A User's Guide to Calibrating Robotics Simulators

Bhairav Mehta
Massachusetts Institute of Technology
bhairavm@mit.edu

Ankur Handa
NVIDIA

Dieter Fox
University of Washington, NVIDIA

Fabio Ramos
University of Sydney, NVIDIA

Abstract: Simulators are a critical component of modern robotics research. Strategies for both perception and decision making can be studied in simulation first before deployed to real world systems, saving on time and costs. Despite significant progress on the development of sim-to-real algorithms, the analysis of different methods is still conducted in an *ad-hoc* manner, without a consistent set of tests and metrics for comparison. This paper fills this gap and proposes a set of benchmarks and a framework for the study of various algorithms aimed to transfer models and policies learnt in simulation to the real world. We conduct experiments on a wide range of well known simulated environments to characterize and offer insights into the performance of different algorithms. Our analysis can be useful for practitioners working in this area and can help make informed choices about the behavior and main properties of sim-to-real algorithms.

Keywords: Robotics Simulation, Sim-to-Real, Parameter Estimation, Benchmark

1 Introduction

Simulators play an important role in a wide range of industries and applications, ranging from drug discovery to aircraft design. In the context of robotics, simulators enable practitioners to test various research ideas, many of which may be too expensive or dangerous to run directly in the real world. As learning-based robotics expands in both interest and application, the role of simulation may become ever more central in driving research progress. However, as the complexity of the task grows, the gap between simulation and real-world becomes increasingly evident. As many recent works have shown, simpler approaches such as uniform domain randomization fail as task difficulty increases, leading to new curriculum and adaptive approaches to minimizing the sim2real gap.

There is recent growing interest in the robot learning community in the field of calibrated simulation and adaptive learning (much of which can be categorized as *machine learning for system identification*). Oftentimes, especially in robotics contexts, it may be reasonable to collect safe demonstrations on hardware using controllers or teleoperation. This line of work, merging the machine learning and system identification communities, is able to incorporate trajectory data into the calibration or policy learning process. Real-world data can allow for better environment sampling within simulators, more accurate uncertainty estimates for environment parameters, and more robust policy learning.

Calibrating and adaptive simulations hold great promise for robot learning, as a correct set of simulation parameters can help bridge the sim2real gap in a wide range of challenging tasks. When combined with powerful advances in fields like meta-learning or imitation learning, simulation calibration stands to expand the frontier for robot learning applications. With calibrated simulators, learning methods need not trade efficiency for robustness, potentially leading to a future where the simulation itself is embedded into the model-based or closed loop adaptive control loops [1].

In this work, we explore current methods in the space of Machine Learning for System Identification (MLSI), and push these algorithms to their limits along a variety of axes. We explore failure modes of each algorithm, and present a "user's guide" on when and where to use each. To present our results

4th Conference on Robot Learning (CoRL 2020), Cambridge MA, USA.

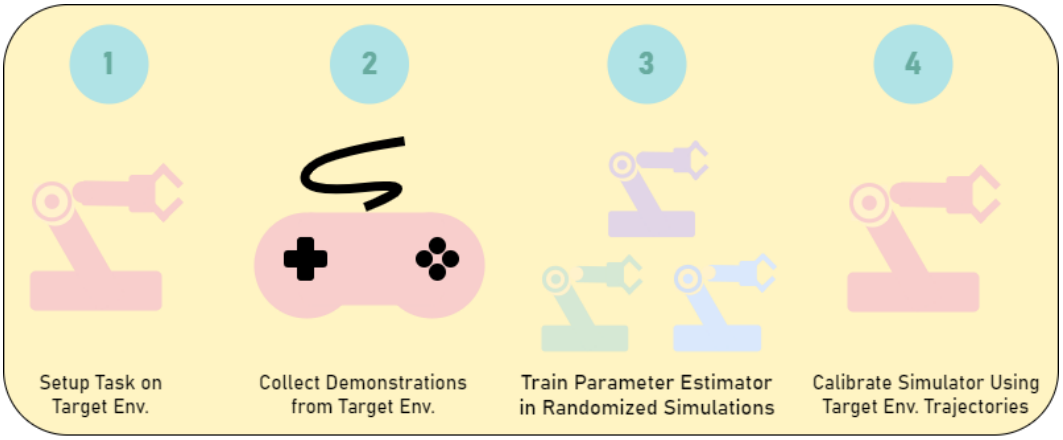


Figure 1: A typical parameter estimation workflow often requires the use of both robotic hardware and simulation, culminating in **calibration** of the simulator. Better calibration, hopefully, leads to better transfer when using simulators to train control algorithms with reinforcement or imitation learning.

cleanly, we introduce the **Si**mulation **P**arameter **E**stimation (**SIPE)** benchmark, which provides tools to efficiently test and compare past, current, and future algorithms in this space.

Contributions: Specifically, this work,

1. introduces the **Si**mulation **P**arameter **E**stimation (**SIPE)** benchmark, enabling standardization of benchmarking parameter estimation, system identification, and simulation calibration algorithms;
2. provides a comprehensive comparison of several current methods across a wide variety of environments, alongside open-source implementations of the algorithms compared, and extensive, varied datasets of trajectories useful in estimation tasks;
3. compares the usefulness of methods in the accuracy-agnostic task of *sim2sim* transfer.

