rewards, events, commands, and curricula rather than from a single monolithic step function. The result is a framework with Isaac Lab’s structural advantages but without its Omniverse runtime dependency or closed-source physics, running on Warp in place of MJX and exposing a PyTorch [88]-native interface for policy training. `mjlab` ships with three robot morphologies and three reference tasks.

The remainder of this chapter describes the design philosophy (Section 5.2), architecture (Section 5.3), core components (Section 5.4), the manager-based API (Section 5.5), shipped tasks (Section 5.6), and software design (Section 5.7).



## 5.2 Design Philosophy and Scope

`mjlab` is designed around three core engineering commitments: (1) minimal installation friction, (2) transparent and inspectable physics, and (3) tight integration with the MuJoCo ecosystem. These commitments motivate deliberate trade-offs in implementation and scope, which we make explicit below.

**Physics backend.** `mjlab` targets a single physics stack, MuJoCo Warp, to prioritize simulation transparency and debuggability. The framework exposes MuJoCo-native `MjModel` and `MjData` structures for direct inspection and state access. A non-goal is cross-simulator portability [89]; `mjlab` favors precise control and interpretability over backend generality.

**Sensing and rendering.** `mjlab` provides ray-cast, depth, and RGB camera sensors for geometric and visual perception. Photorealistic rendering is out of scope. This does not preclude vision-based policies: a common approach is to train privileged policies using full state, then distill into vision-based controllers using external rendering.

**Environment composition.** `mjlab` adopts Isaac Lab’s manager-based API, where environments are built from reusable, composable components rather than monolithic definitions. Modular terms for observations, rewards, events, and curricula reduce code duplication across tasks. The implementation remains fully MuJoCo-native; users work directly with MuJoCo models and conventions.

**Scope and extensibility.** `mjlab` provides infrastructure for rigid-body robot learning that is intended to be extended to custom robots, tasks, sensors, and actuators. The framework includes three example robots and tasks that demonstrate use of the API.

Together, these choices reflect a focus on fast iteration, transparent physics, and MuJoCo-native workflows, rather than feature completeness or broad simulator support.

## 5.3 Architecture

**MuJoCo Warp.** MuJoCo Warp [86] ([docs](#)) is a GPU-accelerated backend for MuJoCo built on [NVIDIA Warp](#) [66]. It preserves MuJoCo’s [10] familiar `MjModel`/`MjData` paradigm: `MjModel` holds the static kinematic and dynamic description of the scene, while `MjData` carries the time-varying simulation state. The key addition is a leading *world* dimension: a single `MjData` object holds the state of  $N$  independent simulation instances in parallel, enabling thousands of environments to be stepped in parallel. Model parameters are shared across all worlds by default, and individual fields can be expanded to vary per-world when domain randomization requires it. `mjlab` further captures the simulation step as a CUDA graph: the kernel execution sequence is recorded once and replayed on subsequent calls, eliminating CPU-side dispatch overhead.

**Scene pipeline.** `mjlab` constructs scenes by composing entity descriptions defined via MJCF into a single `MjSpec`. This specification is compiled into an `MjModel` on the CPU, then transferred to the GPU via MuJoCo Warp’s `put_model` and `put_data` routines.

**Manager-based paradigm.** On top of the simulation layer, `mjlab` adopts the manager-based environment design introduced by Isaac Lab. Users define their environment by composing small, self-contained *terms* (reward functions, observation computations, domain randomization events) and register them with the appropriate manager. Each manager handles the lifecycle of its terms: calling them at the right point in the simulation loop, aggregating their outputs, and exposing diagnostics. Terms can be plain functions for stateless computations or classes that inherit from `ManagerTermBase` when they need to cache expensive setup (e.g., resolving regex patterns to joint indices at initialization) or maintain state across steps with a per-episode `reset()` hook. This abstraction introduces overhead compared to writing monolithic step functions, but benefits in modularity, testability, and rapid iteration outweigh the cost, a lesson reinforced by our experience with the more minimal MuJoCo Playground API, where the absence of such structure made environments harder to maintain and extend.

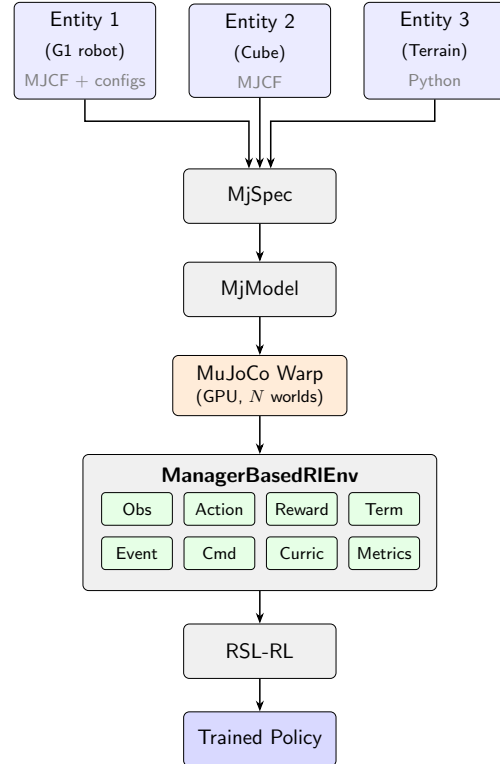


Figure 5.1: Entities are composed into an `MjSpec`, compiled, and transferred to MuJoCo Warp for GPU simulation. The `ManagerBasedRLEnv` orchestrates the MDP; `RSL-RL` handles training.

## 5.4 Components

`mjlab` provides a set of core components that managers operate on during the simulation lifecycle.

**Entities.** An entity represents a physical object in the scene, such as robots, manipulated objects, or static fixtures. While Isaac Lab splits this concept across multiple specialized classes (`Articulation`, `RigidObject`, `DeformableObject`, `RigidObjectCollection`, and others), `mjlab` uses a single `Entity` class that covers all variations through two independent properties: base type (fixed or floating) and articulation (with or without joints). Runtime queries such as `is_fixed_base` and `is_articulated` let downstream code branch where

needed without requiring a class hierarchy. Each entity maintains an `EntityData` structure that aggregates kinematic quantities such as body positions, orientations, and velocities, making them available to observation and reward terms without redundant computation.

**Sensors.** Sensors in `mjlab` follow a layered design. Users can wrap sensors already defined in their MJCF files and read them through `MuJoCo`’s native `sensordata` buffer, or define them entirely in Python configuration. On top of this native layer, `mjlab` provides custom sensor types that go beyond what `MuJoCo` offers: a BVH-accelerated ray-cast sensor that supports grid and pinhole ray patterns and returns per-ray distances, hit positions, and surface normals; a contact sensor that tracks collision forces between body pairs with air-time and force-history bookkeeping; and a GPU-accelerated camera sensor that renders RGB and depth across all worlds in parallel and supports both perspective and orthographic projection. All sensors implement a common interface with per-step caching and optional debug visualization.

**Actuators.** Actuators follow the same layered philosophy. Users can wrap actuators defined in MJCF, preserving the exact behavior specified in the model file, or create `MuJoCo`-native actuator types (motor, position, velocity, muscle) from configuration. Beyond these native options, `mjlab` provides custom implementations that compute torques on the GPU outside of `MuJoCo`: an ideal PD controller, a DC motor model with velocity-dependent torque saturation, and a learned MLP actuator for capturing hardware-specific dynamics from data. All three tiers coexist within the same entity. Actuation delay, common in real robots, is modeled by a wrapper class that buffers control signals and replays them with a latency quantized to the physics timestep.

**Terrain.** The terrain module generates ground geometry for locomotion training. Users can spawn a flat ground plane or compose a grid of sub-terrain patches, each drawn from a library of terrain types. Two families are provided: primitive terrains built from `MuJoCo` box geoms (flat surfaces, stairs, stepping stones, narrow beams, and random-height grids) and heightfield terrains for smoother, continuous profiles (sloped pyramids, Perlin noise, and sinusoidal waves). Every terrain type accepts a difficulty parameter that scales its features (step height, slope angle, noise amplitude) from trivial to challenging. The grid generator supports two modes: random, where each patch is sampled independently, and curriculum, where rows encode increasing difficulty so that the curriculum manager can advance robots to harder terrain as performance improves. Users can subclass the base terrain configuration to define custom terrain types (Figure 5.2).

**Viewer.** `mjlab` supports two visualization backends. The native `MuJoCo` viewer provides the full `simulate` experience: pause and resume, contact point and force visualization, interactive perturbations, and more. For headless servers or remote machines, `mjlab` offers a web-based viewer built on Viser [90] that supports pause, resume, and contact visualization in the browser.

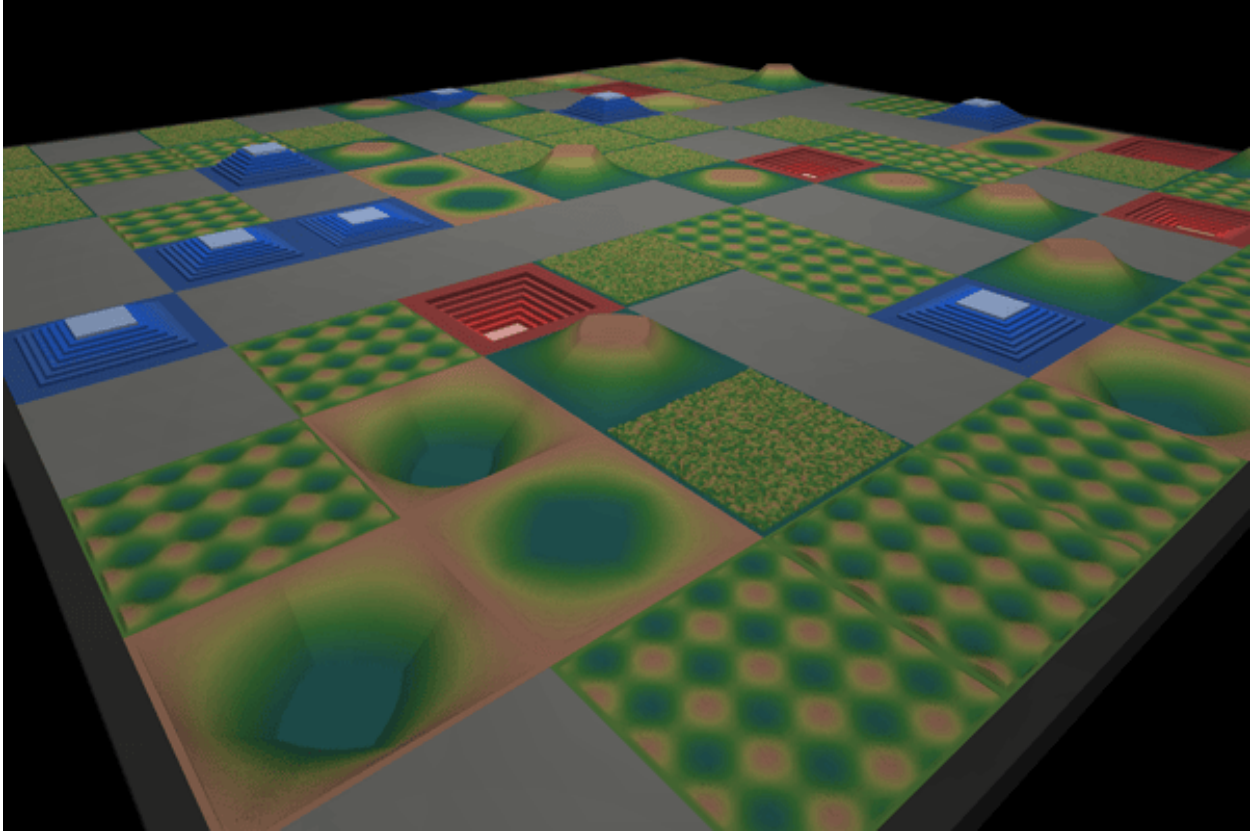


Figure 5.2: A composite terrain grid generated by `mjlab`. Sub-terrain patches include flat ground, stairs, sloped pyramids, heightfield noise, and wave patterns. Difficulty increases along one axis for curriculum-based training.

## 5.5 Manager-Based API

`mjlab` environments implement the Gym [91] interface, a standard API for defining Markov decision processes (MDPs). Environments expose two methods: `reset()` initializes a new episode and returns the first observation, while `step()` applies an action and returns the next observation, reward, and termination signal. Each call to `step()` proceeds through a fixed pipeline:

1. **Act.** The action manager updates the action history and splits the policy output into per-term slices.
2. **Simulate.** For each of  $d$  decimation sub-steps: apply actuator commands, write controls to the simulation, advance physics, and update entity state.
3. **Terminate.** The termination manager evaluates stop conditions and flags environments for reset.
4. **Reward.** The reward manager computes and aggregates weighted reward terms.
5. **Reset.** Flagged environments are reset: the curriculum manager updates difficulty, entity

states are reinitialized, and the event manager applies reset hooks.

6. **Command.** The command manager generates or resamples goal signals.
7. **Apply Events.** The event manager applies periodic events (e.g., external pushes).
8. **Observe.** The observation manager computes observations for the next policy query.

This manager-based orchestration follows the design introduced by Isaac Lab [87], where each manager encapsulates a specific aspect of the MDP and users compose environments by registering modular terms.

