LYCEUM: AN EFFICIENT AND SCALABLE ECOSYSTEM FOR ROBOT LEARNING

Anonymous authors
Paper under double-blind review

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. Lyceum is up to 10-20X faster 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 with physics simulators. Lyceum has a straightforward API and supports parallel computation across multiple cores or machines. The code base, tutorials, and demonstration videos can be found at: `https://sites.google.com/view/lyceum-anon`.

1 INTRODUCTION

Progress in deep learning and artificial intelligence has exploded in recent years, due in large part to growing computational 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., 2017b), have lead to new classes of algorithms by enabling what was once computational 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 (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 (Lowrey et al., 2018a; OpenAI, 2018; Rajeswaran et al., 2016; Sadeghi & Levine, 2016; Tobin et al., 2017). Despite these advances, many algorithms are computationally inefficient and have been unable to scale to complex problem domains. Training control policies with state-of-the-art RL algorithms often takes hours to days of compute time. For example, OpenAI's extremely impressive Dactyl work (OpenAI, 2018) required 50 hours of training time across 6144 CPU cores and 8 powerful NVIDIA V100 GPUs. Such computational budgets are available only to a select few labs. Furthermore, such experiments are seldom run only once in deep learning and especially in deep RL. Indeed, RL algorithms are notoriously sensitive to choices of hyper-parameters and require reward shaping (Henderson et al., 2017; Rajeswaran et al., 2017; 2018; Mania et al., 2018). Thus, many iterations of the learning process may be required, with humans in the loop, to improve reward and hyperparameters, before deploying solutions in 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 (Brockman et al., 2016; Tassa et al., 2018). Exploring avenues to reduce experiment turn around time is thus crucial to scaling up to harder tasks and 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, the computational restrictions manifest as the requirement to compute controls 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 more 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/C++. This allows RL and MPC algorithms implemented in Lyceum to be 10-100X faster compared to Gym and dm_control. We hope that this speedup will enable RL researchers to scale up to harder problems without increased computational costs, as well as enable real-time MPC that looks ahead through a simulator.

2 Related Works

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

Physics simulators: MuJoCo (Todorov et al., 2012) has quickly emerged as a leading physics simulators for robot learning research. It is fast and efficienct, 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., 2018a) or domain randomization (OpenAI, 2018; Mordatch et al., 2015; Nachum et al., 2019). Our framework wraps MuJoCo in Julia and enables programming and research with a high level language yet retaining the speed of a low level language like C. While we primarily focus on MuJoCo, we believe that similar design principles can also be extended to other simulators such as Bullet (Coumans, 2013) and DART (Lee et al., 2018).

Computational ecosystems: Even though MuJoCo has been around for at least 6-7 years, it's mainstream adoption and popularity grew more recently in the past 3-4 years after the introduction of computational ecosystems around the simulator. Specifically, 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 (which itself is written in C) as well as easy-to-use environments and a high-level python API. This has enabled the RL community to quickly access physics-based environments and prototype algorithms. Unfortunately, this flexibility comes at the price of computational efficiency. Existing ecosystems are slow due to inefficiencies and poor parallelization capabilities of Python. Prior works have tried to address some of the shortcomings of Python-based frameworks by attempting to add JIT compilation to the language (Lam et al., 2015; Paszke et al., 2017a; 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 and multi-processing. Perhaps closest to our motivation is the work of Koolen & 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 contact support. In contrast, our work attempts to address the inefficiencies of existing computational ecosystems through use of Julia, and directly wraps a high-quality and extensively supported simulator like MuJoCo with zero overhead.

Algorithmic toolkits and environments: A number of algorithmic toolkits like OpenAI Baselines (Dhariwal et al., 2017), MJRL (Rajeswaran et al., 2017), Soft-Learning (Haarnoja et al., 2018), and RL-lab (Duan et al., 2016); as well as environments like Hand Manipulation Suite (Rajeswaran et al., 2018), Door Gym (Urakami et al., 2019), and Surreal Robosuite (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). Lyceum comes with a few popular robotics algorithms out of the box like PPO and Natural Policy Gradient for RL as well as several MPC algorithms like "Model Predictive Path Integral" (Williams et al., 2016). In the future, we plan to port a number of additional algorithms and advances into our ecosystem.

