Learning Robust Control Policies from Simulations with Perturbed Parameters

Master-Thesis von Felix Treede aus Mainz

Tag der Einreichung:

Gutachten: Prof. Dr. Jan Peters

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Learning Robust Control Policies from Simulations with Perturbed Parameters

Vorgelegte Master-Thesis von Felix Treede aus Mainz

Gutachten: Prof. Dr. Jan Peters

Tag der Einreichung:

Erklärung zur Master-Thesis

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Felix Treede, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Datum / Date: ____________________

Unterschrift / Signature: ______________________________

Abstract

Deep reinforcement learning has a lot of potential to solve robotics motor tasks. However, deep learning algorithms suffer from a high sample complexity, thus training on the real robot is unfeasible. Training in simulated environments has been successful, but transferring the learned policies to the real robot is difficult. Recent research has focused on learning robust policies which can be transferred successfully.

In this thesis, we developed a framework to learn robust policies and evaluate them on environments with varied physics parameters. The simulation was implemented using the Rcs framework, developed by Honda Research Institute Europe, which can control both simulated and real-world robots. The applied learning algorithms are based on the rllab framework. In order to allow using Rcs-based environments, we wrote RcsPySim as a bridge.

Throughout the thesis, we used RcsPySim to evaluate and compare the robust learning algorithms Ensemble Policy Optimization (EPOpt) and Robust Adversarial Reinforcement Learning (RARL) with each other and with a baseline policy trained using the standard policy optimization algorithm Trust Region Policy Optimization (TRPO). The policies were trained and evaluated on the ball-on-plate task, where a robot has to stabilize the ball at the center of the plate which is mounted on the robot's end-effector. We varied different physics parameters in order to analyze the robustness of the learned policies against these changes. Furthermore, we performed a sensitivity analysis of the parameters. The obtained results show that the relevant physics parameters for the ball-on-plate task are the balls friction properties and mass distribution, whereas the ball's mass and radius do not have a significant influence. Moreover, we observed that the baseline policy, trained solely on the nominal physics world, is already quite robust. Both RARL and EPOpt increase the robustness for some parameter ranges, but reduce it in others. EPOpt does not perform as well as expected. Generally, EPOpt always prefers the more cautious approach, which means it can deal better with more unstable simulations, but it will not be able to solve the task in a setup with strong friction. The solution trained by RARL is more aggressive, making it well suited for solving cases with higher friction, but less attractive for more unstable environments with lower friction. All laerned policies could be transferred to the real world. The policy learned by EPOpt should be preferred as it is the most cautious. Since EPOpt doesn't work well with friction values higher then the nominal parameters, choosing their values to be higher then the measured mean would likely increase the robustness over the whole parameter space.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
EPOpt	Ensemble Policy Optimization
RARL	Robust Adversarial Reinforcement Learning
CVaR	Conditional Value at Risk
IK	Inverse Kinematics
MDP	Markov Decision Process
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
REPS	Relative Entropy Policy Search
TRPO	Trust Region Policy Optimization
RL	Reinforcement Learning
Rcs	Robot Control System

List of Symbols

Notation	Description
ϵ	quantile used for CVaR in EPOpt
$\mathscr{P}$	distribution of physics parameter sets
p	physics parameter set for one trajectory in EPOpt
$F_{\mathrm{a,max}}$	maximum force the adversary for RARL can apply
N_{iter}	number of iterations
N_{μ}	number of protagonist optimization steps per iteration
N_{ν}	number of adversary optimization steps per iteration
T	number of timesteps in trajectory

τ_k	a single trajectory
$\mathscr{T}$	set of trajectories

$\mathbf{a}_{\text{des}}$	goal action vector, used by quadratic cost function
$\mathbf{R}$	diagonal action cost weight matrix, used by quadratic cost function
$\mathbf{s}_{\text{des}}$	goal state vector, used by quadratic cost function
$\mathbf{Q}$	diagonal state cost weight matrix, used by quadratic cost function

d_{angular}	angular damping coefficient
F_d	force applied to stop the object from moving due to damping
T_d	torque applied to stop the object from moving due to damping
d_{linear}	linear damping coefficient

μ_f	linear friction coefficient
μ_{rf}	rolling friction coefficient
r_{contact}	curvature radius at contact point
F_f	force applied to stop the object from moving due to linear friction
T_f	torque applied to stop the object from moving due to rolling friction
F_N	contact normal force

A	set of action vectors of a MDP
$\mathbf{a}_t$	vector of actions in timestep t of a MDP
γ	discount of a MDP
S	set of state vectors of a MDP
$\mathbf{s}_t$	vector of states in timestep t of a MDP

v	absolute linear velocity of object
ω	absolute angular velocity of object

r_{min}	minimum exponential reward value, hyperparameter
c_{max}	maximum quadratic cost value, computed from , and
$\mathbf{q}_{\text{des}}$	desired joint positions for robot control
$\dot{\mathbf{q}}_{\text{des}}$	desired joint velocities for robot control
$\mathbf{T}_{\text{des}}$	desired joint torques/forces for robot control
$\mathbf{x}$	joint space state vector
$\mathbf{q}_{\text{ctrl}}$	desired joint space state integrated from
$\boldsymbol{\Delta}\mathbf{q}$	joint state error, output of IK

$\mathbf{q}_{\text{int}}$	current desired joint space state, from last timestep
$\mathbf{q}_{\text{cur}}$	current joint positions from sensors/simulator
$\dot{\mathbf{q}}_{\text{cur}}$	current joint velocities from sensors/simulator
$\mathbf{x}$	task space vector for forward/inverse kinematics
$\mathbf{\Delta x}$	task space error for IK
$\mathbf{x}_{\text{des}}$	desired task space state
$\mathbf{x}_{\text{int}}$	integrated task space state computed from
$x_{\bullet}$	object position along x axis
$y_{\bullet}$	object position along y axis
$z_{\bullet}$	object position along z axis
$\alpha_{\bullet}$	object rotation around x axis
$\beta_{\bullet}$	object rotation around y axis
$\gamma_{\bullet}$	object rotation around z axis

List of Operators

Notation	Description	Operator
exp	the exponential function; e to the power of $\bullet$	$\exp(\bullet)$
ln	the natural logarithm	$\ln(\bullet)$
$\tan^{-1}$	the inverse tangens function	$\tan^{-1}(\bullet)$
R	reward function, computes reward for a whole trajectory	$R(\tau_k)$
r	reward function, computes reward for a single timestep	$r(\mathbf{s}_t, \mathbf{a}_t)$
V^{π}	value function, computes the expected return when starting a trajectory from the given state using the policy π	$V^{\pi}(\mathbf{s}_t)$
r_{exp}	exponential reward function	$r_{\text{exp}}(\mathbf{s}, \mathbf{a})$
C	quadratic cost function	$C(\mathbf{s}, \mathbf{a})$
$\mathbb{E}$	the expected value of a random variable	$\mathbb{E}[\bullet]$
P	generic probability distribution of a random variable	$P(\bullet)$
Q_{α}	the α-quantile of a random variable	$Q_{\alpha}(\bullet)$

1 Introduction

Deep reinforcement learning of robot motor tasks has seen a lot of success in the last years. Modern algorithms like Trust Region Policy Optimization (TRPO) [1] or Relative Entropy Policy Search (REPS) [2] are capable of learning solutions for complex tasks. These algorithms require exploration of the state space, that is, the control policy can test different approaches to the solution autonomously. However, performing state space exploration on a real robot with real objects is problematic. On one hand, there is the risk of the robot damaging itself or its environment during exploration. Safety limits can avoid this problem at least partially, but they are more difficult to implement for objects in the environment than for the joints of the robot itself.

On the other hand, the world state must be reset after every rollout. We can easily do that actuated parts of the robot, but not for objects the robot interacts with. Resetting these either needs intervention by the operator, a complex separate machinery or another existing control policy for the robot itself. The latter would likely also be a solution for the actual task, hence it is unlikely to exist except for very simple tasks, such as closing a door the robot should learn to open [3].

Even if these issues can be solved, it is still very time-consuming to train on a real robot, since it can only perform one rollout at the time. Yahya et.al. used 5 robots working in parallel to speed up the process [3], and Levine et.al. even used 14 [4], but in the general case using multiple robots would be very expensive.

One promising solution is to create a model of the robot and it's environment and train the policy in a simulation. Multibody physics simulators such as Bullet[1], Vortex[2] or MuJoCo[3] can simulate most robotic setups. Simulated experiments do not have any of the aforementioned disadvantages. And on modern processors, rollouts can be simulated much faster than real-time, on multicore processors even truly in parallel. However, every physics simulation is by definition an approximation. In order to be an accurate reproduction of the real world, it needs very exact estimates of the physics parameter values. Additionally, the simulation of non-continuous behaviour like contact reaction is very hard to model accurately. Complex policies with many parameters like neural networks tend to overfit in these cases. Thus, a policy which has been trained to a high performance in the simulation might fail on the real robot. This phenomenon is known as the reality gap. Overcoming this issue has been a focus of the robot learning research.

One approach do bridge the reality gap is to modularize the task [5]. Instead of learning a deep end-to-end policy taking raw sensor data as input and producing raw motor commands as output, one could split off the sensor and actuator handling. The core policy would get object positions as input and command the robot in the task space. Learning sensor data interpretation from simulation is hard, especially for visual tasks. By separating the sensor interpretation from the main policy, any issues stemming from this part are easily avoided. Of course, one still need a solution for interpreting the sensor data in the real world. Likewise, solutions for controlling a robot in it's task space are well explored and readily available. However, modularization does not solve the core problem. The physics simulation still needs to simulate object interaction, which is the source of the issues mentioned earlier.

Another approach is model learning, where a small amount of trajectories is sampled on the real robot and then used to estimate the physics parameters for the simulation. Then, the actual training is done in the simulation. Model learning makes use of the less time-consuming simulation, but it retains the other issues of real-world exploration, albeit in a smaller part of the process. Recent research has seen a lot of success in this area [6] [7].

Finally, one can modify the training process to make the policy itself more robust. The basic idea is not new. A policy training with a random initial state is robust against changes in the initial state. Early research used artificial noise on the robot's observation [8] in order to bridge the reality gap. Recent research has focused on learning policies which are also robust to changes in the physics parameters, and thus should be transferrable to the real world.

The aim of this thesis was to develop a simulation framework for a simple physics task and to compare the performance of several learning algorithms for robust policies from the literature. We compared the Ensemble Policy Optimization (EPOpt) [9] and Robust Adversarial Reinforcement Learning (RARL) [10] algorithms with each other, with a baseline policy trained using the TRPO [1] algorithm and with a policy trained on an environment with high generic transition noise. These algorithms were tested on the task of balancing a ball on a plate. All algorithms were implemented using the rllab framework [11]. In order to facilitate the migration from simulation to real-world robotics, the simulation of the robot task was implemented using the Rcs framework. We wrote RcsPySim as a bridge software to allow using Rcs-based environments in rllab.

[1] `https://github.com/bulletphysics/bullet3`

[2] `https://www.cm-labs.com/vortex-studio/`

[3] `http://www.mujoco.org/`

2 Foundations

2.1 Reinforcement Learning (RL)

We are considering a task that can be described as a Markov Decision Process (MDP) consisting of:

- continuous states $\mathbf{s} \in S$
- continuous actions $\mathbf{a} \in A$
- transition dynamics $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
- a deterministic reward function $r(\mathbf{s}, \mathbf{a})$
- initial state probabilities $P(\mathbf{s}_0)$
- a discount factor γ

In order to solve these tasks, we employ a policy π that computes $\mathbf{a}_t$ based on $\mathbf{s}_t$ at every timestep t. The policy depends on a parameterset θ. As a shorthand notation, we denote a policy with parameters θ as π_θ. The goal of the optimizer is to find a value for θ that maximizes the expected reward over the whole state space. This policy can either be deterministic $\mathbf{a}_t = \pi_\theta(\mathbf{s}_t)$ or stochastic $\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$. A stochastic policy implicitly performs exploration and can keep track of the certainty in the current solution on it's own, thus removing the need of an explicit exploration strategy. Algorithms like TRPO or REPS require the use of a stochastic policy. However, when using the learned policy in a production scenario, stochastic behaviour is undesired as it imposes risk. Many stochastic policies are based on Gaussian noise, which means that they have a learned component producing a mean for the internal distribution. They can easily be turned into a deterministic policy by using the mean as output action values.

For robotics tasks, the transition dynamics are often not known explicitly, so they have to be approximated from a finite set of samples. Many algorithms do so in batches. First, a number of rollouts is performed, generating trajectories $\tau_k = \{\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}\}_{t=0}^{T-1}$. The optimization objective is the expected return over all trajectories in the trajectory set $\mathscr{T}$. The expected discounted return of one algorithm is defined as:

$$R(\tau_k) = \sum_{t=0}^{T-1} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t) \tag{2.1}$$

The discount factor γ is used to make sure short-term rewards get preferred over long-term rewards. The value is chosen as a hyperparameter. When plotting the return of a trajectory in order to measure policy performance, we use $\gamma = 1$ since that makes the results easier to interprete.

The algorithm updates the parameters using the trajectories:

$$\theta_{i+1} = \text{batchPolopt}(\theta_i, \mathscr{T}) \tag{2.2}$$

This class of algorithms is called batch policy optimization. Common implementations of this scheme include TRPO [1] and REPS [2]. Since the interface between the algorithm and the main setup is well defined, the batch policy optimization can be used as interchangeable component in meta-algorithms like those discussed in this thesis.

More details on reinforcement learning, including value function approaches, can be found in the book by Sutton & Barto [12].

2.2 Ensemble Policy Optimization (EPOpt)

Rajeswaran et.al. propose EPOpt [9] as a way to train robust policies. The transition function of the MDP is modified so that it depends on a physics parameter set p. At the begin of each rollout, values for p are sampled from a parameter distribution $\mathscr{P}(p)$. These values are then used to generate a single trajectory. This way, the training is performed on an ensemble of models defined by the parameter distribution. The rollouts from different models are then passed to a batch policy optimization algorithm like it would be done in a setup without perturbation.

Usually, a RL algorithm optimizes the expected return over all rollouts, which is a risk neutral objective. In order to obtain a more robust result, a risk averse objective would be better. Rajeswaran et.al. decided to use the conditional value at risk (CVaR) as objective. The CVaR is defined as

$$\mathrm{CVaR}_\epsilon(x) = \mathbb{E}[x_k | x_k \leq Q_\epsilon(x)] \tag{2.3}$$

where $Q_\epsilon(x)$ is the ϵ-quantile of x. In order to optimize for the conditional value at risk of the returns $R(\tau)$, the batch policy optimizer is called on a subset of all trajectories. Only trajectories with a return lower then $Q_\epsilon(R(\tau))$ are relevant for the CVaR, so the others are filtered out:

$$\mathscr{T}_{\mathrm{CVaR}_\epsilon} = \{\tau_k | R(\tau_k) \leq Q_\epsilon\left(R(\mathscr{T})\right)\} \tag{2.4}$$

The expected return of the trajectory subset $\mathscr{T}_{\mathrm{CVaR}_\epsilon}$ is equal to the CVaR of the original trajectory set:

$$\mathbb{E}\left[\mathscr{T}_{\mathrm{CVaR}_\epsilon}\right] = \mathbb{E}[\tau_k | R(\tau_k) \leq Q_\epsilon(R(\mathscr{T}))] = \mathrm{CVaR}_\epsilon(\mathscr{T}) \tag{2.5}$$

So, when the batch optimizer is run on $\mathscr{T}_{\mathrm{CVaR}_\epsilon}$, it will automatically optimize $\mathrm{CVaR}_\epsilon(\mathscr{T})$. Since only a subset of $\mathscr{T}$ is used, we need to compute more trajectories in order for $\mathscr{T}_{\mathrm{CVaR}_\epsilon}$ to be large enough that the batch optimizer yields a stable result. So, EPOpt is very sample inefficient, since only the fraction of ϵ trajectories is used.

