Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Learning Robotic Bimanual Cloth Manipulation

vorgelegt von

Daniel Brühl

MIN-Fakultät

Fachbereich Informatik

Studiengang: Master Informatik

Matrikelnummer: 7516316

Abgabedatum: 23.03.2026

Erstgutachter: Prof. Dr. Jianwei Zhang

Zweitgutachter: Dr. Norman Hendrich

Betreuer: M. Sc. Niklas Fiedler

Abstract

This thesis extends dynamic cloth folding from a single-arm Franka Panda setup to a bimanual Diana7 robot configuration implemented in both MuJoCo and PyBullet with a shared task interface. Both backends share the same observation and action interface and follow the same BioIK-based control path. Cross-engine transfer is possible, but strongly asymmetric. Three of the four MuJoCo-trained policies transfer to PyBullet with a success rate of 1.0, while none of the PyBullet-trained policies transfer back to MuJoCo successfully. The most likely reason is the difference between the underlying deformable-body models. Domain randomization improves robustness within an engine, but it doesn't reliably bridge the sim-to-sim gap. Across unseen cloth sizes and friction settings, randomized policies tend to show more consistent geometric errors. The `ee` robot observation mode includes the current end effector (EE) position, EE velocity and the current controller target. It is more reliable than `ctrl`, which contains the previous raw action.

Zusammenfassung

Diese Arbeit erweitert das dynamische Falten von Stoffen von einem Einarm-Setup mit dem Franka Panda auf eine bimanuale Diana7-Roboterkonfiguration, die in MuJoCo und PyBullet mit einer gemeinsamen Aufgaben-Schnittstelle implementiert ist. Beide Backends teilen sich dieselbe Beobachtungs- und Aktionsschnittstelle und folgen demselben BioIK-basierten Kontrollpfad. Der Cross-Engine-Transfer ist möglich, aber stark asymmetrisch. Drei der vier in MuJoCo trainierten Policies erreichen in PyBullet eine Erfolgsrate von 1,0, während keine der in PyBullet trainierten Policies erfolgreich nach MuJoCo übertragen wird. Die wahrscheinlichste Ursache ist der Unterschied zwischen den zugrunde liegenden Modellen für deformierbare Körper. Domain Randomization verbessert die Robustheit innerhalb einer Engine, überbrückt die Sim-to-Sim-Lücke aber nicht zuverlässig. Über unbekannte Stoffgrößen und Reibungswerte hinweg zeigen randomisierte Policies tendenziell konsistentere geometrische Fehler. Der Roboterbeobachtungsmodus `ee` enthält die aktuelle Endeffektor (EE) Position, EE-Geschwindigkeit und das aktuelle Controller-Ziel. Er ist zuverlässiger als `ctrl`, welcher nur die vorherige rohe Aktion enthält.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Problem Statement: Single-Arm to Bimanual Dynamic Folding . . . 2
- 1.3. Research Objectives and Contributions . . . 2
  - 1.3.1. Research Objectives . . . 2
  - 1.3.2. Contributions . . . 3
- 1.4. Thesis Structure . . . 3

2. Fundamentals **7**
- 2.1. Reinforcement Learning . . . 7
  - 2.1.1. Partially Observable Markov Decision Processes (POMDP) . . . 8
  - 2.1.2. Goal-conditioned Reinforcement Learning . . . 8
  - 2.1.3. Soft Actor-Critic (SAC) . . . 9
  - 2.1.4. Hindsight Experience Replay (HER) . . . 9
- 2.2. Robotics and Control for Bimanual Manipulation . . . 10
  - 2.2.1. Degrees of Freedom and 7-Joint Arms . . . 12
  - 2.2.2. Inverse Kinematics (IK) and BioIK . . . 12
  - 2.2.3. Operational Space Control (OSC) and Jacobians . . . 14
  - 2.2.4. Bimanual Coordination and Action Parameterization . . . 14
- 2.3. Physics Simulation and Deformable Objects . . . 15
  - 2.3.1. Cloth Modeling Basic . . . 16
  - 2.3.2. MuJoCo vs. PyBullet . . . 17
  - 2.3.3. Domain Randomization (DR) . . . 18
- 2.4. Vision for Visual Feedback Control . . . 19
  - 2.4.1. Camera Model and Projection . . . 19
  - 2.4.2. Observation Preprocessing . . . 21
  - 2.4.3. Image Augmentation . . . 21
  - 2.4.4. CNN Feature Learning and Auxiliary Objectives . . . 22

3. Related Work **23**
- 3.1. Overview . . . 23
- 3.2. Bimanual Cloth Manipulation of Non-Rigid Objects . . . 23
- 3.3. Sim-to-Sim and Sim-to-Real Transfer in Reinforcement Learning . . . 24
- 3.4. Dynamic cloth folding . . . 25

4. Simulation Environment: The Diana Scene **27**
4.1. System Context and Simulation Setup . . . 28
4.1.1. Original environment (reference implementation) . . . 28
4.1.2. Dual-Arm Diana Robot Configuration . . . 29
4.1.3. Workspace and Arena Definition . . . 31
4.2. Cloth / Soft-Body Modeling . . . 33
4.2.1. MuJoCo: Modeling Soft-Body Dynamics . . . 34
4.2.2. PyBullet: Deformable Body Simulation and Porting . . . 34
4.2.3. MuJoCo vs. PyBullet . . . 37
4.3. Porting the Simulation . . . 39
4.3.1. BIOIK Integration . . . 40
4.3.2. Mapping between MuJoCo Templates and PyBullet World . . . 41
4.3.3. Handling Mocap and Grasping Constraints . . . 42
4.4. Domain Randomization (DR) Strategies . . . 43

5. Approach **47**
5.1. System overview . . . 47
5.1.1. Task definition and success criteria . . . 48
5.1.2. Controller and BioIK integration . . . 48
5.2. Observation Space Design . . . 50
5.2.1. Visual Feedback: Image-based Observations . . . 51
5.2.2. Robot Proprioception: End-Effector (ee) vs. Control (ctrl) States . . . 51
5.3. Action Space Configuration . . . 52
5.3.1. Dual-Arm Independent Action Control . . . 52
5.4. Reward Engineering for Dynamic Folding . . . 53
5.4.1. Sparse vs. Dense Rewards . . . 53
5.4.2. Implementation Details . . . 53
5.5. Learning and Training Procedure . . . 54
5.5.1. SAC + HER Instantiation for Cloth Folding . . . 55
5.5.2. Network Architecture and Input Fusion . . . 56
5.5.3. Training Hyperparameters and Evaluation Schedule . . . 59

6. Evaluation **63**
6.1. Evaluation Goals . . . 63
6.2. Experimental Protocol . . . 63
6.2.1. Training Setup . . . 63
6.2.2. Evaluation During Training . . . 64
6.2.3. Cross-Engine Evaluation Protocol . . . 64
6.3. Experimental Conditions . . . 64
6.4. Metrics . . . 65
6.4.1. Success Rate . . . 65
6.4.2. Corner Sum Error . . . 65
6.4.3. Final Constraint Error . . . 66
6.4.4. Episode Return . . . 66

6.5. Results . . . 66
6.5.1. Learning curves per engine and setting . . . 66
6.5.2. In-engine evaluation . . . 68
6.5.3. Cross-engine evaluation . . . 70
6.5.4. Generalization to unseen cloth configurations . . . 72

7. Discussion **81**
7.1. In-Engine Bimanual Dynamic Folding . . . 81
7.2. Cross-Engine Transfer . . . 82
7.3. The Role of Domain Randomization . . . 82
7.4. Robot Observation Modes . . . 83
7.5. 3D vs. 6D Action Layouts . . . 84
7.6. Generalization to Unseen Cloth Configurations . . . 84
7.7. Limitations . . . 84

8. Conclusion **87**

9. Future Work **89**

Bibliography **97**

Appendix **97**
A. Appendix A: Additional Native-Engine Rollouts for Selected Checkpoints . . . 99
A.1. Selected `ctrl` checkpoints by lowest FCE . . . 99
A.2. Cross-engine evaluation of the selected `ctrl` checkpoints . . . 99
B. Appendix B: Additional Training Curves . . . 99
C. Usage of gKI . . . 99

List of Figures

1.1. Task setup in PyBullet [1] with a demo trajectory for the sideways backward task. . . . 5

2.1. Redraw of the standard reinforcement learning interaction loop after Sutton and Barto [2]. The environment provides state s_t and reward r_t. The agent responds with action a_t. . . . 8
2.2. Conceptual control pipeline for bimanual cloth folding. . . . 11
2.3. Redraw of the BioIK memetic evolutionary algorithm after Starke et al. [3]. . . . 13
2.4. Redrawn from the OpenCV "Perspective projection, from object to camera frame" [4]. . . . 20
2.5. Grayscale policy input at step 0. MuJoCo (left). PyBullet (right). . . . 21
2.6. Sampled MuJoCo DR settings. . . . 22

4.1. Redraw of the training and deployment setup proposed by Hietala et al. (Fig. 2 in [5]) . . . 28
4.2. First step evaluation of the original cloth environment: side view . . . 28
4.3. Original arena in the reconstructed PyBullet environment . . . 29
4.4. First step evaluation of the Diana cloth environment: side view . . . 30
4.5. Gripper geometry comparison: lifted between grippers . . . 30
4.6. Fold benchmark visualization over 10 runs . . . 31
4.7. First step in PyBullet of the Diana cloth environment with workspace box . . . 32
4.8. Schematic of the sideways backward fold: keypoints with $j = 0$ and $j = 8$ overlap after folding. . . . 33
4.9. Schematic of the cloth initialization. Y is stretched around an anchored corner. 34
4.10. Grasp constraint abstraction used for sim-to-sim parity. Top: MuJoCo. Bottom: PyBullet. . . . 43

5.1. System overview (adapted from the policy–controller loop in Hietala et al., Fig. 2) [5]. . . . 47
5.2. Mapping from an end-effector target p_W^{ee} (controller target) to a link-7 target p_W^{link7} (BioIK goal) using link7-to-EE offset and Rotation Matrix R_W^{link7} [6]. . . . 50
5.3. SAC+HER data flow Implemented for three environment setups [7]. . . . 56
5.4. Policy network forward pass (`ScriptPolicy`) [7, 8]. . . . 59

6.1. Learning curves for the shared baseline setting `6D ee` with $DR = 0$ and $RCS = 0$. . . . 67
6.2. Learning curves for the shared baseline setting `3D ee` with $DR = 0$ and $RCS = 0$. . . . 68

6.3. Learning curves for the shared randomized setting 3D ee with $DR = 1$ and $RCS = 1$. 69
6.4. Learning curves for the shared randomized setting 6D ee with $DR = 1$ and $RCS = 1$. 70
6.5. Final-epoch in-engine metrics for matched settings in both backends. SR(success rate). CSE(corner-sum error). FCE(final constraint error). 71
6.6. MuJoCo 3d ee 11, epoch 94. (Lowest FCE among MuJoCo configurations.) . 73
6.7. PyBullet 3d ee 11, epoch 97. 74
6.8. MuJoCo 6d ee 11, epoch 99. 75
6.9. PyBullet 6d ee 11, epoch 87. 76
6.10. Cross-engine evaluation results for the selected checkpoints. Left: success rate (SR). Middle: corner-sum error (CSE). Right: final constraint error (FCE). 77
6.11. MuJoCo 6d ee 11 in PyBullet, epoch 99. 78
6.12. PyBullet 6d ee 11 in MuJoCo, epoch 87. 79

.1. MuJoCo 3d ee 00, epoch 33. (Worst FCE among all configurations.) 101
.2. PyBullet 3d ee 00, epoch 62. 102
.3. MuJoCo 6d ee 00, epoch 65. 103
.4. PyBullet 6d ee 00, epoch 55. 104
.5. MuJoCo 3d ctrl 10, epoch 82. (Lowest FCE within the requested MuJoCo 3D `ctrl 1 0` setting.) 105
.6. PyBullet 3d ctrl 11, epoch 82. (Lowest FCE within the requested PyBullet 3D `ctrl 1 1` setting.) 106
.7. PyBullet 6d ctrl 11, epoch 94. (Lowest FCE within the requested PyBullet 6D `ctrl 1 1` setting.) 107
.8. PyBullet 3d ctrl 11 in MuJoCo, epoch 82. 108
.9. PyBullet 6d ctrl 11 in MuJoCo, epoch 94. 109
.10. Learning curves for the additional PyBullet-only setting 3D `ctrl` with $DR = 1$ and $RCS = 1$. 110

1. Introduction

1.1. Motivation

Robot manipulation for deformable objects such as towels remains a major challenge due to their flexibility and unpredictable shape and behavior. The high dimensional state space and the varying material properties of cloth makes it a challenging task to precisely control their behavior in such tasks as folding [9]. Cloth folding systems have a wide range of industrial and commercial applications. They are essential for medical and service robots, such as those used in hotels and hospitals, where autonomous cloth handling is usually required. In order to fold cloths autonomously, the robot must recognize the cloth, plan its movements and ideally, adapt its movements dynamically. This makes handling cloths an ideal test object for research into complex robot systems. Several steps and prerequisites are necessary for a usable application with robotic arms for folding cloths. The paper Folding Clothes Autonomously: A Complete Pipeline [10] breaks the process into five steps that build on one another.

1. Pickup from table
2. Classification and Pose Estimation
3. Unfolding
4. Spreading on table
5. Folding

The scope of this thesis is limited to the final step: folding. The cloth is already spread flat and held at two corners, so the focus is purely on the folding motion itself.

Figure 1.1 provides a short overview of the task addressed in this thesis. The cloth lies flat on the table and is already grasped at two corners. The task is to move the corners (two green markers) to the target positions (two red markers) without losing the folded shape.The orange markers show where the other two corners and two edge points should stay put.

A lot of cloth folding approaches use quasi-static or open-loop strategies with pre-planned motions. They don't account for how the cloth actually moves and can't correct themselves mid-fold. The cloth swings, drapes, and settles based on how fast it moves. To address this, real-time feedback is necessary from cameras or sensors. Even small changes in the velocity of arm movements can cause the cloth to land in a completely different position.

Hietala et al. trained folding policies in simulation using SAC, visual feedback and domain randomization. Their policies learned dynamic trajectories that transferred successfully to real robots [5]. This thesis adopts the dynamic, feedback-driven approach [5] to investigate it with bimanual folding.

1.2. Problem Statement: Single-Arm to Bimanual Dynamic Folding

Recent research has shown that dynamic cloth folding can be learned via visual feedback, reinforcement learning, and a single-arm setup. Extending this approach to a two-arm setup poses challenges in terms of arm control and policy learning. A two-arm setup must coordinate two end effectors that interact with the same deformable object. Various actions can influence the contact dynamics, tension and wrinkle formation of the cloth. Overall, a two-armed setup enables more complex tasks to be performed, but the complexity of executing the task also increases. Two-armed setups double the action dimension, thus increasing the complexity of planning [11] [12]. In this work, a 3-dimensional command, which consists of a position change of the end effector of one arm, is simply converted into a position change of the end effectors of two independent arms, which corresponds to a 6-dimensional command. This can destabilize the learned policy [13], making it more difficult to find an optimal action. The policies also receive image observations (optionally extended with a robot state). Positive reward and success are calculated only from the cloth positions, not from the image input.