**Action manager.** The action manager receives the policy’s output tensor, splits it according to the registered action terms, and routes each slice to the appropriate entity’s actuators. It supports action clipping and tracks the two most recent actions, which observation and reward terms can reference (e.g., to compute an action rate penalty). Shipped action terms include joint-space position and velocity targets as well as a differential inverse kinematics term that converts task-space pose commands into joint displacements via damped least-squares, with optional null-space posture regularization and soft joint-limit avoidance.

**Termination manager.** The termination manager evaluates a set of Boolean conditions each step (contact with illegal bodies, joint limit violations, timeout) and flags environments for reset. It also monitors simulation state for NaN and Inf values, catching numerical instabilities before they propagate. When an instability is detected, `mjlab` captures a rolling buffer of recent states that users can replay in an interactive viewer to diagnose the failure. The observation and reward managers additionally report which specific terms produced invalid values, narrowing the search from “something exploded” to the exact offending computation.

**Reward manager.** Reward terms are functions or stateful classes that return per-environment scalars. The manager computes a weighted sum of all active terms, automatically scaling each by the environment’s control timestep to make reward magnitudes invariant to the simulation frequency. Per-term episodic sums are tracked and logged, making it straightforward to diagnose which terms dominate training. Terms that produce NaN or Inf values are detected and reported individually.

**Curriculum manager.** The curriculum manager adjusts training conditions based on policy performance. Examples include scaling reward term weights, widening command velocity ranges, or advancing robots to harder terrain rows as success rates improve.

**Event manager.** Events are hooks that execute at specific points in the environment lifecycle: at startup, on reset, or at fixed intervals during training. The most common use case is domain randomization. `mjlab` provides a domain randomization module covering body, joint, geometry, camera, light, material, and tendon parameters, each with configurable operations (absolute, additive, multiplicative) and sampling distributions (uniform, log-uniform, Gaussian). When an event modifies a field of the `MuJoCo` model, `mjlab` expands that field from a shared value to a per-world array and rebuilds the CUDA capture graph to reflect the new memory layout. For inertial parameters, the module implements physically consistent randomization based on the pseudo-inertia parameterization of Rucker



Figure 5.3: The three robot morphologies shipped with `mjlab`: Unitree G1 humanoid (left), Unitree Go1 quadruped (center), and YAM robot arm (right). All models are adapted from MuJoCo Menagerie [12].

and Wensing [22], guaranteeing that randomized mass, center of mass, and rotational inertia remain fully physically consistent for any perturbation magnitude. The breadth of randomizable parameters, from link geometry and inertial properties to joint dynamics and actuator gains, combined with physical consistency, makes the module applicable beyond sim-to-real transfer to problems such as system identification and hardware co-design.

**Command manager.** The command manager generates goal signals for the policy (velocity targets, tracking waypoints, or object poses) and resamples them at configurable intervals. Commands are exposed to the observation manager so the policy can condition on the current goal.

**Observation manager.** Observation terms are composable functions that read from entity data, sensors, or commands and return tensors. The manager chains these terms through a configurable pipeline: raw values are optionally clipped, scaled, corrupted with noise, delayed by a configurable number of control steps, and concatenated with a history buffer. Multiple observation groups (e.g., *policy* and *critic*) can coexist, each with its own processing pipeline, enabling asymmetric actor-critic architectures.

**Metrics manager.** The metrics manager computes and logs custom per-episode metrics that are not tied to any specific manager, such as task success rates or tracking errors, providing a unified interface for monitoring training progress.

## 5.6 Tasks

`mjlab` ships with three reference tasks spanning locomotion, whole-body control, and manipulation. The codebase is intentionally lean: three robot morphologies, a humanoid (Unitree G1), a quadruped (Unitree Go1), and a robot arm (YAM), cover a broad range of research use cases while keeping the repository easy to navigate and maintain.

### Locomotion: Velocity Tracking

The velocity tracking task trains a locomotion controller to follow linear and angular velocity commands on flat or rough terrain. The policy observes IMU readings, projected gravity,



joint positions and velocities, the previous action, and the commanded twist. Reward terms encourage accurate velocity tracking while penalizing excessive body angular velocity, angular momentum, joint limit violations, action rate, and foot slip. Terrain variants range from flat ground to composite grids of stairs and rough heightfields, with terrain difficulty tied to the curriculum manager. The reward structure produces a natural running gait on the Unitree G1 humanoid both in simulation and after zero-shot transfer to real hardware.

## Whole-Body Control: Motion Imitation

The motion imitation task trains a humanoid (Unitree G1) to track reference motion clips, implementing the DeepMimic [92] framework with extensions from BeyondMimic [93]. The policy observes an anchor pose from the reference trajectory, base velocities, joint states, and the current action. Reward terms penalize deviations in global root position and orientation, relative body poses, and body velocities. A self-collision cost discourages interpenetration. Trained policies recover dynamic skills from the reference set, including a triple spin kick.

## Manipulation: Cube Lifting

The manipulation task trains a YAM robot arm to lift a cube to a target pose. The policy observes joint positions and velocities, the vector from end-effector to cube, the cube-to-goal error, and the previous action. A staged reward structure first guides the end-effector toward the cube, then rewards lifting the cube to the target height. Contact sensors on the end-effector and ground plane provide auxiliary signals.

## 5.7 Software Design

A framework’s long-term usability depends as much on how its code is organized as on what it can simulate. We made several deliberate departures from Isaac Lab’s software patterns to reduce boilerplate, catch errors earlier, and lower the barrier to writing new tasks.

**PyTorch-native interface.** MuJoCo Warp stores simulation state in Warp arrays, while policy training typically happens in PyTorch. `mjlab` bridges this gap with a `TorchArray` abstraction: a zero-copy wrapper that exposes Warp arrays as PyTorch tensors. Users write rewards, observations, and custom logic in pure PyTorch without touching Warp kernels, and the framework handles memory sharing transparently.

**Instance-based configuration.** Isaac Lab organizes environment parameters through deeply nested dataclass hierarchies: each new task inherits from a base config class and overrides fields via `__post_init__` mutations or field redeclarations. This pattern is fragile: `__post_init__` overrides can silently ignore constructor arguments, and a misspelled field name creates a new field rather than overriding the parent’s, with no error at any stage. `mjlab` replaces class-based configs with plain config *instances*. Observation, reward, event, and other term configurations are held in typed dictionaries (`dict[str, RewardTermCfg]`),

etc.) rather than in bespoke dataclass subclasses. Creating a task variant is a matter of copying an existing config and mutating it, not defining a new class. This eliminates the inheritance pitfalls while making configs easier to read, diff, and compose programmatically.

**CLI-first configuration.** Because configs are ordinary dataclass instances with typed fields, `mjlab` can expose every parameter at the command line automatically via `tyro`. Users override settings like reward weights, observation noise, simulation timestep with a single flag, without writing configuration files or subclassing.

**Co-located definitions.** Isaac Lab separates configuration dataclasses from their implementations: all manager term configs live in a single `manager_term_cfg.py`, while the corresponding classes are spread across other files. Configs must reference their implementation via a `class_type` field, adding indirection. `mjlab` keeps each config dataclass in the same file as its implementation (`ActionTermCfg` alongside `ActionTerm`, for example), so related code is always collocated and navigation is straightforward.

**Static typing and testing.** The entire codebase is statically type-checked with both `pyright` and `ty`, and ships a `py.typed` marker for downstream consumers. A test suite of over 40 files covers managers, sensors, actuators, domain randomization, task execution, and more.

**Minimal dependencies.** A guiding constraint throughout `mjlab`’s development was keeping installation friction-free. The framework avoids heavyweight dependencies, and is designed around `uv` [94], a fast Python package manager that eliminates explicit environment setup. Users can go from a fresh clone to a running training job in seconds:

```
git clone https://github.com/mujocolab/mjlab && cd mjlab
uv run train Mjlab-Velocity-Flat-Unitree-G1 \
  --env.scene.num-envs 4096 \
  --agent.max-iterations 10_000
```

Environment creation, installation, and dependency resolution happen automatically. Training uses RSL-RL [95] for on-policy algorithms, with multi-GPU scaling available via `torchrunx` [96].

## 5.8 Adoption

`mjlab` was released publicly in late 2025. In the months since, the public repository has accumulated over 2,000 GitHub stars and the framework has been cited in seven published works on humanoid motion tracking, retargeting, and teleoperation. It has been used in UC Berkeley’s graduate robotics course (ME 292b/193b), where students new to reinforcement learning trained their first legged locomotion policies, and at Boston Dynamics, Google, and PAL Robotics. External open-source forks have grown alongside, including the official Unitree RL environments for the Go2, G1, and H1, `pal_mjlab` for PAL Robotics platforms, `mjlab_myosuite` for musculoskeletal control, and `mjlab_upkie` for the Upkie wheeled biped.



# Part III

## Task Priors

## Chapter 6

# RoboPianist: Dexterous Piano Playing with Deep Reinforcement Learning

Grounded simulation and fast training infrastructure are not always sufficient. When the action space is high-dimensional and the reward signal is sparse, random exploration fails to discover useful behavior no matter how fast the simulator runs. This chapter presents *RoboPianist*, a benchmark and training system for dexterous bimanual piano playing, and shows that a fingering prior derived from human sheet-music annotations turns an otherwise intractable exploration problem into a tractable one.

### 6.1 Introduction

Despite decades-long research into replicating the dexterity of the human hand, high-dimensional control remains a grand challenge in robotics. This topic has inspired considerable research from both mechanical design [97–99] and control theoretic points of view [100–104]. Learning-based approaches have dominated the recent literature, demonstrating proficiency with in-hand cube orientation and manipulation [105–107] and have scaled to a wide variety of geometries [83, 108–110]. These tasks, however, correspond to a narrow set of dexterous behaviors relative to the breadth of human capabilities. In particular, most tasks are well-specified using a single goal state or termination condition, limiting the complexity of the solution space and often yielding unnatural-looking behaviors so long as they satisfy the goal state. How can we bestow robots with artificial embodied intelligence that exhibits the same precision and agility as the human motor control system?

In this work, we seek to challenge our methods with tasks commensurate with this complexity and with the goal of emergent human-like dexterous capabilities. To this end, we introduce a family of tasks where success exemplifies many of the properties that we seek in high-dimensional control policies. Our unique desiderata are (i) spatial and temporal

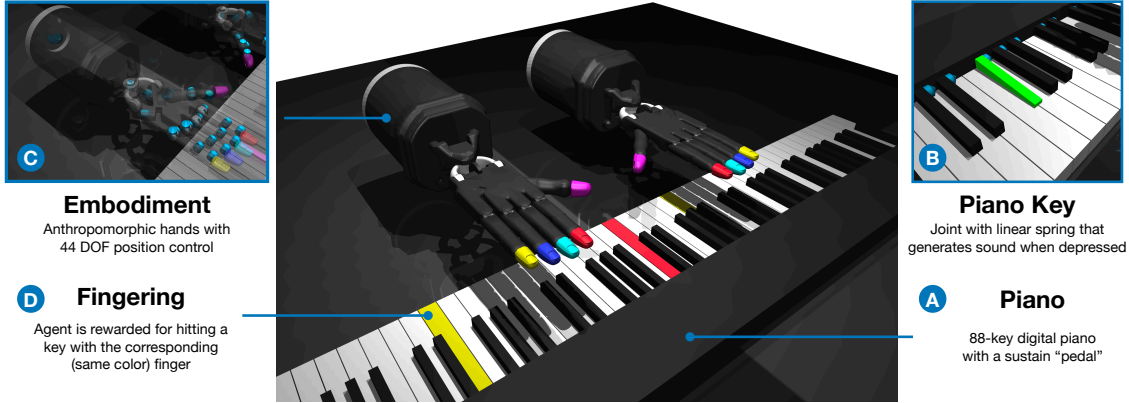


Figure 6.1: **RoboPianist** simulation featuring a full-size digital keyboard (A) with 88 piano keys modeled as linear springs (B). In the piano playing task, two (left and right) anthropomorphic Shadow hands (C) are tasked with playing a musical piece encoded as a trajectory of key presses (D).

precision, (ii) coordination, and (iii) planning. We thus built an anthropomorphic simulated robot system, consisting of two robot hands situated at a piano, whose goal is to play a variety of piano pieces, i.e., correctly pressing sequences of keys on a keyboard, conditioned on sheet music, in the form of a Musical Instrument Digital Interface (MIDI) transcription (see Figure 6.1). The robot hands exhibit high degrees of freedom (22 actuators per hand, for a total of 44), and are partially underactuated, akin to human hands. Controlling this system entails sequencing actions so that the hands are able to hit exactly the right notes at exactly the right times; simultaneously achieving multiple different goals, in this case, fingers on each hand hitting different notes without colliding; planning how to press keys in anticipation of how this would enable the hands to reach later notes under space and time constraints.

We propose **ROBOPIANIST**, an end-to-end system that leverages deep reinforcement learning (RL) to synthesize policies capable of playing a diverse repertoire of musical pieces on the piano. We show that a combination of careful system design and human priors (in the form of fingering annotations) is crucial to its performance. Furthermore, we introduce **ROBOPIANIST-REPERTOIRE-150**, a benchmark of 150 songs, which allows us to comprehensively evaluate our proposed system and show that it surpasses a strong model-based approach by over 83%. Finally, we demonstrate the effectiveness of multi-task imitation learning in training a single policy capable of playing multiple songs. To facilitate further research and provide a challenging benchmark for high-dimensional control, we open source the piano-playing environment along with **ROBOPIANIST-REPERTOIRE-150** at <https://kzakka.com/robopianist/>.