2.3 Robust Adversarial Reinforcement Learning (RARL)

An alternative approach is proposed by Pinto et.al. [10]. Instead of using the conditional value at risk, they formulate a zero-sum game with an adversarial agent, whose goal is to hinder the policy.
In general, a two-player game consists of two players, who each have a set of actions $a_1 \in A_1$ and $a_2 \in A_2$ as well as a reward function $r_1(a_1, a_2)$ and $r_2(a_1, a_2)$. Both players choose an action simultanously, they cannot react to the choice of the other player. The goal of each player is to maximize his own reward.
A zero-sum game is a special case of a two player game. The reward of one player is the negative of the reward of the other player: $r_1(a_1, a_2) = -r_2(a_1, a_2)$. Thus, maximizing the own reward is equivalent to minimizing the opponents reward.
RARL requires a second adversary policy in addition to the regular protagonist policy from the original reinforcement learning setup. The transition function is extended depends on the adversary's actions:

$$P_t(s_{t+1} | s_t, a_{\mathrm{prot},t}, a_{\mathrm{adv},t}) \tag{2.6}$$

In a concrete task setup, the adversary could be defined to use the same action space as the protagonist, or it could be completely independent. One example is an adversary applying a disturbance force to a specific part of the task's physics model. Here, the adversary actions would determine the direction and strength of the force. The adversary's total strength is limited, since an infinitely strong adversary could always prevent the protagonist from reaching it's goal.
The protagonist and the adversary are optimized in turns. First, the protagonist is optimized on trajectories generated using a fixed adversary. Then, the protagonist is fixed and the adversery gets optimized. Each of them only perform a few batch optimization iterations before the focus is switched. These steps are repeated until convergence.

2.4 Generic Transition Noise

A general MDP already covers the effects of the perturbed physics parameters from EPOpt and the adversarial actions from RARL as part of the stochasticity of the transition function $P_t(s_{t+1} | s_t, a_t)$. In usual setups, the variance of the transition function is very low. A physics simulation is generally completely deterministic unless it's parameters change. In most cases, a stochastic transition function is viewed as a deterministic function with added noise. That noise is called transition noise.
Transition noise is not the same as observation noise. The latter is applied to the states before they are passed to the policy, but it does not influence the next state directly. Real world systems usually have observation noise due to sensor noise. While observation noise is also important for bridging the reality gap [8], it does not solve the problem of wrongly modeled transition dynamics.
Adding artificial transition noise to a simulated system also is a way of perturbing the system. However, artificial noise is sampled from a simple normal distribution, whereas the perturbations done by EPOpt and RARL result in a far more complicated transition noise distribution.

3 Framework

3.1 Rcs (Robot Control System)

Rcs[1] is a robotics control framework developed and maintained by the Honda Research Institute Europe.
At the core of Rcs, there is the graph defining the robot and its environment. The graph holds a tree of bodies. Every body has physical and visual properties, and is linked to its parent and its children via joints. A body can have multiple joints to its parents, which allows to model objects with no constraints relative to their parents by defining a joint for each of the six degrees of freedom. Thus, both the kinematic chain of the robot as well as free objects in the scene the robot can interact with can be described as part of the graph. The graphs mutable state consists of the joint states $\mathbf{q}$ and their derivative $\dot{\mathbf{q}}$. Additionally, every body provides it's position and rotation in the world coordinate frame as well as corresponding velocities, which are calculated from $\mathbf{q}$ and $\dot{\mathbf{q}}$.
Rcs also contains a powerful inverse kinematics solver. The task space $\mathbf{x}$ is defined by a list of `Task` components. A number of `Task` subclasses exist to model position or rotation, optionally relative to another body, as well as direct joint positions or advanced concepts such as collision avoidance. Each of these `Tasks` has methods to compute it's task space state $\mathbf{x}$ as well as the Jacobian $\mathbf{J}(\mathbf{q})$ and other task-specific parts required by the solver. Coordinate-based tasks always control a single effector body. The position can also be relative to another body. The inverse kinematics solver operates on a list of tasks to realize their combined $\mathbf{x}$. The default solver uses the right pseudo-inverse of the jacobian to turn a task-space error $\mathbf{\Delta x}$ into a joint space error $\mathbf{\Delta q}$. Ambiguities in the joint configuration are resolved by adding a nullspace term. This approach is called "Resolved Motion Rate (RMR)".
Additionally, Rcs also has a ready to use graphics module, which can render 3D view of the robot using the graphs data on body shapes and positions.

3.1.1 Physics Simulations

Rcs defines an abstract interface for physics simulators called `PhysicsBase`. A simulator step takes joint commands $\mathbf{q}_{des}$, $\dot{\mathbf{q}}_{des}$ and $\mathbf{T}_{des}$ as inputs, computes the dynamics and updated state and returns new joint states $\mathbf{q}_{cur}$ and velocities $\dot{\mathbf{q}}_{cur}$. Additionally, it is possible to add an external force to a body, which can be used to introduce disturbances. Rcs provides two implementations of the described interface using the Vortex and Bullet physics engines.

Vortex
Vortex Studio is a commercial physics simulator developed by CM Labs. The core components are available as freeware under the name Vortex Studio Essentials. Vortex Studio provides a wide range of features, most of which are not relevant for our task. One big advantage of Vortex is it's material system. Contact-related properties like friction model and coefficients are stored in material objects. Every collision shape is assigned one material. The combination of materials for contacts is performed automatically, taking the mimimum value from the two materials for each coefficient, but it is also possible to modify properties for a specific pairing of materials explicitly. This approach allows for accurate simulation of the real-world friction coefficients, which can only be measured per combination of materials.
Vortex also ships with plugins for special physics situations like wheeled objects or cable systems, to allow a more accurate simulation of those cases. However, these are not relevant for the common robotics tasks Rcs was built for.

Bullet
Bullet is an open-source physics simulation developed by the community.
Most physics simulation engines use the kinetic energy of part as foundation for internal computations. However, Bullet's internal state is impulse based, which means that some parameters have unusual units. Dampers, for example, usually apply a force based on a fraction of the current velocity, which is set by the damping coefficient. In bullet, the damping coefficient has no specific physical meaning, instead it can roughly be described as "What fraction of the velocity is removed in each timestep"[13]. Ergo, the damping effects do not depend on the mass properties at all.
Additionally, Bullet does not have the notion of materials. All contact-related properties like the friction coefficient are stored directly in the rigid body object. On contact, the coefficients from the two objects are multiplied to obtain the actual value. In order to have realistic friction between all material types, one must find object friction coefficients such

[1] `https://github.com/HRI-EU/Rcs`

that their product gives the proper value for every material combination. Such values can only be found for a small set of materials, when using too many, some combined coefficients will be wrong. Therefore, Bullet is not suited for setups with a high diversity of materials.

3.2 rllab (Reinforcement Learning Framework)

Duan et.al. introduced rllab as a framework for reinforcement learning algorithm developement and benchmarking [11]. The source code can be found on GitHub[2]. The key feature of rllab is it's high modularity. As opposed to other machine learning frameworks such as tensorforce[3], there is a clear separation between the environment, the policy acting on the environment, and the machine learning algorithm optimizing said policy. This modular structure does not only allow to easily swap out these components to test new combinations, it also simplifies extending the framework. The main components are listed below.

The environment (`rllab.envs.base.Env`) defines the world the policy will be in. It exposes the shape and value range for actions and observations as well as a simple protocol to generate rollouts, as can be seen in Figure 3.1. Rllab also defines a number of environment wrappers. These decorate an environment with additional functionality, such as observation noise or delayed actions. Every wrapper contains a reference to an inner environment, and ultimately forwards all calls to it, only modifying parameters and return values as desired. Because of that, multiple wrapper environments can be used to form a chain.

The policy (`rllab.policies.base.Policy`) defines the structure of the policy function. Any policy will define a `get_action()` method, which computes the actions for a given observation, as well as an interface to get/set the trainable parameters. In addition, a policy is either defined to be stochastic or deterministic. A deterministic policy always provides the same action for a given observation. A stochastic policy computes parameters for the action distribution. The actual action is then sampled from the specified distribution. The rllab framework provides a number of common policy implementations based on configurable multilayer perceptrons.

The interface for algorithms is not defined, since there is a wide range of approaches requiring different structures. Those algorithms performing batch policy search inherit from a common base class, which defines additional components.

The `Sampler` component defines how the samples for the batch optimization are obtained and preprocessed. Modifying this component allows to introduce i.e. the CVaR sample subselection for EPOpt. The default implementation of the sampler is capable of spawning multiple processes in order to perform rollouts in parallel.

The `Baseline` function is the approximation of the value function $V^{\pi}(\mathbf{s}_t)$. During each iteration, it will

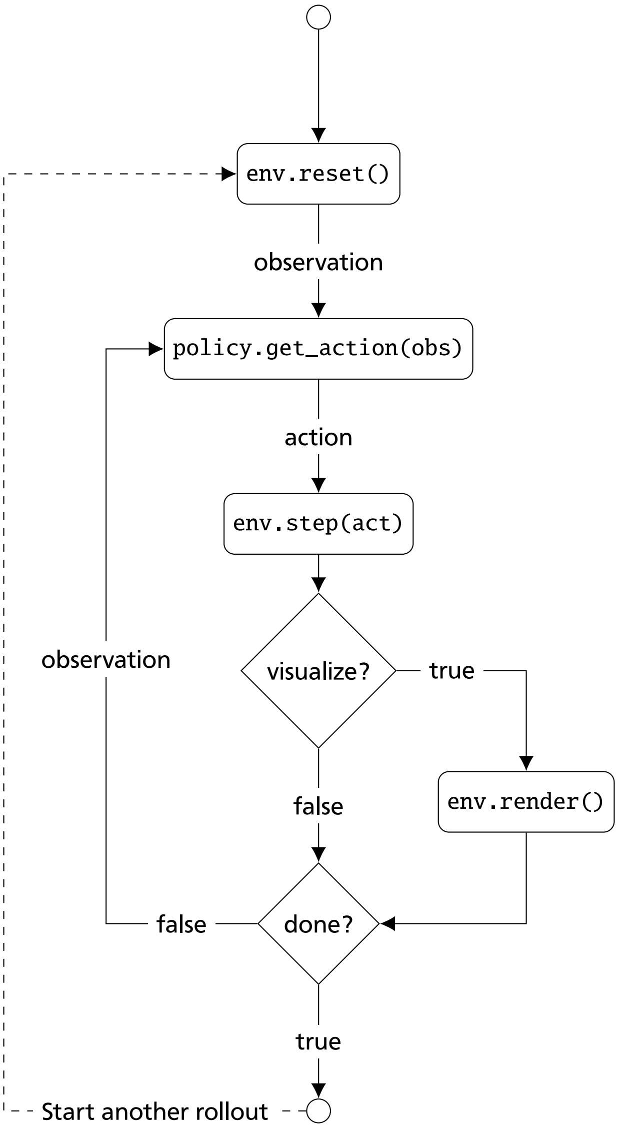


Figure 3.1: Lifecycle of an rllab environment. The `reset()` method is called to prepare the environment for the rollout. It returns the initial observation. The `step()` method will perform the given action and return the new observation, the step reward and a flag whether the task is done or not. This method will be repeatedly called with the action computed by the policy. If visualization of the environment is desired, the `render()` method should be called after each step. When simulating rollouts for training, rendering is omitted to speed up the process.

[2] https://github.com/rll/rllab

[3] https://github.com/reinforceio/tensorforce

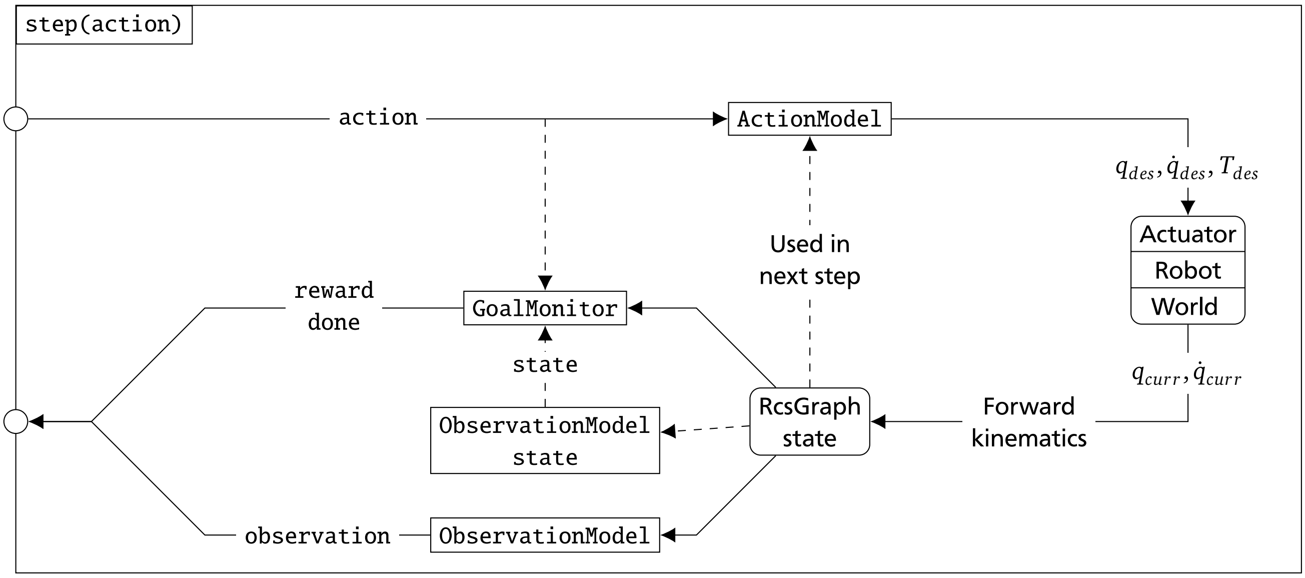


Figure 3.2: Data flow inside the `step()` function. The boxes with hard corners are interchangeable components. The boxes with rounded corners are part of Rcs. Both the physics simulation and the robot drivers take desired joint states $\mathbf{q}_{\mathrm{des}}$, velocities $\dot{\mathbf{q}}_{\mathrm{des}}$ or torques $\mathbf{T}_{\mathrm{des}}$ as input and return the current joint states $\mathbf{q}_{\mathrm{cur}}$ and velocities $\dot{\mathbf{q}}_{\mathrm{cur}}$.

be fitted to the measured values from the rollouts. In the next iteration, the fitted function is used to compute the advantages. The default baseline function uses a linear combination of nonlinear features. The baseline parameters are determined using a least-squares fit on the recorded trajectories. The features for one timestep consist of the states $s_{i,t}$, their squares $s_{i,t}^2$, as well as the time step index dependent values t, t^2, t^3 and a constant value of 1 for a bias.

Since rllab was written for benchmarking, ready-to-use implementations of a lot of algorithms already exist. These include the algorithms TRPO [1], REPS [2], CMA-ES [14] and many more. However, the results generated by this implementation of REPS and CMA-ES are surprisingly bad, which is even noted in the original paper. Investigating source code, it seems that the implementation of REPS is suboptimal[4]. We planned to reproduce the experiments from the original paper to look into these issues. In order to do so, we asked the authors for their hyperparameter settings[5], but so far have received no response. TRPO performs as expected, which is not surprising since that algorithm was developed by the same group that implemented rrlab. Since we planned to use TRPO from the start, we decided not to investicate this issue further.

TRPO requires a stochastic policy formulation. The stochasticity is used for exploration. The common stochastic policy provided by rllab uses a neural network for the mean and a constant, trainable parameter for the standard deviation of a gaussian distribution. Ideally, the standard deviation parameter will automatically shrink as the training progresses. In practice, however, the standard deviation takes way longer to do so then the policy needs to converge. When evaluating the policy, we do not want it to explore further, but we want a stable, reproducible result. In order to get that, we want to use a deterministic policy. So, we implemented the evaluation rollout function to use the mean action of the instead of the randomized action originally returned.

3.3 RcsPySim (Python Adapter for Rcs)

In order to enable using the Rcs simulator from rllab, we implemented a wrapper library called RcsPySim.

The core of this library is the equally-named C++ class RcsPySim defining `reset()`, `step()` and `render()` methods as expected by rllab. These methods are exposed to python using the pybind11[6] framework. Rllab requires environment implementation to extend the `Env` base class. Since it is not possible to do so in C++, we wrote a thin wrapper class in python which extends the proper base and forwards all methods to the C++ implementation. The wrapper also takes care of serialization, which is important for rllab's parallelization capability.