The second problem, which also has to do with complexity, is generalizing between different simulation backends. Deformable objects in a simulation are computationally expensive. Furthermore, the learned policy is sensitive to the physics simulation used. Although physics simulators provide similar interfaces, the same task can fail without retraining. Kaup et al. [14] report that policies trained in MuJoCo [15] were able to maintain their performance in a continuous control task across different simulations, while policies from PyBullet [1] did not. The next section explains the research objectives and summarizes the contributions to solving the challenges.

1.3. Research Objectives and Contributions

1.3.1. Research Objectives

This thesis examines to what extent policies trained in MuJoCo and PyBullet maintain their performance when executed in the other engine. Visual feedback is always used and the effect of additional robot state feedback, which contains the position, velocity and desired position is evaluated. In addition, domain randomization is being investigated to determine whether it can achieve higher cross-engine robustness.

The objectives of this thesis are:

- Extend visual-feedback dynamic cloth folding from a single-arm setup to a bimanual Diana7 setup.
- Measure in-engine and cross-engine performance between MuJoCo and PyBullet.
- Evaluate the effect of domain randomization on training stability and cross-engine robustness.
- Compare robot-state observation variants (visual-only vs. visual + robot state) with respect to performance and transferability.

The specific research questions are:

- Can visual feedback policies for dynamic cloth folding be trained and executed successfully in a bimanual robot system?
- What are the advantages and challenges of bimanual folding compared to single-arm control?
- How large is the performance gap when a policy is transferred between MuJoCo and PyBullet?
- How do domain randomization and different robot-state observations affect cross-engine robustness?
- How well can the policies generalize to unknown cloth configurations?
- How do the trajectories change during cross-engine execution?

1.3.2. Contributions

This thesis makes the following contributions:

- **Bimanual Diana7 cloth-folding environment:** The original single-arm cloth-folding setup is expanded with a dual-arm Diana7 scene. This includes bimanual robot observations, the independent execution of bimanual actions and dual-arm coordination logic.
- **PyBullet port with MuJoCo-compatible API:** A PyBullet version of the Diana cloth-folding environment is implemented to mirror the MuJoCo task interface for sim-to-sim evaluation. The task definition, observation structure, and action interface are aligned closely enough that the same policy checkpoint can be evaluated in either backend.
- **Unified BioIK-based control integration:** A backend controller interface is implemented using Jacobians from either MuJoCo or PyBullet. It replaces the original OSC-based controller, introduced by Hietala et al. [5]. This allows the same high-level control logic to be applied across both engines. The low-level controller, referred to as hybrid-OSC, uses engine-provided Jacobians to compute joint torques for the desired end-effector motion. BioIK is used as a ROS (rospy) inverse-kinematics service.
- **Benchmarking protocol for sim-to-sim transfer:** Evaluation of learned policies under different robot observations, as well as in-engine and cross-engine performance.

1.4. Thesis Structure

The rest of this thesis is structured as follows.

Chapter 2 explains the concepts necessary for this work, which were mainly introduced by Hietala et al. [5]. These include reinforcement learning (including SAC and HER), a necessary background in robotics and kinematics, physics simulation of deformable objects and computer vision components used for visual feedback.

Chapter 3 reviews related work on cloth manipulation, bimanual manipulation, dynamic folding, and transfer across simulation and real-world domains. This positions the thesis with respect to prior works.

Chapter 4 explains the simulation environment in more detail in the thesis. This includes the Diana scene, dual-arm configuration and soft-body modelling in MuJoCo and PyBullet. It further describes the porting process, which serves as the basis for the implementation choices.

Chapter 5 presents the proposed approach for creating policies for the dynamic folding task. It defines the task and success criteria and specifies observation and action spaces. The chapter also explains the reward design and the execution of demos to gain insight into the folding process.

The evaluation process is described in Chapter 6. It describes the test conditions, the in-engine and cross-engine evaluation protocols, and the performance metrics. The results are documented there and compared between the different training configurations.

Chapter 7 discusses the results with regard to the research questions. Success and failure modes are analyzed, which summarizes the limitations and shows the sim-to-sim robustness of the bimanual cloth manipulation setup.

Finally, Chapter 8 concludes and classifies the thesis. Chapter 9 shows the direction future work could take.

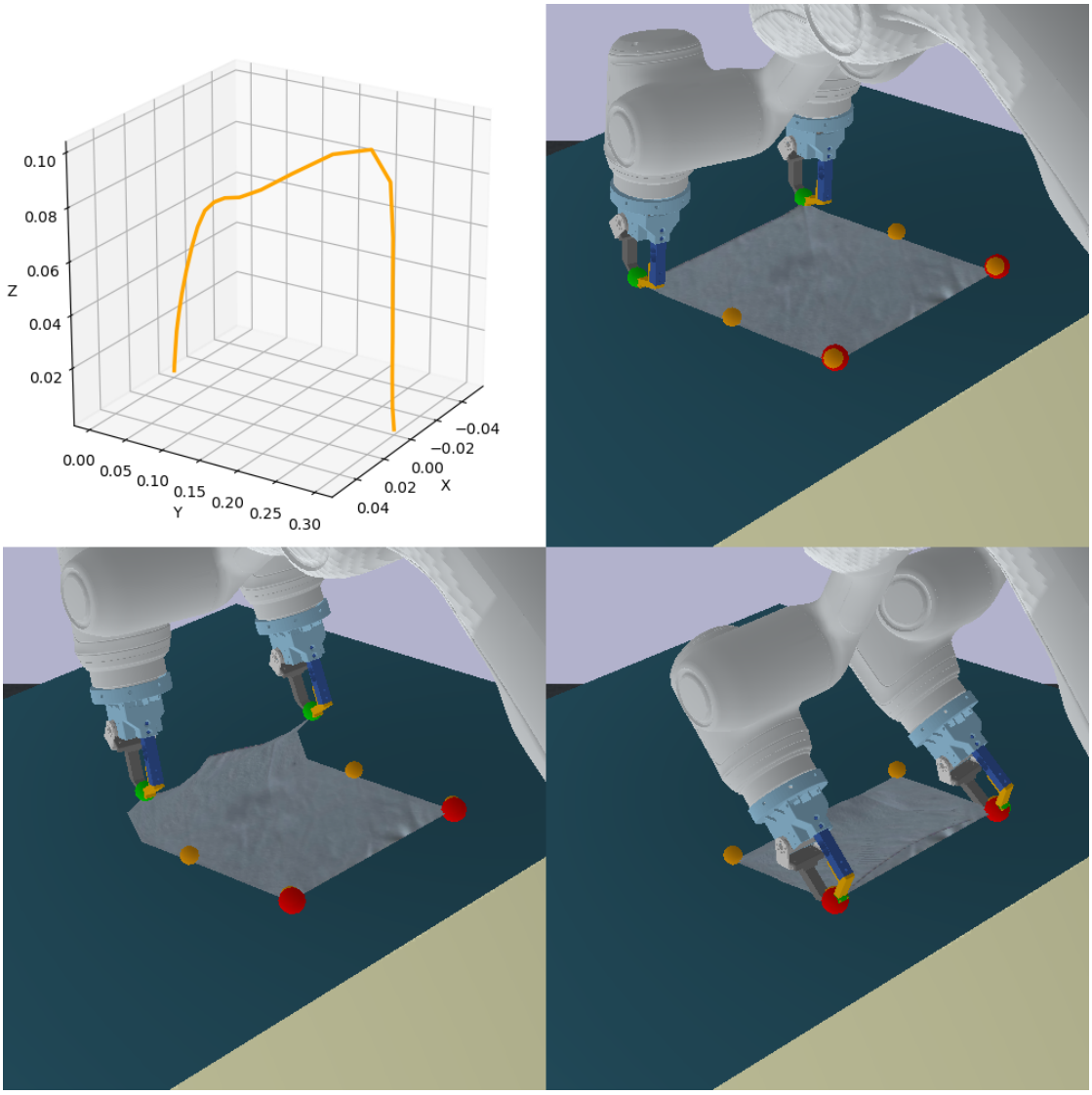


Figure 1.1.: Task setup in PyBullet [1] with a demo trajectory for the sideways backward task.

2. Fundamentals

This chapter bridges the gap between physics simulation, robotic control and machine learning. It is important in order to understand the paper "Learning Visual Feedback Control for Dynamic Cloth Folding" from Hietala et al. [5], the extension involving the Diana robot, two-arm manipulation and the PyBullet port.

2.1. Reinforcement Learning

A sequential decision-making process where an agent interacts with an environment is the fundamental idea behind reinforcement learning (RL) [2]. In each discrete step t, the agent first receives an observation o_t, which may partially reveal the state s_t, and a reward r_t as shown in Figure 2.1. The agent picks an action a_t from its policy $\pi_\theta(a_t \mid o_t)$, the environment moves to a new state s_{t+1} and a new observation o_{t+1} with reward r_{t+1} is returned.

Throughout, lowercase symbols like s_t, o_t, a_t, and r_t refer to specific values at time t.

Figure 2.1 uses the term state. In a partially observable environment, the agent instead receives an observation o_t derived from s_t.

The agent's objective is to learn a policy π_θ that maximizes the expected discounted return in Eq. (2.1) [2].

$$J(\pi_\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right], \tag{2.1}$$

where T is the maximum length of an episode (episode horizon) and $\gamma \in (0, 1]$ is the discount factor, which controls how much future rewards are worth. The Q-function Q^π measures how good an action is in an observation under policy π. It assigns an expected discounted return to each observation–action pair as defined in Eq. (2.2) [2].

$$Q^\pi(o_t, a_t) = \mathbb{E}_\pi\left[\sum_{l=0}^{T-t-1} \gamma^l r_{t+l} \mid o_t, a_t\right]. \tag{2.2}$$

RL fits robotics problems where accurate analytical models are hard to come by, deformable object manipulation being a prime example [16]. In this thesis, the environment includes the simulated robot arms, scene assets (e.g., table and obstacles), as well as the cloth dynamics. The environment also includes camera rendering and low-level torque control that executes high-level commands from the agent.

The agent performs motion commands at a low frequency, while the controller tracks the commands at a high frequency. Together, this forms a closed-loop feedback policy. The low-level controller runs at a faster rate than the policy. A single RL time step corresponds to one policy update.

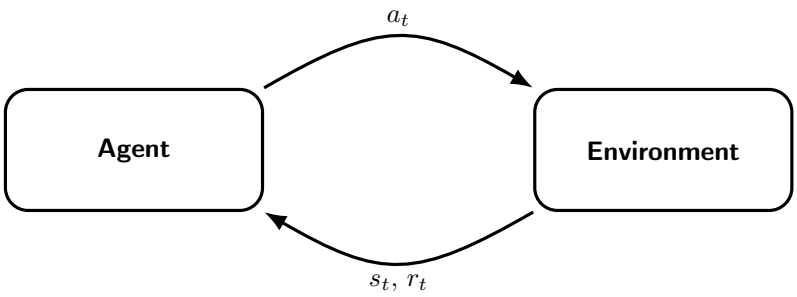


Figure 2.1.: Redraw of the standard reinforcement learning interaction loop after Sutton and Barto [2]. The environment provides state s_t and reward r_t. The agent responds with action a_t.

2.1.1. Partially Observable Markov Decision Processes (POMDP)

In a normal Markov decision process (MDP), the agent observes a Markovian state s_t, such that the distribution of the next state s_{t+1} depends only on (s_t, a_t). In robotics, this is often not possible due to limited sensing and unobserved quantities such as sensor noise, contact forces, or velocities. The standard approach for decision-making under partial observability is the partially observable Markov decision process (POMDP) [17].

A POMDP is defined by the tuple $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, Z, R, \gamma)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and $\mathcal{O}$ is the set of observations. The environment state evolves by the transition model $T(s_{t+1} \mid s_t, a_t)$. It produces observations by the observation model $Z(o_{t+1} \mid s_{t+1})$, which may omit information. Therefore, the observation sequence is generally non-Markovian. $R(s_t, a_t)$ provides the reward r_t and $\gamma \in (0, 1]$ is the discount factor [17, 18].

Since the agent can't see s_t directly, the policy maps from the interaction history $h_t = (o_0, a_0, o_1, \ldots, a_{t-1}, o_t)$ to actions. In practice, many learning-based methods use a temporal context such as a short observation history to reduce the influence of partial observability [18].

Deformable objects manipulation is therefore generally modelled as a POMDP. The complete physical state of cloth is high-dimensional and difficult to measure directly. It involves contacts, tension, and many vertex positions and velocities. Real systems are therefore usually based on partial sensory observations such as camera images [5, 19].

The folding task considered in this thesis is therefore treated as partially observable. The policy receives visual observations, while the simulator state is not considered by the policy. Following Hietala et al., temporal information is provided by short-term contexts such as stacked frames and/or the previous action. This helps the policy to infer unobservable dynamic variables [5].

2.1.2. Goal-conditioned Reinforcement Learning

In many cases, robotic manipulation tasks are specified by a desired outcome, rather than a single reward function. Goal-conditioned reinforcement learning addresses this by introducing an explicit goal variable g. The policy selects actions depending on the goal according to

$\pi_\theta(a_t \mid o_t, g)$. The reward is also defined relative to the goal according to $r_t = r(s_t, a_t, g)$ or $r_t = r(\phi(s_t), g)$ for a goal-achievement mapping ϕ [20, 21].

In goal-conditioned RL, it is common to represent the goal with a desired goal g and an achieved goal $a_t^g = \phi(s_t)$. ϕ extracts goal-relevant features from the environment state [21]. This separates the task specification from the underlying state and allows the same environment dynamics to be reused across different targets. Rewards can be defined via a goal error function, for example using a distance $d(a_t^g, g)$ with a success threshold ε.

In this thesis, the folding task is approached in a goal-conditioned way. Each observation dictionary contains both a desired goal and an achieved goal vector. The achieved goal is computed from the current cloth keypoint (corner/constraint) positions in the simulator. The desired goal is the corresponding target configuration. For each episode, a goal is sampled by generating target positions from a set of cloth constraints. The agent receives the goal together with the partial observation. It learns a policy that drives the cloth configuration toward the desired goal, consistent with the goal-conditioned setup used by Hietala et al. [5].

2.1.3. Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) optimizes a policy using experience collected by a behavior policy and stored in a replay buffer.

This makes it an off-policy method. SAC builds on maximum entropy RL. The agent tries to maximize reward while staying as stochastic as possible, which encourages exploration and keeps it from converging too early [22, 23]. This results in the following objective:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t \left(r_t + \alpha \, \mathcal{H}(\pi(\cdot \mid s_t)) \right) \right], \tag{2.3}$$

where $\mathcal{H}$ denotes the differential entropy of the stochastic policy. α is a temperature parameter that balances reward maximization and entropy (exploration) [23].

The actor is a stochastic policy $\pi_\theta(a \mid s)$. The critic consists of one or more soft Q-functions (see (2.2)). Learning is performed off-policy by reusing past experience from a replay buffer of transitions (s_t, a_t, r_t, s_{t+1}). In addition, the critic uses a soft Bellman backup.

Instead of taking only the best action, the soft Bellman backup includes an entropy term that promotes the likelihood of multiple high-value actions (see equation (2.3)). Q-networks provide the expected value for an action. Typically, two Q-networks are used to reduce positive biases in value estimates with slowly updated target networks for stability [22].

Since SAC supports continuous actions, provides stable optimization, and learns efficiently through repeated reuse of replayed experience, it is well suited for robotics tasks [23]. This is particularly important for this thesis, as the dynamic cloth folding problem under consideration requires a strategy that must output smooth continuous control commands and learn from high-dimensional inputs under partial observability [5].