2 The SiPE Benchmark

Robotics benchmarking has seen large improvements in recent years; focus on building simulation environments [2, 3, 4, 5], robotic trajectory datasets [6, 7], and teleoperation infrastructure [8] has proved fruitful for the community, with hundreds of methods annually building upon these tools. Discrepancies between simulators and real world have also been documented [9, 10, 11], yet a lack of standardized tools exist to test such robotic system identification and transfer tasks.

In this section, we introduce the **S**imulator **P**arameter **E**stimation (**SiPE**) benchmark, a collection of environments, trajectories, and expert policies collected for parameter estimation in robotics settings. As a parameter estimation-focused benchmark, we can abstract away many of the reward design issues that come along with new environment development. In addition, unlike work in reality-simulation discrepancy analysis, the goal is to *automatically* tune parameters of the simulation, rather than manual tuning followed by studying the differences.

2.1 Problem Setup

Parameter estimation in robotics, especially learning based system-identification methods, can often be separated into three distinct stages:

1. Collecting trajectories safely on real robotic hardware.
2. Training an estimation algorithm to learn how to estimate the parameters from trajectories.
3. Using real robot trajectories to estimate the parameters of interest and calibrate the simulator.

As illustrated in Figure 1, simulators play a significant role in Stage 3. To train system identification algorithms [12, 13, 14], a large diversity of environment setups are needed — oftentimes, too large a

Environment	Source	Description	Obs. Dim.	Action Dim.	Parameter Dim.
FetchPush	Plappert *et al.*[18]	Push a block to goal	31	4	2
FetchSlide	Plappert *et al.*[18]	Slide a puck to goal	31	4	2
FetchPick&Place	Plappert *et al.*[18]	Pick&Place block onto other	31	4	2
Relocate	Rajeswaran *et al.*[3]	Move ball with hand to location	39	28	2
Door	Rajeswaran *et al.*[3]	Open door with hand	39	28	7
HalfCheetahLinks	Brockman *et al.*[2]	Locomote until termination	17	6	5
HalfCheetahJoints	Brockman *et al.*[2]	Locomote until termination	17	6	15
HumanoidLinks	Brockman *et al.*[2]	Locomote until termination	376	17	17
HumanoidJoints	Brockman *et al.*[2]	Locomote until termination	376	17	51

Table 1: List of environments used in this work to benchmark.

variety to gather the data in the real world. Instead, *domain randomization* [15, 16, 17] allows us to quickly vary simulation parameters of robotic tasks, enabling the large diversity of data to be gathered quickly.

After training, given the reference trajectories from the real world, the normalized values for environment parameters (mass, friction, etc.) that could have generated the trajectories are inferred. Since the goal of these methods is to calibrate the simulator, we provide each algorithm reference trajectories generated from a test environment, and compare them to trajectories from the environment generated by the estimated parameters. This test environment could be trajectories gathered from the real world robot, or simply a held out set of trajectories from another simulated environment.

Concretely, we define the simulation parameter estimation task as follows: we aim to infer a set of parameters θ, from some real-world trajectories τ_{real}. Given access to a parameterized simulator S, we aim to match the real-world trajectory as closely as possible by estimating the simulation parameters θ' and executing the trajectory in the generated simulator environment $E = S(\theta')$. We use an algorithm $\mathcal{A}$ to estimate the simulation parameters from the trajectories, and calculate a loss $\mathcal{L}$. For notational clarity, we do not denote differences between *training* and *testing* trajectories, which we discuss further in Section 2.3. We summarize the general approach in Algorithm 1.

Algorithm 1 General Pseudo-Code for Parameter Estimation

Input: Parameter Estimation algorithm $\mathcal{A}$, Simulator S, Number of Parameters N, Real robot trajectories τ_{real}
while not converged **do**
    Sample parameters from some prior distribution ($\theta \sim p(\theta)$)
    Generate simulated environment $E = S(\theta)$
    Execute trajectory in simulated environment E, generating trajectory τ_{sim}
    Estimate parameters θ' from trajectory τ_{sim} and estimation algorithm $\mathcal{A}$
    Train $\mathcal{A}$ using loss $\mathcal{L}(\tau_{sim}, \tau_{real})$)
end while
At Inference
Using τ_{real} and $\mathcal{A}$, calibrate simulator using estimated parameters $\hat{\theta}$

2.2 Library Design and Implementation

In order to standardize evaluation of these algorithms, we have compiled a representative set of environments, which range from simple sanity checks to difficult high-dimensional parameter estimation problems. A complete list of characterized environments can be seen in Table 1 and visualized in Figure 3.

Broadly, we consider the following types of tasks:

1. **Object Centric, Simple** - The *Fetch* robotics suite of tasks, from [18], is a low-dimensional task with only cubes. The parameters estimated here are related to the cube's friction and mass, and serve as sanity checks.
2. **Manipulation** - The *Adroit* robotics tasks [3] involve a high-dimensional observation and action space, but is a low-dimensional parameter estimation task involving properties of the object the hand interacts with.
3. **Locomotion, Simple** - The standard reinforcement learning benchmarks are split into two types of parameter estimation tasks. The simple variants require estimation of link lengths.

Estimator	Description	Optimization Goal	Particle Based?	Learned Rewards?
Regression	Linear Regression (Baseline)	Cost Minimization	No	No
BayesOpt	Bayesian Optimization over parameter space	Cost Minimization	No	No
MAML	Regression from [22]	Fast Adaptation	No	No
SimOpt	Particle-based REPS	Cost Minimization	Yes	Both
ADR	Stein Variational Policy Gradient	Cost Minimization, Diversity	Yes	Both
BayesSim	Posterior Estimation of Parameters	Accurate Posterior	No	No

Table 2: An overview of the estimators benchmarked in this paper.

