Lyceum: An efficient and scalable ecosystem for robot learning

Colin Summers[1], **Kendall Lowrey**[1], **Aravind Rajeswaran**[1]
Siddhartha Srinivasa[1], **Emanuel Todorov**[1,2]

{colinxs, klowrey, aravraj, siddh, todorov}@cs.uw.edu
[1] *University of Washington Seattle,* [2] *Roboti LLC*

Abstract

We introduce Lyceum, a high-performance computational ecosystem for robot learning. Lyceum is built on top of the Julia programming language and the MuJoCo physics simulator, combining the ease-of-use of a high-level programming language with the performance of native C. In addition, Lyceum has a straightforward API to support parallel computation across multiple cores and machines. Overall, depending on the complexity of the environment, Lyceum is 5–30X faster as compared to other popular abstractions like OpenAI's Gym and DeepMind's dm-control. This substantially reduces training time for various reinforcement learning algorithms; and is also fast enough to support real-time model predictive control through MuJoCo. The code, tutorials, and demonstration videos can be found at: www.lyceum.ml.

1. Introduction

Progress in artificial intelligence has exploded in recent years, due in large part to advances computational in infrastructure. The advent of massively parallel GPU computing, combined with powerful automatic-differentiation tools like TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019), has lead to new classes of deep learning algorithms by enabling what was once computationally intractable. These tools, alongside fast and accurate physics simulators like MuJoCo (Todorov et al., 2012), and associated frameworks like OpenAI's Gym (Brockman et al., 2016) and DeepMind's dm_control (Tassa et al., 2018), have similarly transformed various aspects of robotic control like Reinforcement Learning (RL), Model-Predictive Control (MPC), and motion planning. These platforms enable researchers to give their ideas computational form, share results with collaborators, and deploy their successes on real systems.

From these advances, simulation to real-world, or "sim2real", transfer has emerged as a promising paradigm for robotic control. A growing body of recent work suggests that robust control policies trained in simulation can successfully transfer to the real world (OpenAI et al., 2020; Rajeswaran et al., 2017a; Sadeghi and Levine, 2016; Lowrey et al., 2018; Tobin et al., 2017; Mordatch et al., 2015). However, many algorithms used in these works for controller synthesis are computationally intensive. Training control policies with state-of-the-art RL algorithms often takes many hours to days of compute time. For example, OpenAI's landmark Dactyl work (OpenAI et al., 2020) required 50 hours of training time across 6144 CPU cores and 8 NVIDIA V100 GPUs. Such computational budgets are only available to a select few labs. Furthermore, such experiments are seldom run only once in deep learning, especially in deep RL. Indeed, RL algorithms are notoriously sensitive to choices of hyper-parameters (Rajeswaran et al., 2017b; Henderson et al., 2017; Mania et al., 2018).

Thus, many iterations of the learning process may be required, with humans in the loop, to improve hyperparameter choices and reward functions, before finally deploying solutions to the real world. This computational bottleneck often leads to a scarcity of hardware results, relative to the number of papers that propose new algorithms on highly simplified and well tuned benchmark tasks. Exploring avenues to reduce experiment turnaround time is thus crucial for scaling up to harder tasks as well as making resource-intensive algorithms and environments accessible to research labs without massive cloud computing budgets.

In a similar vein, computational considerations have also limited progress in model-based control algorithms. For real-time model predictive control (MPC), the computational restrictions manifest as the requirement to compute actions in bounded time with limited local resources. As we will show, existing frameworks such as Gym and dm_control, while providing a convenient abstraction in Python, are too slow to meet this real-time computation requirement. As a result, most planning algorithms are run offline and deployed in open-loop mode on hardware. This is unfortunate, since it does not take feedback into account, which is well known to be critical for stochastic control.