3 Compute Considerations

Robotic control with RL and MPC requires unique computational considerations when designing infrastructure and ecosystems. RL algorithms are typically inherently parallel. Consider the class of policy gradient RL algorithms which have demonstrated very impressive results in a variety of tasks (OpenAI, 2018; OpenAI; Vinyals et al., 2019) or evolutionary methods for policy learning. They require rolling out many trajectories to form a dataset, on which the policy is updated. These rollouts can be parallelized and so we would like a computational infrastructure that yields performance which scales linearly with increasing number of cores. RL algorithms are also notoriously sensitive to many hyperparameter details, as well as reward shaping (Henderson et al., 2017; Rajeswaran et al., 2017; 2018). These invariably need to be done sequentially with a human in the loop, especially to refine and design the reward functions. Thus, experiment turn around times are critical, which are largely limited by the speed of serial operations within each thread.

Similar considerations also apply to real-time MPC, where sampling based algorithms like MPPI (Williams et al., 2016), POLO (Lowrey et al., 2018b), or iLQR (fitted LQR) (Todorov & Li, 2005; Kumar et al., 2016; Levine & Abbeel, 2014) can be parallelized and would benefit from a framework that facilitates this. In addition, real-time MPC also poses the challenge of requiring computation of action to happen within the control loop time period, to incorporate correct real-world feedback. In robotic hardware, this typically must be done with locally available (often on-board) compute, since cloud computing invariably has significantly larger latency than hardware control loop period. This places an even stronger emphasis on extremely efficient serial operations within threads to match the strict bounded time computation requirements.

4 The Lyceum ecosystem

We now describe the structure of our ecosystem and the advantages it provides. Lyceum consists of the following Julia packages which together empower robotics researchers with the ease of use of a high-level, dynamic programming language with all performance of a compiled, low-level language.

1. `Lyceum.jl`, a "base" package which defines a set of abstract environment and controller interfaces along with several utilities.
2. `LyceumAI.jl`, a collection of various algorithms for robotic control.
3. `MuJoCo.jl`, a low-level Julia wrapper for the MuJoCo physics simulator.
4. `LyceumMuJoCo.jl`, a high-level "environment" abstraction similar to Gym and dm_control.

4.1 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.

4.1.1 Just-in-time compilation

Julia feels like a dynamic, interpreted scripting language. Its high-level syntax, garbage-collected memory management, and feature-rich "Read-Eval-Print-Loop" (REPL) command-line interface makes programming in Julia quick and interactive. Under the hood, however, Julia leverages the LLVM backend to "just-in-time" (JIT) compile native machine code that is as fast as C for a variety of hardware platforms Bezanson et al. (2017). This allows researchers in robotics and RL to rapidly express their ideas in code while maintaining the high-performance at runtime that is critical to many problem domains.

4.1.2 JULIA IS FAST

A core philosophy of Julia is that users' code should be just as powerful and fast as the core language. Indeed, most of Julia is itself implemented in Julia! This empowers users to create powerful tools and packages like Lyceum without requiring low-level knowledge of the language or waiting for a library to implement a desired feature. This is highly beneficial for algorithmic development since the kinds of operations available to the researcher are often restricted by what functionality is afforded by the computational platform. For example, use of optimization methods based on Newton's method were out of favor in machine learning until the feature of Hessian-vector products were implemented in low-level languages like C and wrapped into packages like TensorFlow and PyTorch.