4. **Locomotion, Difficult** - The more difficult variants of locomotion estimation tasks involve estimating three parameters for each joint.

For each environment, SiPE provides wide variety of tools and datasets to explore different parameter estimation problem settings. Each SiPE environment comes with:

- *Pretrained Agents*: Agents are trained with DDPG [19] (training is elaborated on in Section 3) and include both a optimal and suboptimal (50% of training completed) agent that can be used to generate trajectories. In this analysis, we use the fully-trained agent to collect trajectories (Stage 2 in Figure 1).
- *Reference, In-Distribution, and Out-Of-Distribution Variants*: To test generalization, we provide environment variants (*i.e.* different parameter settings) that are not seen during training of the estimation algorithm: one within the parameter estimation range, and one outside.
- *Reference, Validation, and Test Trajectories from each environment*: To ensure fair comparison and to mitigate overfitting, each proposed SiPE environment comes with three sets of unique trajectories from each proposed environment. In our work, the reference trajectories are used during training, whereas validation and test trajectories are only used to tune hyper-parameters or evaluate performance.
- *Comprehensive Description of Environments and Parameters*: Each environment has been documented to ensure the experimenter knows the effects of each parameter, including how it interacts with the others.

The reference trajectories are generated from a "default" environment, and each algorithm predicts a (normalized) parameter estimate $\theta \in [0, 1]^N$, where N is the dimensionality of the parameter estimation space. As real robots were not accessible for many of the experiments, we use this reference as the **target** environment.

To compare performance across environments and algorithms, we introduce the SiPE plot, inspired by `bsuite` [20] and implemented with Matplotlib [21], shown in Figure 4(a). Each algorithm is run against all of the benchmark environments in Table 1, and the score for each radial point represents the accuracy on that particular environment. Scores are calculated by averaging across all of the parameters.

However, as many parameter estimation tasks are high dimensional, it can be difficult to compare estimators (especially as compared to reinforcement learning, where cumulative rewards can be compared across agents); SiPE comes with extra comparison modes (`mean`, `max`, `min`, seen in Figure 9 and 10 in the Appendix), which, when compared in conjunction, can provide real insight to estimator strengths and weaknesses. Similar to `bsuite`, SiPE ships with quality-of-life improvements, such as in-depth parameter estimation error plots, LaTeXreport generation, and more.

2.3 Algorithms

In order to get a clear understanding of current approaches, we benchmark a few representative methods, listed in Table 2, and compare them within the SiPE tasks described in the previous section. As the interest in this direction has grown, so have the number of approaches to do calibration or transfer. It is therefore imperative to understand the strengths and weaknesses of these approaches, which we compare across a wide variety of tasks and ablation studies. While this work may not fully represent analysis of all estimation methods both in use and development, we believe that even this sampling (along with Section 3's ablations) constitutes a good representation of today's MLSI landscape.

Briefly, we describe the five algorithms and their variants tested in this survey. Some of these algorithms have been adapted to the particular MLSI setting we use here, but each represents a reasonable

method to do parameter estimation. A full discussion on design choices and hyperparameter search can be found in Appendix A.1.

1. **Regression**: As a baseline, we use a simple, linear regression approach, trained with stochastic gradient descent. Given trajectories, the goal is to regress the environment parameters using the MSE between trajectories generated by executing the same actions in both the reference (target) environment and the estimated parameter environment. We use the iterative, gradient descent based version, rather than the closed form expression.

2. **Bayesian Optimization (BayesOpt)**: A popular system-identification tool, we implement a Bayesian Optimization approach, which is optimized using the upper confidence bound acquisition function. As a substitute for utility, we optimized for the lowest MSE between the trajectories generated by executing the same actions in both the reference (target) environment and the estimated parameter environment.

3. **Model-Agnostic Meta-Learning (MAML)**: Using the regression task setup from [22], we adapt the algorithm to the parameter estimation setting. MAML, as a gradient-based meta-learning algorithm, trains using a *meta-training* set, which consists of randomized simulation parameter settings and trajectories rolled out in those environments. At test time, the trained MAML model is given a trajectory from the target environment, and uses a set amount of gradient updates to regress the correct environment parameters. While there have been large numbers of follow up work to improve on the base algorithm [23, 24], we use the original variant as a comparison.

4. **SimOpt**: SimOpt [25] evolves a distribution of particles — which control the simulation parameters proposed — towards some ground truth distribution that is inferred from the real-world trajectories provided. SimOpt uses Relative Entropy Policy Search [26] to evolve the parameters, using a trajectory discrepancy loss (between real and simulation trajectories) and a trust-region to keep the optimization constrained.

5. **Active Domain Randomization**: Active Domain Randomization (ADR) [27] uses particles, trained with learned rewards based on trajectory discrepancies (similar to the discriminator approach from [28]), to estimate parameter settings. [29] proposes a recent improvement that builds upon ADR that aims to infer the parameter settings by "fitting" to real world trajectories, by flipping the sign of the reward signal. This forces particles to generate parameter settings that generate trajectories that are indistinguishable from the reference (as compared to the original approach that attempted to maximize discrepancies).