The main logic of RcsPySim is distributed to pluggable components. This modular structure has two benefits: first, it allows to simply and clearly swap out parts of a single task, for example to use a different reward model. Second, it

[4] Personal communication

[5] `https://github.com/rll/rllab/issues/223`

[6] `https://github.com/pybind/pybind11`

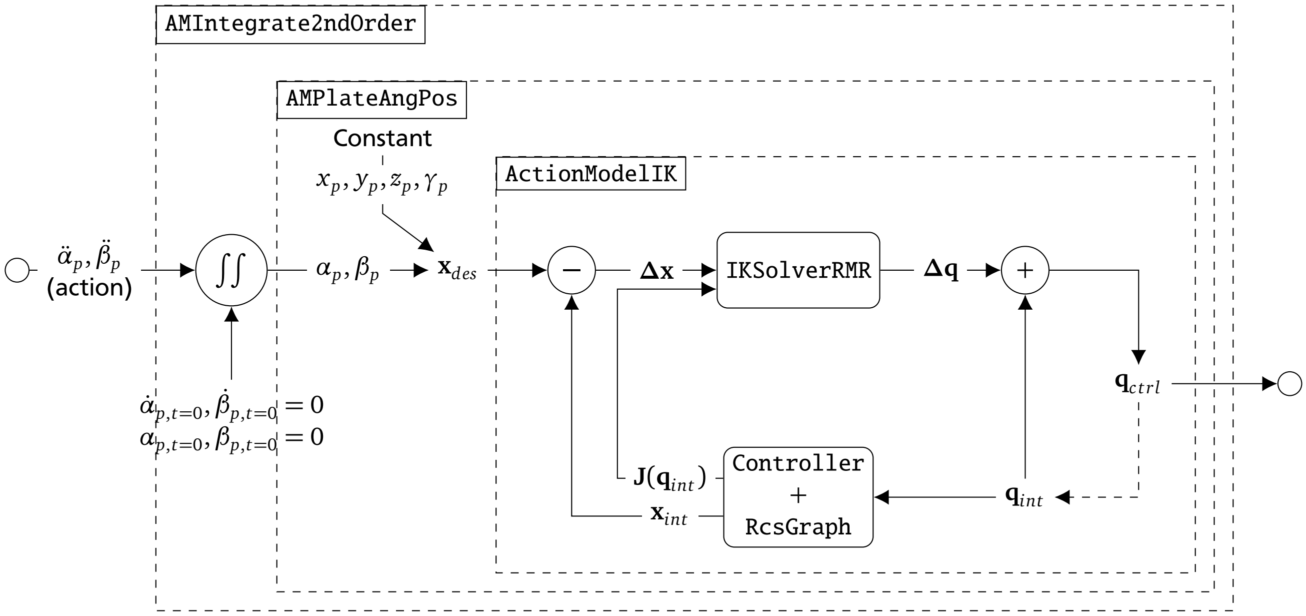


Figure 3.3: Data flow of `AMPlateAngAccelInt`. The task space consists of the six degrees of freedom of the plate: $\mathbf{x} = (x, y, z, \alpha, \beta, \gamma)^T$. α and γ are provided, the other values are fixed to prevent the IK solver from choosing the arbitrarily. The IK solver does not use the measured graph state, it only uses the desired joint positions it computed as foundation for the task state in the next time step.

allows to reuse these components for other tasks. An overview of the interaction of these components can be seen in Figure 3.2.

In Rcs, both the physics simulation and the robot drivers require joint commands to operate. The `ActionModel` converts the action vector passed to the `step()` method into these joint commands. A simple `ActionModel` could just copy the action vector into the q_{des} vector to allow controlling the joints directly. On the other hand, `ActionModelIK` encapsulates Rcs' inverse kinematics system. Here, the actions are task-space inputs, which are then converted to joint space by the inverse kinematics solver. The IK solver requires desired positions. However, it might be desireable to command velocities or accelerations. In order to facilitate that, we implemented decorators that integrate the action once or twice before passing it to a delegate `ActionModel`. A good example of this structure can be seen in Figure 3.3. The action consists of the angular acceleration around the plate's x and y axis. The other parts of the plate position should be fixed. First of all, `AMIntegrate2ndOrder` integrates these values to turn them into the desired euler angles. Then, they are passed to `AMPlateAngPos`, which now adds fixed values for the plate's position and z rotation. Without fixing those values, the inverse kinematics would vary them freely. The resulting inverse kinematics input vector $\mathbf{x}_{des}$ now constrains all six degrees of freedom of the plate body. Now the solver implemented in `ActionModelIK` can do the rest.

The `ObservationModel` is responsible for reading the state of the simulation at the end of the step and formatting it to the observation vector that will be returned to the policy.

The `GoalMonitor` defines the objective of the task. It will be called at the end of the step to determine the step reward, as well as check whether the task is done due to success or failure. The most common reward formula is based on the quadratic cost function:

$$C(\mathbf{s}, \mathbf{a}) = -\left((\mathbf{s} - \mathbf{s}_{\text{des}})^T \mathbf{Q} (\mathbf{s} - \mathbf{s}_{\text{des}}) + (\mathbf{a} - \mathbf{a}_{\text{des}})^T \mathbf{R} (\mathbf{a} - \mathbf{a}_{\text{des}})\right) \tag{3.1}$$

Here, $\mathbf{s}_{\text{des}}$ are the desired state values, $\mathbf{a}_{\text{des}}$ are the desired action values and $\mathbf{Q}$ and $\mathbf{R}$ are diagonal matrices used to weigh single entries of the state and action matrices. Since we use acceleration-based actions, the desired action values are generally $\mathbf{0}$ to encourage a stable state. This reward is similar to the cost function of a Linear-Quadratic Regulator (LQR). A generic implementation of equation (3.1) is provided by `GMGenericQuadr`. The problem with a cost-based reward is that it will always be negative, so every step the policy takes will be penalized. In a task that can reach a failure state early, that is an issue since it might be cheaper to fail fast then to take a long path towards success. The usual way to migitate this issue is to make the reward in a terminal state extremely large, using a negative value on failue and a positive one on success. Unfortunately, early testing showed that the large terminal reward does not work well with some reinforcement learning algorithms such as TRPO. The terminal reward offset introduces a higher variance into the rewards. Additionally, tasks such as balancing do not terminate on success. Here, we'd rather want a positive reward for every step close to the goal, and a smaller one for steps further away. The easiest way to turn a negative cost into a reward like this is using the exponential function:

$$r_{\exp}(\mathbf{s}, \mathbf{a}) = \exp\left(C(\mathbf{s}, \mathbf{a})\right) \tag{3.2}$$

The exponential function ensures that the step reward is always in the range $(0, 1]$. However, if the cost gets too big, the reward will become extremely close to 0, thus improvements are barely visible if the distance to the desired state is high. In order to avoid this problem, a minimum reward r_{min} can be set as hyperparameter. It is used to compute the normalization factor c_{max}:

$$c_{max} = -\frac{\ln(r_{min})}{C(\mathbf{s}_{max}, \mathbf{a}_{max})} \tag{3.3}$$

Here, $\mathbf{s}_{max}$ and $\mathbf{a}_{max}$ should be state and action values which result in the maximal quadratic cost. For the positional parts of the states, these limits are easy to compute. In the ball-on-plate task, the rollout is failed if the ball falls of the plate, so the size of the plate is a natural boundary for the ball position. Similarly, the plate's angular position will never exceed ±90°, since higher angles would be redundant. Selecting sensible maxima for velocities is not that easy. So, the cost is not absolutely guaranteed to be smaller then $C(\mathbf{s}_{max}, \mathbf{a}_{max})$, but in practice, that is not an issue. Combining the result of equation (3.3) with equation (3.2), we get the final equation for the exponential reward function:

$$r_{\exp}(\mathbf{s}, \mathbf{a}) = \exp\left(\frac{C(\mathbf{s}, \mathbf{a})}{c_{max}}\right) \tag{3.4}$$

In Rcs, the internal state is described by the joint states $\mathbf{q}$. However, using joint angle state for the reward function is not always a good idea. For example, we might want to penalize the distance between two moving objects. Instead, we need a task specific state vector, for which we use the observation vector that is computed anyways to be returned to the policy. A generic implementation of equation (3.4) is provided by `GMGenericExp`.

If the initial state of the simulation is fixed, it can just be defined in the graph xml. If the initial state should be random, the `reset()` method will accept a parameter for changing the initial state. The initial state is not necessairily identical with the observable state. On one hand, it can be desireable to have parts of the state fixed. On the other hand, the randomized state variables might not be observable directly. The `InitStateSetter` is the component responsible for this part. It takes the init state vector, parses it and modifies the relevant parts of the internal state. The selection of the initial state is left to the python side. The python wrapper can either take a fixed value or generate a random one. The values are generated in python because the random number generator interface of python has more features. These functions also automatically use the global seed set by rllab.

3.3.1 Physics Parameters

Many of the experiments performed in chapter 4 require to run simulations with different physics parameters, so we need a way to change them. Most physics parameters are set on the `RcsGraph` and read by the physics simulation on construction. In order to apply changes to these parameters, one just needs to create a new physics simulation instance and discard the old one. Some parameters, notably material parameters and the body damping values, cannot be stored in the graph description. These values are stored separately and applied after the simulation has been recreated. The `PhysicsParameterUpdater` is responsible for applying those parameters. It has one subclass for Vortex and Bullet each containing the specific code to set non-standard parameters. More details on the parameter update process can be found in Figure 3.4.

Even though these parameters can be set on any part of the graph, it might be desirable to only make certain properties available. For that reason, there is a list of `PhysicsParameterDescriptors`. Each of them describes a value or a group of values that can be set for a specific body. A list of settable physics parameters can be found in Table 3.1.

The mass of a body is pretty straightforward. By default, Rcs assumes that the density of a body is uniform. However, an inhomogenous mass distribution is one of the more interesting cases of varying physics. One can modify the center of mass and the inertia tensor components directly in order to simulate the desired behaviour. Unfortunately, there are many configurations that are not physically plausible, and it is hard for an external perturber to choose these parameters correctly.

Another thing that may change is the object geometry. For that, we implemented a descriptor for the radius of the ball in the ball-on-plate task. Here, it is not enough to change the radius. Since the coordinate frame of the ball is at it's center, and the ball should initialize lying on the plate, the ball's position also needs to be adjusted by this descriptor.

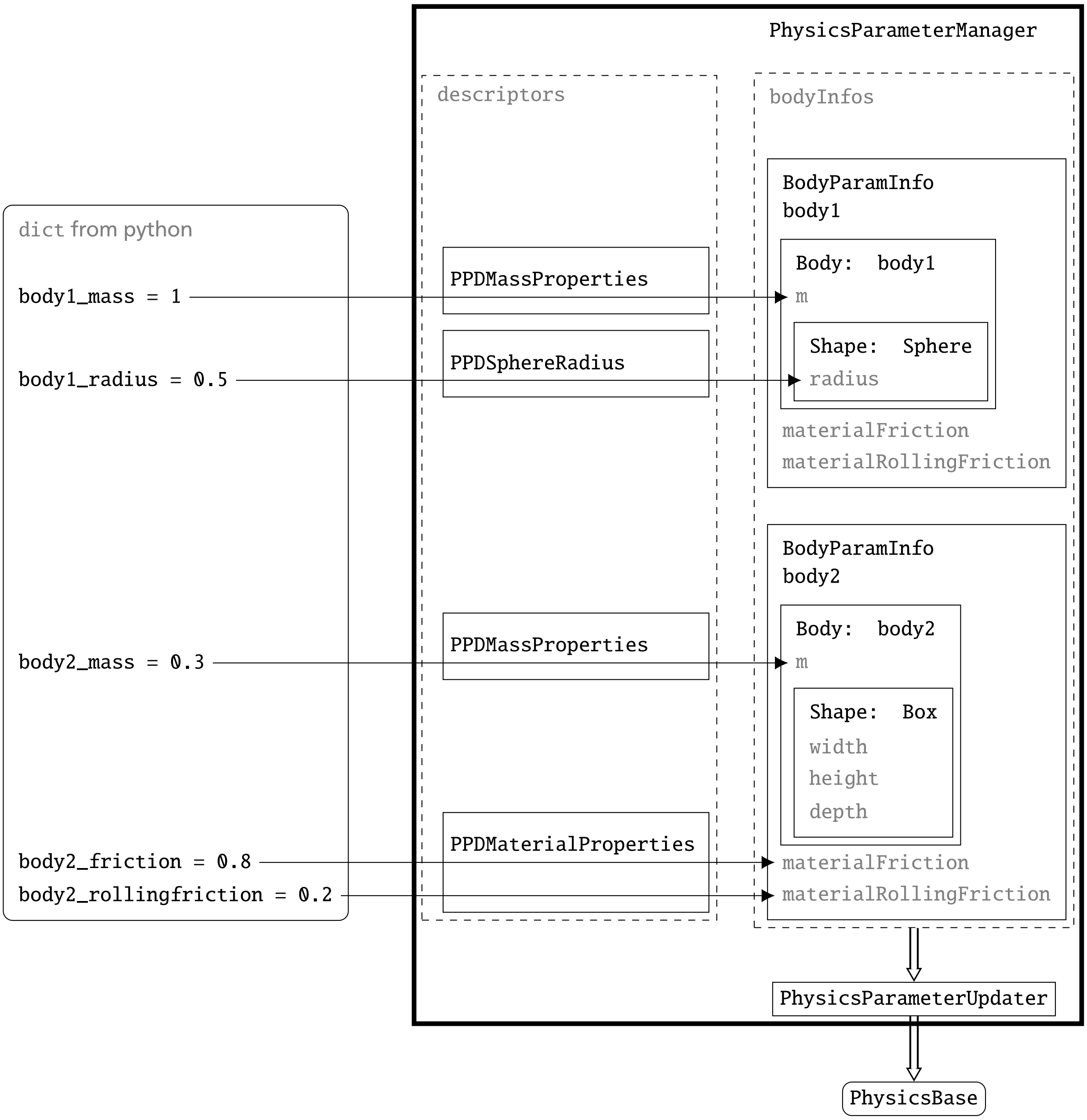


Figure 3.4: Data flow for the physics parameter modification. The values are passed in a `dict` from python. The name of every parameter is prefixed with the name of the body it is set on. The individual `PhysicsParameterDescriptors` look up values in the `dict` and write them into the fields of the `BodyParamInfo` for the body. The `PhysicsParameterUpdater` gets the list of `BodyParamInfos` and transfers the parameetr values to the simulator. The `PhysicsParameterManager` encapsulates the whole operation.

Name	Target	Descriptor	Vortex	Bullet	Notes
Mass	Body	`PPDMassProperties`	✓	✓	
Center of mass	Body	`PPDMassProperties`	✓	✓	3 Values, will be computed from shape and mass if not set
Sphere radius	Shape	`PPDSphereRadius`	✓	✓	
Linear velocity damping	Body	`PPDBodyDamping`	✓	✓	Meaning in Vortex and Bullet is very different
Angular velocity damping	Body	`PPDBodyDamping`	✓	✓	Meaning in Vortex and Bullet is very different
Friction coefficient	Material	`PPDMaterialProperties`	✓	✓	
Rolling friction coefficient	Material	`PPDMaterialProperties`	✓	✓	Actual value is scaled with the sphere radius
Slip	Material	`PPDMaterialProperties`	✓		
Compliance	Material	`PPDMaterialProperties`	✓		

Table 3.1: Physics parameters settable in RcsPySim

The friction coefficient is a unitless scalar describing the ratio between the contact normal force and the maximum friction force. The physics engine contact solver will apply as much force as necessairy to reduce relative linear object motion at the contact point. This coefficient is also called linear friction coefficient later on.

$$F_f \leq F_N * \mu_f \tag{3.5}$$

For a round object like a ball, the linear friction coefficient does not have a high influence. While the object is rolling, the relative linear velocity in the contact plane is already 0, so the friction won't do anything. In order to model those interactions, there is a separate rolling friction coefficient μ_{rf}. Ideally, μ_{rf} functions similar to the linear friction coefficent, so that it is a unitless coefficent descibing the ratio between the contact normal force and the maximum friction torque. However, the actual implementation inside both Bullet and Vortex is defined as:

$$T_f \leq F_N * \mu_{rf} \tag{3.6}$$

Since force is usually measured in N and torque in Nm, this definition of μ_{rf} is not unitless, but has the unit m, so it doesn't match the ideal definition of a friction coefficient. The correct solution is to premultiply μ_{rf} with the radius of the rolling object, as is stated by the Vortex SDK documentation. For Bullet, the documenation is not clear on this, but experimental results show similar behaviour. So, the full formula for the rolling friction is:

$$T_f \leq F_N * \mu_{rf} * r_{\text{contact}} \tag{3.7}$$

Here, r_{contact} is the curvature radius at the contact point. The physics simulation cannot compute it internally, so it needs to be premultiplied. In our case, r_{contact} is simply the radius of the rolling object, i.e., the ball. Since we set that parameter as well, we can compute the value for the physics simulation automatically. All values for the rolling friction defined later use the formula from equation (3.7).