4.1.3 BROADCASTING

While many languages and libraries provide the notion of "broadcasting" or vectorizing certain operations, Julia extends this capability to all functions and can combine multiple operations into a single in-place, fused loop. Consider the following example:

```
X .= 2 .* Y.^2 .+ exp.(Z)
```

which compiles down to a single loop that computes $2y^2 + exp(z)$ for every element in Y and Z, assigning it to it's appropriate location in X. Critically, this allows researchers to flexibly and efficiently apply several operations to a data structure without worrying about which specific vectorized function to use (e.g. BLAS's gemm function) or waiting for a library to implement it if it does not exist.

4.1.4 INTERFACING WITH OTHER LANGUAGES

One the greatest benefits of Python is the massive ecosystem of packages it provides. While Julia has many powerful tools like Flux.jl for deep-learning and Optim.jl for optimization, Julia users are also able to easily interact with Python through PyCall.jl in just a few lines of code:

```
using PyCall
so = pyimport("scipy.optimize")
so.newton(x -> cos(x) - x, 1)
```

Similarly, Julia comes with a straightforward and zero-overhead interface for calling C as shown in this snippet from `MuJoCo.jl`:

```
ccall((:mjr_getError,libmujoco),Cint,())
```

Similar tools also exist for other languages like R and C++. This crucially allows Julia researchers to tap into a deep well of tools and libraries and allows roboticists to interact with low-level hardware drivers.

4.2 PARALLEL COMPUTING

Julia also 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 does an in-place multiplication of X by 2 in parallel across an internal thread pool:

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

This equips researchers with the power to parallelize their implementations and make maximal use of high-core count CPUs without worrying about about over-scheduling their machines with too

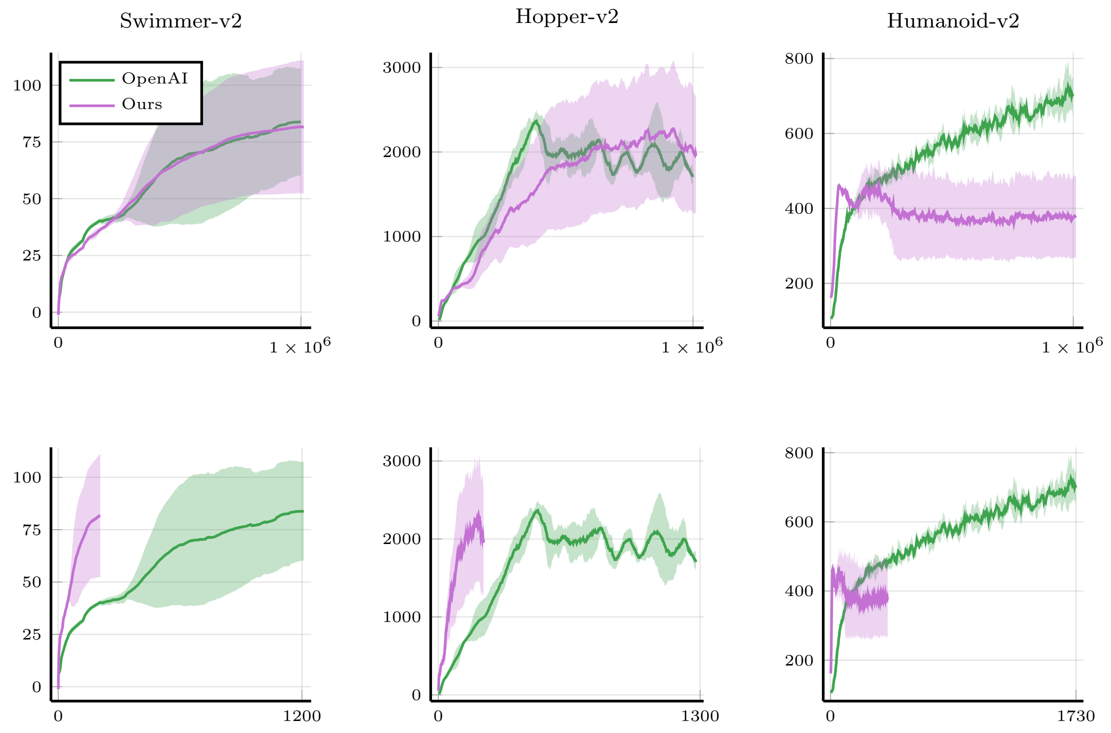

Figure 1: Training performance our's and OpenAI Baseline's implementations of Proximal Policy Optimization (PPO) for 1 million timesteps with Swimmer, Hopper, and Humanoid. The top is plotted against simulator time, while the bottom is bottom is against wall clock time.

many threads. Lyceum is designed with this feature in mind and supports parallel computation out-of-the-box.