As a comparison to the class of parameter estimation algorithms that generate posteriors, we compare the previous algorithms against **BayesSim** [30]. BayesSim learns a posterior over simulation parameters given real-world trajectories. The algorithm trains a Mixture Density Network [31], allowing for efficient sampling of the posterior from the underlying Mixture of Gaussians distribution. BayesSim, and similar Bayesian Inference algorithms, are built on the assumption that having uncertainty over parameters can be critical in preventing overfitting to simulation, especially as many unique combinations of simulator settings can generate the same trajectory. We introduce a BayesSim comparison in Section 3.4, but do not compare it against the previous algorithms in parameter fitting settings, due to the difficulty of comparing Bayesian inference algorithms and their outputs (posteriors over parameters) against the point estimates of parameters found by the other surveyed algorithms.

For approaches that require the use of learned rewards, we train a discriminator to differentiate among types of trajectories, where indistinguishable trajectory sources (i.e the discriminator cannot tell if the trajectory came from the reference environment, or the estimated one) are rewarded higher. For all other estimators, we use a MSE loss between trajectories, executing the same actions from the same starting state in both the reference and proposed environment.

To ensure fair comparison, and to test the generality of each algorithm, we use *FetchPush-v1* and *Relocate-v0* to hyperparameter tune the approaches, and use the best performing set to benchmark on the rest of the environments. However, as noted in Section 2.2, *all* SiPE environments come with validation datasets, which could be used in more practical, results-driven settings.

```
import sipe
import gym

target_env = gym.make('target-env-v1')
trajectories = sipe.collect_demonstrations(target_env, type='expert')

randomizable_env = sipe.make('randomized-env-v1')
parameter_estimator = get_estimator(randomizable_env, estimator_type)
for iteration in range(n_max_iterations):
    parameter_estimator.update_parameter_estimate(env)

final_estimates = parameter_estimator.get_parameter_estimate(env)
sipe.show_results(final_estimates)
```

Figure 2: A code snippet that imports, runs, and plots results using the SiPE library.

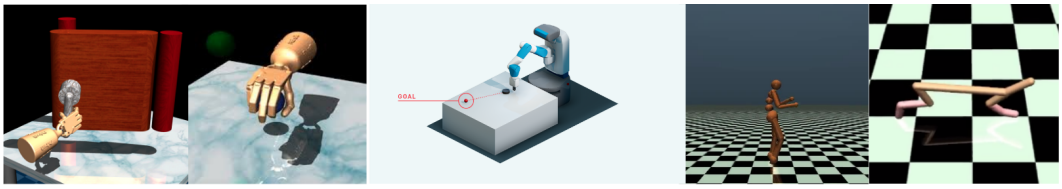


Figure 3: The base environments tested: two complex hand environments from [3], three, lower-dimensional Fetch Robotics environments from [18], and two locomotion environments from [2]. The locomotion environments are edited to have two variants: a simpler parameter estimation task for link masses, and a more difficult, higher-dimensional joint properties estimation task.

3 Results

We benchmark all five algorithms on the environments shown in Figure 3, and present the raw results below in Figures 4 with **results averaged over three trials for each experiment**. In these SiPE plots, only a single estimator is shown per plot, with `mean`, `min`, and `max` accuracies; for example, the *min* plot will take the minimum accuracy across all N parameters estimated. This allows us to conclude general trends of estimators, before exploring and comparing estimators against one another in Sections 3.2 to 3.4 and in Appendix A. **The rest of this section contains our most general analysis of results, and can be used as a self-contained section with which to get started with simulation calibration and MLSI.**[1]

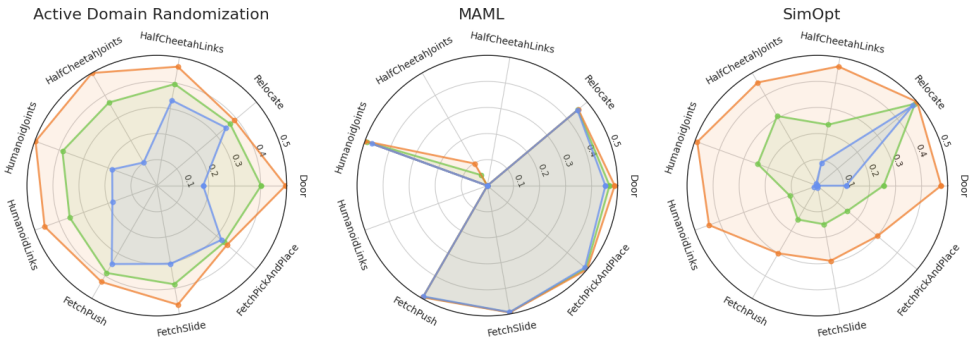
By comparing the *spread* — variance in the results from $\min$ and $\max$ of parameters — of each estimator in Figure 4, we can get an understanding of how these estimators behave, even in high-dimensional parameter estimation tasks. We experiment with the following settings:

- Decoupling policy learning from parameter estimation by directly regressing for parameters using trajectories from the target environment.
- Parameter estimation and policy learning in the loop with hand-tuned rewards.
- Parameter estimation and policy learning in the loop but with learned rewards with a discriminator.
- Assessing the generalisation of the learned parameters on a suite of test environments.

In summary, our conclusions are that particle based methods like ADR and SimOpt tend to do well on average but with larger spread for direct parameter regression. We also notice that simultaneously learning a policy alongside parameter estimation stabilizes all algorithms, despite the fact that performance may drop in certain tested environments. We find no general improvement from learning rewards (when compared to using the MSE error between states).