Our contributions: Our goal in this work is to overcome the aforementioned computational restrictions to enable faster training of policies with RL algorithms, facilitate real-time MPC with a detailed physics simulator, and ultimately enable researchers to engage with complex robotic tasks. To this end, we develop Lyceum, a computational ecosystem that uses the Julia programming language and the MuJoCo physics engine. Lyceum ships with the main OpenAI Gym continuous control tasks, along with other environments representative of challenges in robotics. Julia's unique features allow us to wrap MuJoCo with zero-cost abstractions, providing the flexibility of a high-level programming language to enable easy creation of environments, tasks, and algorithms, while retaining the performance of native C. This allows RL and MPC algorithms implemented in Lyceum to be 5–30X faster compared to Gym and dm_control. We hope that this speedup will enable RL researchers to scale up to harder problems with reduced computational costs and enable real-time MPC.

2. Related Works

Recently, various physics simulators and the computational ecosystems surrounding them have transformed robot learning research. They allow for exercising creativity to quickly generate new and interesting robotic scenes, as well as quickly prototype various learning and control solutions. We summarize the main threads of related work below.

Physics simulators MuJoCo (Todorov et al., 2012) has quickly emerged as a leading physics simulator for robot learning research. It is fast and efficient, and particularly well suited for contact-rich tasks. Numerous recent works have also demonstrated simulation to reality transfer with MuJoCo through physically consistent system identification (Lowrey et al., 2018) or domain randomization (OpenAI et al., 2020; Mordatch et al., 2015; Nachum et al., 2019). Our framework wraps MuJoCo in Julia and enables programming and research with a high level language, while retaining the speed of native C. While we primarily focus on MuJoCo, we believe that similar design principles can be extended to other simulators like Bullet (Coumans and Bai, 2016) and DART (Lee et al., 2018).

Computational ecosystems OpenAI's Gym (Brockman et al., 2016) and DeepMind's dm_control (Tassa et al., 2018) sparked a wave of interest by providing Python bindings for MuJoCo with a high-level API, as well as easy-to-use environments and algorithms. This has enabled the RL community to quickly access physics-based environments and prototype algorithms. Unfortunately, this flexibility comes at the price of computational performance: existing ecosystems are slow due to the inefficiencies of Python combined with its poor support for parallelization. Prior works have tried to address some of the shortcomings of Python frameworks by attempting to add "just-in-time" compilation to the language (Lam et al., 2015; Paszke et al., 2019; Agrawal et al., 2019) but only support a subset of the language, and do not achieve the same performance as Julia. Fan et al. (2018) developed a framework similar to Gym that supports distributed computing, but it still suffers the same performance issues of Python. Perhaps closest to our motivation is the work of Koolen and Deits (2019), which demonstrates the usefulness of Julia as a language for robotics. However, it uses a custom and minimalist rigid body simulator with limited support for contacts. In contrast, our work addresses the inefficiencies of existing computational ecosystems through use the of Julia, and directly wraps a more capable simulator, MuJoCo, with zero overhead.

Algorithmic toolkits and environments A number of algorithmic toolkits like OpenAI Baselines (Dhariwal et al., 2017), mjRL (Rajeswaran et al., 2017b), Soft-Learning (Haarnoja et al., 2018), and RL-lab (Duan et al., 2016); as well as environments like the Hand Manipulation Suite (Rajeswaran et al., 2018), ROBEL (Ahn et al., 2019), DoorGym (Urakami et al., 2019), and SURREAL (Fan et al., 2018) have been developed around existing computational ecosystems. Our framework supports all the underlying functionality needed to transfer these advances into our ecosystem (e.g. simulator wrappers and automatic differentiation through Flux.jl). Lyceum comes with a few popular algorithms out of the box like Natural Policy Gradient (Kakade, 2002; Rajeswaran et al., 2017b) for RL and variants of Model Predictive Path Integral (Lowrey et al., 2019; Williams et al., 2016) for MPC. In the future, we plan to port further algorithms and advances into our ecosystem and look forward to community contributions as well.

3. The Lyceum Ecosystem

The computational considerations for designing infrastructure and ecosystems for robotic control with RL and MPC are unique. We desire a computational ecosystem that is high-level and easy to use for research, but that also provides the speed of native C and support for distributed computing, which is required for control algorithms running in a tight loop on robots. We found Julia to be well-suited for these requirements and summarize some of these main advantages below. Subsequently, we outline the salient features of Lyceum.