4.2.1 Package Management

All these features and more make writing both performant and generic code easy for the programmer, contributing to a rich ecosystem of open-source Julia packages like Lyceum. To handle the 3000+ packages available, Julia comes with a built-in package manager that prevents having to separately manage a build system and helps to avoid "dependency hell" by allowing each package to provide its own binaries and maintain separate versions of its dependencies. This means less time is spent getting things to run and more time for focusing on the task at hand.

4.3 Salient Features of Lyceum

4.3.1 `Lyceum.jl`

At the highest level we provide `Lyceum.jl`, This base package contains several convenience utilities used throughout the Lyceum ecosystem for data logging, multi-threading, and controller benchmarking (i.e. measuring throughput, jitter, etc.), and more. `Lyceum.jl` also contains interface definitions, such as `AbstractEnv` which `LyceumMuJoCo.jl`, discussed below, implements. The full AbstractEnv interface is as follows:

```
statespace(env::AbstractEnv)
getstate!(state, env::AbstractEnv)
getstate(env::AbstractEnv)

observationspace(env::AbstractEnv)
getobs!(observation, env::AbstractEnv)
```

```
getobs(env::AbstractEnv)

actionspace(env::AbstractEnv)
getaction!(action, env::AbstractEnv)
getaction(env::AbstractEnv)
setaction!(env::AbstractEnv, action)

getreward(env::AbstractEnv)

# Task evaluation metric
# which may be the reward,
# or something else entirely
# (e.g. a "sparse" task-completion reward)
evalspace(env::AbstractEnv)
geteval(env::AbstractEnv)

# Reset to a fixed, initial state.
reset!(env::AbstractEnv)
# Reset to `state`
reset!(env::AbstractEnv, state)

# Reset to a random initial state.
randreset!(env::AbstractEnv)

# Step the environment, return reward.
step!(env::AbstractEnv)
# Apply `action`,
# step the environment, return reward.
step!(env::AbstractEnv, action)
```

This interface is similar to the popular Python frameworks Gym and dm_control with a few differences:

1. The ability to arbitrarily get/set the state of the simulator, a necessary feature for model-based methods like MPC or motion planning. Note that an important component of this is defining a proper notion of a 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 user-allocated data structure. This eliminates unnecessary memory allocations and garbage collection, enabling environments to be used in tight, real-time loops.

3. An optional "evaluation" metric. Often times reward functions are heavily "shaped" and hard to interpret. The evaluation metric serves as a measure of the true task-completion reward that you want to optimize for.

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. Additionally, only a subset of these functions need to be overridden should something other than the default behavior be desired. For example, an environment can only override getobs!$(\cdot)$ and observationspace$(\cdot)$, while our framework uses the information provided by observationspace$(\cdot)$ to pre-allocate the required data structure and pass it to getobs!$(\cdot)$. Having both in-place and out-of-place operations allows users to choose convenience vs greater performance as desired.

This separation of interface and implementation 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.