2.1.4. Hindsight Experience Replay (HER)

Goal-conditioned manipulation tasks often give weak learning signals. Most rollouts do not achieve the desired goal and therefore deliver nearly identical rewards across many transitions.

This problem is solved by Hindsight Experience Replay (HER). HER records past experience with alternative goals that were actually achieved during the same episode [21].

The replay buffer stores transitions together with the episode goal, e.g., $(o_t, a_t, r_t, o_{t+1}, g)$, where the reward can be recalculated using the achieved-goal mapping $a_t^g = \phi(s_t)$ (see Section 2.1.2). An additional goal $\tilde{g}$ is taken from the visited states of the trajectory. The original goal g is replaced by $\tilde{g}$ and the reward is recomputed as $\tilde{r}_t = r(\phi(s_t), \tilde{g})$. Even if the episode did not reach the intended goal, it usually reached a different configuration. So even a "failed" rollout becomes a success story, just for a different goal. This turns failed rollouts into useful training data [21].

The cloth folding setup in this thesis uses HER to speed up learning of goal-conditioned visual policies [5]. One relabeling strategy used within HER to increase sample efficiency is future HER. Here, the newly selected target $\tilde{g}$ is chosen from a future time step within the same episode (e.g., $\tilde{g} = a_{t'}^g$ with $t' > t$). The relabeled transition sequence is likely to be goal-consistent, which improves learning efficiency in sparse reward settings [21]. HER is particularly effective for off-policy algorithms, as relabeled transitions added to the replay buffer can be reused for many gradient updates.

2.2. Robotics and Control for Bimanual Manipulation

Robot manipulation is used to generate a motion and interaction, typically by controlling the robot joint configuration. Let the robot configuration be described by the joint coordinate vector q.

$$q \in \mathbb{R}^n \tag{2.4}$$

Here, n is the number of joints [24].

Robot kinematics links joint space to task space. The joint position q results in a pose x (position and orientation) to describe end effectors in the Cartesian task space. Forward kinematics (FK) is used to determine the position of the end effector at a given joint position:

$$x = f(q) \tag{2.5}$$

End effector (EE) velocities $\dot{x}$ are derived from joint velocities $\dot{q}$ and the Jacobian matrix $J(q)$: $\dot{x} = J(q)\dot{q}$ [24, 25]. These concepts are fundamental when a policy commands actions in task space.

The system in bimanual manipulation includes two manipulators with configurations $q_L \in \mathbb{R}^{n_L}$ and $q_R \in \mathbb{R}^{n_R}$ for the left and right arms, respectively. For compact notation, the two arms can be combined into a single configuration vector $q = [q_L^\top, q_R^\top]^\top$. This results in EE poses $x_L = f_L(q_L)$ and $x_R = f_R(q_R)$.

In bimanual manipulation, the arms are coupled through the deformable object. Although they can be controlled independently, grasp points and cloth tension restrict the feasible joint and EE motions.

Hierarchical structures are common in learning-based control. A high-level policy outputs low-dimensional task-space commands such as Cartesian EE increments. A low-level controller runs at a higher rate to track these commands and enforce physical feasibility (e.g., joint limits

and smoothness) [26]. Figure 2.2 summarizes this layered pipeline. The top blocks describe the RL interface (observation o_t and action a_t), while the bottom blocks describe how these commands are realized by the low-level controller. The BioIK and OSC blocks connect directly to Sec. 2.2.2 and Sec. 2.2.3, respectively.

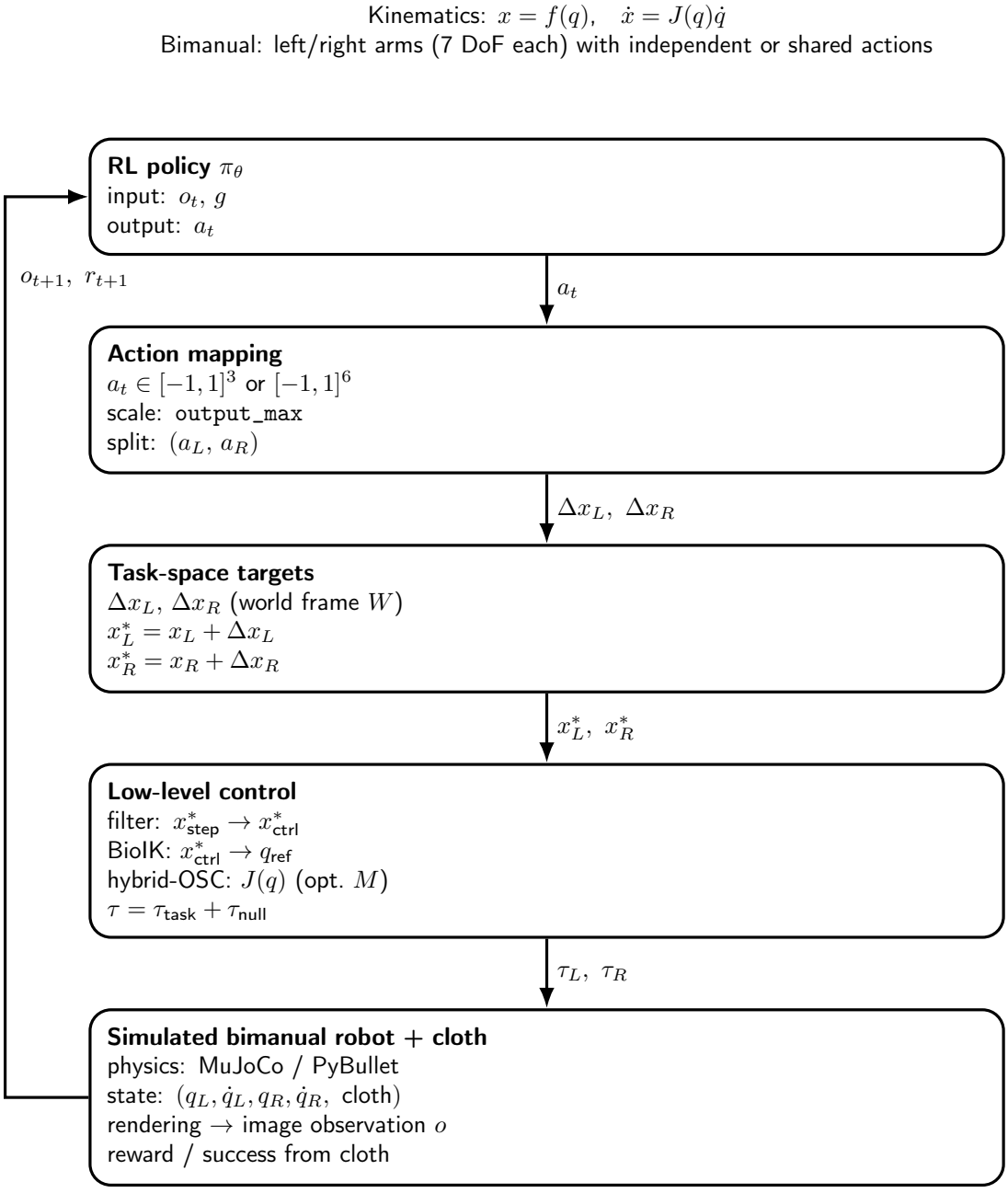


Figure 2.2.: Conceptual control pipeline for bimanual cloth folding.

2.2.1. Degrees of Freedom and 7-Joint Arms

Many modern robot arms use 7 degrees of freedom (DoF), which correspond to the number of independently actuated joint coordinates, i.e., the dimension of the joint configuration vector q (see Eq. (2.4)) [24]. A rigid end-effector (EE) pose has 3 translational and 3 rotational degrees of freedom, so 6 DoF are sufficient to reach arbitrary poses within their workspace. However, many modern manipulators have 7 joints to enable kinematic redundancy. Kinematic redundancy means that multiple joint configurations can realize the same EE pose. Thus, a 7-DoF arm has one extra joint that provides an additional null-space direction that can be exploited without changing the EE pose. This redundancy can be used to optimize secondary objectives. These include avoiding joint limits, reducing self-collisions, improving manipulability, or maintaining comfortable postures [27, 28].

2.2.2. Inverse Kinematics (IK) and BioIK

Inverse kinematics (IK) computes a joint configuration q that achieves a desired EE pose x:

$$f(q) = x^* \tag{2.6}$$

under the forward kinematics map (see Eq. (2.5)). IK in its basic form is the inverse problem of FK, i.e., finding q such that Eq. (2.6) holds [24]. With kinematic redundancy from last section 2.2.1, the IK problem usually has multiple solutions. To select a suitable configuration among multiple solutions, secondary criteria are typically applied (Section 2.2.1) [27].

In practice, deriving an explicit equation (2.6) for q (closed-form) is often not possible for complex or redundant manipulators. A common method is the iterative numerical solution of IK, in which the EE pose errors are reduced step by step.

Starting from an initial estimate q_0, the Jacobian is evaluated at the current iterate q_k (i.e., $J(q_k)$). Then, the joint configuration is refined iteratively to reduce the pose error.

Using a local linearization, the joint update can be chosen via the Jacobian pseudoinverse. To do this, $f(q)$ is locally linearized by q_k. $J(q_k)$ is used as the Jacobian matrix, which relates joint changes to EE changes, and the pseudoinverse $J^\dagger$ is used to determine the update Δq.

This leads to an iteration of the form:

$$q_{k+1} = q_k + J^\dagger(q_k)\,\Delta x_k, \tag{2.7}$$

where Δx_k is a task-space correction derived from the current pose error. $J^\dagger$ is the pseudoinverse of J. The update in Eq. (2.7) is repeated until the pose error is sufficiently small [25].

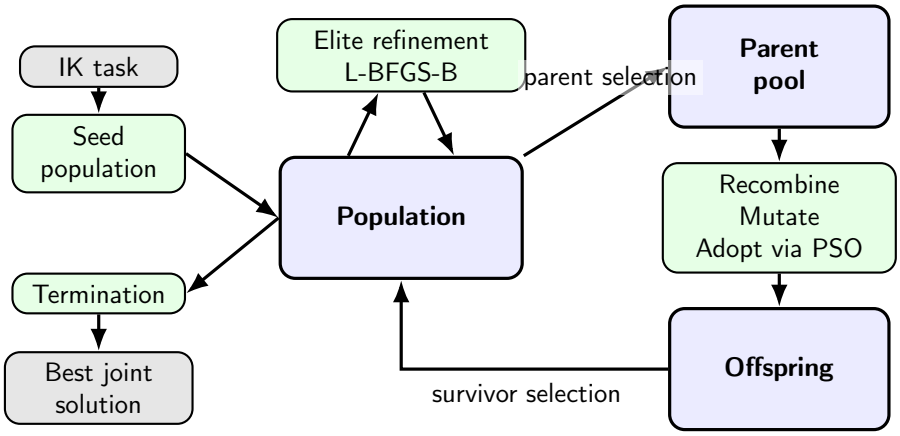


Figure 2.3.: Redraw of the BioIK memetic evolutionary algorithm after Starke et al. [3].

In this thesis, IK is solved using BioIK. BioIK is an optimization-based framework for generic full-body inverse kinematics in robotics and animation. It searches directly in joint space and treats the IK query as a constrained continuous optimization problem to respect joint bounds. The IK task is formulated as a weighted sum of objective terms so that multiple requirements can be considered simultaneously. Primarily, end-effector pose goals are achieved, but BioIK also supports objectives such as minimizing joint displacements for smoother motion. In addition, joint preferences, avoidance of joint limits, and collision-avoidance terms can be included in the optimization [3].

Optimization in BioIK is performed using a memetic strategy. This is illustrated in Figure 2.3. The diagram shows that the evolutionary loop (selection and variation) is emphasized together with the local refinement of elite candidates [3].

A population of candidate configurations is improved using evolutionary operators (selection and variation). At the same time, good candidates are refined using a local optimizer. A bound-constrained gradient-based step (L-BFGS-B) is applied to elite candidates to achieve local improvements. The evolutionary loop also includes an adoption step based on particle swarm optimization (PSO). Task accuracy and feasibility terms are considered in the optimization, while a displacement objective helps to avoid abrupt joint changes between successive control steps. This combination is designed for real-time IK in high-dimensional systems and in situations with multiple simultaneous goals [3].

In the control pipeline shown in Fig. 2.2, BioIK is used to convert desired task-space end-effector goals into a joint-space reference configuration q_{ref}.

BioIK and ROS. BioIK is implemented as a ROS (Robot Operating System) MoveIt Kinematics plugin [3]. The open-source framework ROS serves as a communication layer for interactions between different software components in a robot system. A ROS system consists of multiple processes that connect at runtime in a peer-to-peer topology. Before communication between the processes, a lookup mechanism called master is used so that the processes can find each other [29]. ROS services handle synchronous request-response calls, where a client sends a request and waits until the server replies [30].

BioIK plugs into MoveIt via the kinematics interface and can replace the default Orocos/KDL solver. The `bio_ik_service` package exposes a ROS service for calling BioIK from Python or Java, along with message definitions for BioIK goal types [31]. The `GetIK` service takes an `IKRequest` and returns an `IKResponse`. The `IKRequest` message consists of the MoveIt group name, timeout, approximate-solution flag, fixed joints, a seed `moveit_msgs/RobotState`, an optional collision-check flag, and typed goal lists such as `PositionGoal` and `OrientationGoal`. The `IKResponse` message returns a `moveit_msgs/RobotState` solution, a `moveit_msgs/MoveItErrorCodes` status, and a scalar fitness value [32].

2.2.3. Operational Space Control (OSC) and Jacobians

As illustrated in Fig. 2.2, the policy first outputs actions that specify end-effector (EE) setpoints in Cartesian space. A low-level controller then turns these Cartesian targets into joint torques. Hietala et al. use the same Cartesian torque controller, called Operational Space Control (OSC), both in simulation and on the real robot [5, Fig. 2]. To convert Cartesian EE motion into joint motion, a mapping between joint motion and EE motion is needed.

Jacobians describe how joint motions are mapped to EE motions. The joint angles are $q \in \mathbb{R}^n$ and the EE spatial velocity (linear and angular) is $V \in \mathbb{R}^6$. The geometric Jacobian $J(q)$ connects joint velocities to EE spatial velocity as in Eq. (2.8) [25].

$$V = J(q)\,\dot{q}. \tag{2.8}$$

Equation (2.8) shows the same relationship $\dot{x} = J(q)\dot{q}$ that is shown in the note to Figure 2.2.

When only the EE position $p \in \mathbb{R}^3$ is controlled, the translational (linear-velocity) part of the Jacobian is used, denoted $J_p(q)$ [25].

In OSC, a desired EE force/torque (wrench) $F \in \mathbb{R}^6$ is mapped to joint torques by the Jacobian transpose as in Eq. (2.9) [26].

$$\tau = J^{\top}(q)\,F. \tag{2.9}$$

In practice, F is chosen so that the EE moves toward a desired pose. For example, Khatib uses a linear second-order law with position and velocity feedback to define the commanded operational force [26]. For redundant arms, OSC can add extra joint torques in the Jacobian null space (e.g., for posture control) without changing the resulting EE wrench [26].

2.2.4. Bimanual Coordination and Action Parameterization

In bimanual manipulation, two arms (left l and right r) act at the same time. They often work on the same object, so the end effectors must be coordinated in space and time [33, Sec. 1]. As mentioned earlier, the policy actions are parameterized as Cartesian position increments applied at the policy rate. This is a common choice in learning-based manipulation approaches [5].