3.1. Julia for Robotics and RL

Julia is a general-purpose programming language developed in 2012 at MIT with a focus on technical computing (Bezanson et al., 2017). While a full description of Julia is beyond the scope of this paper, we highlight a few key aspects that we leverage in Lyceum and believe make Julia an excellent tool for robotics and RL researchers.

Just-in-time compilation Julia feels like a dynamic, interpreted scripting language, enabling an interactive programming experience. Under the hood, however, Julia leverages the LLVM backend to "just-in-time" compile native machine code that is as fast as C for a variety of hardware platforms. This enables researchers to quickly prototype ideas and optimize for performance with the same language.

Julia can easily call functions in Python and C In addition to the current ecosystem of Julia packages, users can interact with Python and C as illustrated below. This allows researchers to benefit from the existing body of deep learning research (in Python) and easily interact with low-level robot hardware drivers.

```
using PyCall
so = pyimport("scipy.optimize")
so.newton(x -> cos(x) - x, 1)
ccall((:mj_step, libmujoco), Cvoid, (Ptr{mjModel}, Ptr{mjData}), m, d)
```

Easy parallelization Julia comes with extensive support for distributed and shared-memory multi-threading that allows users to trivially parallelize their code. The following example splits the indices of X across all the available cores and performs in-place multiplication in parallel:

```
@threads for i in eachindex(X)
  X[i] *= 2
end
```

Julia can also transpile to alternative hardware backends, allowing use of parallel processors like GPUs by writing high level Julia code.

3.2. Salient Features of Lyceum

Lyceum consists of the following packages

1. LyceumBase.jl, a lightweight package that defines a set of abstract environment and controller interfaces, along with several utilities.
2. MuJoCo.jl, a low-level Julia wrapper for the MuJoCo physics simulator.
3. LyceumMuJoCo.jl, a high-level "environment" abstraction similar to Gym and dm_control.
4. LyceumMuJoCoViz.jl, a flexible policy and trajectory visualizer with support for interaction.
5. LyceumAI.jl, a collection of various algorithms for robotic control.

LyceumBase.jl At the highest level we provide LyceumBase.jl, which contains several convenience utilities used throughout the Lyceum ecosystem for data logging, multithreading, and controller benchmarking (i.e. measuring throughput, jitter, etc.). LyceumBase.jl also contains interface definitions, such as `AbstractEnvironment` (which LyceumMuJoCo.jl then implements for MuJoCo).

This interface is similar to the popular Python frameworks Gym and dm_control, where an agent's observations are defined, actions are chosen, and the simulator is stepped forward in time. A few key differences are as follows:

1. The ability to arbitrarily get/set the state of the simulator, a necessary feature for model-based methods like MPC or motion planning. An important component of this is a proper definition of state, which is often missing from existing frameworks.
2. Optional, in-place versions for all functions (e.g. getstate!$(\cdot)$) which store the return value in a pre-allocated data structure. This eliminates unnecessary memory allocations and garbage collection, enabling environments to be used in tight, real-time control loops.

We expect most users will be interested in implementing their own environments, which forms a crucial part of robotics research. Indeed, different researchers may be interested in different robots performing different tasks, ranging from whole arm manipulators to legged locomotion to dexterous anthropomorphic hands. To aid this process, we provide sensible defaults for most of the API, making it easy to get started and experiment with different environments. The separation of interface and implementation also allows for other simulators and back-ends (e.g. RigidBodySim.jl or DART) to be used in lieu of the MuJoCo-based environments we provide, should the user desire.

MuJoCo.jl, LyceumMuJoCo.jl, and LyceumMuJoCoViz.jl MuJoCo.jl is a low-level Julia wrapper for MuJoCo that has a one-to-one correspondence to MuJoCo 2.0's C interface and includes soft body dynamics. All data is memory mapped to native Julia objects with no overhead. Support for accessing the simulation elements defined in a MuJoCo XML file with named indices (e.g. `d.qpos[:, :arm]`) is also provided. We then build LyceumMuJoCo.jl, the MuJoCo implementation of our `AbstractEnvironment` API, on top of MuJoCo.jl to create environment and task definitions. Finally, the LyceumMuJoCoViz.jl package provides a feature-rich visualizer that enables playback of previously recorded trajectories and allows for interaction with the simulator and control polices in real time using a mouse and keyboard. Robots in the real world encounter perturbations and disturbances, and with LyceumMuJoCoViz.jl the user can interact with the simulated environment to test the robustness of a controller.