We set all parameters per body. However, there is not one friction and rolling friction coefficient per body, there is one per combination of bodies. Bullet stores each coefficient per body and multiplies them to get the combined value. Vortex stores these parameters in material objects. The combined value is the minimum of the values from the two involved materials. For the ball-on-plate task, we only have two objects - the ball and the plate. We set the friction and rolling friction coefficients of the plate to 1. This way, the combined coefficients will be equal to the ball's coefficients in both Vortex and Bullet.

The linear and angular velocity damping coefficients are mainly intended to improve numerical stability. They do not have a direct physical meaning. Vortex applies a constant force $F_d = d_{\text{linear}} * v_{\text{Object}}$ or torque $T_d = d_{\text{angular}} * \omega_{\text{Object}}$ opposing the object's motion. These formulas model a constant friction without an interaction with any object. In Bullet, they do not have a proper unit, but are defined roughly as "What fraction of the velocity is removed in each timestep"[13]. Thus the damping does not depend on the mass properties of the object in Bullet. Usually, these parameters are used to avoid feedback loops which lead to numerical instability. Due to the fact that they have no equivalent in the real world, we decided to set them to 0 for all experiments. We did not encounter instabilities resulting from this setting.

4 Experiments

4.1 Task

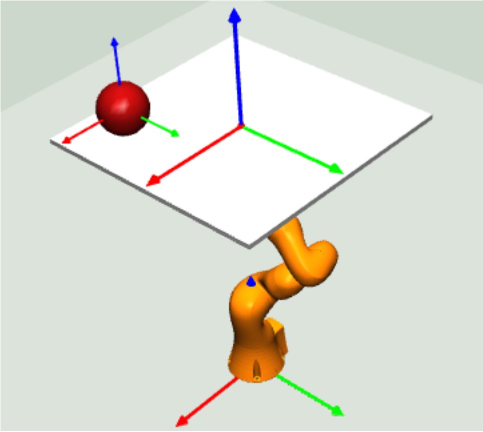


Figure 4.1: Task setup rendered by Rcs. The goal is to get the ball to the marked center of the plate.

The setup consists of a ball lying on a horizontal plate. The plate is held by a robot and can be controlled directly. The goal is to move the ball to the middle of the plate and keep it there. The initial position of the ball is randomized on a circle arount the plate center. The actions are given in the plate's task space. The robot's desired joint angles are computed by Rcs' inverse kinematics solver. Since using the policy output as position command leads to very extreme behaviour, the actions are interpreted as accelerations instead. They are integrated separately to obtain the positions required as input for the inverse kinematics. Thus, the full action vector **a** is defined as:

$$\mathbf{a} = \begin{pmatrix} \ddot{\alpha}_{\text{Plate}} & \ddot{\beta}_{\text{Plate}} \end{pmatrix}^T \tag{4.1}$$

The state vector consists of the plate's controllable state: the angular position and velocity around the x and y axes and the ball's position and velocity relative to the goal state. So, we get the full state vector **s**:

$$\mathbf{s} = \begin{pmatrix} \alpha_{\text{Plate}} & \beta_{\text{Plate}} & x_{\text{Ball}} & y_{\text{Ball}} & z_{\text{Ball}} & \dot{\alpha}_{\text{Plate}} & \dot{\beta}_{\text{Plate}} & \dot{x}_{\text{Ball}} & \dot{y}_{\text{Ball}} & \dot{z}_{\text{Ball}} \end{pmatrix}^T \tag{4.2}$$

The balls angular position and velocity are not included since they would be hard to measure in the real world and they don't provide any needed information for this task.

If the ball falls off the plate, the rollout is terminated. If the ball reaches the target position, the rollout will continue until the maximum step count is reached in order to train the policy in keeping the ball steady. The reward is computed using the exponential reward function as defined in equation (3.4).

4.2 Baseline Policy

First of all, we trained a policy on the task with fixed physics parameters. This policy will serve as baseline to gauge the results of the following experiments. Additionally, good hyperparameters for the simple case will likely also be a good starting point for other training runs.

The physics parameters the baseline policy is trained on can be found in Table 4.1. These values will be called the nominal parameter values later on. Since the real world ball and plate objects were not available for measurements, these values have been selected arbitrarily.

As policy, we're using a feed-forward neural network with two hidden layers, each with 16 neurons and tanh nonlinearity. The output of the network is used as mean of a gaussian distribution, making the policy stochastic. The standard deviation

Parameter	Unit	Value
ball_mass	kg	0.30
ball_radius	m	0.10
ball_friction	-	0.20
ball_rollingfriction	-	0.05
ball_com_z	m	0.00

Table 4.1: Nominal physics parameter values.

is a parameter for each action variable. It does not depend on the observation, but can be modified by the training algorithm. Ideally, the standard deviation will go down as the training is successful.

The policy is trained using TRPO. We use 300 batch training iterations, with at least 6000 total steps per batch. With one rollout taking at most 300 steps, these settings lead to at least 20 trajectories per batch. The sampler will perform more rollouts if necessairy to get the desired amount of data points. This mechanism increses the exploration at the start where a lot of runs will fail after just a few steps. We use a step length of 0.02 s, so the maximum rollout duration is 6 s.

4.3 Influence of Physics Parameters

To interpret the performance of the policy on different physics environments, we first need to measure the effects each parameter has on the task. Using a fixed initial state and a uniform control input that doesn't depend on the observation, we record trajectories using different parametersets.

The first setup has the plate accelerate constantly at $\ddot{\alpha}_{\text{Plate}} = 0.05\,\frac{\circ}{\text{s}^2}$. We then look for the timestep where the ball's linear acceleration is significantly larger then 0. We record the value of α_{Plate} from that step. This value is the angle at which the static friction is overcome. The expected value can be calculated using equation (4.3).

$$\alpha = \tan^{-1}(\mu) \tag{4.3}$$

In the second setup, we initialize the plate at $\alpha_{\text{Plate}} = 15^\circ$ and keep it's position constant. The ball should start rolling down the plate at constant acceleration, which we then measure.

In all of these setups, we first vary each parameter individually, the respective other parameters stay on their nominal value. This way, we can judge every parameter's influence without interference. However, some parameters interact to influence the result. For these, we took two parameters to vary, and then plotted the result for each combination as heatmap.

Name	Unit	μ	σ^2	Min clip
ball_mass	kg	0.30	0.200	0.01
ball_radius	m	0.10	0.070	0.01
ball_friction	-	0.20	0.170	0.00
ball_rollingfriction	-	0.05	0.070	0.00
ball_com_z	m	0.00	0.005	-

Table 4.2: Physics parameter distributions used for EPOpt experiment. All parameters use normal distributions. The min clip value is the absolute minimum, sampled values that fall below it are cut off. The clipping is done to avoid artifacts with i.e. zero mass values.

4.4 Setup for EPOpt

EPOpt requires two parts. For every physics parameter, a normal distribution is used to sample parameters for a rollout. Table 4.2 lists the physics parameters and the distributions used for them. Only the parameters of the ball are modified. The plate is moved by the inverse kinematics and the robots joint angle controllers, thus varying parameters of the plate would not have a large influence. We could vary the plates dimensions, but that isn't very useful either. The only way for the policy to detect changed plate dimensions is to see that the ball falls off, and in that case the rollout would be done and the policy cannot react any more.

The center of mass is only varied in the z direction, which is also the z direction of the world at the start of the rollout. Varying the x or y shift would result in the ball starting to roll immediately. If the initial position is chosen badly, the ball would roll off the plate directly, and avoiding these combinations is not straight forward. Also, the direction of the shift does not matter on a ball, since a simple rotation of the object would limit it to one axis again. We can have positive or negative values for the center of mass shift. A positive value results in a higher center of mass, causing the ball to be stable when not disturbed, but to start rolling quickly if the plate starts shifting. A negative value results in a lower center of mass, making it harder to get the ball rolling in the first place.

EPOpt optimizes the conditional value at risk $\mathrm{CVaR}_\epsilon(R(\tau))$. The selected quantile ϵ is a hyperparameter of the algorithm. To optimize the CVaR, we implemented a custom `Sampler` that will select a subset of rollouts as specified in equation (2.4). In the original paper [9], the training run used $\epsilon = 1$ for 100 iterations at the start before switching to $\epsilon = 0.1$. We used the same approach and compared it with using $\epsilon = 1$ for the whole rollout.

We used TRPO as underlying batch policy optimization algorithm. The hyperparameters are the same as for the baseline policy, with one exception. Since EPOpt uses random physics parameters for each rollout, we need a lot more rollouts in order to get a decent coverage of the physics parameter space. When optimizing the conditional value at risk, only a small subset of rollouts is actually used, increasing the required amount of rollouts even more. This approach is very sample inefficient. The original paper used 240 rollouts per batch optimization, which would result in 24 rollouts actually used with $\epsilon = 0.1$. We used the same value for that setup. For $\epsilon = 1$, we only used 120 rollouts.

4.5 Setup for RARL

Pinto et.al. used a strongly modified version of rllab to perform their experiments [10] . Their source code can be found on GitHub [1], but due to the heavy modifications it does not readily work with our code. Instead, we implemented our own version.

In the ball-on-plate task, we designed the adversary to apply a force to the ball. This force is aligned with the x-axis of the plate's reference frame. We used a one-dimensional variable because it makes the adversary's reaction easier to visualize. We also tested an adversary that could apply force in the z axis of the plate, but quickly discarded that approach since this adversary could lift the ball off the plate, removing any ability for the robot to manipulate it.

As far as possible, we used the same hyperparameters as the original paper [10]. Even though not all parameter values can be found in the paper, the remaining ones have been extracted from the published code on GitHub. However, the source code only contains default values for the parameters. Pinto et.al. could have modified used modified parameters passed via the command line to create their results. Therefore, a bit of research into the parameter selection is necessairy. The most interesting parameters are the iteration counts and the adversary force cap. RARL has three iteration counts: One for the number of consecutive protagonist training steps N_μ, one for the number of consecutive adversary training

[1] `https://github.com/lerrel/rllab-adv`

steps N_ν, and one for the number of overall rounds of protagonist/adversary training runs N_{iter}. Consequentially, the total number of batch policy optimization runs is given by $(N_\mu + N_\nu)N_{\text{iter}}$. Due to the high computation time necessairy per batch iteration, we can only put either N_{iter} or the other two on a high value. The paper states that Pinto et.al. used $N_{\text{iter}} = 100$ for the inverted pendulum task, which is slightly less complex than our ball on beam task, and $N_{\text{iter}} = 500$ for the other task which are more complex than ball on beam. According to their sourcecode, they used $N_\mu = 1$ and $N_\nu = 1$. Since these values are not guaranteed to be exact, we also compared that approach with a training run using $N_{\text{iter}} = 10$, $N_\mu = 40$ and $N_\nu = 40$.

Another change affects the batch size and maximum rollout length. The tasks in the original paper do not have a randomized initial state, so the problems can be learned with less samples. Instead, we used the same parameters as when training our reference policy in Section 4.2.

The last important parameter is the maximum force used by the adversary $F_{\text{a,max}}$. A larger value leads to a stronger adversary, which in theory should lead to a more robust policy. A too large value however will lead to the policy never reaching the goal. We compared multiple values for the maximum force to verify this assumption.

4.6 Setup for Transition Noise

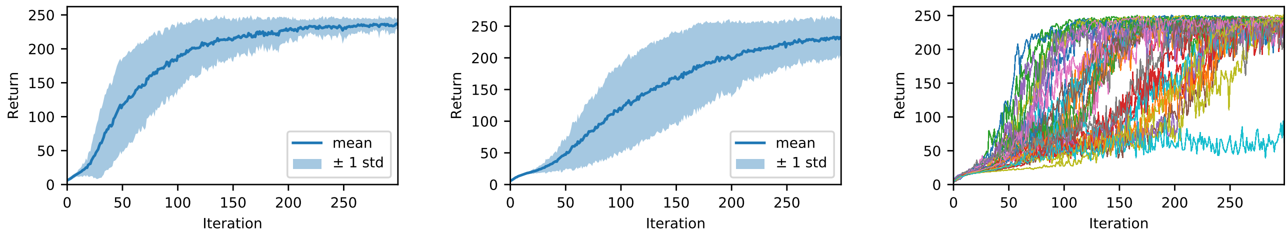
As mentioned in Section 2.4, the policy optimization algorithm sees the manipulations done by EPOpt and RARL as transition noise. So, we should also compare these algorithms with a setup using generic transition noise to see if there is a justification in using more specialized algorithms.

The transition noise is normally distributed and added to the joint state $\mathbf{q}_{\text{cur}}$ and velocity $\dot{\mathbf{q}}_{\text{cur}}$ returned from the physics simulation. The state of the physics simulation is then updated to match the noisy values. For the ball-on-plate task, $\mathbf{q}_{\text{cur}}$ has 13 entries: 7 values describe the joints of the robot, the last 6 are the degrees of freedom of the ball in world coordinates. $\dot{\mathbf{q}}_{\text{cur}}$ follows the same scheme. Since we are not interested in the details of the underlying robot control, we only apply noise to the ball's position. We only modify the x and y components of the state to avoid the ball flying off. Also, since the linear position of the ball is stored in meters and the angular position is stored in radians, we use different standard deviations for these. The standard deviation of the linear position noise is $0.001\,\mathrm{m}$, the standard deviation of the angular position noise is $0.05°$. For the velocities, we used $0.001\,\frac{\mathrm{m}}{\mathrm{s}}$ and $0.05\,\frac{^\circ}{\mathrm{s}}$. In total, we have 10 noise values per step. Using a bigger standard deviation leads to the training not converging, while smaller standard deviation results in no different behaviour from the baseline policy. Finding the best level of transition noise is hard since every variable could have an individual standard deviation. The other hyperparameters were the same as for training the baseline policy. Since transition noise is already modeled in the definition of the MDP, we use the regular TRPO algorithm to train this setup.

5 Results

This chapter is divided into three parts. First, we take a look at the training performance of the baseline policy, to show how well it solves the task. Second, we performe a sensitivity analysis of the parameters and compare the available physics engines Vortex and Bullet. Third, we analyze the robustness of the learned policies. We compare the baseline policy with various policies trained with EPOpt, RARL and transition noise.

5.1 Training the Baseline Policy



(a) Vortex

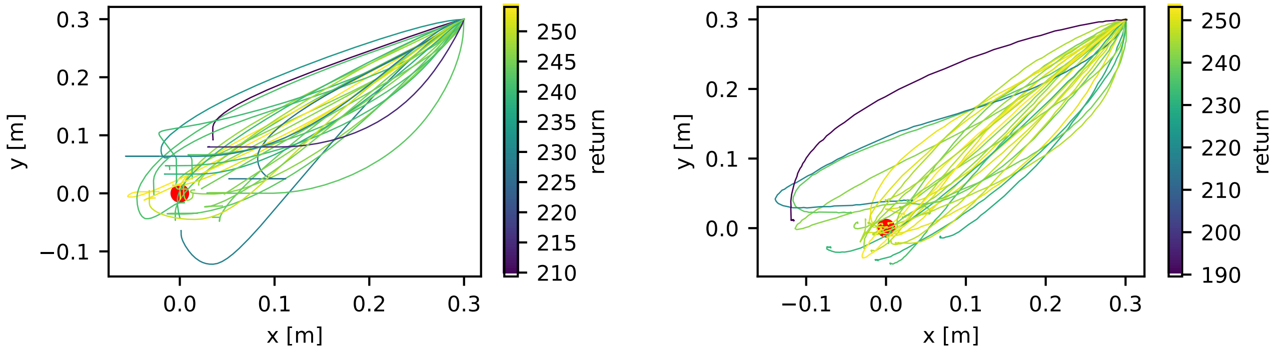
(b) Bullet

(c) Bullet - every single training run

Figure 5.1: Training curve of baseline policy. When using Vortex, the optimization takes about 200 iterations to converge. When using Bullet, we get a much higher variance, which is caused by one training run not converging at all.