For bimanual control in this thesis, the two arms can either share the same action or receive independent commands. In the independent case, the action stacks separate Cartesian

position increments for the left and right end effectors as in Eq. (2.10) [5].

$$a_t = \begin{bmatrix} \Delta p_t^{(l)} \\ \Delta p_t^{(r)} \end{bmatrix} \in \mathbb{R}^6, \qquad \Delta p_t^{(l)}, \Delta p_t^{(r)} \in \mathbb{R}^3, \tag{2.10}$$

This lets the policy move both arms differently within the same RL step. In the shared case, the action is a single increment $a_t \in \mathbb{R}^3$ applied to both arms, i.e., $\Delta p_t^{(l)} = \Delta p_t^{(r)} = a_t$.

Given the increments, the desired end-effector positions are updated by simple integration as in Eq. (2.11)

$$p_{t+1}^{(i)} = p_t^{(i)} + \Delta p_t^{(i)}, \qquad i \in \{l, r\}, \tag{2.11}$$

and the low-level controller tracks these desired positions for each arm. Any coordination between the two arms must be learned through the task objective, while keeping the action representation minimal.

2.3. Physics Simulation and Deformable Objects

A physics simulation steps the world forward in discrete time increments, computing how robots and objects interact through numerical integration and constraint solving (contacts, joints, etc.). In PyBullet, a single call to `stepSimulation` performs collision detection, constraint solving, and integration [34]: In MuJoCo, the state is advanced with `mj_step` using a selectable integrator (e.g., Euler, Runge–Kutta, or implicit variants) [35].

Deformable objects such as cloth significantly increase the number of coupled degrees of freedom. Cloth simulation often involves contact and self-contact, which makes stability and modeling decisions particularly relevant. A common starting point is a continuum formulation. The cloth is first modeled as a continuous material. It is then divided into many small parts (a mesh). The movement of these mesh points is calculated over time. The result is an ordinary differential equation (ODE) [36]:

$$\ddot{\mathbf{x}} = \mathbf{M}^{-1}\left(-\frac{\partial E}{\partial \mathbf{x}} + \mathbf{F}\right), \tag{2.12}$$

where $\mathbf{x}$ is a vector containing the positions of all mesh points like nodes of the cloth. $\mathbf{M}$ is the mass matrix of the discretized cloth. $E(\mathbf{x})$ is the internal potential energy of the cloth, and $-\frac{\partial E}{\partial \mathbf{x}}$ gives the corresponding restoring forces. $\mathbf{F}$ is the sum of external forces, e.g., damping, contact, and gravity [36].

The time-step size and the integrator choice influence simulation stability and accuracy, since time integration directly affects deformable dynamics [37].

Due to the dynamics and complexity of the material, the physics simulation can become unstable if time steps that are too large are used [36]. A policy might learn to exploit simulation instabilities instead of learning real folding behavior. If the time step or integrator is off, the simulation is not able to learn something useful. For instance, a default time step size of $1/240\,\mathrm{s}$ is specified in the PyBullet documentation. Larger time steps (above $1/60\,\mathrm{s}$) tend create artifacts [34].

A common countermeasure is domain randomization. This involves varying simulation parameters during training to increase robustness [38]. The next subsections cover the essentials of how cloth is actually simulated. In addition, the most important differences between MuJoCo and PyBullet are highlighted.

2.3.1. Cloth Modeling Basic

Deformable objects can change their shape under the influence of forces. Therefore, their state must be described not only by rigid motion, but also by deformation [37]. In computer graphics, cloth is often modeled as a deformable surface [36]. In physics-based simulation, on the other hand, Newtonian dynamics (bodies/particles) and continuum mechanics (continuous medium) are combined and solved numerically [37]. In practice, the cloth surface is discretized into a finite number of mesh nodes, and their 3D positions are computed over time. A widely used representation is a triangular particle mesh with n nodes [37]. The complete configuration is stacked into a single vector $\mathbf{x} \in \mathbb{R}^{3n}$. After discretization, the cloth dynamics are often written in the ODE form (2.12) [36].

A material/deformation model defines the internal forces that counteract deformation by defining an energy $E(\mathbf{x})$. The restoring forces are calculated as $\mathbf{f}_{\text{int}}(\mathbf{x}) = -\partial E/\partial \mathbf{x}$ [36, 37]. The dominant effects for textiles are elongation in the plane and bending out of the plane. The properties of cloths depend heavily on the direction in which they are stressed. Cloth resists stretching relatively strongly, while it resists bending only weakly. In mass-spring models and similar particle systems, these effects are not modeled directly from continuum physics. Instead, they are approximated by distance springs and more general spring types such as angular or bending terms, which are defined on the mesh connectivity [37]. Springs maintain distances between nodes, while angle and bending terms control the curvature of the surface. Particle systems are intuitive and cheap to compute, but many mass-spring formulations don't converge as the mesh gets finer—change the resolution or topology and the behavior shifts. Continuum mechanics methods like FEM take a different route: they start from a PDE and discretize the object into elements, turning the continuous problem into a finite one over node values [37].

Beyond internal elasticity, cloth needs collision and contact handling to interact with its surroundings [36]. Contacts between cloth and solid objects can be handled with constraints that keep colliding nodes on the surface. This means that one does not have to rely solely on penalty forces that are artificially generated, such as when nodes penetrate the surface [36]. Self-collision works similarly with vertex-triangle constraints that keep vertices on the right side of nearby faces. Instead of computing contact forces directly, a constraint-based approach generates collision constraints each step and projects penetrating points back to valid positions. This position-based approach was used by Müller et al. for real-time cloth simulation in games, highlighting its practicality for interactive cloth behavior with collisions [39]. Discretization, internal models, and constraint handling are recurring building blocks in the literature on the simulation of deformable objects [37].

2.3.2. MuJoCo vs. PyBullet

In sim-to-sim evaluation, a policy is trained in one physics engine and then evaluated in another engine to measure cross-engine transfer. The engines differ in how the continuous equations of motion are advanced in discrete time steps, i.e., in the numerical integration scheme. They also differ in how they enforce constraints in the model, such as joints, equality constraints (e.g., fixed-length constraints), and contact constraints. Finally, they differ in contact handling, i.e., how collisions are detected and how contact forces are resolved. Therefore, even with identical high-level parameters, the resulting trajectories can differ [40].

Rigid-body simulation and contacts. MuJoCo is a free, open-source physics engine built for fast and accurate simulation, originally developed with model-based optimization through contacts in mind. MuJoCo simulates dynamics in generalized coordinates. Constraints like contact and friction get formulated continuously and solved via convex optimization, meaning any local optimum is also global [15, 41]. It has several solver options (Newton, Conjugate Gradient, Projected Gauss–Seidel) and integrators like Runge–Kutta [15, 42].

PyBullet is a Python module for robotics sim and ML. It loads URDF, SDF, MJCF models and exposes routines for forward/inverse dynamics and kinematics. Each simulation step in PyBullet performs collision detection, constraint solving, and integration [1]. The documented default time step is $1/240\,\mathrm{s}$, so covering a fixed amount of simulated time requires many integration steps. PyBullet exposes the Bullet physics engine, implemented in C++, through a Python API [43]. The sequential-impulse constraint solver in Bullet implements a Projected Gauss–Seidel method for constraint solving [44]. This shows that constraint-solving strategies can differ between engines, even if the user-facing APIs look similar.

Why deformable objects amplify the engine gap. Equation (2.12) highlights that cloth simulation combines an internal deformation model with contact and collision constraints [37]. In practice, the same high-level cloth specification (mesh resolution, mass, stiffness, damping, friction) may map to different low-level models and solvers in MuJoCo and PyBullet [1, 45].

Deformable objects in MuJoCo. The cloth implementation used on the MuJoCo side of this thesis is based on the older `composite` mechanism. The current MuJoCo documentation recommends using 2D flex deformable objects instead for thin elastic structures such as cloth. In the scene, the cloth is defined as a `composite` of type `grid`, which is expanded into model elements during compilation [45, 46]. The XML reference describes this grid type as a 1D or 2D grid of bodies with a sphere geom, a sphere site, and three orthogonal sliding joints by default. Adjacent grid bodies are connected by equality-constrained spatial tendons. The main tendons run along the grid axes, and optional shear tendons run along the diagonals [46]. In this thesis, `arena_diana.xml` uses exactly that structure with `joint kind="main"`, `tendon kind="main"`, and `tendon kind="shear"` [5, 6]. A `skin` is added as a visual surface, but collision handling and physics still come from the generated element bodies and their geoms [45]. Therefore, the contact behavior is set at the geom level, including per-geom friction and the solver parameters `solref` and `solimp` [45, 46]. The

reference acceleration parameter `solref` is often expressed in terms of a time constant and a damping ratio. `solimp` sets the impedance of how constraints respond [45]. As a result, the MuJoCo cloth in this thesis is best understood as a solver-tuned tendon-connected grid model [45, 46].

Deformable objects in PyBullet. The PyBullet Quickstart Guide describes support for deformable objects. This includes cloth and volumetric objects. The loadSoftBody function allows deformable objects to be loaded from OBJ or VTK meshes. Additionally, PyBullet offers the option of choosing between different deformable body models. It offers two deformable body models. One based on the finite element method (FEM) and one using position-based dynamics (PBD). In the finite element method (FEM), deformation is described by a material model, which can result in internal forces. PBD, on the other hand, calculates deformation by solving position constraints without explicitly modeling internal forces. The guide notes that some loadSoftBody parameters are intended for the FEM model and are ineffective for PBD simulations. The loadSoftBody options include model- and material-related switches such as the activation of a Neo-Hookean simulation. In addition, there are stiffness and damping parameters, which make the choice of model an explicit part of the API [1]. Within Bullet, support for deformable objects is primarily focused on the btSoftBody class. This class provides functions for simulating cloth and volumetric soft bodies with bidirectional interaction between soft and rigid objects. This interaction forms the basis for coupling cloth with a table and grippers [47].

Implications for sim-to-sim evaluation. MuJoCo and PyBullet use different formulations for deformable bodies and solve contacts and constraints with different numerical methods. Cross-engine evaluation can therefore be interpreted as robustness to a simulator-induced domain shift. Even for rigid continuous control benchmarks without deformables, cross-engine transfer can fail asymmetrically. Mohammed and Valdenegro-Toro report poor generalization when training in PyBullet and evaluating in MuJoCo for multiple tasks and algorithms [40]. They also report that the opposite direction can be better in some cases [40]. Therefore, in sim-to-sim analyses, both the simulator backend and the choice of deformable model should be considered as sources of variation. The same settings can lead to different results, depending on how the engine implements integration, constraints, and contacts [1, 45].

2.3.3. Domain Randomization (DR)

Domain randomization, means that the simulator is trained with its key settings changed on purpose from one run to the next. The idea is straightforward. If the agent trains under enough variation in simulation, it won't break when conditions shift at deployment time. Instead of aiming for exact calibration, variability in robotics is usually linked to the randomization of simulation parameters. From this perspective, robustness is coming from an exposure to a large number of realistic simulated variations. Without it, the policy can easily overfit to one specific simulator configuration [38, 48].

More formally, DR introduces a domain parameter vector ξ that covers things like physics constants, sensor noise, and visual appearance. Before each rollout, the simulator draws $\xi \sim p(\xi)$ and runs the episode under those sampled settings. The aim is a policy that does well on average over ξ (Eq. (2.13)) [38, 48].

$$\pi^\star \;=\; \arg\max_{\pi} \; \mathbb{E}_{\xi \sim p(\xi)}[J(\pi;\xi)]\,. \tag{2.13}$$

$J(\pi;\xi)$ is the expected return of π under the settings ξ [38, 48].

DR can vary both observations and dynamics. Typical randomized variables include rigid-body and contact parameters (e.g., friction and restitution), inertia parameters (e.g., mass properties), sensor noise, and timing effects such as latency or delays. Typically visual domain randomization includes randomizing lighting, textures, and camera positions. When the randomized variability is broad enough, it can also work with low-fidelity rendering [38, 48, 49].

The design decision of when to apply randomization is very important. The variance of the learned policy can be increased with randomly assigning parameters to each simulation step. In practice, though, some parameters need a full simulator reset or recompilation, so randomizing per step isn't always feasible. The common workaround is episode-level randomization. Sample ξ once at the start of each rollout and keep it fixed throughout [38].

For deformable objects, DR is particularly important because their dynamics are highly sensitive to material and contact parameters. Stiffness, shear resistance, and friction are some of the factors which have a significant impact on the behavior of deformable objects. Even small modeling errors can produce wildly different deformations. That is why cloth manipulation usually randomizes both the scene and the cloth itself. Camera pose, lighting, and similar visual details are perturbed, and the cloth model is varied through its physical parameters. This has been used to narrow the sim-to-real gap, and the learned policies were later run on real cloth without training on real deformable data [5, 8].

Temporal effects can be handled as part of domain randomization if the policy depends on feedback. Delays in actions and observations are common in real-world systems. Random delay models and training methods have been explicitly investigated in reinforcement learning. This is why some robotics pipelines also use temporal randomization (e.g., action frequency or observation delay) in simulation to reduce sensitivity to latency mismatches [5, 38, 50].

2.4. Vision for Visual Feedback Control

As already mentioned, manipulating cloth is challenging due to the high dimensionality of the configuration space and the dynamics caused by material properties. The complete state of a cloth cannot be directly observed in practice. Therefore, vision is a common means of obtaining feedback in dynamic folding [5].

2.4.1. Camera Model and Projection

Figure 2.4 summarizes the pinhole (perspective) camera model used by OpenCV to map 3D points to pixel coordinates [4].

In homogeneous coordinates, a 2D pixel (u, v) is represented as $\tilde{\mathbf{x}} = [u\ v\ 1]^\top$ (up to a nonzero scale). Given a 3D point $\mathbf{X}_c = [X_c\ Y_c\ Z_c]^\top$ in the camera frame $\mathcal{F}_c$, the projection follows Eq. (2.14) [4].

$$s\,\tilde{\mathbf{x}} = \mathbf{K}\,\mathbf{X}_c, \tag{2.14}$$

The intrinsic matrix $\mathbf{K}$ encodes the focal lengths (f_x, f_y) in pixels and the principal point (c_x, c_y). Pixel coordinates follow OpenCV convention with the origin in the upper-left corner, matching the $(\mathbf{u}, \mathbf{v})$ axes in Fig. 2.4 [4].