[1]For brevity, our results regarding Bayesian Optimization and Linear Regression can be found in the Appendix.

3.1 Direct Parameter Estimation



(a) (b) (c)

Figure 4: All results for each estimator across each environment for experiments with direct regression without learning any policy; here, the maximum error (minimum accuracy) is shown in blue, the mean accuracy shown in green, and the maximum accuracy shown in orange.

We see that while MAML performs strongly in lower dimensional settings, it struggles in higher-dimensional settings (Figure 4(b)); however, as the spread of the algorithm is remarkably low, it may be that with more extensive, particularly task-specific hyperparameter tuning, the algorithm can be made to work well in many parameter estimation settings.

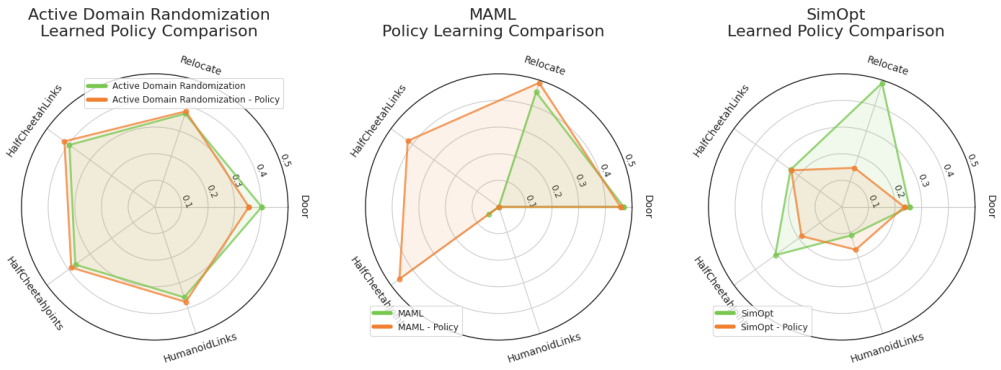
Both Active Domain Randomization (ADR) and SimOpt (Figures 4(a) and 4(c) respecitively) do well on average, but have much larger spread - the variability between the individual particles is much higher here than with other algorithms tested. Both algorithms are *particle-based* estimators, rolling out and maintaining multiple estimates in a diversity-inducing (ADR) or evolutionary (SimOpt) manner. While some particles achieve high accuracy (the orange lines in each figure), the diversity-inducing behavior also works against both algorithms, leading to poor mean performance and a generally higher spread. SimOpt also tends to do poorly on lower-dimensional environments, likely due to the surjectiveness of the parameter estimation problem: many possible parameter settings, especially in low-dimensional problems, may lead to the same generated trajectory. ADR's explicit enforcement of diversity may allow it to refrain from latching onto these incorrect parameter estimates, whereas SimOpt inherently has no such protective mechanism.

3.2 The Effect of Policy Learning

The results in Section 3.1 were generated using static, held-out trajectories. However, a recent trend of work has been to do *online* system-identification, especially in reinforcement learning contexts [32, 33]. Here, rather than using expert data generated by a fully trained policy, we explore the effect of using an iteratively-learned policy to generate the trajectories for system identification. In this section, we relax the assumption that the actions must be the same across trajectories, and rather use the policy to generate actions naturally (but control for starting states).

We use the `OurDDPG.py` variant from the open-source TD3 [34] repository, which is an improvement on the original Deep Deterministic Policy Gradient [19] algorithm. At the beginning of each episode, we randomize the simulator, and then roll out the policy in the generated environment. We use these same transitions to populate the replay buffer and train the policy. All trials are mean-averaged across three seeds, and each trial is used if and only if the policy trained converges to a "environment-solving" solution.

As seen in Figure 5, we see that using a policy for trajectory generation stabilizes many of the algorithms — the variance is significantly reduced as compared to Fig. 4 — despite the fact that the performance degrades in certain environments.

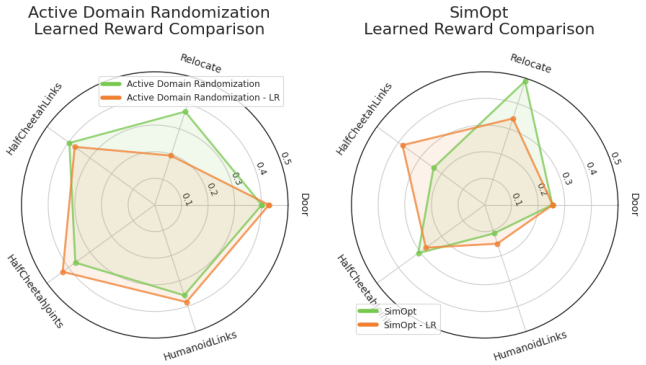


(a) (b) (c)

Figure 5: Across five environments, we compare results for the effects of policy learning. The green is the mean over trials when using demonstrations, while orange shows the mean over trials when using the data generated by learning a policy. The results for linear regression and Bayesian Optimization can be found in Appendix A.2.

3.3 The Effect of Learned Rewards

In this section, we benchmark variants of Active Domain Randomization (ADR) and SimOpt using learned rewards (rather than the cost function being the MSE between states seen along the trajectory when executing the same actions). As the cost function is generated by training a discriminator, we roll out the actions from the same starting state, and train the discriminator to differentiate between the sources of trajectories. The benefit of using learned rewards is that starting states and actions no longer need to be held constant; however, this can make the parameter estimation problem more difficult.