Figure 5.1 shows the training progress of the baseline policy described in Section 4.2. Since the performance varies depending on the used random seed, we trained 30 policies individually. The maximal performance is reached after at most 200 iterations when using Vortex. In this instance, the training curve of the runs using the Bullet physics engine has a higher variance near the end. If we plot all learning curves individually, as done in Figure 5.1c, we can see that all training runs except for one eventually converge. It seems that training curve will stay low around a return of 75, until the algorithm has success once, at which point it will quickly converge to the maximum. The outlier also explains the higher variance bullet has in Figure 5.1b.

Additionally, Figure 5.2 shows the trajectories for the different policies from the same initial position. These plots shows that the different training runs did learn a variety of solutions, but they all get reasonably close to the target.



(a) Vortex

(b) Bullet

Figure 5.2: Paths for every training run from single initial position. The red dot marks the desired ball position. The color of the path encodes the return the policy would get for that approach.

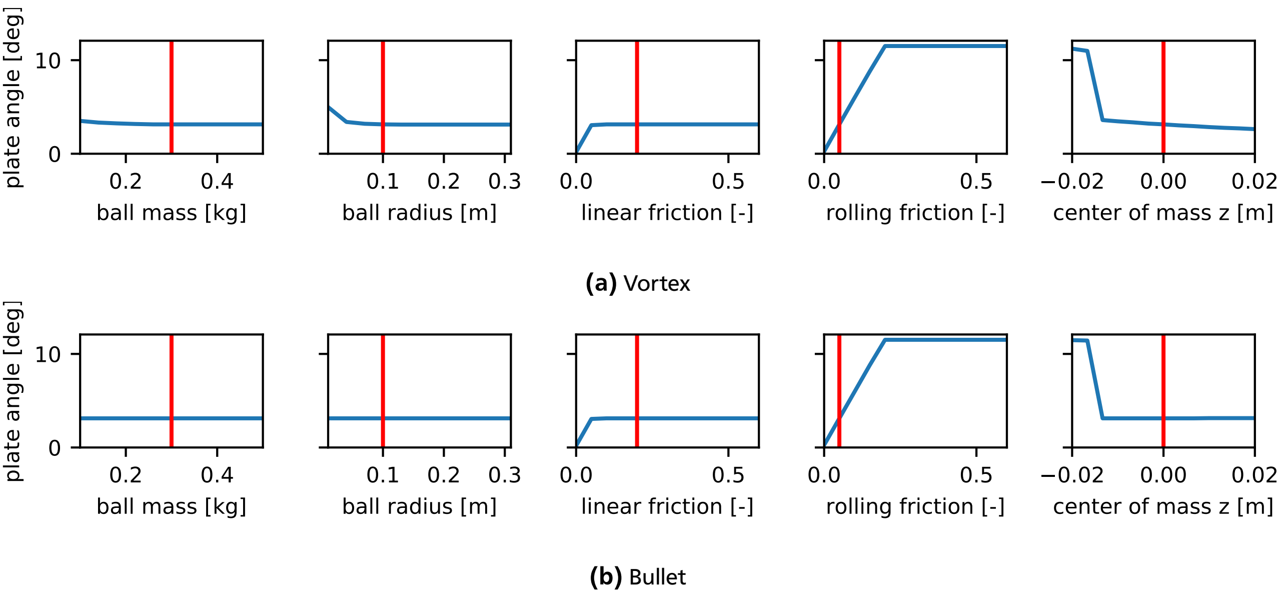


Figure 5.3: Plate angle at which the static friction is overcome, depending on a single perturbed physics parameter. The ball mass and radius parameters do not have a high influence while linear friction, rolling friction and center of mass position do.

5.2 Influence of Physics Parameters

We performed three experiments to quantify the influence of the physics parameters on the overall performance. For each setup, we varied every parameter individually while keeping the others at their nominal value. The red vertical line always marks the nominal parameter value as specified in table 4.2.

5.2.1 Overcoming Static Friction

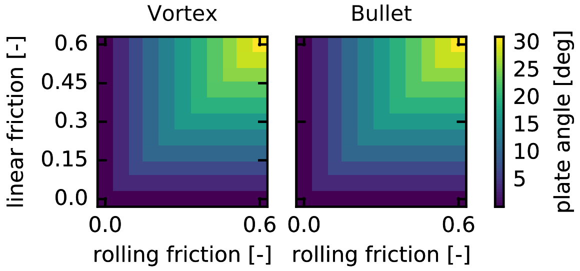
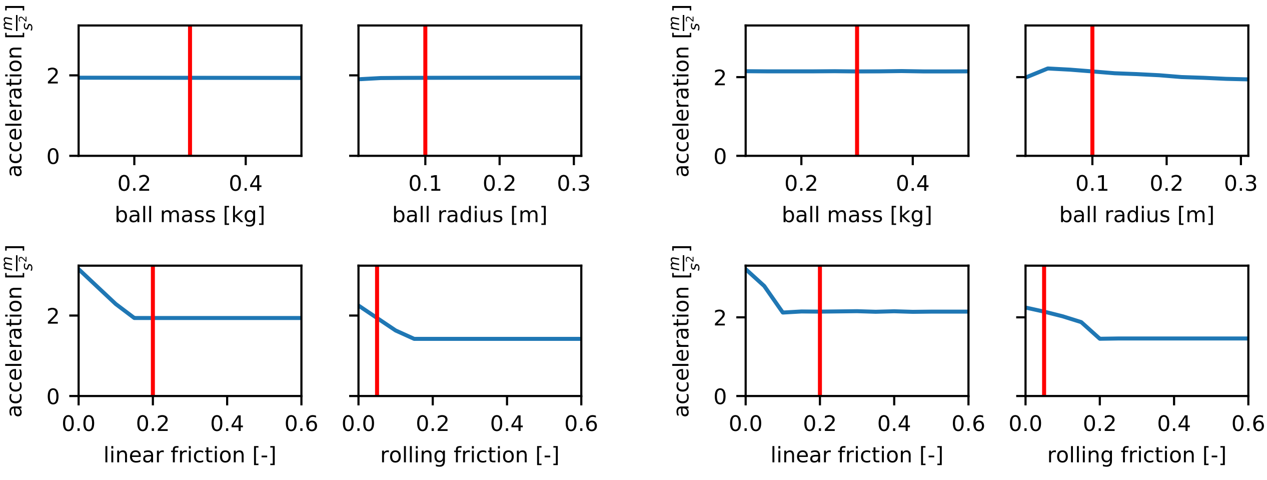


Figure 5.4: Plate angle at which the static friction is overcome, depending on linear friction and rolling friction. We can clearly observe that only the smaller friction coefficent determines the result.

Figure 5.3 shows the angle of the plate at which the ball can overcome the static friction. In general, it appears that there are some artifacts in the Bullet version. One can clearly see that the ball mass has no influence on the results. The friction and rolling friction coefficients each have a high influence. However, both of them apruptly stop making a difference when they get higher than a certain value, which is 0.05 for the friction and 0.2 for the rolling friction coefficient. These values are identical to the nominal value for the respective other friction coefficient.

In order to analyze this behaviour closer, we plotted the value for every combination of friction and rolling friction in Figure 5.4. The measured values are consistent with Equation (4.3). Only the smaller of the two friction coefficients determines the result. This observation also fits the expectations, since the smaller valued friction part will give in first. If the rolling friction is lower, the ball will start to roll. If the linear friction is lower, the ball will start to slide.

The center of mass offset also shows differences between Vortex and Bullet. Both show a big jump between −0.016 m and −0.013 m. For lower values, the roll start angle is about 11.5°, which is the same value as we get for the nominal friction if the rolling friction is larger. In that case, the center of mass is located so low that the ball will not start to roll at all, and only move due to sliding when the angle is high enough for that. For higher values, the roll start angle is similar to the nominal value. For Bullet, is is constant at that value. For Vortex, we see a slight slope, with the ball starting to roll a bit earlier for higher center of mass positions. This behaviour matches the expectations, as a higher center of mass should excert a higher torque on the ball when tilted, and thus overcome the rolling friction earlier. In Bullet, we cannot make the same obervation. Instead, the angle is constant for all center of mass positions higher than the jump at −0.013 m.



(a) Vortex

(b) Bullet

Figure 5.5: Acceleration of the ball with plate at a constant angle, depending on a single perturbed phyiscs parameter. Again, only linear and rolling friction have a high influence, but for Bullet, the ball radius also modifies the outcome.

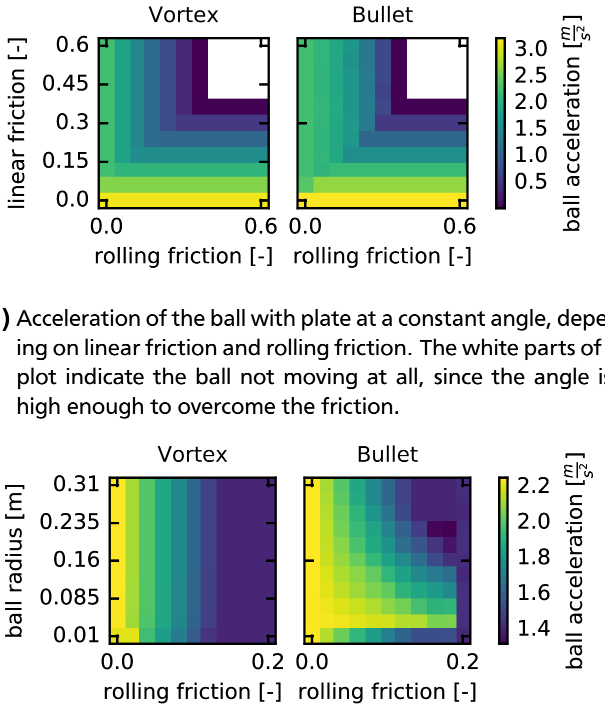
5.2.2 Acceleration on Tilted Plate

Figure 5.5 shows the linear acceleration of the ball on a plate tilted at a constant angle of 20°. Since the acceleration of the ball is not part of the task state, we computed it manually using finite differences of the ball's velocity. In theory, the acceleration should be constant over the whole rollout, but in practice it might vary a bit, so we took the mean of the accelerations over all time steps. The center of mass was not included into this test since it would violate that assumption.

With Vortex, the ball mass and radius have no influence on the result. Both linear and rolling friction do. Notably, the influence of the linear friction coefficient does not end once it gets larger than the rolling friction. Instead, the influence of both seems to stop at a coefficient value of about 0.16. For Bullet, the curves are a lot more uneven. The friction coefficients look similar, but both curves have a dent right before they become static. There also is a small reaction to a varying ball radius.

In order to take a closer look at the parameter interaction, we plotted the acceleration for varying linear and rolling friction coefficients. The results can be seen in 5.6a. We have same separation between dominating linear and rolling friction as we saw in Figure 5.4. However, the border is not running perfectly diagonal. This observation can be explained by the ball both starting to turn and to slip due to the low values of both parameters.

We also looked at the influence of the ball radius in comparison with the rolling friction coefficient, the results of which can be seen in Figure 5.6b. We used a smaller value range for the rolling friction coefficient since higher values don't have any influence. As expected, the radius has no influence in Vortex. For Bullet, ball radii smaller than 0.25 m lead to an increased acceleration. This effect disappears appruptly for the smallest ball radius of 0.01 m. This behaviour demands further investigation, which will be done in Section 5.2.4.



(a) Acceleration of the ball with plate at a constant angle, depending on linear friction and rolling friction. The white parts of the plot indicate the ball not moving at all, since the angle isn't high enough to overcome the friction.

(b) Acceleration of the ball with plate at a constant angle, depending on ball radius and rolling friction. We can see large differences between Bullet and Vortex for small ball radii.

Figure 5.6: Acceleration of the ball with plate at a constant angle, depending on combined physics parameters.

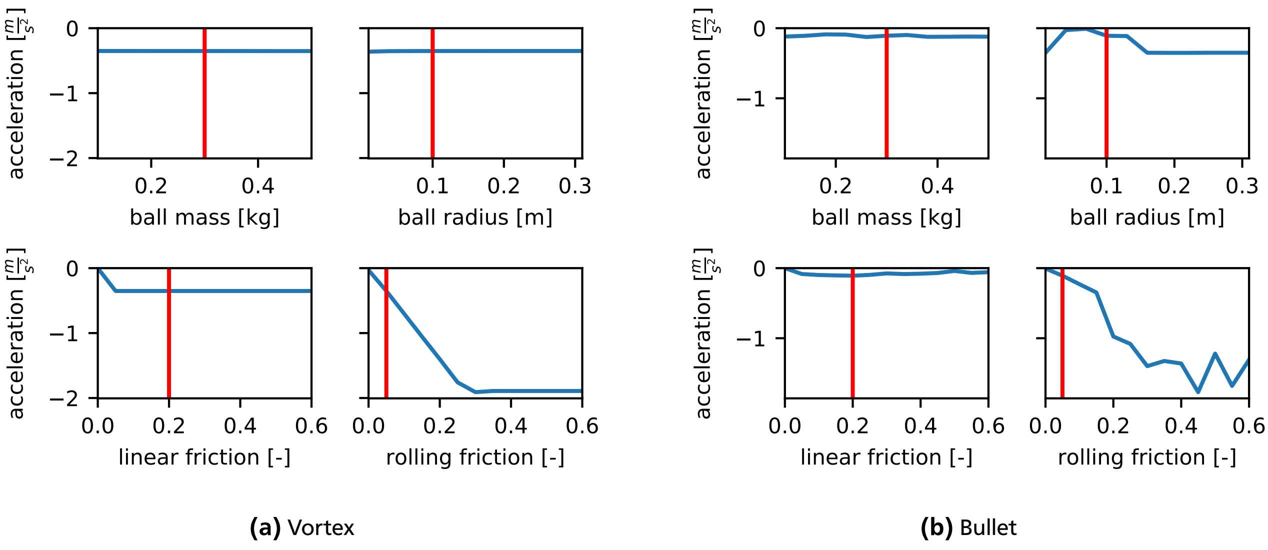
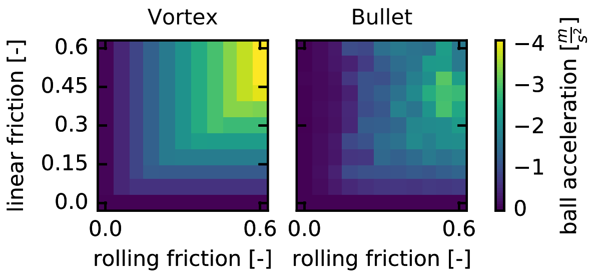
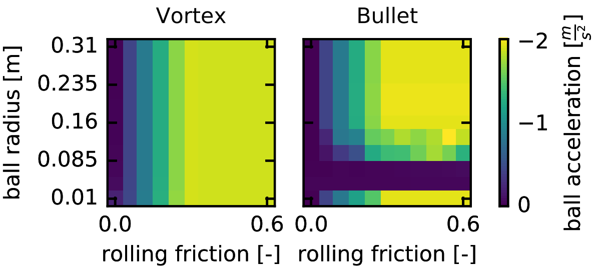


Figure 5.7: Deceleration of the ball on a level plate, depending on a single perturbed phyiscs parameter. While the results for Vortex are very smooth, Bullet produces some very erratic measurements, especially for the varying rolling friction.

5.2.3 Deceleration on Level Plate



(a) Deceleration of the ball on a level plate, depending on linear friction and rolling friction. Vortex produces a clearly structured result shape, while the values from Bullet are rather chaotic.



(b) Deceleration of the ball on a level plate, depending on ball radius and rolling friction. While the results are similar for most parameter combinations, Bullet has a huge anomaly for ball radii between 0.04 m and 0.13 m.

Figure 5.8: Deceleration of the ball on a level plate, depending on combined physics parameters.

In Figure 5.7, we can see the negative acceleration slowing the ball down on a level plate. The acceleration was computed in the same way as for the last experiment, but only of the part where the ball is still moving. Again, the center of mass was not included into this test since it would violate the assumption of constant acceleration. For Vortex, we see that the linear friction appears to have no effect on the acceleration, except for a value of 0 which results in the ball not decelerating at all. Only the rolling friction has visible influence in the investigated range. The results for Bullet are extremely erratic. While the general trends are similar to the results from Vortex, it is not apparent what caused this chaotic behaviour.