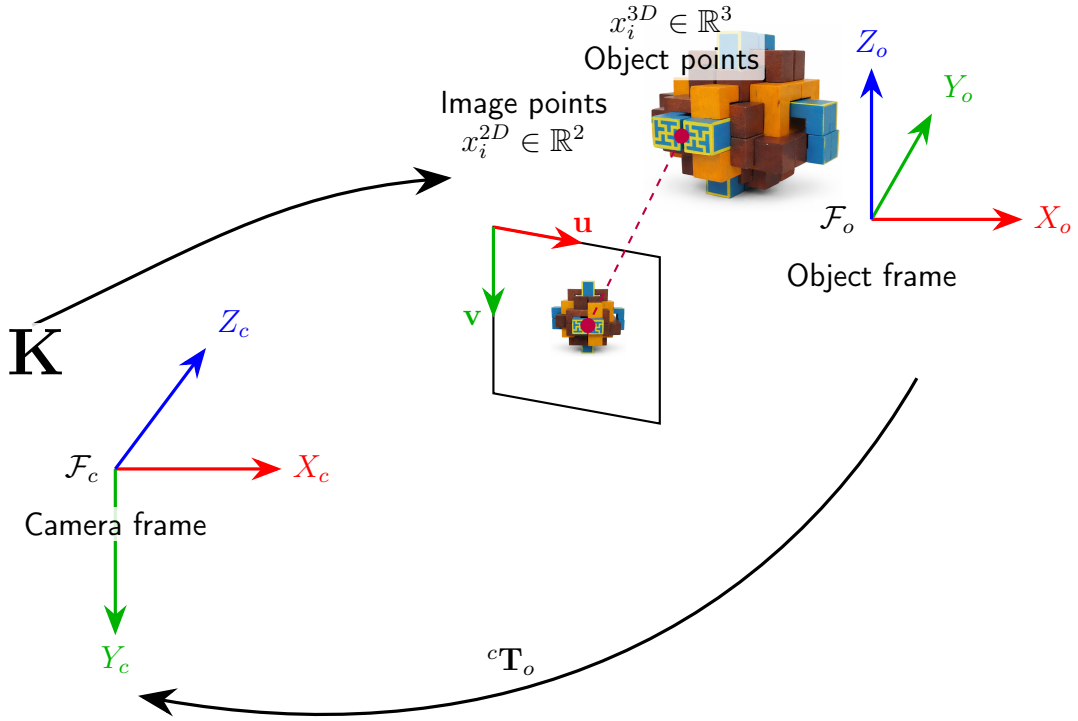


Figure 2.4.: Redrawn from the OpenCV "Perspective projection, from object to camera frame" [4].

Figure 2.4 displays the pinhole camera model. It summarizes the relationship between object/world coordinates, camera coordinates, and image coordinates. For a 3D point expressed in a world/object frame, OpenCV describes transforming it into the camera frame using the extrinsic pose ${}^c\mathbf{T}_o$. This transformation is represented in homogeneous form by a rotation $\mathbf{R}$ and translation $\mathbf{t}$. Using homogeneous 4-vectors $\tilde{\mathbf{P}}_w = [X_w\ Y_w\ Z_w\ 1]^\top$ and $\tilde{\mathbf{P}}_c$, this is written as $\tilde{\mathbf{P}}_c = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}_w$ before projection [4].

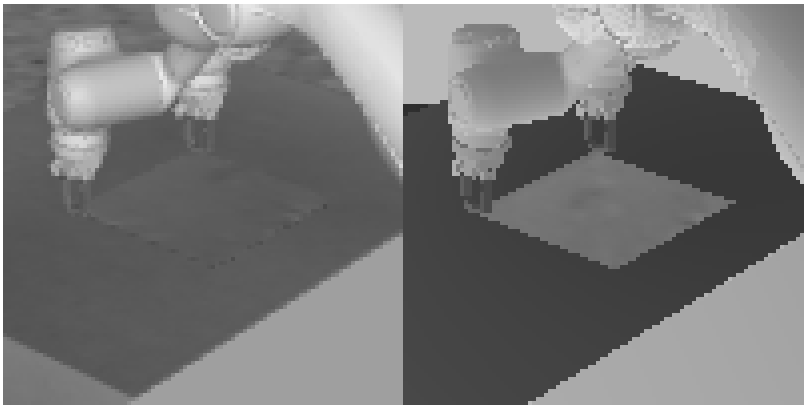


Figure 2.5.: Grayscale policy input at step 0. MuJoCo (left). PyBullet (right).

2.4.2. Observation Preprocessing

Most vision-based RL pipelines preprocess raw images to reduce input dimensionality and to address observation artifacts. Mnih et al., for example, extract the Y luminance channel and then scale each frame to 84×84 pixels. The Y (luminance) channel yields a grayscale image that encodes scene brightness. The last four processed frames are stacked to provide short-term motion information. To remove flickering which is caused by the limited number of sprites the Atari 2600 can display, they encode each frame using a pixel-wise maximum over the current and previous frame. This max-over-two-frames step is applied before the frame is converted to luminance and resized [51]. Hietala et al. use a 100×100 grayscale image as the visual policy input in their dynamic cloth folding setup. Their policy additionally takes the goal g and the previous action a_{t-1} as inputs alongside the current image [5].

Figure 2.5 shows the observations after a reset in both backends. The grayscale policy input shows similar scene geometry between MuJoCo and PyBullet, but the shading, contrast, and the ground are not perfectly aligned. This visual gap is what a transferred policy has to deal with.

2.4.3. Image Augmentation

Data augmentation, as described by Buslaev et al., increases the size and diversity of labeled training data. A label-preserving transformation is applied to the existing data to generate new examples. Image augmentation acts as regularization. It can be used to reduce overfitting in deep learning models [52]. Shorten and Khoshgoftaar group common augmentation methods into families. These include geometric transforms, color transforms, mixed approaches, and random erasing [53].

Matas et al. use DR to vary the visual appearance of the training observations in their setup. They describe sampling simulation parameters (e.g., camera pose, lighting, textures) from probability distributions centred around noisy ground-truth estimates. They argue that this encourages the agent to ignore minor variations and become robust to domain changes, including sim-to-real transfer. For their hanger environment, they list variations in textures,

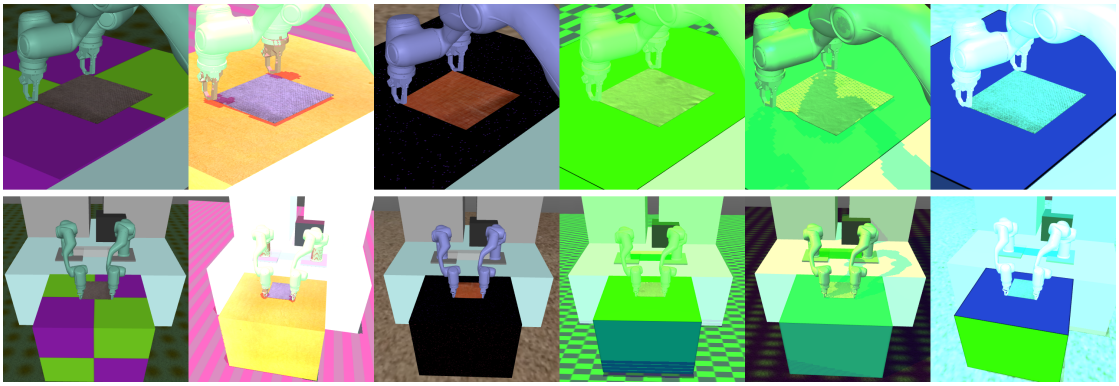


Figure 2.6.: Sampled MuJoCo DR settings.

colors, lighting, camera pose, object sizes, and initial robot configurations as part of the randomization. They also randomize camera position, orientation, and intrinsics. That same setup is used in both MuJoCo and PyBullet here. Figure 2.6 gives six examples from one MuJoCo training run. The top strip shows the `side` camera and the bottom strip shows `eval_camera`.

The task itself does not change between the panels, only the presentation. This involves changing the lighting conditions, colors, floor patterns and camera pose.

Matas et al. report training agents entirely in simulation with DR and deploying them in the real world without using real deformable-object training data [8].

2.4.4. CNN Feature Learning and Auxiliary Objectives

A Convolutional neural networks (CNNs) is a type of deep learning network. The input of a convolutional neural network (CNN) are images, on which they learn to output relevant features. Compared to fully-connected layers, CNNs reduce the number of parameters [54]. Kernels are used to achieve this, by applying a set of filters to the image. This result is a reduction of the number of connections between pixels and neurons. In contrast to fully connected layers, where each pixel is connected to every neuron in the next layer [55]. Goodfellow et al. describe convolutional layers as a combination of sparse connectivity and parameter sharing.

The kernels can be used across multiple input positions, which means that the same filter weights are reused at many image locations. This is called parameter sharing [54].

Hietala et al. model their policy using three mappings (h_1, h_2, h_3) built around an image latent representation. For the first mapping, an image encoder h_1 is used to convert the grayscale image into a latent vector, which is then used for control. A separate policy head h_2 maps the latent representation (together with a_{t-1} and g) to the next action a_t. The auxiliary head h_3 predicts cloth-point (corner) locations in pixel coordinates (u, v). They state that these auxiliary outputs are only used during training to guide the policy toward detecting relevant visual features.

3. Related Work

3.1. Overview

Regarding manipulation of deformable objects, there are various approaches to robotic clothing manipulation. Longhini et al. recently published a review discussing shortcomings in advances in textile manipulation. According to this review, textile manipulation often tailors textiles to a group of textiles. They also emphasize that benchmarking of textile manipulation is also limited, as there are no consistent and objective evaluation processes [56].

The early approach from Doumanoglou et al. presented a pipeline for autonomous cloth folding. They use Active Random Forests for garment classification and a model matching technique to determine the folding trajectory. However, the lack of adaptive feedback for real-time adjustment, which results in accumulated errors, is a shortcoming of the system [10].

Lips et al. used keypoint detection to find corners in 2D positions for the folding process. Despite it worked well in simulation, there was a large performance drop due to the differences between synthetic and real environments. An open-loop architecture was also used here, so the system is less adaptive and limited in dynamic tasks [57].

A feedback loop to monitor execution appears to produce positive results in terms of the robustness and generalizability to fold clothing in half, as shown by Longhini et al. Their system AdaFold works inside a feedback loop. It takes front and back RGB-D observations, evaluates point clouds to find good pick-and-place positions and outputs an optimized folding trajectory. Because it combines cloth perception with semantic descriptors, the planner can refine its actions iteratively as it goes [56].

3.2. Bimanual Cloth Manipulation of Non-Rigid Objects

Bimanual systems can enable improved cloth handling for complex manipulation tasks. Especially with increased manipulation space for two-point constraints in clothing, two arms can improve handling [58].

Dynamic movements with both arms can also improve handling when spreading out and smoothing clothing. Using high-speed dynamic motions, Ha et al. demonstrated a self-supervised system for unfolding clothing. The dual-arm system stretches and flings the garment to increase the area of the garment. It has achieved to unfold garments larger than the robots kinematic reach by leveraging dynamic motions [59].

Bimanual approaches for cloth folding such as *FabricFlowNet: Bimanual Cloth Manipulation with a Flow-based Policy* [13] were able to improve the handling of deformable materials. FabricFlowNet uses optical flow, which is traditionally used for motion estimation, to estimate fabric points between the current state and the target state and to control the trajectory of

the two arms for folding. However, it remains computationally intensive and struggles with occlusions and self collisions in folds. Small disturbances can result in significantly different movement results, such as a poor grip on clothing.

3.3. Sim-to-Sim and Sim-to-Real Transfer in Reinforcement Learning

The training in simulations has many benefits compared to training on real hardware. Data generation in simulation is safer, cheaper, and faster than collecting data on real hardware. However, since simulations are based on a simplified model, they cannot represent the complexity of the real world [38]. This discrepancy between simulation and reality is known as the reality gap [49]. According to Muratore et al., this gap is caused by "omitted physical phenomena, inaccurate parameter estimates, and numerical effects from typical solvers" [38]. The same problem also appears in perception, where synthetic images may not reproduce real sensor noise and visual complexity [49]. The goal, therefore, is to train a policy that remains robust to the reality gap, rather than overfitting to a simulator instance [38].

Mohammed et al. directly tested whether RL policies transfer between simulation engines (see also Section 2.3.2). Trained models were evaluated using the same task and interfaces, but different simulation engines. The result was that generalization across different simulators can be significantly worse than in-engine evaluation.. According to Mohammed et al., transferability also depends on the task and the learning algorithm. Cross-engine evaluation is therefore a useful way to measure how engine-specific modeling choices affect policy performance [40].

Several strategies exist for bridging the reality gap. Tobin et al. describe a "system identification" approach in which the simulator parameters are adapted to the real world. However, this is said to be time-consuming and error-prone [49]. A complementary approach is training on a distribution of simulators, which corresponds to domain randomization, as formalized in Section 2.3.3 and Eq. (2.13) [38, 48].

Muratore et al. divide domain randomization into three categories. The common static domain randomization uses a fixed distribution of domain parameters. There the domain parameters are sampled from the distribution at the beginning of each rollout. In contrast, adaptive domain randomization updates the distribution of domain parameters during learning. Adversarial domain randomization deliberately introduces simulation parameters and disturbances to make the task more difficult and improve the robustness of the policy [38]. Further approaches using domain randomization are described in Section 2.3.3, in particular for visual and dynamics randomization.

Beyond domain randomization, there is domain adaptation, in which representations between simulated and real observations are adjusted to bridge the reality gap. Tobin et al. present approaches based on the use of target domain data or fine-tuning to adapt the domains [49]. James et al. introduced Randomized-to-Canonical Adaptation Networks (RCANs), which take randomized simulation images and map them back to a consistent version. Using this method, sim-to-real transfer was achieved with little real-world fine-tuning [60].

3.4. Dynamic cloth folding

Relevant for this thesis, the approach from Hietala et al. proposes a repository [1] containing code and instructions for dynamic cloth folding using visual feedback, reinforcement learning (SAC), and domain randomization. As stated in the paper from Hietala et al. and other relevant works, the robustness and efficiency increase through feedback-driven and dynamic control. They achieve successful folding across different cloths using fast movements that leverage inertia and adapt in real time based on visual observations. The policy is only trained in simulation and transferred directly to the real world without further fine-tuning [5].

Hietala et al. trained a visual feedback policy for dynamic cloth folding using SAC in simulation. In order to maintain consistency between sim and real setups, the policy uses a shared Cartesian controller and is guided by images and goal configurations. They address issues of dynamic manipulation such as visual ambiguity and high-dimensional cloth dynamics by using domain randomization to train rules and then implementing them in the real environment. However, the method requires a sophisticated training setup that includes non-trivial transfer procedures, simulation of cloth physics with precise parameter sampling, and dense reward shaping [5].

[1] `https://github.com/hietalajulius/dynamic-cloth-folding`

4. Simulation Environment: The Diana Scene

The starting point for this thesis is the dynamic cloth-folding pipeline by Hietala et al. shown in Fig. 4.1, where policies are trained in simulation and transferred to real hardware. In their setup, a single Franka Emika Panda arm in MuJoCo folds a square cloth that is pre-grasped at one corner. For feedback control, the policy receives rendered camera observations and outputs Cartesian displacement targets that drive robot–cloth interactions [5].

In this thesis, the environment is extended to a bimanual Diana7 setup and implemented with two interchangeable physics backends (MuJoCo and PyBullet). Instead of sim-to-real transfer, the focus here is sim-to-sim evaluation between the two physics engines. Both backends expose a Gym-style interface with `reset/step` [61]. Observation and action tensor shapes are aligned across backends so that the same policy can be evaluated unchanged in either simulator. As in the reference setup, intermediate targets between policy steps are generated by repeated filtering/interpolation to keep commanded motion continuous [5].

In the current implementation, both backends share the same BioIK target-generation path and closely matched hybrid operational-space torque-control logic. The task-space PD law regulates the gripper position using the end-effector Jacobian, and a nullspace term regularizes the joint posture. Optionally, the MuJoCo joint-space mass matrix is used to compute the operational-space inertia $\Lambda = (JM^{-1}J^{\top})^{-1}$; otherwise M defaults to the identity. BioIK is called as a ROS service (`/bio_ik/get_bio_ik`) to compute a joint-space reference configuration for the current Cartesian target using a `bio_ik_msgs/IKRequest` that embeds a `moveit_msgs/RobotState`, and solver status is checked via `moveit_msgs/MoveItErrorCodes` [62, 63].