## 6.2 Related Work

We address related work within two primary areas: dexterous high-dimensional control, and robotic pianists.

**Dexterous Manipulation as a High-Dimensional Control Problem.** The vast majority of the manipulation literature uses lower-dimensional systems (i.e., single-arm, simple end-effectors), which circumvents challenges that arise in more complex systems. Specifically, only a handful of general-purpose policy optimization methods have been shown to work on high-dimensional hands, even for a single hand [83, 103, 105–107, 109, 111, 112], and of these, only a subset has demonstrated results in the real world [83, 105–107, 111]. Results with bi-manual hands are even rarer [110, 113]. In addition, the class of problems generally tackled in these settings corresponds to a definition of dexterity pertaining to traditional manipulation skills [114], such as re-orientation, relocation, manipulating simply-articulated objects (e.g., door-opening, ball throwing and catching), and using simple tools (e.g., hammer) [83, 107, 110, 115–117]. This effectively reduces the search space for controls to predominantly a single “basin-of-attraction” in behavior space per task. In contrast, our piano-playing task encompasses a more complex notion of a goal, extendable to arbitrary difficulty by only varying the musical score.

**Robotic Piano Playing.** Robotic pianists have a rich history within the literature, with several works dedicated to the design of specialized hardware [118–123], and/or customized controllers for playing back a song using pre-programmed commands [124, 125]. The works of Scholz [126], Yeon [127] use a dexterous hand to play the piano by leveraging a combination of inverse kinematics and offline trajectory planning. In Xu et al. [128], the authors formulate piano playing as an RL problem for a single Allegro hand on a miniature piano and leverage tactile sensor feedback. The piano playing tasks considered in these prior works are relatively simple (e.g., play up to six successive notes, or three successive chords with only two simultaneous keys pressed for each chord). On the other hand, ROBOPIANIST allows a general bi-manual controllable agent to emulate a pianist’s growing proficiency by providing a large range of musical pieces with graded difficulties.

## 6.3 Experimental Setup

In this section, we introduce the simulated piano-playing environment as well as the musical suite used to train and evaluate our agent.

**Simulation details.** We build our simulated piano-playing environment (depicted in Figure 6.1) using the open-source MuJoCo [10, 129] physics engine. The piano model is a full-size digital keyboard with 52 white keys and 36 black keys. We use a Kawai manual [130] as reference for the keys’ positioning and dimensions. Each key is modeled as a joint with a linear spring and is considered “active” when its joint position is within  $0.5^\circ$  of its maximum range, at which point a synthesizer is used to generate a musical note. We also implement a mechanism to sustain the sound of any currently active note to mimic the mechanical effect of a sustain pedal on a real piano. The left and right hands are Shadow Dexterous Hand [131] models from MuJoCo Menagerie [12], which have been designed to closely reproduce the kinematics of the human hand.

**Musical representation.** We use the Musical Instrument Digital Interface (MIDI) standard to represent a musical piece as a sequence of time-stamped messages corresponding to **note-on** or **note-off** events. A message carries additional pieces of information such as the pitch of a note and its velocity. We convert the MIDI file into a time-indexed note trajectory (a.k.a, piano roll), where each note is represented as a one-hot vector of length 88. This trajectory is used as the goal representation for the agent, informing it which keys must be pressed at each time step.

**Musical evaluation.** We use precision, recall, and F1 scores to evaluate the proficiency of our agent. These metrics are computed by comparing the state of the piano keys at every time step with the corresponding ground-truth state, averaged across all time steps. If at any given time there are keys that should be “on” and keys that should be “off”, precision measures how good the agent is at not hitting any of the “off” keys, while recall measures how good the agent is at hitting the “on” keys. The F1 score combines precision and recall into one metric, and ranges from 0 (if either precision or recall is 0) to 1 (perfect precision and recall). We primarily use the F1 score for our evaluations as it is a common heuristic accuracy score in the audio information retrieval literature [132], and we found empirically that it correlates with qualitative performance on our tasks.

**MDP formulation.** We model piano-playing as a finite-horizon Markov Decision Process (MDP) defined by a tuple  $(\mathcal{S}, \mathcal{A}, \rho, p, r, \gamma, H)$  where  $\mathcal{S} \subset \mathbb{R}^n$  is the state space,  $\mathcal{A} \subset \mathbb{R}^m$  is the action space,  $\rho(\cdot)$  is the initial state distribution,  $p(\cdot|s, a)$  governs the dynamics,  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  defines the rewards,  $\gamma \in [0, 1)$  is the discount factor, and  $H$  is the horizon. The goal of an agent is to maximize its total expected discounted reward over the horizon:  $\mathbb{E} \left[ \sum_{t=0}^H \gamma^t r(s_t, a_t) \right]$ .

The agent’s observations consist of proprioceptive and goal state information. The proprioceptive state contains hand and keyboard joint positions. The goal state information contains a vector of key goal states obtained by indexing the piano roll at the current time step, as well as a discrete vector indicating which fingers of the hands should be used at that timestep. To successfully play the piano, the agent must be aware of at least a few seconds’ worth of its next goals in order to be able to plan appropriately. Thus the goal state is stacked for some lookahead horizon  $L$ . A detailed description of the observation space is given in Table 6.1. The agent’s action is 45 dimensional and consists of target joint angles for the hand with an additional scalar value for the sustain pedal. The agent predicts target joint angles at 20Hz and the targets are converted to torques using PD controllers running at 500Hz. Since the reward function is crucial for learning performance, we discuss its design in Section 6.4.

| Observations               | Unit     | Size         |
|----------------------------|----------|--------------|
| Hand and forearm joints    | rad      | 52           |
| Forearm Cartesian position | m        | 6            |
| Piano key joints           | rad      | 88           |
| Active fingers             | discrete | $L \cdot 10$ |
| Piano key goal state       | discrete | $L \cdot 88$ |

Table 6.1: The agent’s observation space.  $L$  corresponds to the lookahead horizon.

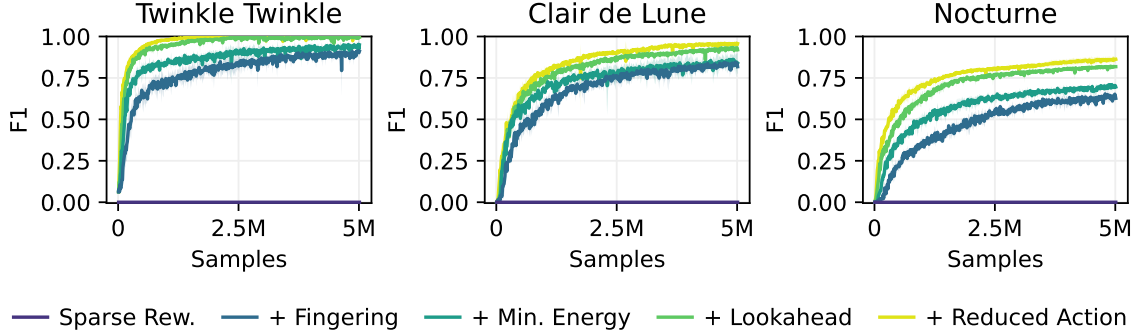


Figure 6.2: **RoboPianist Design Considerations.** The F1 performance for 3 songs of increasing difficulty. From left to right as specified by the legend, each curve **inherits all MDP attributes of the curve before it** but makes one additional modification as described by its label.

**Fingering labels and dataset.** Piano fingering refers to the assignment of fingers to notes, e.g., “C4 played by the index finger of the right hand”. Sheet music will typically provide sparse fingering labels for the tricky sections of a piece to help guide pianists, and pianists will often develop their own fingering preferences for a given piece. Since fingering labels aren’t available in MIDI files by default, we used annotations from the PIG dataset [133] to create a corpus of 150 annotated MIDI files for use in the simulated environment. Overall this dataset we call **REPertoire-150** contains piano pieces from 24 Western composers spanning baroque, classical and romantic periods. The pieces vary in difficulty, ranging from relatively easy (e.g., Mozart’s Piano Sonata K 545 in C major) to significantly harder (e.g., Scriabin’s Piano Sonata No. 5) and range in length from tens of seconds to over 3 minutes.

## 6.4 RoboPianist System Design

Our aim is to enable robots to exhibit sophisticated, high-dimensional control necessary for successfully performing challenging musical pieces. Mastery of the piano requires (i) spatial and temporal precision (hitting the right notes, at the right time), (ii) coordination (simultaneously achieving multiple different goals, in this case, fingers on each hand hitting different notes, without colliding), and (iii) planning (how a key is pressed should be conditioned on the expectation of how it would enable the policy to reach future notes). These behaviors do not emerge if we solely optimize with a sparse reward for pressing the right keys at the right times. The main challenge is exploration, which is further exacerbated by the high-dimensional nature of the control space.

We overcome this challenge with careful system design and human priors, which we detail in this section. The main results are illustrated in Figure 6.2. We pick 3 songs in increasing difficulty from **ROBOPIANIST-REPertoire-150**. We note that “Twinkle Twinkle” is the easiest while “Nocturne” is the hardest. We train for 5M samples, 3 seeds. We evaluate the F1 every 10K training steps for 1 episode (no stochasticity in the environment).

## Human priors

We found that the agent struggled to play the piano with a sparse reward signal due to the exploration challenge associated with the high-dimensional action space. To overcome this issue, we incorporated the fingering labels within the reward formulation. When we remove this prior and only reward the agent for the key press, the agent’s F1 stays at zero and no substantial learning progress is made. We suspect that the benefit of fingering comes not only from helping the agent achieve the current goal, but facilitating key presses in subsequent timesteps. Having the policy discover its own preferred fingering, like an experienced pianist, is an exciting direction for future research.

## Reward design

We first include a reward proportional to how depressed the keys that should be active are. We then add a constant penalty if any inactive keys are pressed hard enough to produce sound. This gives the agent some leeway to rest its fingers on inactive keys so long as they don’t produce sound. We found that giving a constant penalty regardless of the number of false positives was crucial for learning; otherwise, the agent would become too conservative and hover above the keyboard without pressing any keys. In contrast, the smooth reward for pressing active keys plays an important role in exploration by providing a dense learning signal. We introduce two additional shaping terms: (i) we encourage the fingers to be spatially close to the keys they need to press (as prescribed by the fingering labels) to help exploration, and (ii) we minimize energy expenditure, which reduces variance (across seeds) and erratic behavior control policies trained with RL are prone to generate. The total reward at a given time step is a weighted sum over the aforementioned components.

## Peeking into the future

We observe additional improvements in performance and variance from including future goal states in the observation, i.e., increasing the lookahead horizon  $L$ . Intuitively, this allows the policy to better plan for future notes – for example by placing the non-finger joints (e.g., the wrist) in a manner that allows more timely reaching of notes at the next timestep.

## Constraining the action space

To alleviate exploration even further, we explore disabling degrees of freedom [134] in the Shadow Hand that either do not exist in the human hand (e.g., the little finger being opposable) or are not strictly necessary for most songs. We additionally reduce the joint range of the thumb. While this speeds up learning considerably, we observe that with additional training time, the full action space eventually achieves similar F1 performance.

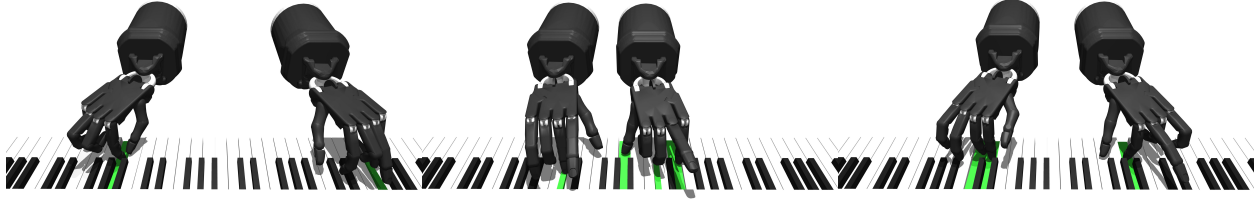


Figure 6.3: Policies in the ROBOPIANIST environment displaying skilled piano behaviors such as (left) simultaneously controlling both hands to reach for notes on opposite ends of the keyboard, (middle) playing a chord with the left hand by precisely and simultaneously hitting a note triplet, and (right) playing a trill with the right hand, which involves rapidly alternating between two adjacent notes.