Looking at the interaction of linear and rolling friction in Vortex plotted in Figure 5.8a, we see that the border between dominant linear and rolling friction is running closer to the rolling friction side. For a large rolling friction coefficient, the friction coefficient matters less even though it is still a bit smaller. The reason for this observation lies in the fact that the rolling friction can only stop the angular motion. The linear motion is stopped due to the linear friction between the plate and the slower rollig ball surface. However, the angular motion is not cancelled immediately, so the relative linear velocity at the contact point is lower than the relative velocity of the objects, thus only requiring a smaller friction coefficient to stop. While the same general trend can be observed in Bullet, the values are too erratic to be interpreted in detail.

Since the ball radius had a large influence on the results of Section 5.2.2, it is reasonable to assume a similar effect on the deceleration. The interations between radius and rolling friction coefficient are visualized in Figure 5.8b. Here, the irregularities in Bullet are even more prominent. The measurements for radii of 0.07 m and 0.04 m show no deceleration at all, no matter the friction coefficient. For the slightly larger radii of 0.1 m and 0.13 m, the values are very uneven. Interestingly, for a radius of 0.01 m, the results are as expected, just as we saw in Figure 5.6b. Unfortunately, our nominal value for the ball radius of 0.1 m falls into this range of problematic values.

5.2.4 Analyzing Outliers

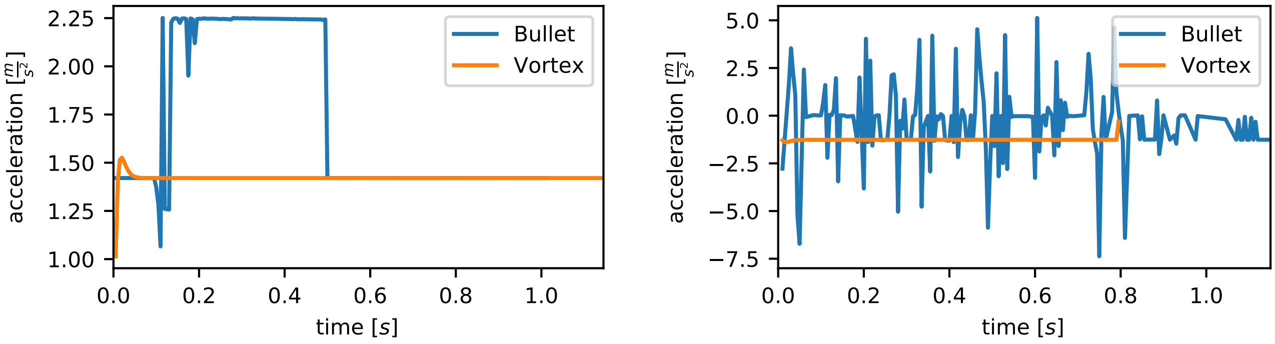
We have seen that Vortex handles the testcases a lot better than Bullet does. In some cases, the results for Bullet are dramatically different. However, they are not random, since rerunning the experiments always yields the same results, even when changing random seeds. In order to analyze the issues Bullet has, we decided to look at the trajectories for physics parameter sets that produce notable outliers in Bullet.

Figure 5.9a shows the trajectory of an outlier from Section 5.2.2. Using Vortex, we get a slight but smooth irregularity right at the start, but afterwards the acceleration is constant as expected at $\approx 1.42\,\frac{\mathrm{m}}{\mathrm{s}^2}$. With Bullet, the trajectory does not meet this expectation. There are several outliers, most of which have a value of $\approx 2.25\,\frac{\mathrm{m}}{\mathrm{s}^2}$. Coincidentally, this value is very close to the acceleration we get when the rolling friction coefficient is set to 0, so it seems that the ball is slipping in the area of these outliers. The slipping is likely caused by inaccuracies of the Bullet engine, since Vortex works with the same parameter set. However, it might also be a bug in the interface code between Rcs and Bullet.

Figure 5.9b shows the trajectory of an outlier from Section 5.2.3. Again, the trajectory from Vortex is constant as expected, the little nook at the last data point is caused by the ball coming to a complete stop. The trajectory from Bullet is even more erratic here. It is repeatedly switching between $\approx -1.25\,\frac{\mathrm{m}}{\mathrm{s}^2}$, which is the same value we get with Vortex, and $0\,\frac{\mathrm{m}}{\mathrm{s}^2}$, which is the value we get for a rolling friction coefficient of 0.

In conclusion, we find that the outliers in Bullet are likely caused by the ball starting to slip or bounce. The Bullet wiki [1] states that moving object sizes should be in a range between 0.05 m and 10 m. If that is not possible, the whole simulation should be scaled, changing the units of measurement. Unfortunately, Rcs does not support changing the scale of the simulation. Also, the anomalies occur even with larger objects. We see them up to a radius of 0.25 m in Figure 5.6b or 0.13 m in Figure 5.8b. Those anomalies does make it very hard to find a scaling that would be guaranteed to work.

Because of the irregular behaviour of Bullet, all further experiments were conducted using the Vortex physics engine only.



(a) Acceleration on tilted plate, using a ball radius of 0.02 m and a rolling friction coefficient of 0.16.

(b) Deceleration on level plate, using a linear friction coefficient of 0.45 and a rolling friction coefficient of 0.18.

Figure 5.9: Trajectory comparison on outliers from Bullet. In both setups, the value from bullet is switching between two different values. One of those values is the same as measured from Vortex, the other one turns out to be the value we get for a rolling friction coefficient of 0.

[1] `http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Scaling_The_World`, 25.05.2018

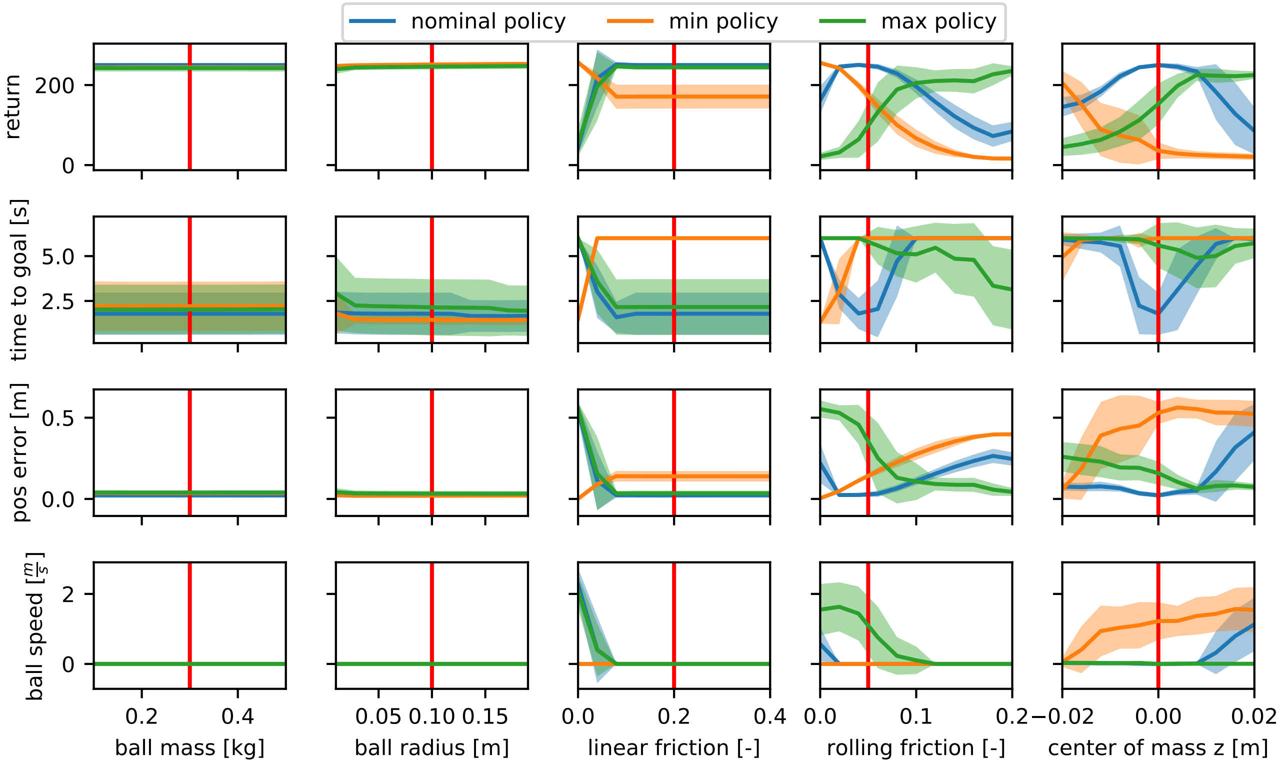


Figure 5.10: Robustness of the baseline policy. The lines show the mean performance and the area contains the mean performance plus/minus the standard deviation. Note that the time to goal is set to to the maximum time of 6 s if the rollout did not reach the goal. The min and max policies were trained for each parameter seperately, the min policy with the minimum and the max policy with the maximum from the respective parameter value axis. The ball mass and radius parameters have no influence. For friction and rolling friction, each policy is best at the value it trained for. If the coefficient is larger, the goal is not reached, if it is smaller, the policy fails to reach a stable state. The effects of the center of mass are similar, but here, higher values cause an unstable final state.

5.3 Robustness Analysis

5.3.1 Robustness of Baseline Policy

Before we measure the robustness of policies trained with EPOpt or RARL, we should first find out how robust the baseline policy is by itself, so we tested the baseline policy on varying parameters. We look at different features of the rollout. Again, each parameter is varied individually. Every parameter set is executed using 30 different initial ball positions, which are equally spaced on a circle around the plate center. We also compared the results for each parameter with two policies trained on the minimum and maximum parameter values respectively. These extreme cases should be better than the nominal policy while close to their trained parameter value, but get worse more in the middle.

In Figure 5.10, we have four rows of plots showing the result. The top row contains the accumulated reward over the whole rollout. However, the reward is calculated from many sources and therefore not a measure of robustness. Thus, the second row shows the time it took the policy to reach a step reward larger than 0.95 and stay at it. The third row shows the the distance between the ball and the target position in the final timestep, and the last row contains the linear velocity of the ball in the final timestep. All curves show the mean and the standard deviation over 30 rollouts with random initial ball positions.

The ball's mass and radius parameters have almost no influence at all. This observation is consistent with the observation from Section 5.2, and means that the nominal policy is robust to changes of ball mass and radius. The friction coefficient only matters while lower than the rolling friction coefficient, so there is no significant difference between the policies trained with nominal and maximum friction coefficient values. However, they both fail for smaller friction values. In this area, the policy trained with a friction coefficient of 0 is significantly better. These observations show that the baseline policy is not robust to changes of the friction coefficient.

For the rolling friction coefficient, the effect is even more visible. At the lower end of the scale, the policy trained with a cofficient of 0 is the only one succeeding. Around the nominal value, the nominal policy works best. For higher values, the policy trained with a rolling friction coefficient of 0.2 is best. The other policies stop reaching the goal when leaving the area. We conclude from those observations that the baseline policy is not robust to changes of the rolling friction coefficient. It is notable that the baseline policy's performance is deteriorating a lot faster for values higher than the nominal policy. Also, the max policy performs well over a larger range of values, though only getting within $\approx 0.15\,\text{m}$ of the objective. Looking at the plot of the final velocity, we see that all policies can bring the velocity to 0 for high values, but they fail to do so for smaller values. This behaviour is not surprising since a high friction helps with stopping the ball. For the center of mass, we obtained an interesting result. We only varied the z coordinate. If the value is negative, the center of mass is closer to the plate, so the ball will only start moving at a higher plate angle and tend to stay stable. The nominal policy starts declining at values lower than $-0.01\,\text{m}$, but still gets with $\approx 0.1\,\text{m}$ of the goal. The min policy trained at $-0.02\,\text{m}$ is only good right at it's known value and declines quickly for higher values.If the center of mass z value is positive, the ball's initial state is more unstable. Once it's angle has changed a slight bit, it will immediately roll until the center of mass is at the bottom even on a level plate. Interestingly, the nominal policy is still working well up to a value of $0.01\,\text{m}$. For higher values, it fails to reach the goal. The maximum policy trained at $0.02\,\text{m}$ has an almost constant performance for values larger than 0.01, but for smaller values it doesn't reach the goal. The plot of the time to goal also shows that the nominal policy takes a lot longer to reach the goal when further from the nominal value. We can also see that the max policy manages to stop the ball for all parameter values, the nominal policy only fails to do so for values larger than $0.01\,\text{m}$, and the min policy only manages it for the value it was trained at.

5.3.2 Robustness of EPOpt

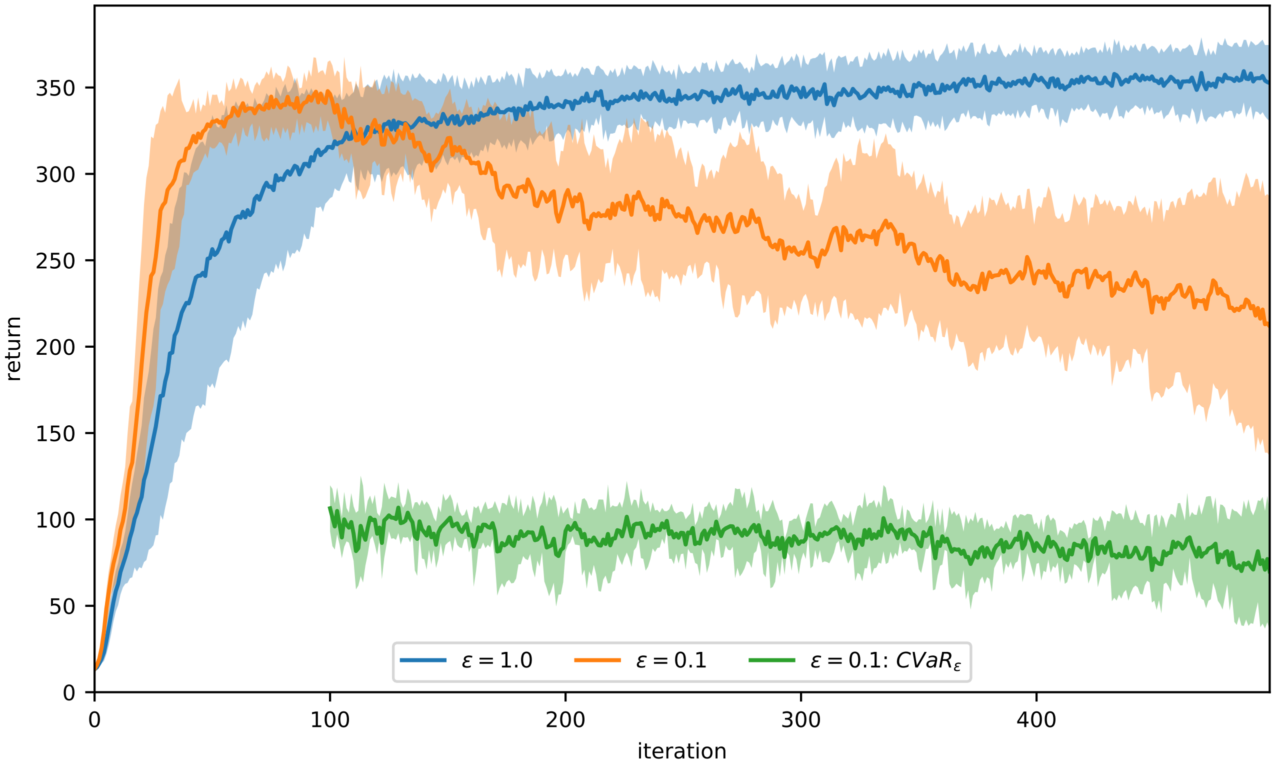


Figure 5.11: Training curve of EPOpt with different ϵ. The lines show the mean return over 5 training runs, the area contains the mean plus/minus the standard deviation. The policy training with $\text{CVaR}_{0.1}(\mathcal{T})$ is losing performance as the training progresses.

The training curve for EPOpt can be seen in Figure 5.11. We performed 5 training runs using different random seeds. As one can see, EPOpt trains faster. The faster training is likely caused by the higher sample count per batch. Since we used 500 time steps per rollout istead of 300, the return the algorithm converged at is higher. However, a converged return of 350 is only 70% of the absolute maximum return achievable if the step reward is always 1, whereas the baseline policy reaches 83%. For the version optimizing $\text{CVaR}_{0.1}(\mathcal{T})$, the mean return starts falling after the bootstrap phase of 100 iterations is over. The value of the objective function $\text{CVaR}_{0.1}(\mathcal{T})$ stays almost the same during the rest of the training run, but get a higher variance in the end.