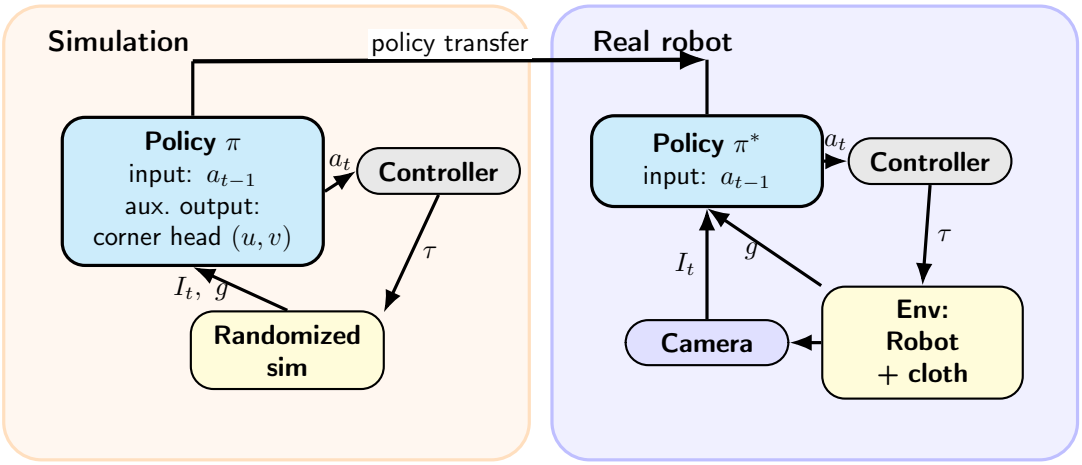


Figure 4.1.: Redraw of the training and deployment setup proposed by Hietala et al. (Fig. 2 in [5])

4.1. System Context and Simulation Setup

4.1.1. Original environment (reference implementation)

Fig. 4.1 summarizes the sim-to-real training pipeline and policy interface proposed by Hietala et al. In simulation, cloth dynamics and visual appearance are randomized before the learned policy is deployed on the real system. For the sideways-folding task, the policy receives a 100×100 grayscale image $\mathcal{I}_t$, the goal g, and the previous action a_{t-1}. It outputs the next Cartesian action a_t, while an auxiliary head predicts cloth-corner pixel coordinates (u, v) during training. The Cartesian controller maps each setpoint to joint torques τ, and those torques are what actually drive the arm in both simulation and on the real robot [5].

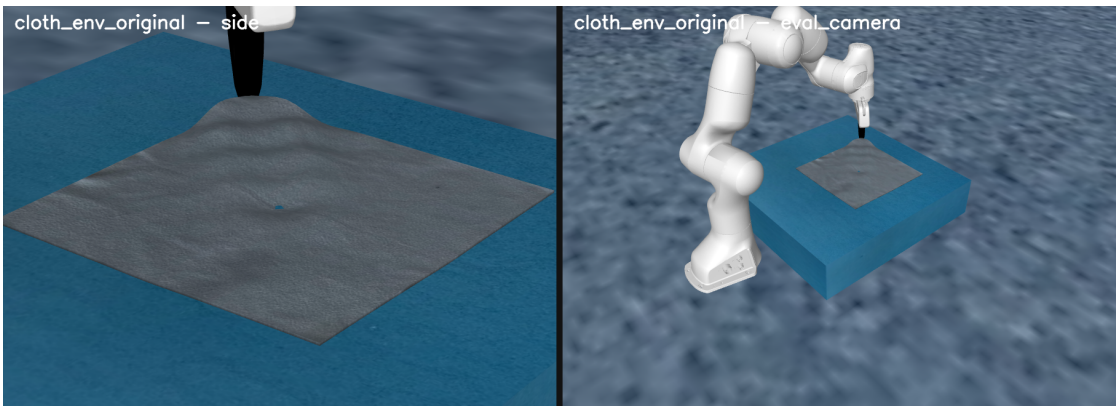


Figure 4.2.: First step evaluation of the original cloth environment: side view

In the reference task, the code starts from a simple setup. A $20,\mathrm{cm} \times 20,\mathrm{cm}$ cloth is already

grasped, so the robot only has to execute the fold. The auxiliary corner prediction head is used only during training to shape the learned representation [5]. The reference implementation is publicly available in the `dynamic-cloth-folding` repository [64]. Fig. 4.3 shows the reset scene from that same evaluation camera in the reconstructed PyBullet environment.

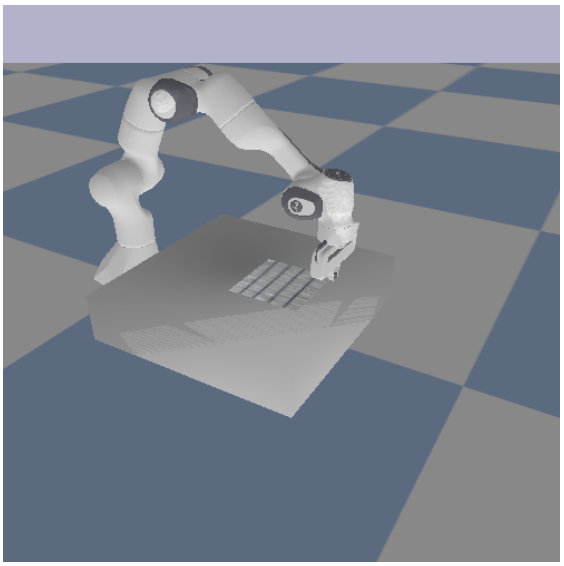


Figure 4.3.: Original arena in the reconstructed PyBullet environment

Before the PyBullet backend was extended with the Diana scene, the original environment from Hietala et al. was rebuilt in PyBullet. This version uses PyBullet's inverse kinematics to control the Franka Panda arm loaded from `franka_panda/panda.urdf`. It was mainly used to replay the reference demonstration, compare a small number of rollouts, and find a PyBullet parameter setting that behaves close enough to the original task. The `demos.csv` file from the reference repository was used as the demonstration trajectory for these checks. The reset, reward, and timing logic were reimplemented in the PyBullet environment itself. The resulting code was used as the basis for the later porting of the Diana scene. The reconstructed version of the reference environment is available in the `bruehldev/dynamic-cloth-folding` repository [65].

The simulator runs at a higher frequency than the policy in the reference implementation. Hietala et al. report a simulation frequency of $100\,\mathrm{Hz}$ and a policy frequency of $10\,\mathrm{Hz}$. To avoid abrupt jumps to a new target, the environment interpolates between successive policy outputs and executes intermediate targets for smoother motion [5].

4.1.2. Dual-Arm Diana Robot Configuration

In this thesis, the dual-arm Diana7 scene is implemented in both PyBullet and MuJoCo, and both backends share the same dual-arm stepping logic in the environment `step` function.

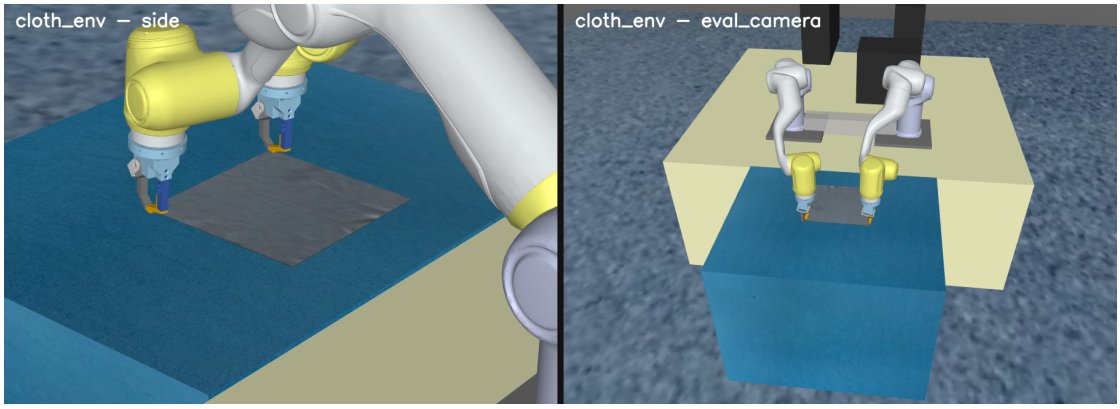


Figure 4.4.: First step evaluation of the Diana cloth environment: side view

In the Diana7 version, the cloth edge length is increased to $30\,\mathrm{cm}$ to provide enough workspace for bimanual folding. Both backends expose the same goal-based task dictionary with keys `achieved_goal`, `desired_goal`, `observation`, `robot_observation`, and `image`, and they support the same `ee`, `ctrl`, and `none` robot-observation variants. Dual-arm execution is enforced in both environments by requiring `DIANA_DUAL_CONTROL=1` and `DIANA_DUAL_GRASP=1` at startup. Actions are configured either as coupled 3-DoF Cartesian deltas applied to both arms or as independent 6-DoF deltas split across arms, selected via `DIANA_DUAL_ACTION_INDEPENDENT`. Each arm is controlled via a gripper control point (`grip_geom`) to which the Cartesian delta action is applied. In Fig. 4.5, the lower sphere visualizes `grip_geom`, which defines the gripper control point. The upper sphere is `grip_site`, the end-effector reference used for forward kinematics and Jacobians in the original Franka setup. In the Diana7 scene, both coincide at the same position [6].

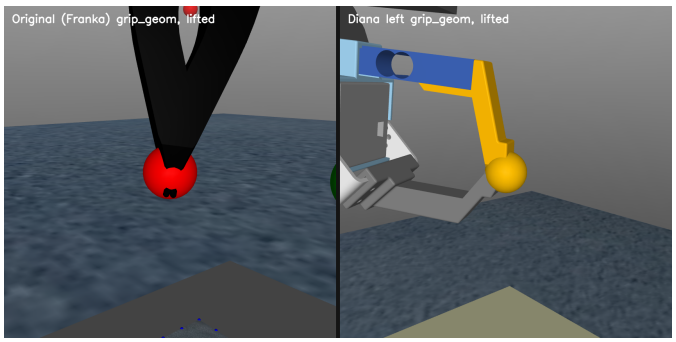


Figure 4.5.: Gripper geometry comparison: lifted between grippers

The initial grasp configuration is defined by the selected grasped cloth corners, while `DIANA_FOLD_DIRECTION` specifies the fold direction used by the task. Fig. 4.6 shows a small implementation benchmark over 10 scripted folding rollouts per configuration. In this benchmark, fold direction and domain-randomization settings were varied across runs. It compares two simple diagnostics. The first is accumulated control-point misplacement, meaning

how far the gripper control point drifts from the cloth corner it is supposed to carry over the rollout. The second is peak acceleration. In MuJoCo, this is read from `dQACC_MAX`, so it serves as a rough proxy for how abrupt the motion becomes during the fold. In MuJoCo, "backward" yields a lower average misplacement ($0.006\,\mathrm{m}$) than "forward" ($0.012\,\mathrm{m}$). Peak `dQACC_MAX` follows the same pattern, at about $1.0\,\mathrm{m/s^2}$ for "backward" versus about $2.5\,\mathrm{m/s^2}$ for "forward". In PyBullet, the gap between directions is smaller, but "backward" still looks a bit more stable in these runs.

That is enough reason to keep "backward" as the default fold direction for training in both backends.

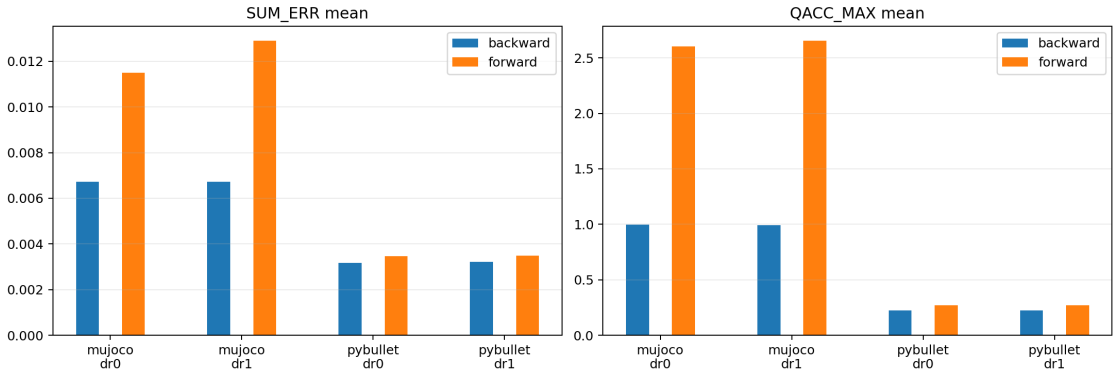


Figure 4.6.: Fold benchmark visualization over 10 runs

At the start of each episode, the environment checks whether the gripper–corner assignment is valid. The distance between each gripper control point and its assigned cloth corner (selected via `DIANA_GRASP_CORNER_LEFT` and `DIANA_GRASP_CORNER_RIGHT`) is compared against `DIANA_GRASP_INIT_TOL_M`. If the distance exceeds the tolerance, the environment performs one additional reset attempt.

BioIK is used for inverse kinematics in both backends, and joint 7 can be locked via `DIANA_BIOIK_LOCK_J7` to reduce wrist rotation during IK [6, 62, 63]. This matches the policy-to-controller layers in Fig. 2.2.

4.1.3. Workspace and Arena Definition

In this project, the Diana MuJoCo scene is instantiated by rendering the MJCF template `arena_diana.xml` with episode-specific parameters at reset time. Fig. 4.4 and Fig. 4.7 show that the scene layout is nearly identical in both backends. The table placement, camera placement, and cloth-corner indexing in the task definition are consistent across backends. In MuJoCo, the yellow workspace posts were removed because their collision geometry is larger than their visual mesh and appeared in the training camera image [6].

Table 4.1.: Environment variables that directly affect the bimanual simulation setup and controller configuration.

Variable	Effect on setup
`DIANA_DUAL_CONTROL`	Enables dual-arm stepping.
`DIANA_DUAL_GRASP`	Enables two-corner grasp mode.
`DIANA_DUAL_ACTION_INDEPENDENT`	Chooses coupled 3-DoF vs. independent 6-DoF bimanual actions.
`DIANA_GRASP_CORNER_LEFT`	Selects the left grasped cloth corner index.
`DIANA_GRASP_CORNER_RIGHT`	Selects the right grasped cloth corner index.
`DIANA_GRASP_INIT_TOL_M`	Sets the reset-time gripper–corner distance tolerance.
`DIANA_FOLD_DIRECTION`	Selects the fold direction used by the task, e.g. `backward`.
`DIANA_BIOIK_LOCK_J7`	Locks joint 7 during BioIK solves to reduce wrist rotation.