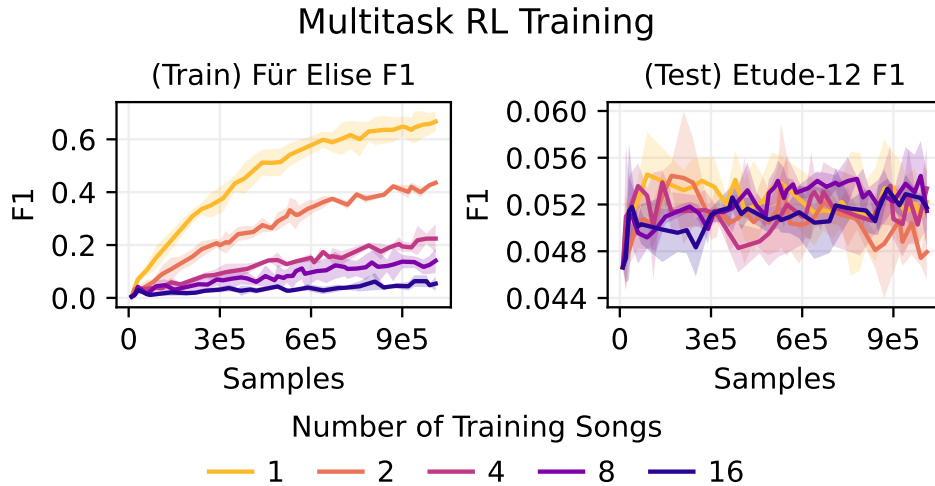


Figure 6.4: **Multi-song training with RL.** We test the ability to learn multiple songs simultaneously. The training performance on the shared song, Für Elise, is shown on the left. We see adding more songs degrades performance on individual songs. On the right, we evaluate on the held out ROBOPIANIST-ETUDE-12 songs, suggesting the difficulty of generalization with multitask RL training.

## 6.5 Results

In this section, we present our experimental findings on ROBOPIANIST-ETUDE-12, a subset of ROBOPIANIST-REPERTOIRE-150 consisting of 12 songs. We design our experiments to answer the following questions:

- (1) How does our method compare to a strong baseline in being able to play individual pieces?
- (2) How can we enable a single policy to learn to play more than one song?
- (3) What effects do our design decisions have on the feasibility of acquiring highly complex, dexterous control policies?



## Specialist Policy Learning

For our policy optimizer, we use a state-of-the-art model-free RL algorithm DroQ [135], one of several regularized variants of the widely-used Soft-Actor-Critic [136] algorithm. We evaluate online predictive control (MPC) as a meaningful point of comparison. Specifically, we use the implementation from Howell et al. [137] that leverages the physics engine as a dynamics model, and which was shown to solve previously-considered-challenging dexterous manipulation tasks [105, 107, 108] in simulation. Amongst various planner options in [137], we found the most success with **Predictive Sampling**, a derivative-free sampling-based method. Our method uses 5 million samples to train for each song using the same set of hyperparameters, and the MPC baseline is run at one-tenth of real-time speed to give the planner adequate search time.

The quantitative results are shown in Figure 6.5. We observe that the ROBOPIANIST agent significantly outperforms the MPC baseline, achieving an average F1 score of 0.79 compared to 0.43 for MPC. We hypothesize that the main bottleneck for MPC is compute: the planner struggles with the large search space which means the quality of the solutions that can be found in the limited time budget is poor. Qualitatively, our learned agent displays remarkably skilled piano behaviors such as (1) simultaneously controlling both hands to reach for notes on opposite ends of the keyboard, (2) playing *chords* by precisely and simultaneously hitting note triplets, and (3) playing *trills* by rapidly alternating between adjacent notes (see Figure 6.3). We encourage the reader to listen to these policies on the supplementary website<sup>1</sup>.

## Multi-Song Policy Learning

**Multitask RL** Ideally, a single agent should be able to learn how to play all songs. Secondly, given enough songs to practice on, we would like this agent to zero-shot generalize to new ones. To investigate whether this is possible, we create a multi-task environment where the number of tasks in the environment corresponds to the number of songs available in the training set. Note that in this multi-song setting, environments of increasing size are additive (i.e., a 2-song environment contains the same song as the 1-song environment plus an additional one). We use Für Elise as the base song and report the performance of a single agent trained on increasing amounts of training songs in Figure 6.4. We observe that training on an increasing amount of songs is significantly harder than training specialist policies on individual songs. Indeed, the F1 score on Für Elise continues to drop as the number of training songs increases, from roughly 0.7 F1 for 1 song (i.e., the specialist) to almost 0 F1 for 16 songs. We also evaluate the agent’s zero-shot performance (i.e., no fine-tuning) on ROBOPIANIST-ETUDE-12 (which does not have overlap with the training songs) to test our RL agent’s ability to generalize. We see, perhaps surprisingly, that multitask RL training fails to positively transfer on the test set regardless of the size of the pre-training tasks.

---

<sup>1</sup><https://kzakka.com/robopianist/>

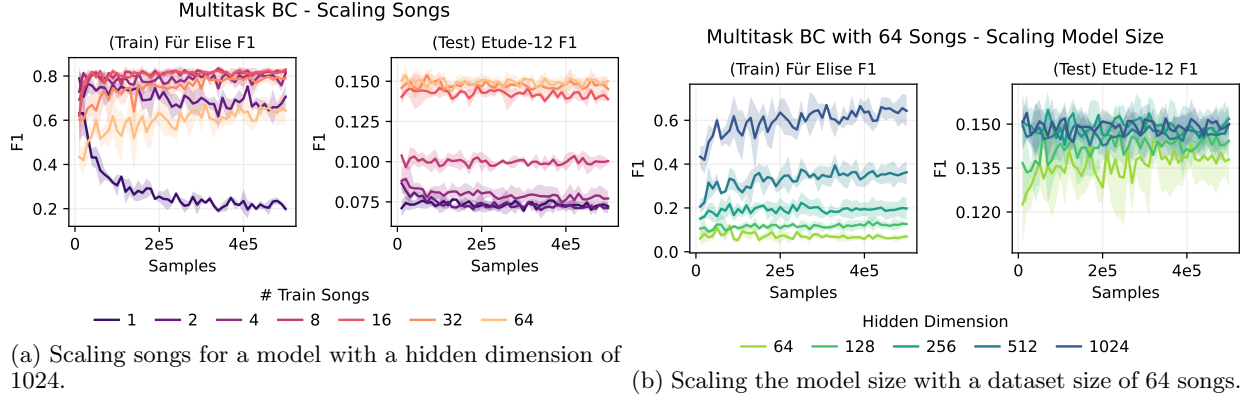


Figure 6.6: Here we show the learning curves of BC, showing the interplay between the attempted number of training songs and model size. The song Für Elise, is in the training dataset for each experiment. We show the F1 score across Für Elise and ROBOPIANIST-ETUDE-12. We see that for a small number of songs, the model quickly overfits, suggesting that for the given model size the amount of data and diversity is lacking. We see that adding more songs results in a higher test F1, suggesting that the model is beginning to generalize. However, zero-shot test performance is far from that of the expert.

As we increase the model size, the F1 score on the training song improves, suggesting that larger models can better capture the complexity of the piano playing task across multiple songs.

**Multitask Behavioral Cloning** Since multitask training with RL is challenging, we instead distill the specialist RL policies trained in Section 6.5 into a single multitask policy with Behavioral Cloning (BC) [138]. To do so, we collect 100 trajectories for each song in the ROBOPIANIST-REPertoire-150, hold out 2 for validation, and use the ROBOPIANIST-ETUDE-12 trajectories for testing. We then train a feed-forward neural network to predict the expert actions conditioned on the state and goal using a mean squared error loss.

For a more direct comparison with multi-task RL, the dataset subsets use trajectories from the same songs used in the equivalently sized multi-task RL experiments. We observe in Figure 6.6a that as we increase the number of songs in the training dataset, the model’s ability to generalize improves, resulting in higher test F1 scores. We note that for a large model (hidden size of 1024), training on too few songs results in overfitting because there isn’t enough data. Using a smaller model alleviates this issue, but smaller models are unable to perform well on multiple songs. Despite having better generalization performance than RL, zero-shot performance on ROBOPIANIST-ETUDE-12 falls

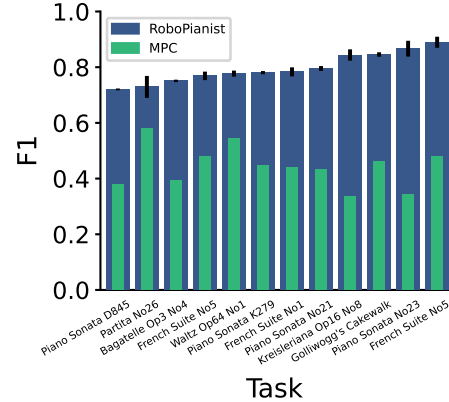


Figure 6.5: The F1 scores achieved by ROBOPIANIST (blue) and MPC (green) for each of the ROBOPIANIST-ETUDE-12 tasks.

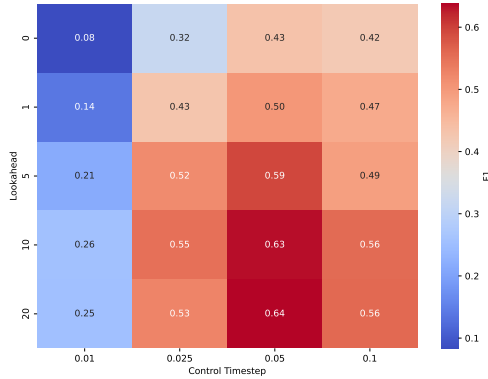


Figure 6.7: **Lookahead & Control Timestep.** These parameters jointly modulate the foresight and control granularity of the agent. We find a control timestep of 0.05 offers optimal control granularity, and performance generally improves with the lookahead up to a point.

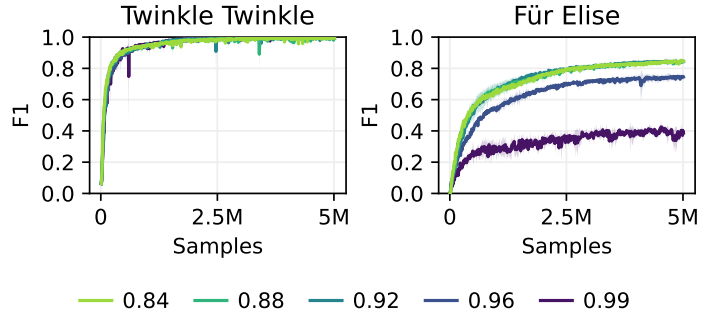


Figure 6.8: **Discount factor.** We find the discount factor to have a high impact on F1, especially as the task complexity increases. While on simpler songs (Twinkle Twinkle) the discount factor does not have any visible effect, it is critical for faster-paced songs (Für Elise). The curves show that any discount in the range of 0.84-0.92 can be used with similar effect.

far below the performance achieved by the specialist policy. Additionally, we investigate the effect of model size on the multitask BC performance. We train models with different hidden dimensions (fixing the number of hidden layers) on a dataset of 64 songs. As shown in Figure 6.6b, a smaller hidden dimension results in lower F1 performance which most likely indicates underfitting.

## Further analysis

In this section, we discuss the effect of certain hyperparameters on the performance of the RL agent.

**Control frequency and lookahead horizon:** Figure 6.7 illustrates the interplay between the control frequency (defined as the reciprocal of control timestep) and the lookahead horizon  $L$ , and their effect on the F1 score. Too large of a control timestep can make it impossible to play faster songs, while a smaller control timestep increases the effective task horizon, thereby increasing the computational complexity of the task. Lookahead on the other hand controls how far into the future the agent can see goal states. We observe that as the control timestep decreases, the lookahead must be increased to maintain the agent ability to see and reason about future goal states. A control frequency of 20Hz (0.05 seconds) is a sweet spot, with notably 100Hz (0.01 seconds) drastically reducing the final score. At 100Hz, the MDP becomes too long-horizon, which complicates exploration, and at 10Hz, the discretization of the MIDI file becomes too coarse, which negatively impacts the timing of the notes.

**Discount factor:** As shown in Figure 6.8, the discount factor has a significant effect on the F1 score. We sweep over a range of discount factors on two songs of varying difficulty.

Notably, we find that discounts in the range 0.84 to 0.92 produce policies with roughly the same performance and high discount factors (e.g., 0.99, 0.96) result in lower F1 performance. Qualitatively, on Für Elise, we noticed that agents trained with higher discounts were often conservative, opting to skip entire note sub-segments. However, agents trained with lower discount factors were willing to risk making mistakes in the early stages of training and thus quickly learned how to correctly strike the notes and attain higher F1 scores.

## 6.6 Discussion

**Limitations** While ROBOPIANIST produces agents that push the boundaries of bi-manual dexterous control, it does so in a simplified simulation of the real world. For example, the velocity of a note, which modulates the strength of the key press, is ignored in the current reward formulation. Thus, the dynamic markings of the composition are ignored. Furthermore, our RL training approach can be considered wasteful, in that we learn by attempting to play the entire piece at the start of every episode, rather than focusing on the parts of the song that need more practicing. Finally, our results highlight the challenges of multitask learning especially in the RL setting.

**Conclusion** In this chapter, we introduced ROBOPIANIST, which provides a simulation framework and suite of tasks in the form of a corpus of songs, together with a high-quality baseline and various axes of evaluation, for studying the challenging high-dimensional control problem of mastering piano-playing with two hands. Our results demonstrate the effectiveness of our approach in learning a broad repertoire of musical pieces, and highlight the importance of various design choices required for achieving this performance. There is an array of exciting future directions to explore with ROBOPIANIST including: leveraging human priors to accelerate learning (e.g., motion priors from YouTube), studying zero-shot generalization to new songs, incorporating multimodal data such as sound and touch. We believe that ROBOPIANIST serves as a valuable platform for the research community, enabling further advancements in high-dimensional control and dexterous manipulation.