(a) (b)

Figure 6: Across five environments, we compare results for the effects of learning rewards over using state-action differences. The green is the mean over trials when using demonstrations, while orange shows the mean over trials when using the data generated by learning a discriminator.

In our experiments, shown in Figure 6, this difficulty seems to manifest itself in terms of **larger variance** between trials and generally **higher instability**. When using learned rewards, trends get less consistent. While both algorithms see less seeds converge, SimOpt seems to improve greatly from the use of a discriminator, whereas ADR suffers from the use of learned rewards when compared to a cost function based on ground truth state differences.

3.4 Generalization Experiments for Parameter Estimation via Domain Randomization

Oftentimes, parameter estimation, system identification, or simulation calibration is the *first* step of an engineering or research project. In this section, we benchmark each estimation algorithm in a *zero-shot*

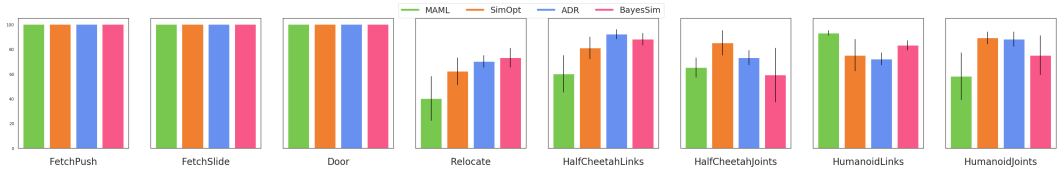


Figure 7: Across all environments, we compare the percentage return of an agent trained in the proposed environment settings with respect to the return of an agent that is trained solely in the testing environment.

transfer setting: using the estimated parameters, another agent is trained in the resulting parameter settings, and then tested on the held-out environment from which the reference (target) trajectories are generated. We compare the percentage return of an agent trained in the proposed environment settings normalised by the return of an agent that is trained solely in the target environment.

For MAML, as a point estimation algorithm, we train a reinforcement learning agent using the singular environment that is proposed after adapting to the reference trajectories. Since ADR and SimOpt are particle-based, we sample uniformly from the converged particles. In this section, we also compare against BayesSim, as comparing the agent's task performance is agnostic to the sampling strategy of environment generation.

We see in Figure 7 that, generally, having a distribution to sample from improves test performance; however, our results seem to suggest that in these settings, recovery of the posterior is not critical for strong test time performance. Particles seem to suffice, and the two particle-based methods tested perform well on the high-dimensional, locomotion estimation tasks.

4 Conclusion

We introduce the **Si**mulation **P**arameter **E**stimation (**SIPE)** benchmark and provide extensive analysis of current methods in common and difficult parameter estimation tasks. We believe our codebase, library, and analysis will serve as a strong starting point for practitioners in the field.

Acknowledgments

The authors would like to thank Yevgen Chebotar and Ian Abraham for their helpful comments and code implementation.

References

[1] S. Kuindersma. Recent progress on atlas, the world's most dynamic humanoid robot. Robotics Today https://www.youtube.com/watch?v=EGABAx52GKI, 2020. Accessed: 2020-06-30.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

[3] A. Rajeswaran, V. Kumar, A. Gupta, J. Schulman, E. Todorov, and S. Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *CoRR*, abs/1709.10087, 2017. URL http://arxiv.org/abs/1709.10087.

[4] Y. Tassa, S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, and N. Heess. dm_control: Software and tasks for continuous control. *arXiv*, 2020.

[5] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison. RLBench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5:3019–3026, Apr 2020.

[6] S. Cabi, S. Gómez Colmenarejo, A. Novikov, K. Konyushova, S. Reed, R. Jeong, K. Zolna, Y. Aytar, D. Budden, M. Vecerik, and et al. Scaling data-driven robotics with reward sketching and batch reinforcement learning. *Robotics: Science and Systems XVI*, Jul 2020.

[7] S. Dasari, F. Ebert, S. Tian, S. Nair, B. Bucher, K. Schmeckpeper, S. Singh, S. Levine, and C. Finn. RoboNet: Large-scale multi-robot learning, 2019.

[8] A. Mandlekar, Y. Zhu, A. Garg, J. Booher, M. Spero, A. Tung, J. Gao, J. Emmons, A. Gupta, E. Orbay, S. Savarese, and L. Fei-Fei. RoboTurk: A crowdsourcing platform for robotic skill learning through imitation. In *Conference on Robot Learning*, 2018.

[9] J. Collins, J. McVicar, D. Wedlock, R. Brown, D. Howard, and J. Leitner. Benchmarking simulated robotic manipulation through a real world dataset. *IEEE Robotics and Automation Letters*, page 250–257, Jan 2020.

[10] R. Menzenbach. Benchmarking sim-2-real algorithms on real-world platforms. Master's thesis, 2019.

[11] F. Muratore, M. Gienger, and J. Peters. Assessing transferability from simulation to reality for reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2019.

[12] N. Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive behavior*, pages 325–368, 1997.

[13] I. Abraham, A. Handa, N. Ratliff, K. Lowrey, T. D. Murphey, and D. Fox. Model-based generalization under parameter uncertainty using path integral control. *IEEE Robotics and Automation Letters*, pages 2864–2871, 2020.

[14] W. Zhou, L. Pinto, and A. Gupta. Environment probing interaction policies, 2019.