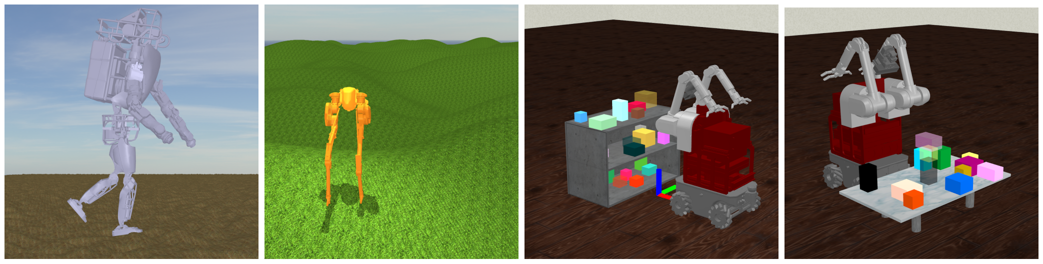

Figure 2: Two instantiations of the procedurally generated variable terrain environments using ATLAS and Cassie (top) as well as shelf and tabletop versions of the Reconfiguration Planning environment utilizing HERB as the robot.

4.3.2 `MuJoCo.jl` AND `LyceumMuJoCo.jl`

The next package we provide is `MuJoCo.jl`, a low-level Julia wrapper for MuJoCo which has a one-to-one correspondence to MuJoCo's C interface. We then build `LyceumMuJoCo.jl`, the MuJoCo implementation of our AbstractEnv API, on top of `MuJoCo.jl`.

4.3.3 ENVIRONMENTS AND TASK DOMAINS

Along with this interface we provide ports of CartPole, Swimmer, Ant, HalfCheetah, and Humanoid from the popular OpenAI Gym framework, along with two new environments, each with their own task-specific reward and evaluation functions. The first is the problem domain of reconfiguration planning (Figure 2), which has been studied earlier in motion planning literature (kin; nie). The task is defined by a set of N total movable objects and $M \subset N$ target objects. Each target object has a specified goal configuration while the remaining $N \setminus M$ objects represent movable obstacles, possibly with their own set of constraints (e.g. on the right side of Figure 2 the objects must stay on the tabletop). The reconfiguration planning domain explores interesting challenges related to: (a) reward under-specification – where only desired goal configurations of a subset of objects are specified; thereby having a multiplicity of equally good and valid solutions; (b) sequential manipulation – in certain cases, some objects may have to be moved out of the way to reach and manipulate other objects; (c) variable number of objects – most current work in RL is concerned with manipulating a fixed number of known objects, whereas in reconfiguration planning there can be a wide variety in number of objects and scenes. We also provide a locomotion environment (Figure 2) in which an agent must navigate robustly through rough terrain, hills, and terraces, represented as height maps. These environments are procedurally-generated, allowing for the environment to be modified by changing a few parameters, support multi-threading out-of-the-box, are non-allocating, and have zero-overhead over using MuJoCo in C directly. Note that our goal is to not solve these task domains in this work, but rather provide an interesting set of environments the research community can work on.

4.3.4 `LyceumAI.jl`

Coupled with these environments is `LyceumAI.jl`, a collection of algorithms for robotic control that similarly leverage Julia's performance and multi-threading abilities. Currently we provide implementations of "Model Predictive Path Integral Control" (MPPI), a stochastic shooting method for model-predictive control and Natural Policy Gradient.

5 BENCHMARK EXPERIMENTS

We designed our experiments and benchmarks to address the following questions: (a) Does Julia faciliate high-performance programming in the robotics and RL domains (as discussed in Section 3)? (b) Does our computational ecosystem and wrapper lead to faster implementation and experiment time than Gym and dm_control, and in particular is Lyceum faster than real-time for MPC?