# Chapter 7

## mink: Inverse Kinematics with MuJoCo

Chapter 6 showed that fingering annotations make an otherwise intractable exploration problem tractable on the piano-playing task. Priors of that level of specificity have limited reach: they require domain expertise to produce, they exist for only narrow sets of tasks, and they describe behavior at the level of which finger strikes which key, not how a robot’s joints should move. A more general source of task priors is human motion data (motion-capture recordings, teleoperation trajectories, and video), and one common path from such data to robot reference motions is through inverse kinematics. This chapter introduces **mink**, a Python library for differential inverse kinematics built natively on **MuJoCo** [10], and describes its role both within this thesis and across the wider robotics ecosystem.

### 7.1 Motivation

Inverse kinematics is a classical problem in robotics, with algorithmic foundations established decades ago. Its position in the robot-learning pipeline, however, has shifted. Where IK was previously deployed mostly as a self-contained controller component, it now also appears earlier in the pipeline, as one of the standard bridges between human-derived data and robot reference trajectories. Motion-imitation systems often use IK to retarget motion-capture data onto a robot’s kinematic structure. Some vision-language-action data pipelines use it to convert recorded end-effector poses, drawn from UMI-style grippers or teleoperation sessions, into joint-space demonstrations on a target robot. As human-derived data has become more central to robot learning, IK has correspondingly reappeared as a recurring component on the data side of the pipeline, not only the control side.

**mink** is a direct port of **Pink** [139], which is itself built on **Pinocchio** [31]. **Pink** established a clean, task-and-limit abstraction for differential IK in the **Pinocchio** ecosystem; **mink** brings that same design into the **MuJoCo** ecosystem. Rather than maintaining a parallel robot description or bridging across physics engines, **mink** delegates all forward kinematics, Jacobian

computation, collision detection, and constraint evaluation directly to MuJoCo. The result is a solver that integrates without friction into any existing MuJoCo workflow and is immediately compatible with any MuJoCo Menagerie [12] model.

The remainder of this chapter states the QP formulation (Section 7.2), describes the tasks (Section 7.3) and limits (Section 7.4) that `mink` provides, covers implementation choices and performance (Section 7.5), and presents applications across robot morphologies and downstream pipelines (Section 7.6). Related work, limitations, and a closing discussion follow.

## 7.2 The QP

Let the robot have generalized coordinates  $q \in \mathcal{Q}$  on a configuration manifold  $\mathcal{Q}$  (not necessarily Euclidean, since  $\mathcal{Q}$  may include unit-quaternion joints and free-floating bases) with tangent-space dimension  $n_v$ . A differential IK step finds a small displacement  $\Delta q \in \mathbb{R}^{n_v}$  that drives a set of task errors toward zero, subject to physical constraints. Each task  $i$  defines an error  $e_i(q) \in \mathbb{R}^{k_i}$  and its Jacobian  $J_i(q) = \partial e_i / \partial q$ . A first-order expansion  $e_i(q \oplus \Delta q) \approx e_i(q) + J_i(q) \Delta q$  (where  $\oplus$  denotes manifold addition), driving each error toward zero with gain  $\alpha_i \in (0, 1]$  under a diagonal cost  $W_i = \text{diag}(w_i)$ , gives the weighted least-squares objective

$$\min_{\Delta q} \sum_i \frac{1}{2} W_i (J_i \Delta q + \alpha_i e_i)^2 + \frac{\mu}{2} \|\Delta q\|^2, \quad (7.1)$$

with  $\mu \geq 0$  a Levenberg–Marquardt damping term that regularizes the problem near kinematic singularities [140]. Expanding the squared norms gives a standard quadratic program,

$$\min_{\Delta q} \frac{1}{2} \Delta q^\top H \Delta q + c^\top \Delta q \quad \text{subject to} \quad G \Delta q \leq h, \quad (7.2)$$

with  $H = \sum_i J_i^\top W_i^2 J_i + \mu I_{n_v}$  and  $c = \sum_i J_i^\top W_i^2 (\alpha_i e_i)$ . Inequalities  $G \Delta q \leq h$  encode physical limits; equalities  $A \Delta q = b$  are added when strict satisfaction is required. The solution corresponds to a velocity  $v = \Delta q / \Delta t$ , integrated as  $q_{t+1} = q_t \oplus v \Delta t$  by MuJoCo’s `mj_integratePos` so that quaternion-parameterized joints and free bases remain valid. Each task contributes one  $J_i^\top W_i^2 J_i$  block to  $H$  and one  $J_i^\top W_i^2 \alpha_i e_i$  block to  $c$ , and each limit contributes rows to  $G$  and  $h$ , so tasks and limits compose additively.

## 7.3 Tasks

`mink` provides task types covering the common use cases that arise in differential IK. Each task implements an error and Jacobian and contributes additively to the QP. Each task carries a per-dimension cost weight, a gain  $\alpha_i$ , and an optional task-level Levenberg–Marquardt damping  $\mu_i$  that scales with the current error magnitude, damping the step more strongly when the task is far from its target.

**Frame task on SE(3).** The most common task regulates the 6-DoF pose of a frame (body, site, or geom) to a target  $T_{\text{target}} \in \text{SE}(3)$ . Error and Jacobian follow the right-trivialized SE(3) construction in Solà et al. [141], with the geometric Jacobian supplied by MuJoCo. The error is a 6D twist expressed in the body frame, singularity-free and smoothly differentiable everywhere except at rotations of exactly  $\pi$ ; in the small-error regime the task Jacobian reduces to the geometric Jacobian and the update collapses to the standard resolved-rate formula.

**Posture, CoM, equality, and regularization tasks.** A posture task drives the configuration toward a reference  $q_{\text{ref}}$  with error  $q \ominus q_{\text{ref}}$  and absorbs redundant degrees of freedom at small cost. A center-of-mass task regulates the body CoM to a 3D target via MuJoCo’s CoM Jacobian, and is useful for balance on floating-base robots. For robots with kinematic loops (four-bar linkages, parallel mechanisms, tendon couplings), an *equality-constraint task* reads the residual and Jacobian that MuJoCo computes internally from `efc_pos/efc_J`, and is typically used with a high cost weight or enforced as a hard QP equality when strict satisfaction is required, surfacing closed-chain constraints through the same interface as other tasks. `mink` additionally provides a damping task ( $\|\Delta q\|^2$ ) and a kinetic-energy regularization task ( $\Delta q^\top M(q) \Delta q / \Delta t^2$ , so the cost has units of energy) that act as stabilizers on the QP solution.

## 7.4 Limits

Beyond task objectives, real robots have physical limits that must be respected at every control step. `mink` expresses these as linear inequalities  $G \Delta q \leq h$  added to the QP alongside the task objective.

**Configuration and velocity limits.** Joint bounds linearize on the configuration manifold to standard inequalities  $|\Delta q_j| \leq q_j^{\text{max}} \ominus q_j$  (and the symmetric lower bound), with  $\ominus$  computed via MuJoCo’s `mj_differentiatePos` so that quaternion-parameterized joints are handled correctly. Velocity bounds map directly to displacement bounds  $|\Delta q_j| \leq v_j^{\text{max}} \Delta t$ .

**Collision avoidance.** The most involved limit is collision avoidance. The formulation follows Fang et al. [142], and the implementation uses MuJoCo’s native distance and contact queries throughout. For a pair of geoms  $(g_1, g_2)$  with closest points  $p_1, p_2$  at signed distance  $d = \|p_2 - p_1\|$  and contact normal  $n = (p_2 - p_1)/d$ , the approach velocity along the normal is constrained by

$$-n^\top (J_{p_2}(q) - J_{p_1}(q)) \Delta q \leq \begin{cases} \gamma (d - d_{\text{min}}) & \text{if } d > d_{\text{min}}, \\ 0 & \text{otherwise,} \end{cases} \quad (7.3)$$



where  $J_{p_i} \in \mathbb{R}^{3 \times n_v}$  is the translational Jacobian at the closest point on geom  $g_i$ ,  $d_{\min}$  is a safety margin, and  $\gamma \in (0, 1]$  controls the allowed approach speed. Beyond  $d_{\min}$  approach is permitted at a rate proportional to the remaining clearance; inside it further approach is blocked. The constraint set is rebuilt at every control step as closest points shift and pairs enter or leave the detection range.

A naive implementation would check every geom pair, which scales as the square of the geom count. `mink` prunes pairs on the same rigid body, on welded or parent–child bodies, or that fail MuJoCo’s `contype/conaffinity` bitwise check; users specify which geom groups to monitor, and a configurable detection distance excludes pairs beyond a threshold. The remaining pair set is typically one to two orders of magnitude smaller than the unfiltered count, but the pairwise distance and Jacobian computations still dominate collision-avoidance overhead in practice (Section 7.5).

## 7.5 Implementation

`mink` is implemented in Python on MuJoCo, with a small C extension for the hot-path Lie-group operations. All kinematic computations (forward kinematics, frame Jacobians, closest-point queries, configuration differences, manifold integration) are delegated to MuJoCo, so users work directly with `MjModel` and `MjData` without a parallel model representation and are immediately compatible with any MuJoCo Menagerie model. The assembled QP is passed to a `qpsolvers.Problem` object [143], decoupling formulation from solution and allowing any supported backend to be selected at runtime. `mink` ships DAQP [144] as the recommended solver and uses it throughout its examples, since it is fast on the small dense QPs that arise in real-time control and is MIT-licensed (the popular `quadprog` alternative ships under a more restrictive license). The  $SE(3)$  and  $SO(3)$  operations are implemented in a standalone module adapted from `jaxlie` [145] following the conventions of Solà et al. [141], with an optional C extension on the hot path that yields roughly 30–130× speedup and falls back transparently when not available.

Table 7.1 reports end-to-end IK step timings on an Apple M1 Max processor. Across all three scenarios `mink` solves well above what a typical 200–500 Hz control loop requires; only the most demanding collision setup (ALOHA with 1,104 monitored pairs) exceeds the millisecond mark. QP assembly dominates the step in most cases: for the G1 humanoid, assembly takes 214  $\mu s$  median versus 55  $\mu s$  for the solve; ALOHA is dominated by collision-avoidance overhead.

Table 7.1: Per-step IK timing on Apple M1 Max (median / p95, in  $\mu s$ ).

| Scenario                              | $n_v$ | Assembly    | Solve     | Total       |
|---------------------------------------|-------|-------------|-----------|-------------|
| UR5e arm (1 frame task + limits)      | 6     | 38 / 40     | 11 / 12   | 64 / 67     |
| G1 humanoid (6 tasks + collision)     | 49    | 214 / 225   | 55 / 59   | 307 / 325   |
| ALOHA dual-arm (1104 collision pairs) | 12    | 1018 / 1553 | 206 / 242 | 1249 / 1819 |



A typical control loop constructs task and limit objects once, updates their targets each step, and calls `mink.solve_ik` to obtain a velocity update that is integrated in place on the configuration.

## 7.6 Applications and Adoption

**Morphological coverage.** `mink` ships with over twenty worked examples drawn from MuJoCo Menagerie [12], spanning single-arm manipulators (Franka Panda, UR5e, KUKA iiwa14, ALOHA 2), dual-arm platforms, dexterous hands (Shadow Hand, Allegro, LEAP), humanoids (Unitree G1 and H1, Apptронik Apollo), legged robots (Unitree Go1, Boston Dynamics Spot, Agility Cassie), and mobile manipulators (Hello Robot Stretch, TidyBot). The same task-and-limit API covers all morphologies: the user specifies tasks and limits, calls the QP solver, and integrates the returned velocity update.

**In-thesis uses.** Follow-on work to RoboPianist (Chapter 6), *PianoMime* [11], uses `mink` to retarget hand keypress sequences extracted from internet piano videos into joint-space trajectories for the Shadow Hands, replacing the hand-specified fingering annotations the original work relied on. The value is workflow compatibility rather than algorithmic novelty: retargeting runs inside the same MuJoCo model that downstream training and evaluation use, with no model translation in between.

**Adoption.** `mink`’s public repository has accumulated approximately 1,300 GitHub stars. It is the inverse kinematics back end of GMR (General Motion Retargeting) [146], which has in turn become a building block for downstream humanoid motion-imitation pipelines, and it is used in TWIST [147] for whole-body teleoperated imitation. The library has additionally seen use in teleoperation data collection, the data-pipeline side of vision-language-action training, and coursework where the small dependency footprint and the readability of the task-and-limit API make it easy to drop into a new project.

## 7.7 Related Work

**Differential IK.** Whitney [148] introduced resolved motion rate control. Liégeois [149] proposed gradient projection for redundancy resolution. Nakamura and Hanafusa [150] introduced damped least-squares for singularity robustness, later extended by Sugihara [140] to a Levenberg–Marquardt scheme with adaptive damping. Kanoun et al. [151] cast task-priority IK as a QP with inequality constraints under strict hierarchical priorities, an approach pursued further by Escande et al. [152]; `mink` adopts the QP formulation with inequality limits but balances multiple tasks through soft weighted least-squares rather than strict priorities, following the weighted formulation used by Pink. Chiaverini et al. [153] provide a comprehensive survey.