[15] F. Sadeghi and S. Levine. $(CAD)^2$ RL: Real Single-Image Flight Without a Single Real Image. *CoRR*, 2016. URL http://arxiv.org/abs/1611.04201.

[16] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, 2017. URL http://arxiv.org/abs/1703.06907.

[17] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *IEEE international conference on robotics and automation (ICRA)*, 2018.

[18] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *CoRR*, abs/1802.09464, 2018. URL http://arxiv.org/abs/1802.09464.

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[20] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepezvari, S. Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9 (3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[22] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017. URL http://arxiv.org/abs/1703.03400.

[23] A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms. *CoRR*, abs/1803.02999, 2018. URL http://arxiv.org/abs/1803.02999.

[24] C. Finn, A. Rajeswaran, S. M. Kakade, and S. Levine. Online meta-learning. *CoRR*, abs/1902.08438, 2019. URL http://arxiv.org/abs/1902.08438.

[25] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. D. Ratliff, and D. Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. *CoRR*, abs/1810.05687, 2018. URL http://arxiv.org/abs/1810.05687.

[26] J. Peters, K. Mülling, and Y. Altün. Relative entropy policy search. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, page 1607–1612. AAAI Press, 2010.

[27] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull. Active domain randomization. *CoRR*, abs/1904.04762, 2019. URL http://arxiv.org/abs/1904.04762.

[28] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. *CoRR*, abs/1802.06070, 2018. URL http://arxiv.org/abs/1802.06070.

[29] B. Mehta. Active domain randomization and safety critical few shot learning. Master's thesis, Universite de Montreal, 2020. URL https://bhairavmehta95.github.io/static/thesis.pdf.

[30] F. Ramos, R. C. Possas, and D. Fox. Bayessim: adaptive domain randomization via probabilistic inference for robotics simulators. *CoRR*, abs/1906.01728, 2019. URL http://arxiv.org/abs/1906.01728.

[31] C. M. Bishop. Mixture density networks. 1994.

[32] K. Rakelly, A. Zhou, C. Finn, S. Levine, and D. Quillen. Efficient off-policy meta-reinforcement learning via probabilistic context variables. In *International conference on machine learning*, pages 5331–5340, 2019.

[33] A. Nagabandi, K. Konolige, S. Levine, and V. Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pages 1101–1112, 2020.

[34] S. Fujimoto, H. Van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[35] F. Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014–. URL https://github.com/fmfn/BayesianOptimization.

[36] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

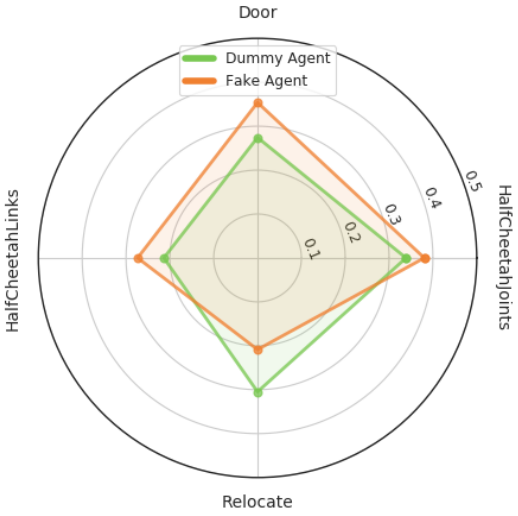


Figure 8: SiPE can also capture the spread of algorithms, enabling a better understanding of large-scale effects in high-dimensional settings, while still being able to understand the performance in one or two plots. Here, alongside `mean` averaging, we see that `max` (Figure 9) and `min` (Figure 10) provide additional insight to the two fake estimators shown here.

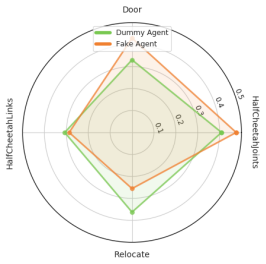


Figure 9: The `max` plotting capability of SiPE.

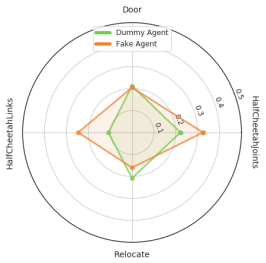


Figure 10: The `min` plotting capability of SiPE.

A Additional Results

A.1 Hyperparameters Discussion

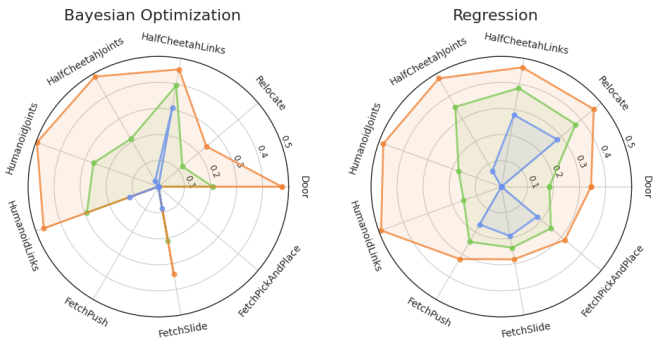
As described in the main text, we used the validation set of trajectories from the SiPE benchmark to hyperparameter tune, choosing the best set of hyperparameters as measured by mean parameter accuracy on *FetchPush-v1* and *Relocate-v0*. We then held these parameters constant across the entire benchmark, and reported the performance both throughout the main text and Figure 11. Within the following list, the **bolded** values are the chosen (benchmark) hyperparameters used in the study.