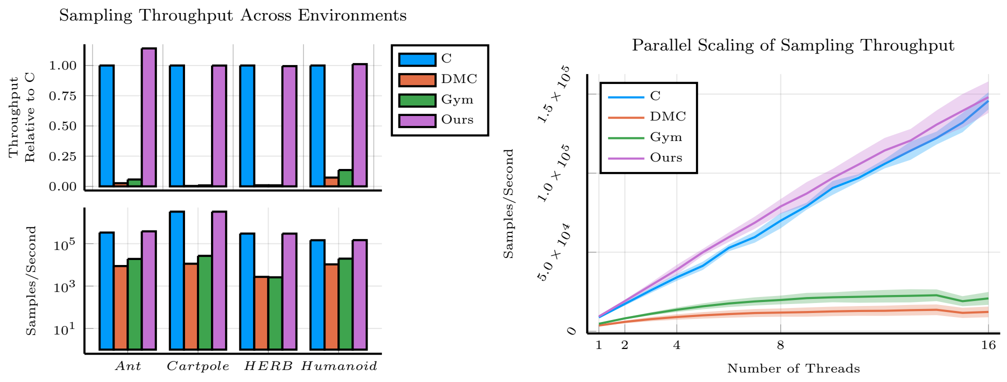

Figure 3: Sampling throughput for a batch of 1024 samples collected in parallel for four environments of various complexity as well as throughput scaling with increasing cores. Top left: sampling throughput relative to native C. Bottom left: sampling throughput on a log scale. Right: scaling with increasing cores.

5.1 Benchmark Tasks

To answer the above question, we will consider different models of increasing complexity: CartPole, Ant, our reconfiguration environment using HERB, and Humanoid. 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. As Gym and dm_control do not come with support for parallel computing, the authors implement this functionality using Python's `multiprocessing` library as recommended in several GitHub Issues by the respective library authors.

In the first task, we explore the parallel scaling performance of `LyceumMuJoCo.jl` against Gym, dm_control, and a native C implementation using a OpenMP thread pool. We collect 1024 samples in parallel with 1 through 16 threads using the humanoid.xml model that ships with MuJoCo.

In the second task, we compare the relative sampling performance for each of the models listed above by again collecting 1024 samples in parallel but with a fixed 16 threads.

The third task examines the real time factor using our Julia MPPI implementation and a similar implementation in Python using Gym for each of the models listed above. The real-time factor is defined as:

$$\frac{\Delta t_{\text{Simulator}}}{\Delta t_{\text{Controller}}}$$

where $\Delta t_{\text{Simulator}}$ is the timestep that our simulator uses and $\Delta t_{\text{Controller}}$ is the timestep per iteration of MPPI measured by wall-clock time. A real-time factor of 1 indicates that the controller runs at the same speed as the simulator, while a real-time factor of 2 indicates that the controller runs twice as fast as the simulator. While we collect the data at the same, fixed 1.2GHz, we scale all timings up to what we would expect at 3.3GHz, the sustained clock-speed of our processor, for a more realistic presentation.

6 Benchmark Results

Figure 3 illustrates how sampling throughput scales with increasing core counts for Gym, DMC, Lyceum, and a native C implementation used OpenMP. While Lyceum and C scale 98% and 95% of linear between 1 and 16 threads, Gym and DMC only achieve 29% and 23% scaling, respectively. Additionally, the authors noted that this gap grew when experiments were run without the CPU frequency was unlocked and allowed to scale to 3.3GHz. Upon profiling the Python implementation it was discovered that a significant component of the slowdown is due to inter-process communication overhead and waiting on locks. Thus it appears that the distributed Python implementations are largely IO-bound.

Figure 3 compares sampling throughput across environments of different complexity. We provide this data in two forms: as a fraction of native C's throughput and as samples per second on a log scale. As seen, Lyceum and native C significantly outperform Gym and dm_control in all cases. The relative performance between the best-performing and worst-performing implementations for Ant, CartPole, HERB, and Humanoid are 44x, 286x, 116x, and 14x, respectively. The smaller differences in the more complex environments is likely due to the fact that each process in the Gym/dm_control environments is performing more work, amortizing the overhead of inter-process communication. It should also be noted that Lyceum and C scale linearly through the entire range of threads, while Gym and dm_control appear to flatline around 10 threads. This suggests that these frameworks may be unable to benefit from higher core count CPUs.