**Software.** Pink [139] pioneered the task-based API design that `mink`’s interface ports; Pink itself uses Pinocchio [31] for kinematics, which requires a separate model representation from MuJoCo. OpenSoT [154] provides a C++ stack-of-tasks framework with strict hierarchical priorities, and PyRoki [155] offers differentiable IK through JAX with GPU batching. `mink`’s contribution sits on a different axis: a MuJoCo-native Python implementation that integrates with the rest of the MuJoCo workflow without a separate model representation, with closed-chain IK via MuJoCo’s equality constraints, a C-optimized Lie algebra module on the hot path, and the collision-avoidance formulation described in Section 7.4.

## 7.8 Limitations and Discussion

`mink`’s weighted least-squares formulation balances tasks through cost weights rather than strict hierarchies [152]: lower-priority tasks can degrade higher-priority ones if weights are tuned poorly, though in practice tuning is straightforward. Adding a new task type requires deriving its error and Jacobian by hand; the built-in set covers the common cases. When constraints are mutually incompatible the QP solver fails and `mink` raises, leaving the user to relax constraints or adjust targets. All computation runs on a single CPU: `mink` sustains kilohertz rates for single-instance real-time control on commodity hardware, but for batched IK over thousands of configurations, GPU-accelerated approaches such as PyRoki [155] are more suitable.

The contribution of `mink` is not a new algorithm but an implementation: a MuJoCo-native differential IK solver that fits into the data-to-reference pipelines the rest of this thesis builds on. Human motion data is the most scalable source of task priors for robot learning, and inverse kinematics is the standard interface between such data and a target robot. `mink` fills the gap that Pink previously filled for the Pinocchio ecosystem, removing the need to bridge out of MuJoCo when retargeting human-derived data onto a specific robot.

# Chapter 8

## Conclusion

This thesis began with a question about the sim-to-real reinforcement learning pipeline: *what does it take to make every stage of the sim-to-real pipeline a shared, reusable foundation rather than a per-project engineering effort?* The six chapters that followed contributed answers, each targeting a specific stage and shipping as a standalone open-source artifact. In this chapter, we revisit those contributions, reflect on what building them taught us, and discuss the directions that remain open.

Part I addressed the grounding stage of the pipeline. Chapter 2 introduced *MuJoCo Menagerie*, the shared library of robot models that the rest of the pipeline draws on. Menagerie grew from a response to a concrete absence of high-quality MJCF models into the de facto model source for MuJoCo-based robotics research and into a corpus that several external simulators have since chosen to support natively. Chapter 3 introduced `mujoco.sysid`, which calibrates those starting models against measurements from a specific physical robot. The toolbox casts identification as a box-constrained nonlinear least-squares problem and reports confidence intervals that distinguish parameters the data constrained from those it did not. Together the two chapters cover both halves of the grounding stage: a high-quality nominal model, and a principled procedure for closing the gap between that nominal model and the hardware in the laboratory.

Part II addressed training infrastructure. Chapter 4 introduced *MuJoCo Playground*, a JAX-based, GPU-accelerated stack designed for minimal abstractions and rapid prototyping, which demonstrated that a single deliberately simple sim-to-real recipe transfers across six robotic platforms with no per-robot algorithmic modifications. Chapter 5 introduced `mjlab`, the complementary point on the abstraction spectrum: a manager-based, declarative API built on MuJoCo Warp for codebases that span multiple robots and task families. Together, the two libraries cover the range from quick one-off experiments to structured multi-robot development.

Part III addressed task priors. Chapter 6 presented *RoboPianist*, which showed that a task-specific prior in the form of fingering annotations from sheet music turns an otherwise intractable exploration problem in dexterous bimanual piano playing into a tractable one. Chapter 7 generalized from task-specific to broader human-motion priors, providing `mink`, a

MuJoCo-native differential inverse kinematics library broadly adopted for humanoid motion retargeting and teleoperation-based data collection for imitation learning.

A recurring theme across these contributions, one that was not obvious at the outset of any of them, is that the decisive gains in sim-to-real performance came less from algorithmic novelty than from execution. Calibrating a robot model’s physical parameters correctly, specifying a reward function precisely enough that the algorithm learns the intended behavior, and retargeting motion-capture data faithfully onto a target robot are operations whose underlying ideas are not new. What required sustained effort was executing them correctly, on real hardware, at the scale and reliability that reproducibility demands. The pipeline framing this thesis develops is a response to that observation: rather than treating any single stage as the locus of difficulty, it treats the combined problem of providing open, inspectable tooling across all three stages as the unit at which progress can be measured.

A second recurring theme is the compounding effect of open infrastructure. Menagerie was released with no expectation that external simulators would adopt MJCF as a portable robot description format, yet Drake, Newton, and Isaac Lab each did so within a few years. The system identification toolbox was developed against the YAM arm but ships with MuJoCo itself and is now used on platforms it was never designed for. `mjlab` was released in late 2025 and within six months had been adopted at Boston Dynamics, Google, and PAL Robotics, with the official Unitree reinforcement-learning environments built on top of it. Follow-on work to RoboPianist extends piano playing to learning from internet video [11]. `mink` is a port of an existing library, but its placement inside the MuJoCo ecosystem removed the engine-bridging step from pipelines that train and evaluate against MuJoCo-native models. It has since been adopted for humanoid motion retargeting and teleoperation-based data collection. Read together, these signals indicate that the open infrastructure has compounded across labs: each artifact has been picked up, extended, or integrated by groups that had no involvement in its original development.

Three directions stand out, each building on what the preceding chapters put in place.

**Understanding which simulation errors matter for control.** Chapter 3 can recover a large number of a robot’s physical parameters, but it leaves open the more basic question of which discrepancies are worth correcting. The resulting picture is not yet well understood. A coarse, visibly inaccurate model can yield a policy that transfers well on one task and a policy that fails catastrophically on another. Which discrepancies matter is task-dependent, and we have no principled way to predict that dependence, or to say in advance that a given model is accurate enough for a given task. When a policy does fail on hardware, the failure is often treated as a generic distribution shift without identifying which discrepancy caused it. This is a gap in scientific understanding, not in tooling, and closing it means studying identification and reinforcement learning together rather than as separate stages: which model errors a task is sensitive to, and why.

**Real-world reinforcement learning as the complement to simulation.** The pipeline this thesis develops trains in simulation and transfers once; it does not continue learning on hardware afterward. This is a limitation wherever simulation has a fidelity ceiling that identification alone cannot raise: deformable objects, granular media, and tactile interaction are difficult to simulate at the speed and accuracy training demands, and the visual diversity of real environments is difficult to reproduce synthetically. Two responses are especially relevant here. One is to improve the simulator from real data, the real-to-sim direction, which remains largely open. The other is to fine-tune directly on hardware, taking the transferred policy as a starting point so that real-world reinforcement learning begins from competent rather than random behavior. The second is the natural complement to this thesis: simulation supplies a strong prior, and real interaction closes what remains. Its practicality depends on two problems the open pipeline currently hands to the simulator. State and reward are both free in simulation, where the full state is known by construction; on hardware each must be estimated from sensors, and the learning loop is only as reliable as those estimates.

**From raw control to learned action abstractions.** The low-level controllers this thesis trains, in Parts II and III, act in raw control space, outputting actuator commands directly. Part III made hard tasks tractable by adding a human prior on top of this: fingering labels in RoboPianist and retargeted human motion through *mink*. In both cases, however, the prior is supplied separately for each task. A more general approach would build such structure into the action representation itself: use retargeting to express large collections of human motion on the robot, train a generative model of that motion, and let the policy act through this model rather than directly in raw commands. A policy that explores through learned motion primitives or short-horizon action sequences, rather than arbitrary control vectors, starts from a more structured prior without requiring that prior to be specified by hand for each task. This raises a second question: how to condition such a prior, turning a task description, observation, or goal into the motions it should produce. Vision-language-action models are a natural candidate for this interface, since they may connect semantic task descriptions and visual observations to the latent commands of a low-level motor prior. How to learn such priors from retargeted motion, and how to connect them to higher-level models within the open pipeline developed here, is a direction this thesis points toward but does not reach.

This thesis asked what it would take to make each stage of the sim-to-real pipeline a shared foundation rather than a per-project effort. The answer it offers is six open artifacts spanning the three stages of the pipeline. The directions above show how much of the pipeline is still open; what this thesis provides is a foundation on which the work of closing those gaps can proceed without first rebuilding the layers underneath.

# Bibliography

- [1] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.
- [2] Takahiro Miki, Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science robotics*, 7(62):eabk2822, 2022.
- [3] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.
- [4] Ankur Handa, Arthur Allshire, Viktor Makoviychuk, Aleksei Petrenko, Ritvik Singh, Jingzhou Liu, Denys Makoviichuk, Karl Van Wyk, Alexander Zhurkevich, Balakumar Sundaralingam, et al. Dextreme: Transfer of agile in-hand manipulation from simulation to reality. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5977–5984. IEEE, 2023.
- [5] Ilija Radosavovic, Sarthak Kamat, Trevor Darrell, and Jitendra Malik. Learning humanoid locomotion over challenging terrain. *arXiv preprint arXiv:2410.03654*, 2024.
- [6] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Champion-level drone racing using deep reinforcement learning. *Nature*, 620(7976):982–987, 2023.
- [7] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.
- [8] Mayank Mittal, Calvin Yu, Qinxu Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, et al. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023.

- [9] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning*, pages 270–282. PMLR, 2018.
- [10] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.
- [11] Cheng Qian, Julen Urain, Kevin Zakka, and Jan Peters. PianoMime: Learning a generalist, dexterous piano player from internet demonstrations, 2024. URL <https://arxiv.org/abs/2407.18178>.
- [12] Kevin Zakka, Yuval Tassa, and MuJoCo Menagerie Contributors. MuJoCo Menagerie: A collection of high-quality simulation models for MuJoCo, September 2022. URL [https://github.com/google-deepmind/mujoco\\_menagerie](https://github.com/google-deepmind/mujoco_menagerie).
- [13] Kevin Zakka, Baruch Tabanpour, Qiayuan Liao, Mustafa Haiderbhai, Samuel Holt, Jing Yuan Luo, Arthur Allshire, Erik Frey, Koushil Sreenath, Lueder A. Kahrs, Carmelo Sferrazza, Yuval Tassa, and Pieter Abbeel. Demonstrating MuJoCo playground. In *Robotics: Science and Systems*, 2025. URL <https://roboticsproceedings.org/rss21/p020.pdf>.
- [14] Kevin Zakka, Qiayuan Liao, Brent Yi, Louis Le Lay, Koushil Sreenath, and Pieter Abbeel. mjlab: A lightweight framework for GPU-accelerated robot learning, 2026. URL <https://arxiv.org/abs/2601.22074>.
- [15] Kevin Zakka, Philipp Wu, Laura Smith, Nimrod Gileadi, Taylor Howell, Xue Bin Peng, Sumeet Singh, Yuval Tassa, Pete Florence, Andy Zeng, and Pieter Abbeel. RoboPianist: Dexterous piano playing with deep reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2023.
- [16] Kevin Zakka. Mink: Python inverse kinematics based on MuJoCo, 2024. URL <https://github.com/kevinzakka/mink>.
- [17] Erwin Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, page 1. 2015.
- [18] Khaled Mamou. V-HACD: Volumetric hierarchical approximate convex decomposition, 2016. URL <https://github.com/kmammou/v-hacd>.
- [19] Xinyue Wei, Minghua Liu, Zhan Ling, and Hao Su. Approximate convex decomposition for 3D meshes with collision-aware concavity and tree search. *ACM Transactions on Graphics (TOG)*, 41(4):1–18, 2022.

- [20] Stéphane Caron et al. `robot_descriptions.py`: A Python library for loading robot descriptions, 2022. URL [https://github.com/robot-descriptions/robot\\_descriptions.py](https://github.com/robot-descriptions/robot_descriptions.py).
- [21] I2RT Robotics. YAM – yet another manipulator, 2025. URL <https://i2rt.com/products/yam-manipulator-pre-order>. Accessed: April 30, 2025.
- [22] Caleb Rucker and Patrick M. Wensing. Smooth parameterization of rigid-body inertia. *IEEE Robotics and Automation Letters*, 7(2):2771–2778, 2022. doi: 10.1109/LRA.2022.3144517.
- [23] Lennart Ljung. *System identification (2nd ed.): theory for the user*. Prentice Hall PTR, USA, 1999. ISBN 0136566952.
- [24] C G Atkeson, C H An, and J M Hollerbach. Estimation of inertial parameters of manipulator loads and links. *Int. J. Rob. Res.*, 5(3):101–119, September 1986. ISSN 0278-3649. doi: 10.1177/027836498600500306. URL <https://doi.org/10.1177/027836498600500306>.
- [25] Maxime Gautier. Numerical calculation of the base inertial parameters of robots. *Journal of robotic systems*, 8(4):485–506, 1991.
- [26] Patrick M. Wensing, Sangbae Kim, and Jean-Jacques E. Slotine. Linear matrix inequalities for physically consistent inertial parameter identification: A statistical perspective on the mass distribution. *IEEE Robotics and Automation Letters*, 3(1):60–67, 2018. doi: 10.1109/LRA.2017.2729659.
- [27] J. Swevers, C. Ganseman, D.B. Tukul, J. de Schutter, and H. Van Brussel. Optimal robot excitation and identification. *IEEE Transactions on Robotics and Automation*, 13(5):730–740, 1997. doi: 10.1109/70.631234.
- [28] Taeyoon Lee, Bryan D. Lee, and Frank C. Park. Optimal excitation trajectories for mechanical systems identification. *Automatica*, 131:109773, 2021. ISSN 0005-1098. doi: <https://doi.org/10.1016/j.automatica.2021.109773>. URL <https://www.sciencedirect.com/science/article/pii/S0005109821002934>.
- [29] Huanyu Tian, Martin Huber, Christopher E. Mower, Zhe Han, Changsheng Li, Xingguang Duan, and Christos Bergeles. Excitation trajectory optimization for dynamic parameter identification using virtual constraints in hands-on robotic system, 2024. URL <https://arxiv.org/abs/2401.16566>.
- [30] Taeyoon Lee, Jaewoon Kwon, Patrick M. Wensing, and Frank C. Park. Robot model identification and learning: A modern perspective. *Annual Review of Control, Robotics, and Autonomous Systems*, 7(Volume 7, 2024):311–334, 2024. ISSN 2573-5144. doi: <https://doi.org/10.1146/annurev-control-061523-102310>.