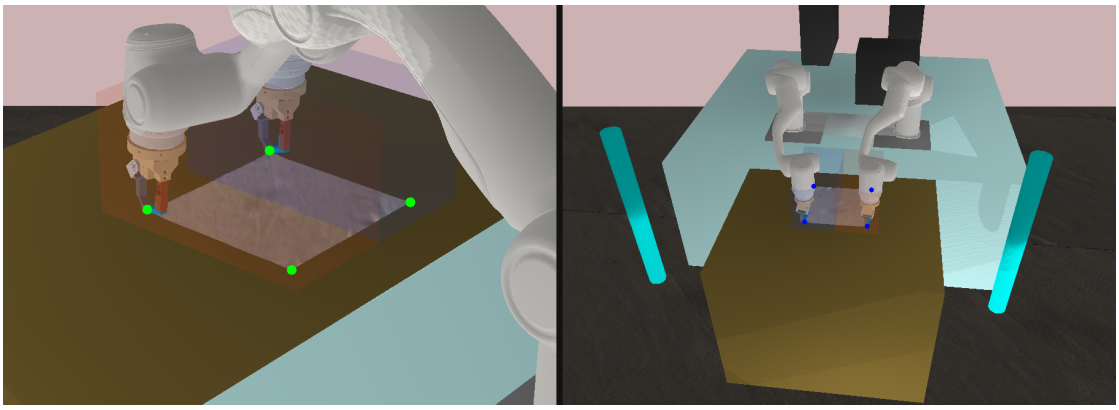


Figure 4.7.: First step in PyBullet of the Diana cloth environment with workspace box

Workspace boundaries are computed from the current cloth-corner positions plus a fixed padding. These bounds are converted into absolute per-arm axis-aligned ranges in Cartesian space and used to clip the target positions of the gripper control points. As shown in Fig. 4.7, each arm is assigned its own workspace. Because cloth size can vary across episodes, the workspace bounds are recomputed from the current cloth geometry as needed. Both implementations include watchdog checks so that slow or numerically unstable substeps are detected and terminated early to keep long training runs stable [6]. Table 4.1 summarizes the environment variables used to configure dual-arm control, grasp selection, fold direction, and the BioIK controller.

4.2. Cloth / Soft-Body Modeling

The cloth is modeled as a discretized deformable object in each simulator, using the native deformable-body mechanisms of MuJoCo and Bullet, while exposing the same task-level keypoints and corner indexing across backends [43,45]. In the reference pipeline by Hietala et al., the MuJoCo simulator represents the cloth as a 2D grid of bodies connected via a spring–damper system, with one corner connected to the robot end-effector to model the initial grasp. Between policy outputs, the robot executes intermediate targets at a higher control rate than the policy, so stable cloth dynamics at physics substeps are important for learning [5]. Cloth motion results from internal deformation forces and external/contact forces, consistent with the node-based discretization introduced in Eq. (2.12) [37]. The environment exposes a 3×3 subset of cloth keypoints as MuJoCo sites named `S{i}_{j}` with $i, j \in \{0, 4, 8\}$, corresponding to the four corners, four edge midpoints, and the center of the 9×9 grid [6,7]. The folding process is visualized in Fig. 4.8.

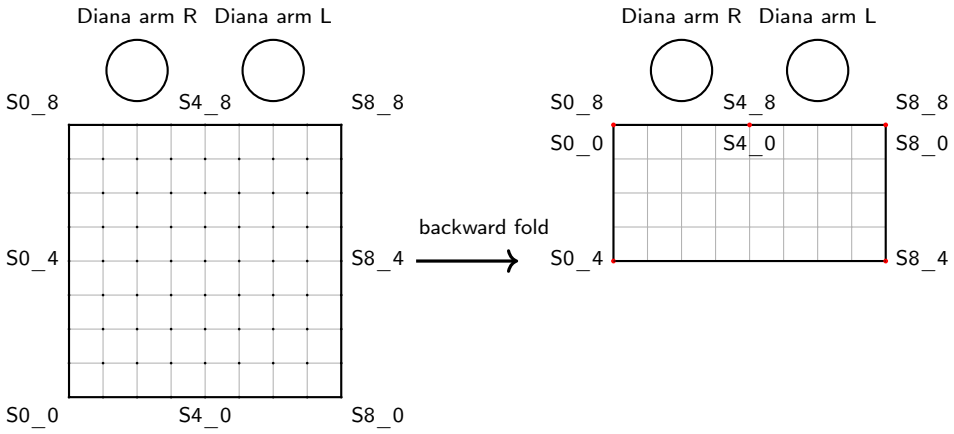


Figure 4.8.: Schematic of the sideways backward fold: keypoints with $j = 0$ and $j = 8$ overlap after folding.

In the reference implementation, the composite-grid spacing is computed from the sampled square cloth side length `cloth_size` [7]. In the Diana MuJoCo environment, `DIANA_RANDOMIZE_SIZE` enables rectangular cloth sampling by randomizing only the Y side length. The X side length is fixed to $0.30\,\mathrm{m}$, and the sampled `cloth_size` parameter is interpreted as the Y side length [6]. Accordingly, `geom_spacing` is computed from the fixed X span, and after model load all cloth node positions are scaled along Y around an anchored corner (`B8_0` for backward folds and `B8_8` for forward folds) [6]. Fig. 4.9 summarizes this fixed-X and stretched-Y initialization used in the Diana environments, followed by a tendon rest-length reset in code so the cloth starts approximately relaxed [6].

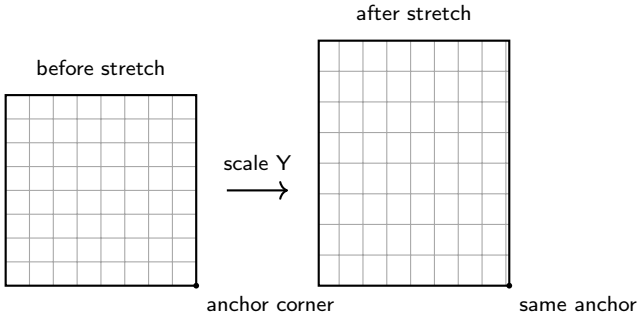


Figure 4.9.: Schematic of the cloth initialization. Y is stretched around an anchored corner.

4.2.1. MuJoCo: Modeling Soft-Body Dynamics

In the current Diana MuJoCo scene and the reference implementation, the cloth is specified in MJCF (MuJoCo XML) using a `composite` grid [45]. Concretely, the cloth is a 9×9 grid (`count="9 9 1"` in MJCF for `composite type="grid"`), with a `joint kind="main"` sub-element specifying the degrees of freedom for each generated body [7, 46]. Two tendon families couple the grid: `tendon kind="main"` forms axis-aligned connections, and optional diagonal `tendon kind="shear"` resists shear deformation. A `skin` sub-element defines the rendered surface mesh and texture coordinates for visualization [46]. This yields a mass–spring style cloth approximation, where the composite tendons act as length constraints with spring–damper parameters [5]. MuJoCo computes collisions using the per-element geoms generated by the composite, while the `skin` is a visualization surface. The `skin` is only used for rendering [46].

The MuJoCo modeling guide classifies `composite` as an MJCF mechanism that generates repeated structures from a compact specification. Recent MuJoCo documentation recommends using 2D `flex` deformables for cloth-like thin sheets and notes that the legacy composite `grid/cloth` mechanisms are deprecated or removed in newer releases [45]. For this thesis, the composite-grid cloth model is retained to preserve task and API parity with the original pipeline. Both implementations use the same 3×3 keypoint subset (corners, edge midpoints, center) for observations and constraint evaluation, as illustrated in Fig. 4.8.

4.2.2. PyBullet: Deformable Body Simulation and Porting

To use PyBullet's deformable-body mode, the simulation is reset with `resetSimulation(p.RESET_USE_DEFORMABLE_WORLD)` [1]. After that, gravity and the simulation time step are set and solver parameters can be adjusted through `setPhysicsEngineParameter`. In PyBullet, `stepSimulation` advances the simulation by one step. The time step itself and the solver settings are set through `setTimeStep` and `setPhysicsEngineParameter` [1]. The cloth is instantiated as a soft body via `p.loadSoftBody` in `DeformableCloth`. Each cloth instance uses a triangular surface mesh stored in Wavefront `.obj` format. Textures are added to an `.obj` using `.mtl` to define the visual appearance. The pre-baked grid meshes were recreated based on PyBullet's `cloth_z_up.obj` from `pybullet_data`. The mesh files were generated

locally, but their triangle layout follows PyBullet's `cloth_z_up.obj`. That original mesh is a 5×5 grid with 25 vertices and 32 triangles. The meshes used here are 9×9 grids with 81 vertices and 128 triangles so they line up with the MuJoCo-side site indexing. Cloth texture images and material definitions were reused from the original reference implementation. The textured mesh assets are stored as Wavefront `.obj/.mtl` files, and the code rewrites `map_Kd` paths so textures still resolve after mesh caching. Cloth meshes were pre-baked so textured cloth can be rendered via EGL. PyBullet supports headless GPU rendering via an EGL renderer plugin on Linux, which enables OpenGL rendering without an on-screen GUI [1]. It replaces the CPU-only `TinyRenderer` in headless setups. Physical properties like mass, friction, and stiffness are passed to `loadSoftBody` [1, 6].

It was a manual process to determine the correct values. The reference environment from Hietala et al. was reconstructed in PyBullet, and the provided `demos.csv` trajectory was played back repeatedly. After each run, parameters were tweaked until the fold looked close to the MuJoCo version. Particular attention was paid to ensuring that the final corner error and the folding progression were similar. These values are summarized in Table 4.2 in the column "Reference reconstruction". The result of the reference reconstruction was a fairly stiff cloth.

These values were intentionally adjusted to make the cloth's dynamics smoother. Values in the "Current Diana PyBullet default" column in Table 4.2 show the current default values. The time step was increased from $1/210\,\mathrm{s}$ to $1/150\,\mathrm{s}$ to speed up training while keeping the simulation stable. The cloth mass was reduced, while the cloth friction and table lateral friction were increased. The biggest change on the cloth side is the reduction in damping stiffness and bending stiffness. The current backend also uses a smaller collision margin and sets `settle_steps` to 0. In this thesis, the collision margin is tied to the local mesh spacing rather than the full cloth side length. The local edge length is therefore the more relevant scale for soft-body contact than the global cloth size.

Table 4.2.: Comparison between the reference-reconstruction settings and the current Diana PyBullet defaults.

Parameter	Reference reconstruction	Current Diana PyBullet default
Task name	`sideways`	`sideways_backward`
Cloth size	$0.20\,\mathrm{m}$	$0.30\,\mathrm{m}$, with Y-size range $[0.30, 0.40]$ when `DIANA_RANDOMIZE_SIZE=1`
Control frequency	$10\,\mathrm{Hz}$	$10\,\mathrm{Hz}$
PyBullet timestep	$1/210\,\mathrm{s}$	$1/150\,\mathrm{s}$
`ctrl_filter`	0.03	0.03
`output_max`	$0.03\,\mathrm{m}$	$0.03\,\mathrm{m}$
`success_distance`	$0.05\,\mathrm{m}$	$0.05\,\mathrm{m}$
`max_close_steps`	10	15
Cloth mass	0.5	0.3 $(0.25, 0.35)$
Cloth friction	0.6	1.1 $(1.0, 1.2)$
Elastic stiffness	80.0	80.0 $(75.0, 85.0)$
Damping stiffness	1.1	0.12 $(0.11, 0.13)$
Bending stiffness	10.0	0.06 $(0.055, 0.065)$
Self-collision / face contact	on / on	on / on
Settle steps	80	0
Collision margin	$\approx 0.006\,\mathrm{m}$	$\approx 0.003\,\mathrm{m}$, set to $0.08 \cdot$ edge_length, where `edge_length` refers to the local mesh edge length for the $0.30\,\mathrm{m}$ cloth
Table lateral friction	1.8	2.4
Gravity	$(0, 0, -9.81)\,\mathrm{m/s^2}$	deterministic fallback $(0, 0, -9.81)\,\mathrm{m/s^2}$, randomized in $[-10.1, -9.5]$ along Z
ERP / contact ERP	$0.26/0.28$	$0.26/0.28$
`numSolverIterations`	50	50
`globalCFM`	10^{-5}	10^{-5}
`solverResidual Threshold`	10^{-5}	10^{-5}
`contactBreaking Threshold`	0.035	0.035

The default cloth size is $2\,\mathrm{m} \times 2\,\mathrm{m}$. The mesh is rescaled to a target span `target_span_xy` before loading, with X fixed and Y randomized in this backend. The rescaling is performed programmatically before each episode rollout. Before each rollout, the code overwrites the cloth settings for mass, friction, elastic stiffness, damping, bending stiffness, and collision margin. Clothing in PyBullet is created by the engine's soft-body class `btSoftBody`. Two-way interaction between rigid bodies and other collision objects are supported. The class provides structures for nodes, connections, surfaces, and anchors [47].

Reward and target definitions remain the same as in MuJoCo by mapping Bullet vertices onto MuJoCo-style sites `S{i}_{j}` on a 9×9 grid. The mapping preserves the corner ordering `S0_8, S8_8, S0_0, S8_0` used by the task definitions. This provides a deterministic site-to-

vertex mapping for extracting corner, edge-midpoint, and center keypoints. Task constraints are generated through the same `task_definitions` interface as in the MuJoCo path [6].

The control path remains consistent across engines. Target filtering and BioIK-based joint reference updates are shared across engines. In the PyBullet backend, `calculateMassMatrix` returns the joint-space mass matrix of the full system. For control, only the 7×7 submatrix for the seven joints of each arm is used. Before rollout, the grasped vertices are pinned back to the end-effector targets to remove residual drift. With the simulation-mesh flag enabled, `getMeshData` returns the current cloth mesh, including the node positions at that step [1].

4.2.3. MuJoCo vs. PyBullet

The theoretical engine comparison has already been discussed in Section 2.3.2. This subsection compares the two backends at the implementation level and highlights the design decisions that matter for sim-to-sim evaluation in this work.

The PyBullet backend is implemented as a port of the MuJoCo environment that preserves the same external environment interface (reset/step semantics, observation dict, and action format). The same policy and evaluation code can switch between engines by changing `PHYSICS`, without requiring changes to the environment wrapper used by the RL pipeline. The entry point `train.py` selects between `ClothEnv` and `ClothEnvBullet` based on `PHYSICS`. Both environments expose the same observation keys and action interface, so the RL stack with SAC+HER pipeline remains compatible [6]. The BioIK-based torque control interface are kept identical across engines, As described in Section 4.1.2, both backends share the same dual-arm initial assumptions and the same coupled-vs-independent action switch. This preserves Hietala et al.'s feedback control structure, where policy updates run at a lower rate than physics/control update [5].

Table 4.3.: Implementation-level MuJoCo–PyBullet comparison used for sim-to-sim evaluation.

Aspect	MuJoCo backend (`cloth_env`)	PyBullet backend (`cloth_env_pybullet`)	Sim-to-sim relevance
Backend selection	Selected by `PHYSICS=mujoco` in `train.py`.	Selected by `PHYSICS=bullet` in `train.py`.	Only physics backend changes.
Cloth representation	Cloth is built from an MJCF composite grid (spheres + equality-constrained tendons) with indexed cloth sites [45].	Cloth is loaded via `p.loadSoftBody` from mesh files (`.obj`) and remapped to MuJoCo-style `S{i}_{j}` sites [1].	Goal extraction is comparable while internal deformable models differ.
Cloth visualization	Composite skin is used for rendering. Collisions use the element-body geoms [45].	Deformable mesh for rendering, with vertex access via `getMeshData` [1].	Same keypoints. Internal representations differ [34, 45].
Default deformable model	Determined by composite generator and constraint settings [45].	Position-based dynamics by default, with optional FEM mode [1].	Both engines support multiple deformable models. Default is engine-specific.
Grasp mechanism	Cloth-corner bodies become mocap bodies and are synchronized via `mjData.mocap_pos` [35].	Grasp vertices are anchored to kinematic helper bodies that follow the gripper links each substep [1].	Both use kinematic corner control, but with engine-specific constraint primitives.
Physics stepping	Control loop uses a split `mj_step1`/`mj_step2` forward-dynamics pipeline, derived from positions and velocities computed in the first phase [42].	The control loop advances with the standard Bullet stepping path, with separate timestep and solver configuration.	Integration and constraint-solving details is backend-specific with same high-level control logic [34, 35].
Controller dynamics terms	BioIK torque controller queries MuJoCo Jacobians (`mj_jacSite`) and mass matrix (`_mj_fullM`) [42].	The same controller queries Bullet Jacobians and the joint-space mass matrix [1].	High-level controller logic is shared while dynamics quantities are engine-provided.
Runtime safeguards	Watchdogs and cloth-health checks terminate unstable episodes. Implementation applies additional memory cleanup to prevent long-run instability.	Equivalent watchdogs are used, plus periodic hard resets to reclaim deformable-world memory [6].	Both engines are stable on long-run. Recovery mechanisms differ.