While increased sampling throughput is critical for many algorithms, end-to-end performance matters most. Figure 1 compares an implementation of Proximal Policy Optimzation (PPO) in our framework against OpenAI Baseline's implementation for 1 million timesteps in the OpenAI Gym environments Swimmer, Hopper, and Humanoid. Each experiment was averaged over three random seeds and performed on a single core with the default OpenAI Baseline hyper-parameters. For Hopper and Swimmer, our implementation yields similar training curves to OpenAI, while lagging behind for the Humanoid experiment. This difference is likely due to the number of additions made to the core PPO algorithm in OpenAI Baseline's including reward and observation scaling, value function gradient clipping, orthonormal paramter initialization, and more. The important difference, however, is the far shorter training time as measured by wall clock time. This speed up comes from the increased sampling efficieny provided by Lyceum, as well as the high performance auto-differentiation framework Flux.jl.

7 Conclusion and Future Work

We intruced 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 this ecosystem can obtain 10-20X speedups compared to existing ecosystems like OpenAI gym and dm_control. We also demonstrated that this speed up enables faster experimental times for RL algorithms, as well as real-time model predictive control. In the future, we hope to port over algorithmic infrastructures like OpenAI's baselines (Dhariwal et al., 2017) as well as environments like hand manipulation suite (Rajeswaran et al., 2018) and DoorGym (Urakami et al., 2019).

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 *OSDI*, 2016.

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. *ArXiv*, abs/1903.01855, 2019.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. URL https://doi.org/10.1137/141000671.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Erwin Coumans. Bullet physics library, 2013.

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

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *ArXiv*, abs/1604.06778, 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. Technical report, 2018.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *ArXiv*, abs/1709.06560, 2017.

Twan Koolen and Robin Deits. Julia for robotics: simulation and real-time control in a high-level programming language. *2019 International Conference on Robotics and Automation (ICRA)*, pp. 604–611, 2019.

Vikash Kumar, Emanuel Todorov, and Sergey Levine. Optimal control with learned local models: Application to dexterous manipulation. In *ICRA*, 2016.

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*, LLVM '15, pp. 7:1–7:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4005-2. 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. *J. Open Source Software*, 3:500, 2018.

Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *NIPS*, 2014.

Kendall Lowrey, Svetoslav Kolev, Jeremy Dao, Aravind Rajeswaran, and Emanuel Todorov. Reinforcement learning for non-prehensile manipulation: Transfer from simulation to physical system. *CoRR*, abs/1803.10371, 2018a.

Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan Online, Learn Offline: Efficient Learning and Exploration via Model-Based Control. *CoRR*, abs/1811.01848, 2018b.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *NeurIPS*, 2018.

Igor Mordatch, Kendall Lowrey, and Emanuel Todorov. Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids. *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5307–5314, 2015.

Ofir Nachum, Michael J. Ahn, Hugo Ponte, Shixiang Gu, and Vikash Kumar. Multi-agent manipulation via locomotion using hierarchical sim2real. *ArXiv*, abs/1908.05224, 2019.

OpenAI. Openai five. `https://blog.openai.com/openai-five/`.

OpenAI. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017a.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017b.

Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine. Epopt: Learning robust neural network policies using model ensembles. In *ICLR*, 2016.

Aravind Rajeswaran, Kendall Lowrey, Emanuel Todorov, and Sham Kakade. Towards Generalization and Simplicity in Continuous Control. In *NIPS*, 2017.

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 (RSS)*, 2018.

Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *ArXiv*, abs/1611.04201, 2016.

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. Technical report, DeepMind, January 2018. URL `https://arxiv.org/abs/1801.00690`.

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. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3482–3489, 2017.

Emanuel Todorov and Weiwei Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *ACC*, 2005.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IROS*, 2012.

Yusuke Urakami, Alec Hodgkinson, Casey Carlin, Randall Leu, Luca Rigazio, and Pieter Abbeel. Doorgym: A scalable door opening environment and baseline agent. *ArXiv*, abs/1908.01887, 2019.

Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

Grady Williams, Paul Drews, Brian Goldfain, James M Rehg, and Evangelos A Theodorou. Aggressive driving with model predictive path integral control. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pp. 1433–1440. IEEE, 2016.