- URL <https://www.annualreviews.org/content/journals/10.1146/annurev-control-061523-102310>.
- [31] Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiriaux, Olivier Stasse, and Nicolas Mansard. The Pinocchio C++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. *IEEE/SICE International Symposium on System Integration*, pages 614–619, 2019.
  - [32] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566, 2018. doi: 10.1109/ICRA.2018.8463189.
  - [33] Rhys Newbury, Jack Collins, Kerry He, Jiahe Pan, Ingmar Posner, David Howard, and Akansel Cosgun. A review of differentiable simulators. *IEEE Access*, 12:97581–97604, 2024. doi: 10.1109/ACCESS.2024.3425448.
  - [34] Quentin Le Lidec, Igor Kalevatykh, Ivan Laptev, Cordelia Schmid, and Justin Carpentier. Differentiable simulation for physical system identification. *IEEE Robotics and Automation Letters*, 6(2):3413–3420, 2021.
  - [35] Taylor A Howell, Simon Le Cleac’h, J Zico Kolter, Mac Schwager, and Zachary Manchester. Dojo: A differentiable simulator for robotics. *arXiv preprint arXiv:2203.00806*, 9(2):4, 2022.
  - [36] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, page 3803–3810. IEEE, May 2018. doi: 10.1109/icra.2018.8460528. URL <http://dx.doi.org/10.1109/ICRA.2018.8460528>.
  - [37] Joshua Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017. URL <https://api.semanticscholar.org/CorpusID:2413610>.
  - [38] Wenhao Yu, C. Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. *ArXiv*, abs/1702.02453, 2017. URL <https://api.semanticscholar.org/CorpusID:10615022>.
  - [39] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979, 2019. doi: 10.1109/ICRA.2019.8793789.

- [40] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *ArXiv*, abs/2107.04034, 2021. URL <https://api.semanticscholar.org/CorpusID:235650916>.
- [41] Marc Duclausaud, Grégoire Passault, Vincent Padois, and Olivier Ly. Extended friction models for the physics simulation of servo actuators. *ArXiv*, abs/2410.08650, 2024. URL <https://api.semanticscholar.org/CorpusID:273323725>.
- [42] 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), January 2019. ISSN 2470-9476. doi: 10.1126/scirobotics.aau5872. URL <http://dx.doi.org/10.1126/scirobotics.aau5872>.
- [43] Patrick M. Wensing, Albert Wang, Sangok Seok, David Otten, Jeffrey Lang, and Sangbae Kim. Proprioceptive actuator design in the mit cheetah: Impact mitigation and high-bandwidth physical interaction for dynamic legged robots. *Ieee transactions on robotics*, 33(3):509–522, 2017.
- [44] Jo-Anne Ting, Michael N Mistry, Jan Peters, Stefan Schaal, and Jun Nakanishi. A bayesian approach to nonlinear parameter identification for rigid body dynamics. In *Robotics: Science and systems*, pages 32–39. Citeseer, 2006.
- [45] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.
- [46] 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, 2021. URL <http://github.com/google/brax>, 6, 2021.
- [47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [48] 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 arXiv:1804.10332*, 2018.
- [49] MuJoCo XLA Authors. MuJoCo XLA (MJX). <https://mujoco.readthedocs.io/en/stable/mjx.html>. Accessed: December 16, 2024.
- [50] Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG)*, 42(4):1–13, 2023.

- [51] Ananye Agarwal, Ashish Kumar, Jitendra Malik, and Deepak Pathak. Legged locomotion in challenging terrains using egocentric vision. In *Conference on robot learning*, pages 403–415. PMLR, 2023.
- [52] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.
- [53] 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 arXiv:2305.14654*, 2023.
- [54] Qiayuan Liao, Bike Zhang, Xuanyu Huang, Xiaoyu Huang, Zhongyu Li, and Koushil Sreenath. Berkeley humanoid: A research platform for learning-based control. *arXiv preprint arXiv:2407.21781*, 2024.
- [55] Kenneth Shaw, Ananye Agarwal, and Deepak Pathak. Leap hand: Low-cost, efficient, and anthropomorphic hand for robot learning. *Robotics: Science and Systems (RSS)*, 2023.
- [56] Jorge ALOHA 2 Team, Aldaco, Travis Armstrong, Robert Baruch, Jeff Bingham, Sanky Chan, Kenneth Draper, Debidatta Dwibedi, Chelsea Finn, Pete Florence, Spencer Goodrich, et al. Aloha 2: An enhanced low-cost hardware for bimanual teleoperation. *arXiv preprint arXiv:2405.02292*, 2024.
- [57] Luc Guy Rosenzweig, Brennan Shacklett, Warren Xia, and Kayvon Fatahalian. High-throughput batch rendering for embodied ai. 2024.
- [58] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Neca, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL <http://github.com/jax-ml/jax>.
- [59] Gwanghyeon Ji, Juhyeok Mun, Hyeongjun Kim, and Jemin Hwangbo. Concurrent training of a control policy and a state estimator for dynamic and robust legged locomotion. *IEEE Robotics and Automation Letters*, 7(2):4630–4637, April 2022. ISSN 2377-3774. doi: 10.1109/lra.2022.3151396. URL <http://dx.doi.org/10.1109/LRA.2022.3151396>.
- [60] Joonho Lee, Jemin Hwangbo, and Marco Hutter. Robust recovery controller for a quadrupedal robot using deep reinforcement learning. *arXiv preprint arXiv:1901.07517*, 2019.

- [61] Laura Smith, J Chase Kew, Xue Bin Peng, Sehoon Ha, Jie Tan, and Sergey Levine. Legged robots that keep on learning: Fine-tuning locomotion policies in the real world. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 1593–1599. IEEE, 2022.
- [62] Albert Hao Li, Preston Culbertson, Vince Kurtz, and Aaron D. Ames. Drop: Dexterous reorientation via online planning. *arXiv preprint arXiv:2409.14562*, 2024. Available at: <https://arxiv.org/abs/2409.14562>.
- [63] Scott Niekum and Isaac I.Y. Saito. *ar\_track\_alvar*, 2016. URL [https://github.com/ros-perception/ar\\_track\\_alvar](https://github.com/ros-perception/ar_track_alvar).
- [64] Samuel Holt, Todor Davchev, Dhruva Tirumala, Ben Moran, Yixin Lin, Antoine Laurens, Atil Iscen, Erik Frey, Markus Wulfmeier, Francesco Romano, and Nicolas Heess. Evolving control: Evolved high frequency control for continuous control tasks. In *CoRL Workshop on Safe and Robust Robot Learning for Operation in the Real World*, 2024. URL <https://openreview.net/forum?id=gzADUWLD9X>.
- [65] leggedrobotics. *rsl\_rl*: Fast and simple implementation of rl algorithms, designed to run fully on gpu. [https://github.com/leggedrobotics/rsl\\_rl](https://github.com/leggedrobotics/rsl_rl), 2023. Accessed: January 10, 2025.
- [66] Miles Macklin. Warp: A high-performance python framework for gpu simulation and graphics. <https://github.com/nvidia/warp>, March 2022. NVIDIA GPU Technology Conference (GTC).
- [67] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.
- [68] Carmelo Sferrazza, Dun-Ming Huang, Xingyu Lin, Youngwoon Lee, and Pieter Abbeel. Humanoidbench: Simulated humanoid benchmark for whole-body locomotion and manipulation. *Robotics: Science and Systems (RSS)*, 2024.
- [69] Haoru Xue, Chaoyi Pan, Zeji Yi, Guannan Qu, and Guanya Shi. Full-order sampling-based mpc for torque-level locomotion control via diffusion-style annealing. *arXiv preprint arXiv:2409.15610*, 2024.
- [70] Genesis-Authors. Genesis: A universal and generative physics engine for robotics and beyond, December 2024. URL <https://github.com/Genesis-Embodied-AI/Genesis>.
- [71] Xuxin Cheng, Kexin Shi, Ananye Agarwal, and Deepak Pathak. Extreme parkour with legged robots. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11443–11450. IEEE, 2024.

- [72] Ziwen Zhuang, Zipeng Fu, Jianren Wang, Christopher Atkeson, Soeren Schwertfeger, Chelsea Finn, and Hang Zhao. Robot parkour learning. *arXiv preprint arXiv:2309.05665*, 2023.
- [73] Zhongyu Li, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. Reinforcement learning for versatile, dynamic, and robust bipedal locomotion control. *The International Journal of Robotics Research*, page 02783649241285161, 2024.
- [74] Junfeng Long, Junli Ren, Moji Shi, Zirui Wang, Tao Huang, Ping Luo, and Jiangmiao Pang. Learning humanoid locomotion with perceptive internal model. *arXiv preprint arXiv:2411.14386*, 2024.
- [75] Ritvik Singh, Arthur Allshire, Ankur Handa, Nathan Ratliff, and Karl Van Wyk. Dextrah-rgb: Visuomotor policies to grasp anything with dexterous hands. *arXiv preprint arXiv:2412.01791*, 2024.
- [76] Yoonyoung Cho, Junhyek Han, Yoontae Cho, and Beomjoon Kim. Corn: Contact-based object representation for nonprehensile manipulation of general unseen objects. *arXiv preprint arXiv:2403.10760*, 2024.
- [77] Denis Yarats, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto. Mastering visual continuous control: Improved data-augmented reinforcement learning. *arXiv preprint arXiv:2107.09645*, 2021.
- [78] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *arXiv:2004.14990*.
- [79] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.
- [80] Nicklas Hansen, Hao Su, and Xiaolong Wang. Td-mpc2: Scalable, robust world models for continuous control, 2024.
- [81] Shengjie Wang, Shaohuai Liu, Weirui Ye, Jiacheng You, and Yang Gao. Efficientzero v2: Mastering discrete and continuous control with limited data. *arXiv preprint arXiv:2403.00564*, 2024.
- [82] Mustafa Haiderbhai, Radian Gondokaryono, Andrew Wu, and Lueder A. Kahrs. Sim2real rope cutting with a surgical robot using vision-based reinforcement learning. *Transactions on Automation Science and Engineering*, 2024. doi: 10.1109/TASE.2024.3410297.
- [83] Tao Chen, Jie Xu, and Pulkit Agrawal. A system for general in-hand object re-orientation. In *Conference on Robot Learning*, pages 297–307. PMLR, 2022.

- [84] Tuomas Haarnoja, Ben Moran, Guy Lever, Sandy H Huang, Dhruva Tirumala, Jan Humplik, Markus Wulfmeier, Saran Tunyasuvunakool, Noah Y Siegel, Roland Hafner, et al. Learning agile soccer skills for a bipedal robot with deep reinforcement learning. *Science Robotics*, 9(89):eadi8022, 2024.
- [85] Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, Xander Hinrichsen, Xiaodi Yuan, Chen Bao, Xinsong Lin, Yulin Liu, Tse-kai Chan, et al. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *arXiv preprint arXiv:2410.00425*, 2024.
- [86] Google DeepMind and NVIDIA. MuJoCo Warp, 2025. URL [https://github.com/google-deepmind/mujoco\\_warp](https://github.com/google-deepmind/mujoco_warp).
- [87] Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, et al. Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831*, 2025. URL <https://arxiv.org/abs/2511.04831>.
- [88] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhersch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Byte-code Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. doi: 10.1145/3620665.3640366. URL <https://docs.pytorch.org/assets/pytorch2-2.pdf>.
- [89] Amazon FAR, Pieter Abbeel, Juyue Chen, Rocky Duan, Alejandro Escontrela, Manan Gandhi, Samuel Gundry, Xiaoyu Huang, Angjoo Kanazawa, Tomasz Lewicki, Jiaman Li, Karen Liu, Clay Rosenthal, Younggyo Seo, Carlo Sferrazza, Guanya Shi, Linda Shih, Jonathan Tseng, Zhen Wu, Lujie Yang, Brent Yi, and Yuanhang Zhang. Holo-soma.
- [90] Brent Yi, Chung Min Kim, Justin Kerr, Gina Wu, Rebecca Feng, Anthony Zhang, Jonas Kulhanek, Hongsuk Choi, Yi Ma, Matthew Tancik, and Angjoo Kanazawa. Viser: Imperative, web-based 3d visualization in python. *arXiv preprint arXiv:2507.22885*, 2025.