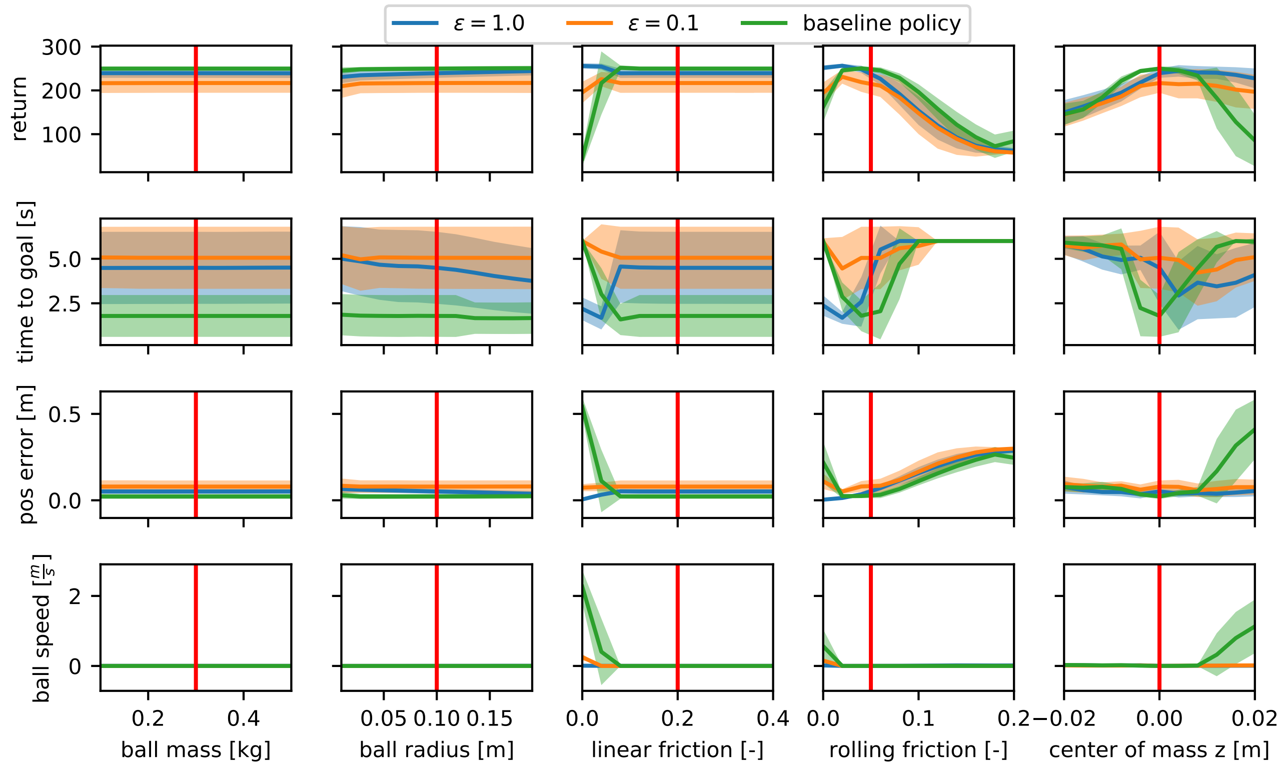


Figure 5.12: Robustness of policy trained with EPOpt, compared with the baseline policy. The lines show the mean over different initial positions, the area contains the mean plus/minus the standard deviation. In general EPOpt performs a lot better than the baseline for small friction coefficients and high center of mass values, and is slightly worse otherwise.

To analyze the robustness of the resulting policy, we plotted the reward depending on the physics parameter in Figure 5.12. We also added the baseline policy's performance for comparison. The baseline policy performs best when using the nominal parameters, which is to be expected since it did train to fit this specific configuration. The ball mass has no influence. Interestingly, the ball radius does have some influence on the performance of the policy trained with $\epsilon = 1$, which is completing slightly faster for higher ball radii. However, it does not have any effect on the final position.

The policies trained with EPOpt are a lot more robust against changes of the linear friction coefficient. The variant with $\epsilon = 1$ always manages to get within $\approx 0.05\,\text{m}$ of the objective, and is even better for a friction coefficient of 0. With the rolling friction coefficient, the EPOpt policies are only more robust for values smaller than the nominal value, but the baseline is better for coefficients larger than the nominal value. However, all three variants terminate further from the target the higher the rolling friction coefficient gets, and the baseline policy only gets 0.05 m closer. Regarding the center of mass, the policies trained with EPOpt manage to get within $\approx 0.1\,\text{m}$ of the goal and stop the ball for all parameter values, even where the baseline fails. EPOpt is completely robust against these parameter changes. It is notable that the EPOpt policies almost always manage to reach a final ball velocity of 0.

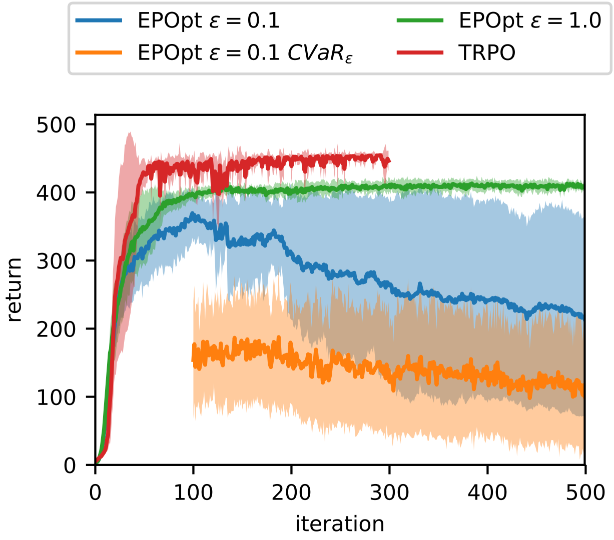
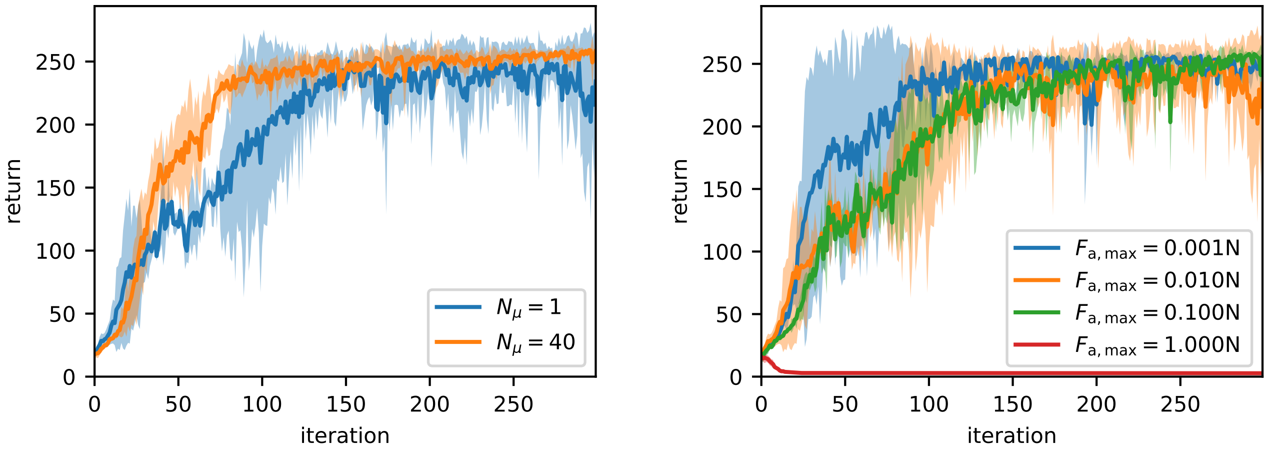


Figure 5.13: Training curve of EPOpt and TRPO with fixed initial state. The lines show the mean return over 5 training runs, the area contains the mean plus/minus the standard deviation. The TRPO policy was only trained for 300 iteration since it converges faster. There is no visible difference to Figure 5.11

In general, it seems that the policy trained with $\epsilon = 0.1$ is worse than the one trained with $\epsilon = 1.0$. This observation is suprising since the use of the CVaR objective was supposed to make the policy more robust. Figure 5.11 also showed that the algorithm didn't manage to improve the CVaR significantly. An indication of the issue is given by the higher variance in the returns for $\epsilon = 0.1$. The increased variance indicates that the policy performs better for some initial ball positions and worse for others. The baseline and the policy trained with $\epsilon = 1.0$ both have a lot less variance. In fact, we have two sources of randomness in these training runs. One is the physics parameter perturbation, the other one is the varying initial state. Using the CVaR subsampling defined in Equation (2.4), we do not only get the trajectories with the most inconvenient physics parameters, but also with the most inconvenient starting positions, which might all be located in one area of the state space. This sample selection might impede the exploration of the state space. Notably, all of the tasks EPOpt was evaluated in the original paper [9] used a fixed initial state.

In order to check whether the initial state randomization is the source of the issues, we trained both EPOpt and TRPO using a fixed initial state. The resulting training curves can be seen in Figure 5.13. Here, the policy training with $\epsilon = 0.1$ reaches a much higher final performance. However, the CVaR is still falling with progressing iterations, so the initial state randomization was not the cause.

5.3.3 Robustness of RARL



(a) Training curve for different N_μ.

(b) Training curve for different $F_{\text{a,max}}$

Figure 5.14: Training curve of RARL with different hyperparameters. The lines show the mean return over 5 training runs, the area contains the mean plus/minus the standard deviation. The setup using $F_{\text{a,max}} = 1\,\text{N}$ failed to learn any solution, while the others all reach a similar average return. The policy using $F_{\text{a,max}} = 0.001\,\text{N}$ trains fastest. Using $N_\mu = 40$ also increases the training speed.

For RARL, we analyzed two hyperparameters. First of all, we look at the iteration structure, while keeping $F_{\text{a,max}} = 0.01\,\text{N}$. In Figure 5.14a, we compare training runs using $N_\mu = N_\nu = 40$, $N_{\text{iter}} = 10$ with training runs using $N_\mu = N_\nu = 1$ and $N_{\text{iter}} = 300$. The training using $N_\mu = 40$ reaches a higher average return, which is not surprising since it does have multiple optimization iterations during which the adversary remains constant. Secondly, we looked at different values for the maximum adversary force $F_{\text{a,max}}$, while using use $N_\mu = N_\nu = 1$. The results can be seen in Figure 5.14b. The variant with $F_{\text{a,max}} = 0.001\,\text{N}$ converged the fastest, $F_{\text{a,max}} = 0.01\,\text{N}$ and $F_{\text{a,max}} = 0.1\,\text{N}$ are very similar. The variant using $F_{\text{a,max}} = 1\,\text{N}$ did not learn a solution. This failure indicates that the adversary was too strong, so the protagonist was unable to learn a countermeasure.

In Figure 5.15, we show the influence of single perturbed physics parameters on the policies trained with different hyperparameters. In general, the variant with $F_{\text{a,max}} = 0.01\,\text{N}$ performs best. The other two variants generally have a higher final position error and ball speed. The latter is about $0.5\,\frac{\text{m}}{\text{s}}$ at the nominal parameters, so those two variants actually did not manage to solve the task at all. For the iteration count, we see that the variant using $N_\mu = 40$ reaches about the same return as it's counterpart trained with $N_\mu = 1$. Near the nominal values, the $N_\mu = 40$ variant does manage to get a bit closer to the goal. However, it generally performs wose when further away from the area, so the policy trained with $N_\mu = 1$ is slightly more robust.

Again, the ball mass has no influence. The ball radius does influence the policies trained with RARL. For larger radii, the policy terminates further away from the goal position. The policy trained with $F_{\text{a,max}} = 0.001\,\text{N}$ only gets within 0.2 m of the goal for a ball radius of 0.19 m. The linear friction coefficient only matters for values smaller than the nominal rolling friction. The baseline is more robust than the RARL policies here. For the rolling friction coefficent, we see that the RARL policies are more robust for values larger than the nominal value. Even at the worst datapoint, the policy trained with $F_{\text{a,max}} = 0.01\,\text{N}$ gets within 0.1 m of the goal position, while the error for the baseline is more than double that value. However, for values lower than the nominal value, all RARL policies fail getting close to the goal position, making the baseline policy significantly more robust in these areas. For the center of mass, the RARL policies reach a much higher reward for negative values. The policy trained with $F_{\text{a,max}} = 0.01\,\text{N}$ even gets a bit closer to the goal even for very low values, but most of the higher reward can be attributed to the goal being reached a lot faster. However, the RARL policies fail to reach the goal for positive values, the policies trained with $F_{\text{a,max}} = 0.1\,\text{N}$ and $F_{\text{a,max}} = 0.001\,\text{N}$ even fail for the nominal value. In conclusion, we see that the policy trained with $F_{\text{a,max}} = 0.01\,\text{N}$ performed best of all the RARL policies.

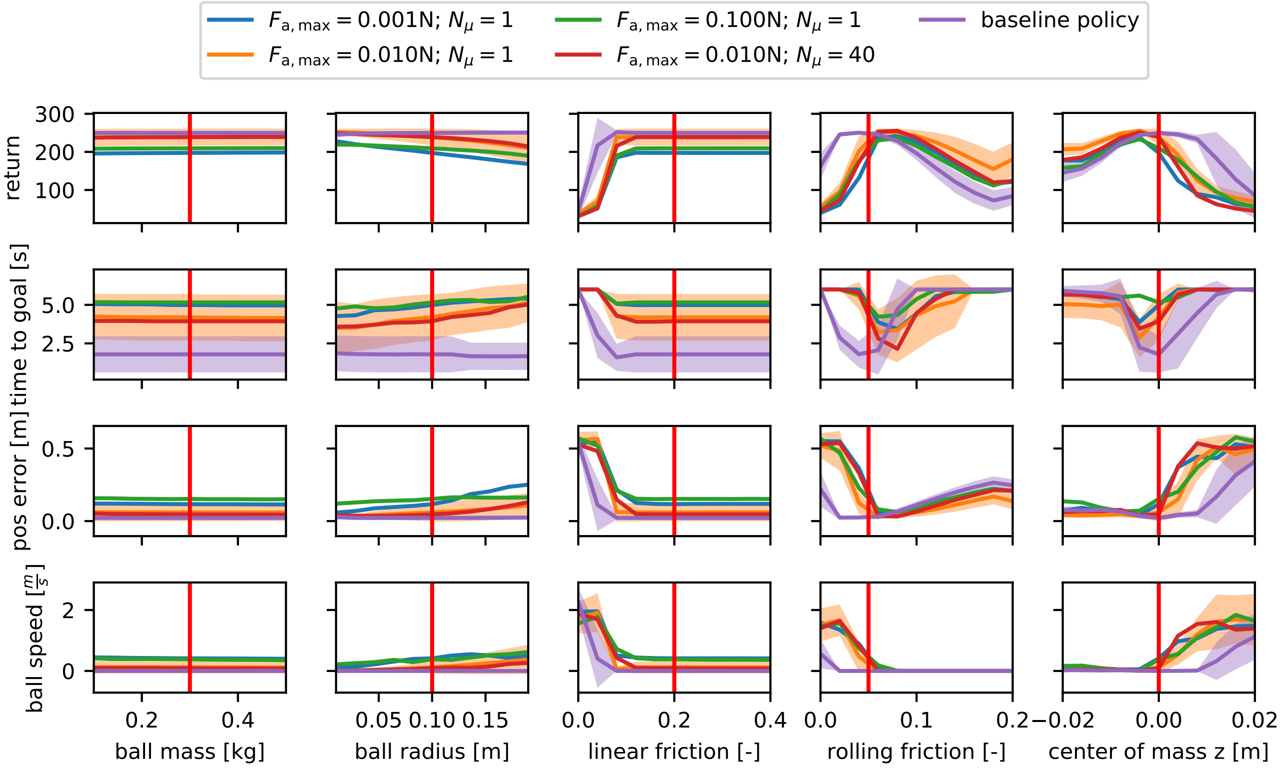


Figure 5.15: Robustness of policy trained with RARL, compared with the baseline policy. The lines show the mean performance. For the baseline policy and the RARL policy with a max adversary force of $0.01\,\text{N}$, the area contains the mean performance plus/minus the standard deviation. The other standard deviations were omitted for readability. The policy trained with $F_{\text{a,max}} = 0.01\,\text{N}$ and $N_\mu = 1$ performs best.