LyceumAI.jl Coupled with these environments is LyceumAI.jl, a collection of algorithms for robotic control that similarly leverage Julia's performance and multithreading abilities. Currently we provide implementations of Model Predictive Path Integral Control (MPPI) (Williams et al., 2016), a stochastic shooting method for model-predictive control, and Natural Policy Gradient (Kakade, 2002; Rajeswaran et al., 2017b). We compare these methods with a Python implementation in the next sections. The combination of efficient compute, flexible high-level programming, and a rich ecosystem of tools for deep learning and optimization should allow both robotics and RL researchers to experiment with different robotic systems and algorithm designs, and hopefully deploy to the real world.

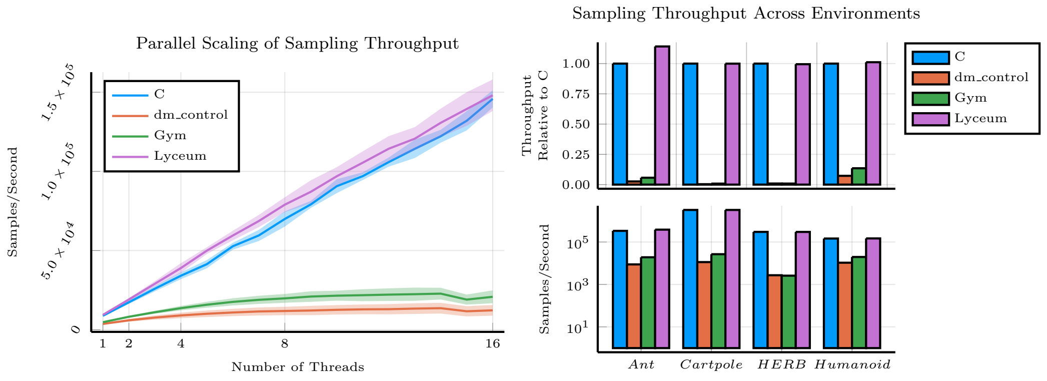
4. Benchmark Experiments and Results

We designed our experiments and timing benchmarks to answer the following questions: (a) Do the implementations of Gym environments and algorithms for RL and MPC in Lyceum produce comparable results? (b) Does Lyceum lead to faster environment sampling and experiment turn-around time when compared to Gym and dm_control?

Methodology All experiments are performed on a 16-core Intel i9-7960X with the CPU governor pinned at 1.2GHz so as to prevent dynamic scaling or thermal throttling from affecting results. Parallelization for Lyceum, C, dm_control, and Gym is implemented as follows:

1. Lyceum: Julia's built-in `Threads.@threads` macro.
2. C: OpenMP's `#pragma omp parallel for`.
3. dm_control: A modification of deepmind/dm_control/92.
4. Gym: A hybrid of Gym's `AsyncVectorEnv` and OpenAI Baseline's `SubprocVecEnv`.

Below we describe the various benchmarks we considered and their results.



	Scaling Factor
C	0.97 ± 0.1
dm_control	0.08 ± 0.04
Gym	0.15 ± 0.05
Lyceum	0.96 ± 0.07

Figure 1: Comparison of parallel sampling throughput across frameworks. Throughput first was measured using 1–16 cores for the `Humanoid` model (left), with the scaling factor reported in the table, as well across models of varying complexity using all 16 cores. The raw throughput on a log scale (bottom right) as well as the performance relative to native C (top right) is shown. We find that Lyceum can match the performance of native C, while still retaining the benefits of writing research code in a high-level language.