1. **Regression**: For regression, we trained a single layer network (using a concatenated, per-timestep $s - a - s'$) to regress to the parameters directly. The hyperparameter search was over the batch size (**320**, 160, 80, 40) and learning rate (**0.001**, 0.01, 0.05, 0.005). We use the gradient descent version of linear regression, rather than using the closed form expression to solve for parameters.
2. **Bayesian Optimization (BayesOpt)**: For Bayesian Optimization, we use the open-source `bayesopt` [35] package. We searched over the utility function (**upper-confidence bound**, expected improvement), κ (**2.5**, 5, 10, 1, 0.5), and ξ (**0**, 0.5, 1).
3. **Model-Agnostic Meta-Learning (MAML)**: Using the regression task setup from [22], we adapt the algorithm to the parameter estimation setting. For MAML, we use a two-layer hidden network (sizes searched over 40, **60**, and 100) and searched over a learning rate of 0.001, **0.005**, and 0.0001. We use the same learning rate for both the inner and outer loop. MAML was the only algorithm in the study implemented in Jax [36], while all other algorithms were implemented in Pytorch [37].
4. **SimOpt**: For SimOpt, our hyperparameter search was over the number of REPS updates (4, 8, **12**), the number of particles (4, **6**, 8), the mean initialization of the particles (mean sampled from U(0, 1) or **U(0, 0.5)** and covariance initialization of the particles (0.05, 0.1, **0.2**).
5. **Active Domain Randomization**: For Active Domain Randomization, we searched over the number of particles (4, 6, 8, **10**) and discriminator learning rate for the learned reward experiments (searched over **0.001** and 0.005). We use the same policy network, temperature,

discriminator learning rate and architecture as in [27], and hold discriminator hyperparameters constant for the *SimOpt Learned Reward* experiments.

6. **BayesSim**: We use the open-source implementation of BayesSim and do not modify any of the hyperparameters.

A.2 All Environment Results: Bayesian Optimization and Linear Regresssion



(a) (b)

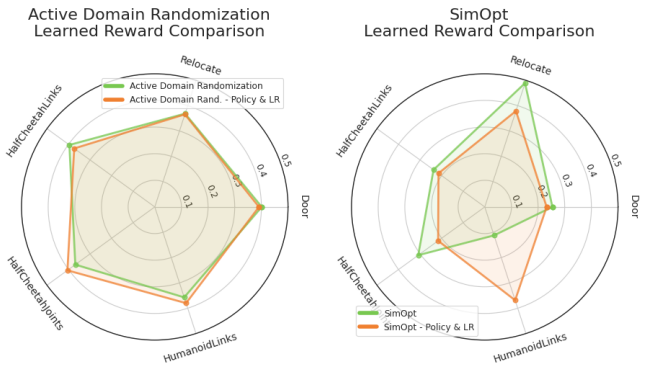
Figure 11: All results for Linear Regression and Bayesian Optimization across each environment for experiments with direct regression without learning any policy; here, the maximum error (minimum accuracy) is shown in blue, the mean accuracy shown in green, and the maximum accuracy shown in orange

.

For completeness, we provide results analogous to those seen in Section 3.1 for Linear Regression and Bayesian Optimization. As seen in Figures 11(a) and 11(b), these simplest approaches seems to do well on high dimensional environments (Bayesian Optimization) or well generally (Regression). While a high spread, these results provide a practical suggestion: sometimes, simple methods can work just as well, especially with further tuning to a specific problem of interest.

A.3 Policy Learning and Learned Rewards

When we combine learning a policy and using learned rewards, we see that SimOpt varies highly in performance, greatly improving on the high-dimensional estimation task of `HumanoidLinks`.



(a) (b)

Figure 12: Across five environments, we compare results for the effects of learning rewards **and** using trajectories from a policy over using state-action differences. The green is the mean over trials when using demonstrations, while orange shows the mean over trials when using the data generated by learning a discriminator.

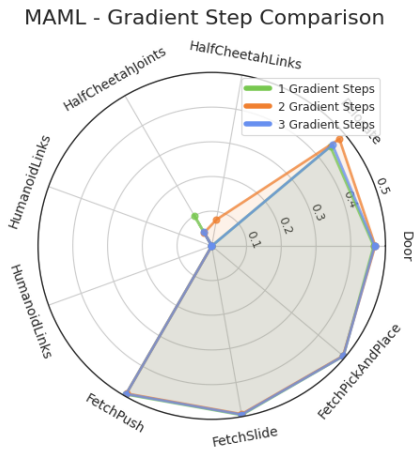


Figure 13: Across all environments, we compare results for the effects of gradient steps when using MAML.

For ADR, we see that learning rewards alongside a policy does not provide extra benefit over learning solely a policy. The two in conjunction seem to destabilize approaches (i.e fewer seeds reach the *Convergence* score), which we attribute to the learned rewards.

A.4 Fine-tune or Adapt?

When we test MAML with varying numbers of *test-time* gradient steps, we see that while the gradient steps hardly affect performance, MAML exhibits *all-or-nothing* behavior. As noted in the experimental setup, our hyperparameters are tuned on two environments which receive almost 100% performance. However, as shown in Figure 13, MAML seems to have issues with the higher-dimensional parameter estimation tasks. This may be a result of fundamental issues with the algorithm, or a need for more extensive tuning of hyperparameters.