In Figure 5.16, we see the training curve from the transition noise setup. It is notable that the standard deviation over the 5 runs with different seeds is very low, even though this setup is using 10 random noise values per timestep, which is a lot more than EPOpt with 5 perturbed physics parameters per trajectory.
The behaviour of the trained policy on single changing parameters is shown in Figure 5.17. The peak performance is lower than that of the baseline policy. We can see that all parameters have some measure of influence, even the ball mass which did not show any influence on the other parameters. However, except for small values for the linear friction coefficient, the transition noise policy almost always produces a return of about 200, which would make it the most robust solution. Unfortunately, the time to goal is always very close to the maximum time of 6 s. And while the solution does get within 0.1 m of the goal position in the mean, the mean ball speed at the final step ranges from 0.2 $\frac{m}{s}$ to 0.5 $\frac{m}{s}$. So, the policy does not actually manage to balance the ball into a stable position and cannot be considered a solution of the task.

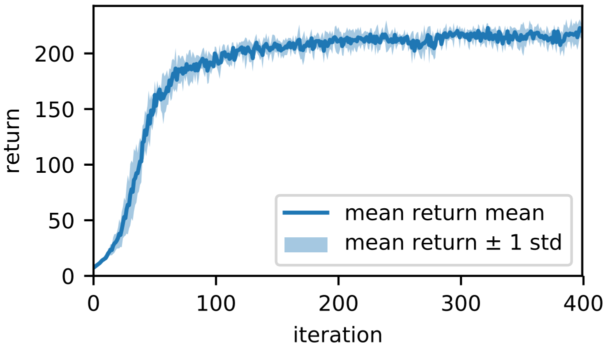


Figure 5.16: Training curve for general transition noise. The variance over multiple training runs is notably low.

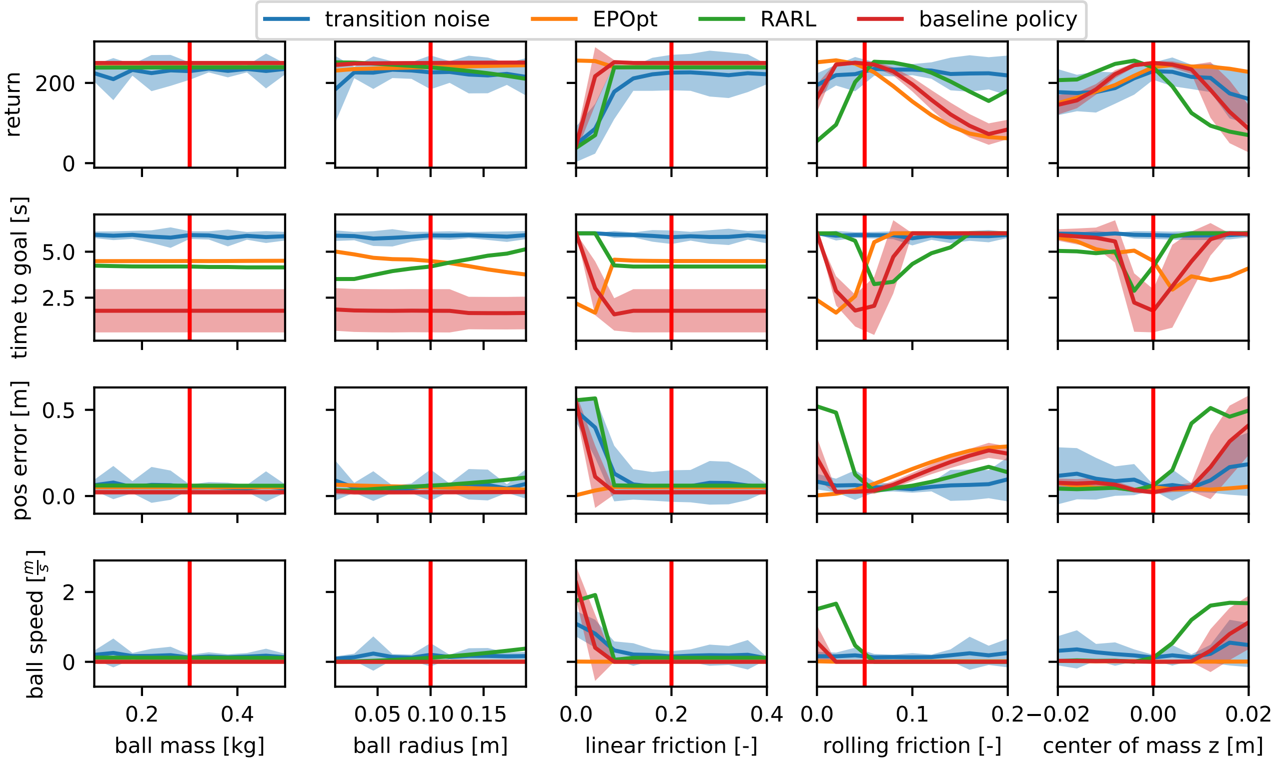
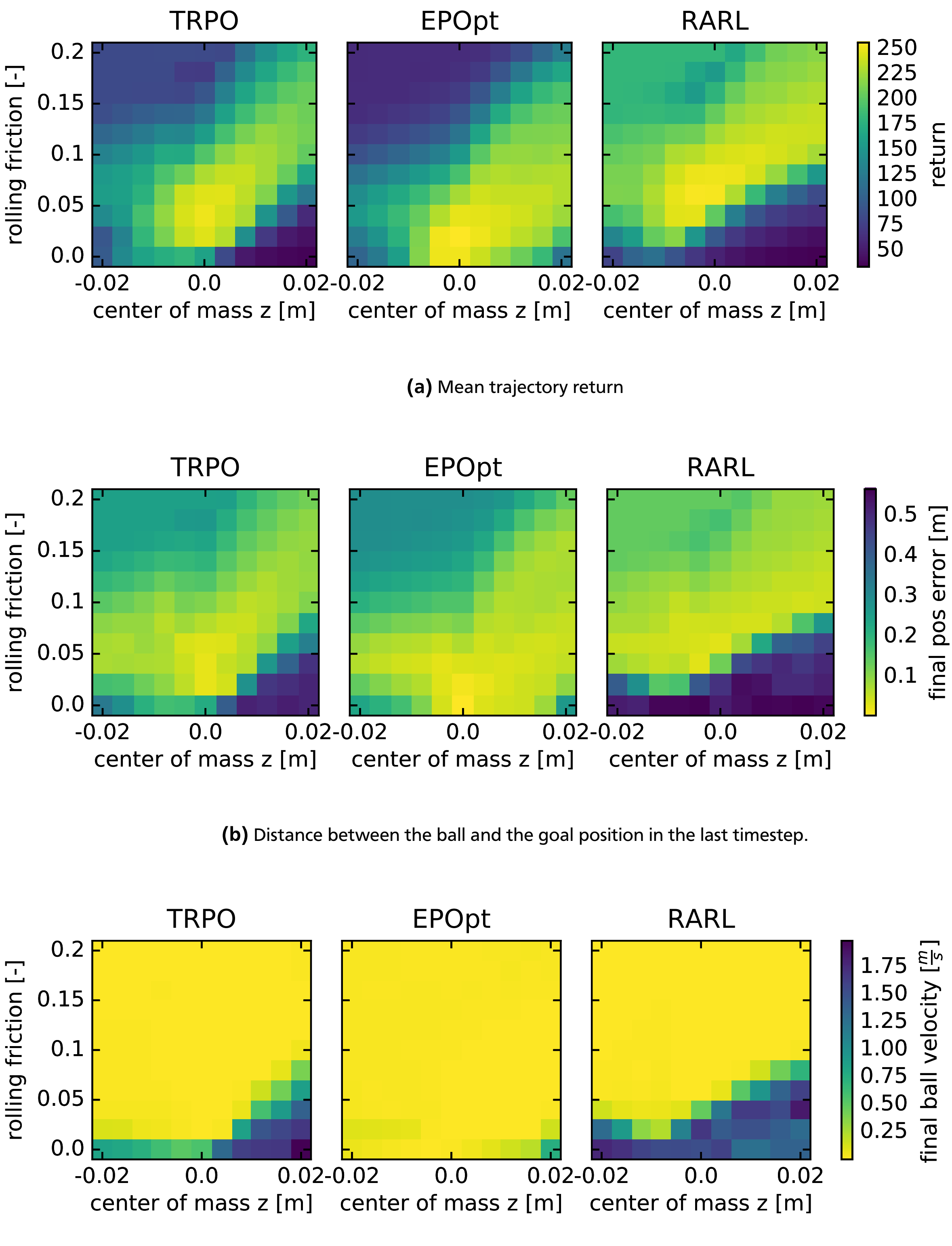


Figure 5.17: Robustness of policy trained with transition noise, compared with the baseline policy. The lines show the mean over different initial positions, the area contains the mean plus/minus the standard deviation. The mean performance of the best EPOpt and RARL policies have been added for reference. We observe that the transition noise policy always has a final step velocity of 0.2 $\frac{m}{s}$ or more, which means that it doesn't solve the task since it doesn't end on a stable state.



(a) Mean trajectory return

(b) Distance between the ball and the goal position in the last timestep.

(c) Linear velocity of the ball in the last timestep.

Figure 5.18: Comparison of all policies depending on rolling friction and ball center of mass. All other physics parameters are set to their nominal values. The displayed values are the mean over 30 initial ball positions. EPOpt is best for high center of mass positions and low friction values, RARL is best for the opposite. Also, the EPOpt policy is the only one that manages to terminate on a stable state for almost all parameter configurations.

5.3.5 Comparing all Algorithms

The experiments have shown that the rolling friction coefficient and the center of mass are the most influential physics parameters. To further investigate the interactions between these parameters, we plotted the return of the policies trained with EPOpt and RARL as well as the baseline policy next to each other. The results can be seen in Figure 5.18. In general, the highest is found along a diagonal running through the nominal params ro the upper right. This behaviour is reasonable since a center of mass in a higher z-position makes the system more unstable, and a higher rolling friction coefficient can counter that. The bottom right has the highest instability with a center of mass position above the ball's center and no rolling friction. The top left has a center of mass position below the ball's center and high friction and is therefore the most stable. We also see that EPOpt has the best solution for the lower right, where the system is the most unstable, but is less performant than the baseline in the upper left, where the system is more stable. RARL on the other hand is the best in the upper left stable parameter area, but quickly fails in the unstable parameter area. The policy learned by EPOpt is also the only one that terminates on a non-moving ball for the whole parameter area, even for badly supported parameter values, whereas RARL fails for it's unsupported parameter values, but this observation likely results from the general behaviour caused by those parameter values and is not a property of the algorithms themselves.

	EPOpt	RARL
A higher rolling friction coefficent leads to	stopping before the goal is reached	getting close to the goal
A lower rolling friction coefficent leads to	success	failure
A higher center of mass position leads to	getting close to the goal	failure
A lower center of mass position leads to	stopping before the goal is reached	getting close to the goal
Is robust for systems that are	more unstable	more stable

Table 5.1: Summary of the behaviour of policies trained by RARL and EPOpt under perturbed physics parameters.

6 Discussion & Outlook

We analyzed the randomized physics parameters in simulation and their influence on the task. It turns out that the mass of the ball has no influence at all, and the radius has almost none. In contrast, the friction coefficients and the mass distribution have a high influence.

We also used the physics parameter analysis to compare the two physics simulators useable with Rcs. Both simulators showed a similar reaction to the same parameters. However, there were significant irregularities in the results from the Bullet simulator. It turns out that our nominal value for the ball radius was close to the lowest value supported as size in Bullet. The documentation suggest rescaling the entire simulation to avoid this issue. However, artifacts also appeared for values that were more then 4 times larger then the lower border. However, we cannot completely rule out that the irregularities are caused by an error in the interface from Rcs to Bullet. Vortex performed did not have any artifacts. The Vortex documentation only lists limits for the ratio between the smallest and the largest object, which suggests that can it adjust it's internal scaling automatically.

By comparing the two robust algorithms with the baseline, we found out that both EPOpt and RARL only improve the robustness in specific areas. Also, the baseline policy is already pretty robust on it's own. EPOpt increases robustness against lower values of rolling friction and a center of mass above the ball's center. This observation makes sense since these two parameters make the physics environment more unstable. EPOpt, having trained with a multitude of physics parameters, is more cautious, and can handle these unstable cases better. However, if the ball is less inclined to move due to a higher rolling friction or a center of mass below the ball's center, it fails since it is too cautious to get the ball moving at all.

RARL increases robustness against higher values of the rolling friction coefficient and negative values for the center of mass position. This bahaviour is not surprising, since increased rolling friction makes the ball harder to move, and so does a lower center of mass. The adversary learned by RARL tries to hinder the ball from reaching the target by applying a force in the opposite direction, and this disturbance is seen by the protagonist policy as a similar effect. For a lower rolling friction, the ball will roll a lot quicker, and the policy can't deal with it as easily since it doesn't expect it. The same happens for a higher center of mass.

The policy trained with transition noise does produce a stable return on changed physics parameters, but it doesn't manage to bring the ball into a stable state, so it cannot be considered a solution. The reason for this failure is that while the environment is manipulated in those algorithms, the basic functionality remains the same. A ball on a level plate that is not moving will stay that way, and both EPOpt and RARL learned that. RARL is slightly worse in it since it is used to dealing with the adversary, but since the adversary's behaviour follows a system, it can adapt. Even if the standard deviation of the transition noise is reduced significantly, it will still not be able to model a completely stable state. Because of this instability, the perturbations applied by RARL and EPOpt are clearly superiour to the generic transition noise approach.

So in conclusion, both EPOpt and RARL do improve the robustness, but none of them is capable of solving the task in a generic way. EPOpt is more cautious and can better deal with an unstable environment, and RARL is more proactive and can better handle an overdampened environment. These observations do raise the question whether there is a generally robust policy at all. That policy would have to somehow detect the grade of instability in the environment. It seems that our feed-forward neural network policy is not capable of this detection. Peng et.al. successfully used a recurrent network to bridge the reality gap [15], so that could be a way to find a more generic solution.

The remaining question is which policy would transfer best to the real robot. For the ball-on-plate task, the solution learned by EPOpt should be preferred, since it is more stable then RARL and prefers to take a careful and slow approach. Since EPOpt doesn't work well with friction values higher then the nominal parameters, choosing their values to be higher then the measured mean would likely increase the robustness over the whole parameter space.

Bibliography

[1] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, pp. 1889–1897, 2015.

[2] J. Peters, K. Mülling, and Y. Altun, "Relative entropy policy search.," in *AAAI*, pp. 1607–1612, Atlanta, 2010.

[3] A. Yahya, A. Li, M. Kalakrishnan, Y. Chebotar, and S. Levine, "Collective robot reinforcement learning with distributed asynchronous guided policy search," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 79–86, IEEE, 2017.

[4] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, "Learning hand-eye coordination for robotic grasping with large-scale data collection," in *International Symposium on Experimental Robotics*, pp. 173–184, Springer, 2016.

[5] I. Clavera, D. Held, and P. Abbeel, "Policy transfer via modularity and reward guiding," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 1537–1544, IEEE, 2017.

[6] K. Chatzilygeroudis, R. Rama, R. Kaushik, D. Goepp, V. Vassiliades, and J.-B. Mouret, "Black-box data-efficient policy search for robotics," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 51–58, IEEE, 2017.

[7] M. Saveriano, Y. Yin, P. Falco, and D. Lee, "Data-efficient control policy search using residual dynamics learning," in *Proc. of IROS*, 2017.

[8] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *European Conference on Artificial Life*, pp. 704–720, Springer, 1995.

[9] A. Rajeswaran, S. Ghotra, B. Ravindran, and S. Levine, "Epopt: Learning robust neural network policies using model ensembles," *arXiv preprint arXiv:1610.01283*, 2016.

[10] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," *arXiv preprint arXiv:1703.02702*, 2017.

[11] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, pp. 1329–1338, 2016.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[13] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of bullet," *Havok, MuJoCo, ODE and PhysX, Washington Univ*, 2015.

[14] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)," *Evolutionary computation*, vol. 11, no. 1, pp. 1–18, 2003.

[15] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," *arXiv preprint arXiv:1710.06537*, 2017.