Sampling throughput In the first benchmark, we study the sampling throughput and parallel scaling performance of LyceumMuJoCo.jl against Gym, dm_control, and a native C implementation. To do so, we consider various models of increasing complexity: `CartPole`, `Ant`, `HERB`, and `Humanoid`. In the first experiment, we study how the sampling throughput scales with the number of cores for the various implementations. To do so, we consider the `Humanoid` environment and measure the number of samples that can be generated per second while varying the number of cores utilized. The results are plotted in Figure 1 (left) and summarized in the table below, where we see substantial gains for Lyceum. In particular, the performances scales linearly with the number of cores for C and Lyceum, while there are diminishing returns for Gym and dm_control. This is due to the inherent parallelization limitations of Python. When using more cores (e.g. on a cluster), the performance difference is likely to be even larger.

In the second experiment, we use all 16 of the available cores to measure the number of samples we can collect per second. Figure 1 (right) shows the results, which are presented in two forms: as a fraction of native C's throughput, and as samples per second. We see that Lyceum and native C significantly outperform Gym and dm_control in all cases. In particular, for `CartPole`, Lyceum is more than 200x faster as compared to Gym.

Reinforcement learning with policy gradients In the second benchmark, we compare the learning curves and wall clock time of Natural Policy Gradient (NPG) (Kakade, 2002; Rajeswaran et al., 2017b), which is closely related to Trust Region Policy Optimization (Schulman et al., 2015), between Gym and Lyceum. Our implementation of NPG, closely based on the algorithm as described

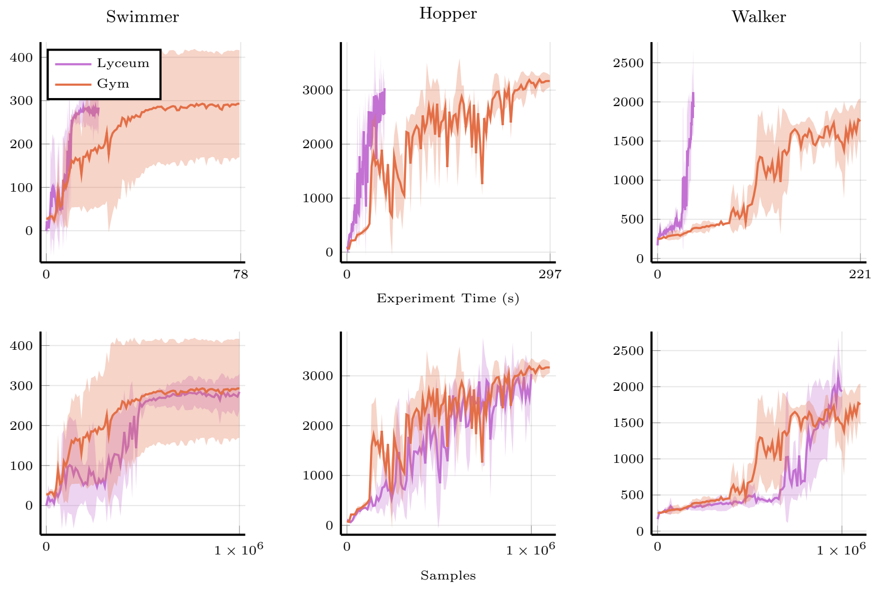
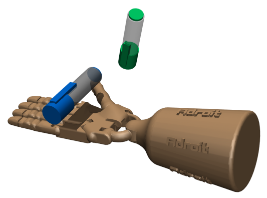


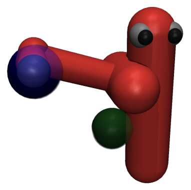
Figure 2: Reinforcement learning in Gym and Lyceum using the Natural Policy Gradient algorithm, trained for one million time-steps. Environment reward is plotted against both wall-clock time (top) and simulator timesteps (bottom), showing that similar performance is achieved in significantly less time using Lyceum. Performance of the underlying deterministic policy is reported.

in Rajeswaran et al. (2017b) and consistent with majority practice in the community, considers two layer neural network policies. Details about hyperparameters are provided in on the website. We compare based on three representative tasks (`Swimmer`, `Hopper`, and `Walker`) and find that the learning curves match across the two frameworks. The results are summarized in Figure 2 show that the performance curves match well. We note that RL algorithms are known to be sensitive to many implementation details (Henderson et al., 2017; Ilyas et al., 2018), and thus even approximately matching results is a promising sign for both the original code base and Lyceum. The important conclusion, however, is that similar results can be achieved in a fraction of the time.