- [91] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
- [92] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics*, 37(4), 2018.
- [93] Qiayuan Liao, Takara E. Truong, Xiaoyu Huang, Yuman Gao, Guy Tevet, Koushil Sreenath, and C. Karen Liu. Beyondmimic: From motion tracking to versatile humanoid control via guided diffusion, 2025. URL <https://arxiv.org/abs/2508.08241>.
- [94] Astral. uv: An extremely fast python package and project manager, 2025. URL <https://docs.astral.sh/uv/>.
- [95] Clemens Schwarke, Mayank Mittal, Nikita Rudin, David Hoeller, and Marco Hutter. Rsl-rl: A learning library for robotics research. *arXiv preprint arXiv:2509.10771*, 2025.
- [96] Khandelwal, Apoorv and Curtin, Peter. torchrunx: Functional distributed pytorch utility, 2025. URL <https://torchrun.xyz/>.
- [97] Shenli Yuan, Lin Shao, Connor L. Yako, Alexander M. Gruebele, and John Kenneth Salisbury. Design and control of roller grasper v2 for in-hand manipulation. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9151–9158, 2020.
- [98] Zhe Xu and Emanuel Todorov. Design of a highly biomimetic anthropomorphic robotic hand towards artificial limb regeneration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3485–3492, 2016. doi: 10.1109/ICRA.2016.7487528.
- [99] Connor McCann, Vatsal Patel, and Aaron Dollar. The stewart hand: A highly dexterous, six-degrees-of-freedom manipulator based on the stewart-gough platform. *icra*, 28(2):23–36, 2021. doi: 10.1109/MRA.2021.3064750.
- [100] R. Fearing. Implementing a force strategy for object re-orientation. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, pages 96–102, 1986. doi: 10.1109/ROBOT.1986.1087655.
- [101] Daniela Rus. In-hand dexterous manipulation of piecewise-smooth 3-d objects. *The International Journal of Robotics Research*, 18(4):355–381, 1999.
- [102] A.M. Okamura, N. Smaby, and M.R. Cutkosky. An overview of dexterous manipulation. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 1, pages 255–262 vol.1, 2000. doi: 10.1109/ROBOT.2000.844067.



- [103] Tao Pang, H. J. Terry Suh, Lujie Yang, and Russ Tedrake. Global planning for contact-rich manipulation via local smoothing of quasi-dynamic contact models. *ArXiv*, abs/2206.10787, 2022.
- [104] Raymond R. Ma and Aaron M. Dollar. On dexterity and dexterous manipulation. In *2011 15th International Conference on Advanced Robotics (ICAR)*, pages 1–7, 2011. doi: 10.1109/ICAR.2011.6088576.
- [105] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Joshua Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39:20 – 3, 2018.
- [106] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas A. Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei M. Zhang. Solving rubik’s cube with a robot hand. *ArXiv*, abs/1910.07113, 2019.
- [107] Ankur Handa, Arthur Allshire, Viktor Makoviychuk, Aleksei Petrenko, Ritvik Singh, Jingzhou Liu, Denys Makoviichuk, Karl Van Wyk, Alexander Zhurkevich, Balakumar Sundaralingam, et al. Dextreme: Transfer of agile in-hand manipulation from simulation to reality. *arXiv preprint arXiv:2210.13702*, 2022.
- [108] Anusha Nagabandi, K. Konolige, Sergey Levine, and V. Kumar. Deep dynamics models for learning dexterous manipulation. abs/1909.11652, 2019.
- [109] Tao Chen, Megha Tippur, Siyang Wu, Vikash Kumar, Edward H. Adelson, and Pulkit Agrawal. Visual dexterity: In-hand dexterous manipulation from depth. *ArXiv*, abs/2211.11744, 2022.
- [110] Yuanpei Chen, Yaodong Yang, Tianhao Wu, Shengjie Wang, Xidong Feng, Jiechuang Jiang, Stephen Marcus McAleer, Hao Dong, Zongqing Lu, and Song-Chun Zhu. Towards human-level bimanual dexterous manipulation with reinforcement learning. *arXiv preprint arXiv:2206.08686*, 2022.
- [111] Haozhi Qi, Ashish Kumar, Roberto Calandra, Yi Ma, and Jitendra Malik. In-Hand Object Rotation via Rapid Motor Adaptation. In *Conference on Robot Learning (CoRL)*, 2022.
- [112] Igor Mordatch, Emanuel Todorov, and Zoran Popović. Discovery of complex behaviors through contact-invariant optimization. *ACM Trans. Graph.*, 31(4), jul 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185539. URL <https://doi.org/10.1145/2185520.2185539>.



- [113] Alejandro M Castro, Frank N Permenter, and Xuchen Han. An unconstrained convex formulation of compliant contact. *IEEE Transactions on Robotics*, 2022.
- [114] Raymond R Ma and Aaron M Dollar. On dexterity and dexterous manipulation. In *2011 15th International Conference on Advanced Robotics (ICAR)*, pages 1–7. IEEE, 2011.
- [115] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4RL: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.
- [116] Christian Smith, Yiannis Karayiannidis, Lazaros Nalpantidis, Xavi Gratal, Peng Qi, Dimos V Dimarogonas, and Danica Kragic. Dual arm manipulation—a survey. *Robotics and Autonomous systems*, 60(10):1340–1353, 2012.
- [117] Henry J Charlesworth and Giovanni Montana. Solving challenging dexterous manipulation tasks with trajectory optimisation and reinforcement learning. In *International Conference on Machine Learning*, pages 1496–1506. PMLR, 2021.
- [118] Ichiro Kato, Sadamu Ohteru, Katsuhiko Shirai, Toshiaki Matsushima, Seinosuke Narita, Shigeki Sugano, Tetsunori Kobayashi, and Eizo Fujisawa. The robot musician ‘wabot-2’(waseda robot-2). *Robotics*, 3(2):143–155, 1987.
- [119] Jen-Chang Lin, Hsing-Hsin Huang, Yen-Fang Li, Jen-Chao Tai, and Li-Wei Liu. Electronic piano playing robot. In *2010 International Symposium on Computer, Communication, Control and Automation (3CA)*, volume 2, pages 353–356. IEEE, 2010.
- [120] Thomas Maloney. Piano-playing robotic arm. Technical report, Worcester Polytechnic Institute, 2019.
- [121] JAE Hughes and P Maiolino. An anthropomorphic soft skeleton hand exploiting conditional models for piano playing. *Science Robotics*, 3(25), 2018.
- [122] Ruben Castro Ornelas. *Robotic Finger Hardware and Controls Design for Dynamic Piano Playing*. PhD thesis, Massachusetts Institute of Technology, 2022.
- [123] Dan Zhang, Jianhe Lei, Beizhi Li, Daniel Lau, and Craig Cameron. Design and analysis of a piano playing robot. In *2009 International Conference on Information and Automation*, pages 757–761, 2009. doi: 10.1109/ICINFA.2009.5205022.
- [124] Yen-Fang Li and Li-Lan Chuang. Controller design for music playing robot—applied to the anthropomorphic piano robot. In *2013 IEEE 10th International Conference on Power Electronics and Drive Systems (PEDS)*, pages 968–973. IEEE, 2013.
- [125] Ada Zhang, Mark Malhotra, and Yoky Matsuoka. Musical piano performance by the act hand. In *2011 IEEE international conference on robotics and automation*, pages 3536–3541. IEEE, 2011.

- [126] B. Scholz. *Playing Piano with a Shadow Dexterous Hand*. PhD thesis, Universität Hamburg, 2019.
- [127] Seongho Yeon. Playing piano with a robotic hand, 2022. URL <https://github.com/seongho-yeon/playing-piano-with-a-robotic-hand>.
- [128] Huazhe Xu, Yuping Luo, Shaoxiong Wang, Trevor Darrell, and Roberto Calandra. Towards learning to play piano with dexterous hands and touch. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10410–10416. IEEE, 2022.
- [129] Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dm\_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020. ISSN 2665-9638. doi: <https://doi.org/10.1016/j.simpa.2020.100022>. URL <https://www.sciencedirect.com/science/article/pii/S2665963820300099>.
- [130] Kawai Musical Instruments. Kawai Vertical Piano Regulation Manual, 2019. URL <https://kawaius.com/wp-content/uploads/2019/04/Kawai-Upright-Piano-Regulation-Manual.pdf>.
- [131] Shadow Robot Company, 2005. URL <https://www.shadowrobot.com/dexterous-hand-series/>.
- [132] Colin Raffel, Brian McFee, Eric J. Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, and Daniel P. W. Ellis. Mir\_eval: A transparent implementation of common mir metrics. In *International Society for Music Information Retrieval Conference*, 2014.
- [133] Eita Nakamura, Yasuyuki Saito, and Kazuyoshi Yoshii. Statistical learning and estimation of piano fingering. *ArXiv*, abs/1904.10237, 2019.
- [134] Laura Smith, Ilya Kostrikov, and Sergey Levine. A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *ArXiv*, abs/2208.07860, 2022.
- [135] Takuya Hiraoka, Takahisa Imagawa, Taisei Hashimoto, Takashi Onishi, and Yoshimasa Tsuruoka. Dropout q-functions for doubly efficient reinforcement learning. 2022.
- [136] Tuomas Haarnoja, Aurick Zhou, P. Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.
- [137] Taylor A. Howell, Nimrod Gileadi, Saran Tunyasuvunakool, Kevin Zakka, Tom Erez, and Yuval Tassa. Predictive sampling: Real-time behaviour synthesis with mujoco. *ArXiv*, abs/2212.00541, 2022.

- [138] Dean A. Pomerleau. Alvin: An autonomous land vehicle in a neural network. In *NIPS*, 1988.
- [139] Stéphane Caron, Yann De Mont-Marin, Rohan Budhiraja, Seung Hyeon Bang, Ivan Domrachev, Simeon Nedelchev, Peter Du, Joris Vaillant, and Bruce Wingo. Pink: Python inverse kinematics based on Pinocchio, 2025. URL <https://github.com/stephane-caron/pink>.
- [140] Tomomichi Sugihara. Solvability-unconcerned inverse kinematics by the Levenberg–Marquardt method. *IEEE Transactions on Robotics*, 27:984–991, 2011.
- [141] Joan Solà, Jérémie Deray, and Dinesh Atchuthan. A micro Lie theory for state estimation in robotics. *arXiv preprint arXiv:1812.01537*, 2018.
- [142] Cheng Fang, Alessio Rocchi, Enrico Mingo Hoffman, Nikos G. Tsagarakis, and Darwin G. Caldwell. Efficient self-collision avoidance based on focus of interest for humanoid robots. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 1060–1066, 2015. doi: 10.1109/HUMANOIDS.2015.7363500.
- [143] Stéphane Caron et al. qpsolvers: Quadratic programming solvers in Python, 2025. URL <https://github.com/qpsolvers/qpsolvers>.
- [144] Daniel Arnström, Alberto Bemporad, and Daniel Axehill. A dual active-set solver for embedded quadratic programming using recursive  $LDL^t$  updates. *IEEE Transactions on Automatic Control*, 67:4362–4369, 2022.
- [145] Brent Yi, Michelle Lee, Alina Kloss, Roberto Martín-Martín, and Jeannette Bohg. Differentiable factor graph optimization for learning smoothers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.
- [146] Joao Pedro Araujo, Yanjie Ze, Pei Xu, Jiajun Wu, and C. Karen Liu. Retargeting matters: General motion retargeting for humanoid motion tracking, 2025. URL <https://arxiv.org/abs/2510.02252>.
- [147] Yanjie Ze, Zixuan Chen, João Pedro Araújo, Zi-ang Cao, Xue Bin Peng, Jiajun Wu, and C. Karen Liu. TWIST: Teleoperated whole-body imitation system, 2025. URL <https://arxiv.org/abs/2505.02833>.
- [148] Daniel E. Whitney. Resolved motion rate control of manipulators and human prostheses. *IEEE Transactions on Man-Machine Systems*, 10(2):47–53, 1969. doi: 10.1109/TMMS.1969.299896.
- [149] Alain Liégeois. Automatic supervisory control of the configuration and behavior of multi-body mechanisms. 1977.

- [150] Yoshihiko Nakamura and Hideo Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of Dynamic Systems, Measurement, and Control*, 108:163–171, 1986.
- [151] Oussama Kanoun, Florent Lamiraux, and Pierre-Brice Wieber. Kinematic control of redundant manipulators: Generalizing the task-priority framework to inequality task. *IEEE Transactions on Robotics*, 27(4):785–792, 2011. doi: 10.1109/TRO.2011.2142450.
- [152] Adrien Escande, Nicolas Mansard, and Pierre-Brice Wieber. Hierarchical quadratic programming: Fast online humanoid-robot motion generation. *The International Journal of Robotics Research*, 33(7):1006–1028, 2014. doi: 10.1177/0278364914521306.
- [153] Stefano Chiaverini, Giuseppe Oriolo, and Ian D. Walker. Kinematically redundant manipulators. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 245–268. Springer, 2008.
- [154] Enrico Mingo Hoffman, Arturo Laurenzi, and Nikos G. Tsagarakis. The open stack of tasks library: OpenSoT. *IEEE Robotics & Automation Magazine*, 32:24–34, 2025.
- [155] Chung Min Kim, Brent Yi, Hongsuk Choi, Yi Ma, Ken Goldberg, and Angjoo Kanazawa. PyRoki: A modular toolkit for robot kinematic optimization, 2025.