Model Predictive Control In the final benchmark, we compare the performance of a model-based trajectory optimizer in Gym and Lyceum. For this purpose, we consider the Model Predictive Path Integral (MPPI) algorithm (Williams et al., 2016), which in conjunction with learning-based techniques has demonstrated impressive results in tasks like aggressive driving and dexterous hand manipulation (Lowrey et al., 2018; Nagabandi et al., 2019). MPPI is a sampling-based algorithm where different candidate action sequences are considered to generate many potential trajectories starting from the current state. Rewards are calculated for each of these trajectories and the candidate action sequences are combined with exponentially-weighted trajectory rewards.

We consider two tasks for the MPPI comparison: a 7-DOF sawyer arm where the goal is to reach various spatial goals with its end effector, and a 30-DOF in-hand manipulation task that requires a Shadow Hand (Adroit) (Kumar, 2016) to perform in-hand manipulation of a pen in order to match a target configuration. We compare the times taken by MPPI to optimize a trajectory as well as the fraction of times MPPI generated a successful trajectory. The results are provided in Figure 3. In





Task	Time (s)	Success (%)
Hand (Gym)	40.4	92 ± 5
Hand (Lyceum)	**14.3**	88 ± 6
Reacher (Gym)	3.67	100
Reacher (Lyceum)	**0.119**	100

Figure 3: (Left) Illustration of the 30-DOF in-hand manipulation task with a Shadow Hand (Adroit). The goal is to manipulate the (blue) pen to match the (green) desired pose. (Middle) Illustration of the reaching task with a 7-DOF Sawyer arm. Goal is to make the end-effector (blue) reach the (green) target. (Right) comparison of time taken and success percentage in Gym and Lyceum. Time refers to the time taken to execute a single episode with the MPPI controller (in MPC mode). Success % measures the number of successful episodes when the robot is controlled using the MPPI algorithm. 95% confidence intervals are also reported. See website for additional details and hyperparameters.

summary, we find that the MPPI success percentages are comparable between Gym and Lyceum, while the Lyceum implementation is approximately 30x faster for the Sawyer arm task and 3x faster for the Shadow Hand task. This trend is consistent with the earlier trend, where the relative differences are larger for lower dimensional systems with fewer contacts. This is because for complex models with many contacts like the Shadow Hand, most of the computational work is performed by MuJoCo, thereby diminishing the impact of overhead in Gym and Python. We note, however, that we found the parallel scaling performance to be significantly better in Lyceum as compared to Gym, and thus the difference between the frameworks is likely larger when using more cores (e.g. on a cluster).

5. Conclusion and Future Work

We introduced Lyceum, a new computational ecosystem for robot learning in Julia that provides the rapid prototyping and ease-of-use benefits of a high-level programming language, yet retaining the performance of a low-level language like C. We demonstrated that Lyceum enables substantial performance gains as compared to existing ecosystems like OpenAI Gym and dm_control. We also demonstrated that this speed up enables faster experimentation times for RL and MPC algorithms. In the future, we hope port over additional algorithmic infrastructure like OpenAI Baselines (Dhariwal et al., 2017). We also hope to include and support models and environments involving real robots like Cassie (Agility Robotics, 2017) and ROBEL (Ahn et al., 2019).

Acknowledgments

The authors thank Ben Evans for continued valuable feedback over the course of the project. The authors also thank Vikash Kumar and Svet Kolev for valuable comments about the paper draft, and Motoya Ohnishi for early adoption of Lyceum and feedback. This work was (partially) funded by the National Institute of Health R01 (#R01EB019335), National Science Foundation CPS (#1544797), National Science Foundation NRI (#1637748), the Office of Naval Research, the RCTA, Amazon, and Honda Research Institute USA.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

Agility Robotics. Cassie, 2017. URL https://www.agilityrobotics.com/robots#cassie.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning, 2019.

Michael Ahn, Henry Zhu, Kristian Hartikainen, Hugo Ponte, Abhishek Gupta, Sergey Levine, and Vikash Kumar. ROBEL: Robotics benchmarks for learning with low-cost robots. In *Conference on Robot Learning*, 2019.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 2017. doi: 10.1137/141000671.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016. URL https://gym.openai.com/.

Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning, 2016. URL http://pybullet.org.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. *GitHub repository*, 2017. URL https://github.com/openai/baselines.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control, 2016.

Linxi Fan, Yuke Zhu, Jiren Zhu, Zihua Liu, Orien Zeng, Anchit Gupta, Joan Creus-Costa, Silvio Savarese, and Li Fei-Fei. Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In *Conference on Robot Learning*, 2018.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2018.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2017.

Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Are deep policy gradient algorithms truly policy gradient algorithms?, 2018.

Sham M. Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems*, 2002.

Twan Koolen and Robin Deits. Julia for Robotics: Simulation and real-time control in a high-level programming language. *International Conference on Robotics and Automation*, 2019. doi: 10.1109/ICRA.2019.8793875.

Vikash Kumar. Manipulators and manipulation in high dimensional spaces, 2016. URL https://digital.lib.washington.edu/researchworks/handle/1773/38104.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015. doi: 10.1145/2833157.2833162.

Jeongseok Lee, Michael X. Grey, Sehoon Ha, Tobias Kunz, Sumit Jain, Yuting Ye, Siddhartha S. Srinivasa, Mike Stilman, and Chuanjian Liu. DART: Dynamic animation and robotics toolkit. *Journal of Open Source Software*, 2018. doi: 10.21105/joss.00500.

Kendall Lowrey, Svetoslav Kolev, Jeremy Dao, Aravind Rajeswaran, and Emanuel Todorov. Reinforcement learning for non-prehensile manipulation: Transfer from simulation to physical system. In *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2018.

Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan online, learn offline: Efficient learning and exploration via model-based control. In *International Conference on Learning Representations*, 2019.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Conference and Workshop on Neural Information Processing Systems*, 2018.

Igor Mordatch, Kendall Lowrey, and Emanuel Todorov. Ensemble-CIO: Full-body dynamic motion planning that transfers to physical humanoids. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015. doi: 10.1109/IROS.2015.7354126.

Ofir Nachum, Michael Ahn, Hugo Ponte, Shixiang Gu, and Vikash Kumar. Multi-agent manipulation via locomotion using hierarchical sim2real, 2019.

Anusha Nagabandi, Kurt Konoglie, Sergey Levine, and Vikash Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, 2019.

OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *International Journal of Robotics Research*, 2020. doi: 10.1177/0278364919887447.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine. EPOpt: Learning robust neural network policies using model ensembles. In *International Conference on Learning Representations*, 2017a.

Aravind Rajeswaran, Kendall Lowrey, Emanuel V. Todorov, and Sham M. Kakade. Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, 2017b.

Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. In *Proceedings of Robotics: Science and Systems*, 2018. doi: 10.15607/RSS.2018.XIV.049.

Fereshteh Sadeghi and Sergey Levine. CAD2RL: Real single-image flight without a single real image, 2016.

John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *International Conference on Machine Learning*, 2015.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. Deepmind control suite, 2018.

Josh Tobin, Lukas Biewald, Rocky Duan, Marcin Andrychowicz, Ankur Handa, Vikash Kumar, Bob McGrew, Alex Ray, Jonas Schneider, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Domain randomization and generative models for robotic grasping. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017. doi: 10.1109/IROS.2018.8593933.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012. doi: 10.1109/IROS.2012.6386109.

Yusuke Urakami, Alec Hodgkinson, Casey Carlin, Randall Leu, Luca Rigazio, and Pieter Abbeel. DoorGym: A scalable door opening environment and baseline agent, 2019.

Grady Williams, Paul Drews, Brian Goldfain, James M. Rehg, and Evangelos A. Theodorou. Aggressive driving with model predictive path integral control. In *IEEE International Conference on Robotics and Automation*, 2016. doi: 10.1109/ICRA.2016.7487277.