The table 4.3 shows that the environment interface of the two backends is consistent. However, deformable body internals and the implementation of constraints remain engine-specific. In this thesis, sim-to-sim evaluation measures robustness to differences between simulation backends. In our setting, the dominant engine gap is expected to come from the deformable-body model and the grasp/contact constraints rather than from the task API. Prior work reports that generalization across physics engines can be limited even when the task definition is unchanged, and cloth benchmarks evaluate simulator-dependent accuracy and stability under table contact [40, 66].

4.3. Porting the Simulation

The goal of the port is to keep the learning pipeline, the task interface, and the policy interface unchanged, while allowing the backend to switch between MuJoCo and PyBullet. Since both implementations use the same observation structure, action interpretation, control logic, and tensor shapes, the RL pipeline remains unchanged. MuJoCo and PyBullet use different mechanisms for scene description, so the port is non-trivial.

The starting point for the scene is the `diana7_dual_setup.xacro` file from the `tams_diana_7` repository, which contains the complete dual-arm setup except for the cloth definition [67]. To use the `tams_diana_7` stack, the repositories listed in Table 4.4 must be built in the Catkin workspace.

Table 4.4.: Software stack for dual-arm Diana7 scene

Repository	Short description
`tams_diana_7`	Collection of packages for the dual-arm setup (bringup, description, MoveIt configs, teleop, tools) [67].
`libdiana7`	C++ wrapper around the C DianaAPI, supporting robot-control functionality [68].
`diana7_ros`	Core ROS driver for the Diana7 robot [69].
`q_gripper`	Q-gripper model and control package [70].
`agile_gripper`	Provides package `diana_gripper` as optional alternative gripper/fingertip stack [71].
`realsense-ros`	Provides `realsense2_description` for wrist-camera Xacro includes (D435) [72].

The Xacro description was converted into a standalone URDF so that it can be loaded by PyBullet. For this purpose, the utility `xacro_to_urdf` was used to generate a self-contained URDF with resolved package paths [73]. Small inertials were added to each link that did not define inertial parameters in the generated URDF to avoid PyBullet warnings. The visuals of the wrist cameras were removed from the URDF so that they are not rendered in simulation, allowing for an unobstructed view of the cloth. Since a visible gap appeared between the gripper fingertips and the gripper base, the z mounting offset of `q_gripper_[l/r]_base_joint` was changed from -0.016 to 0.004 to close the gap.

The patched URDF was converted into an MJCF file with the MuJoCo framework, so that it can be loaded by the MuJoCo backend. Starting from the reference MuJoCo scene

arena.xml, the Franka Emika Panda arm was replaced by the dual-arm Diana7, while keeping link and asset names aligned with the original scene where possible. Geoms that were originally specified without an explicit body frame were wrapped into dedicated bodies such that their position, name, and group match the original geom definition in arena.xml. The loading of textures and materials, including the floor/ground and cloth, was largely retained. The domain randomization parameter template was also taken from the reference implementation. The meshes for the Diana arm were taken from the repositories in Table 4.4. Contact exclusions were taken from the diana7.srdf of the tams_diana_7 repository and entered directly into the MJCF XML. For the PyBullet implementation, the same contact exclusions were set programmatically.

Some visuals in the MJCF XML were removed to improve the view of the cloth in the training camera. As in Section 4.1.3, this includes removing the yellow workspace posts whose collision geometry otherwise appears in the rendered image.

In both the MJCF XML and the URDF, the gripper fingers were closed and locked. To achieve this, the joints of the fingers were changed from revolute to fixed in the URDF. In the MJCF XML, the joint was completely removed and only the geoms of the fingers were retained in order to follow the reference implementation.

Each mesh used in the URDF was duplicated and referenced only once, because the EGL renderer can crash with a Segmentation fault when multiple geoms refer to the same mesh.

4.3.1. BIOIK Integration

In order to use BioIK in both backends, the corresponding ROS packages from the bio_ik and bio_ik_service repositories in Table 4.3.1 were integrated into the Catkin workspace. After integration, the workspace was built [31, 62].

BioIK software stack

- bio_ik: MoveIt kinematics plugin implementing the BioIK solver [62].
- bio_ik_service: ROS service node exposing BioIK through /bio_ik/get_bio_ik and compatibility interfaces [31].

In order to successfully build and use the BioIK packages, it is necessary to install ROS and load the ROS environment (e.g., source /opt/ros/noetic/setup.bash). The Diana7 MoveIt demo setup can be started with the following command:

```
roslaunch tams_diana7_dual_moveit_config demo.launch
   moveit_controller_manager:=fake use_rviz:=false
```

BioIK service node can be started with the following command:

```
rosrun bio_ik_service bio_ik_service
```

MoveIt and the standard ROS messages/TF components are the primary ROS dependencies for this stack. The scene diana7_environment.xacro used by bio_ik_service contains the complete definition of the environment. In that scene, an extra table link was lowered

from $0.80\,\mathrm{m}$ to $0.70\,\mathrm{m}$ (size z: $0.79 \rightarrow 0.69$, origin z: $0.01 \rightarrow -0.04$) so that BioIK can find collision-free solutions. The workspace limitation from Figure 4.4 alone ensures that the grippers cannot reach through the table.

4.3.2. Mapping between MuJoCo Templates and PyBullet World

The resulting URDF file `diana_patched_path.urdf` and the MJCF file `arena_diana.xml` from Section 4.3 are loaded into the respective backends as scene descriptions. The MuJoCo implementation was always performed first, and PyBullet was ported afterwards to ensure the same scene and parameters. In MuJoCo, the MJCF is instantiated by rendering a parameterized XML template at reset time and compiling it into an `mjModel` [46]. In PyBullet, the URDF is loaded and the scene is procedurally constructed from rigid components (URDF and primitives), while the cloth is loaded as a soft body via `p.loadSoftBody` and the plane is initialized via `p.loadURDF("plane.urdf")` [1]. An important step in porting is extracting a canonical scene specification from the MuJoCo template and recreating it in PyBullet. This encompasses four aspects: (i) world frame system and units, (ii) rigid scene geometry (table, floor, robot base), (iii) camera parameters, and (iv) cloth spawn pose and geometry. To achieve a template-based scene setup in PyBullet, the template parameters for camera, lighting, appearance, and other settings are defined in `bullet_model_kwargs.py`. World creation then uses PyBullet's standard calls for resetting and stepping in deformable mode, namely `p.resetSimulation`, `p.setTimeStep`, `p.setPhysicsEngineParameter`, `p.loadURDF`, and `p.stepSimulation` [1].

World frame and units. Both backends use the International System of Units (meters, kilograms, seconds) and a right-handed coordinate system with z pointing upwards. Accordingly, we treat both backends as sharing the same world coordinate frame W, and all logged positions are expressed in this frame.

Rigid scene geometry The table position `_ARENA_TABLE_POS` and size `_ARENA_TABLE_HALF_EXTENTS` are taken from the MuJoCo scene and reused in PyBullet to ensure identical table placement without querying simulator-specific geometry. The value `_ARENA_CLOTH_OFFSET_BASE`, which defines the relative position of the garment to the table, was also taken from the MuJoCo implementation to ensure the same starting position of the garment. Although both backends ultimately originate from the same Xacro/URDF description, small backend-specific adjustments (e.g., collision-only helper geometry) are applied explicitly where required.

Camera parameters Camera mapping preserves the MuJoCo views by reusing the same absolute camera poses for `side` and `front` in the PyBullet camera configuration. The policy observation pipeline always uses the `side` camera with a defined field of view (`fovy`). Both backends also use the same target point (MuJoCo: `lookatbody`; PyBullet: a per-episode look-at point) corresponding to the cloth center. In PyBullet, this is implemented

using `p.computeViewMatrix` and `p.computeProjectionMatrixFOV` to match the MuJoCo camera setup [1].

Cloth spawn pose and geometry In MuJoCo, the composite cloth depends on the cloth size and the composite element radius (`geom_size`). This has been retained in PyBullet. MuJoCo's cloth size logic, where the X side is fixed and the Y side is variable, has also been retained in PyBullet. The spawn position is computed using the same corner-based convention so that the reference corner remains consistent across cloth sizes, yielding:

$$p_{\text{spawn}} = \left(x_0 - \tfrac{1}{2}\,\texttt{span}_x,\ y_0 + \tfrac{1}{2}\,\texttt{span}_y,\ z_0\right),$$

where (x_0, y_0, z_0) is the arena offset base position and $\texttt{span}_x$ and $\texttt{span}_y$ are the X and Y side lengths of the garment. Constraint evaluation remains backend-consistent by keeping MuJoCo-style cloth site names `S{i}_{j}` and the same corner ordering. In both cases, a settling phase is carried out so that the garment lies flat on the table. Other engine-specific details are intentionally retained to preserve the sim-to-sim domain shift; these are discussed in Section 2.3.2.

4.3.3. Handling Mocap and Grasping Constraints

This thesis focuses on learning dynamic folding policies rather than solving grasp planning. Following Hietala et al., each episode starts from a grasped cloth state so that learning focuses on the folding motion rather than grasp acquisition [5]. Compared to the reference implementation, which connects a single cloth corner to the end-effector, the same grasp abstraction is applied to two corners simultaneously for bimanual grasping [5]. The grasping mechanism should enforce the same high-level rule in both backends. Grasping is modeled as a prescribed kinematic constraint, so the grasped corners track the commanded gripper targets while the constraint target itself is not modified by cloth reaction forces [6, 46]. Figure 4.10 summarizes this backend-aligned grasp abstraction.

For bimanual grasping in MuJoCo, the two selected cloth-corner bodies are converted into mocap bodies and synchronized with the left and right gripper control points during stepping. Grasping is modeled as a kinematic constraint by removing joints for the selected corner bodies and driving their mocap poses. At each physics substep, the current gripper control-point position is written to `mjData.mocap_pos` so the cloth corners follow the grippers. A mocap body in MJCF is created by specifying `mocap="true"` in the body definition, which is only permitted for bodies that are children of `worldbody` and have no joints. MuJoCo sets the position and orientation of mocap bodies from `mjData.mocap_{pos,quat}` at each time step. This implements a one-way kinematic attachment because mocap bodies are fixed from the viewpoint of dynamics while their pose is prescribed externally. The cloth is driven by the prescribed mocap targets, and the mocap targets are not accelerated by cloth reaction forces through the mocap mechanism [6, 35, 46]. The MuJoCo-side data flow is illustrated in Figure 4.10.

To match MuJoCo's mocap-style (one-way) grasping in PyBullet, we avoid anchoring cloth vertices directly to the gripper links. Instead, kinematic proxy bodies are created, the grasp

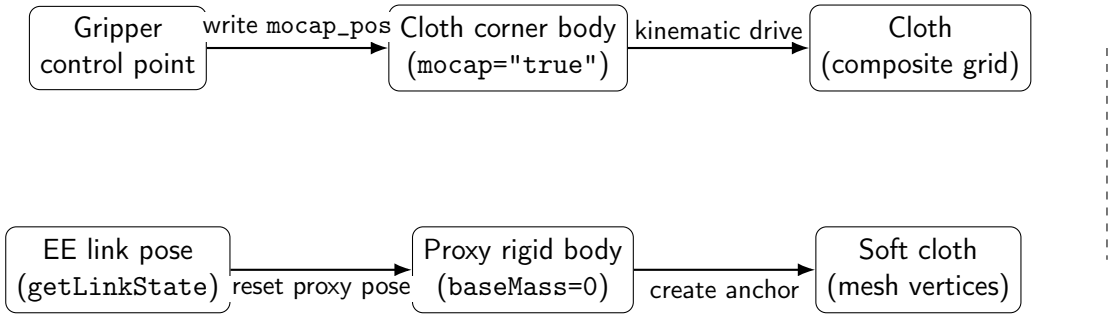


Figure 4.10.: Grasp constraint abstraction used for sim-to-sim parity. Top: MuJoCo. Bottom: PyBullet.

vertices are anchored to these proxies using `createSoftBodyAnchor`, and the proxies are synchronized to the gripper link poses at every physics substep [43]. This mirrors MuJoCo's mocap corner concept using Bullet primitives such as `createMultiBody` with `baseMass=0`, `getLinkState`, and `resetBasePositionAndOrientation` [6]. The PyBullet Quickstart Guide uses `baseMass=0` in `createMultiBody` examples to denote a static body and states that `resetBasePositionAndOrientation` overrides physics simulation while setting the base linear and angular velocity of that proxy body to zero [1]. That only affects the kinematic helper body, not the cloth vertices themselves. The cloth can still acquire velocities through various effects such as anchors, gravity, contacts, and internal soft-body dynamics. The corresponding PyBullet proxy-and-anchor path is also shown in Figure 4.10.

Both implementations perform a short settle phase after reset by stepping the physics for several substeps before starting the episode. This reduces motion after initialization so the episode starts from a consistent resting configuration on the table [6, 7]. As described in Section 4.1.2, a grasp validity check is performed at the beginning of a rollout, measuring the distance between the grasp points and the assigned corners. Let $d_L = \|p^W_{\text{grip},L} - p^W_{\text{corner},L}\|_2$ and $d_R = \|p^W_{\text{grip},R} - p^W_{\text{corner},R}\|_2$, and accept the initialization if $\max(d_L, d_R) \leq \varepsilon_{\text{grasp}}$ holds. This yields a backend-consistent bimanual grasp initialization for the policy learning phase [6]. Apart from the prescribed grasp constraint, we rely on each simulator's native contact and deformable-body model, so remaining differences reflect the underlying engines [1, 45].

4.4. Domain Randomization (DR) Strategies

This section describes the domain randomization (DR) strategies implemented in the training pipeline. The basics and objectives of DR have already been explained in Section 2.3.3 and Equation (2.13). In summary, DR trains the policy across varied environment instances to improve robustness [49]. In this thesis, DR is used so that a single policy can fold different cloth instances. The DR implementation follows the categories used by Hietala et al.: physical, visual, and temporal randomization [5].

Domain randomization is controlled via the environment variable `DR`. If `DR=0` is set, the DR flags for dynamics, materials, camera, lighting, look-at, gravity, texture, and albumentations

are disabled. If `DR=1` is set, the backend uses its default DR configuration.

Table 4.5.: Implemented DR variables used for training.

Category	MuJoCo backend (`cloth_env.py`)	PyBullet backend (`cloth_env_pybullet.py`)
Activation and timing	A training wrapper applies DR at reset (`randomize_on_reset=True`, `randomize_every_n_steps=0`), so parameters are sampled once per episode. A new MJCF model is compiled from the `arena_diana.xml` template using episode-specific `model_kwargs`.	DR is applied in `reset()`, and parameters are sampled once per episode and then kept fixed during the rollout (equivalent to the MuJoCo schedule).
Cloth size randomization	Gated by `DIANA_RANDOMIZE_SIZE=1`; X is fixed to $0.30\,\mathrm{m}$, Y is sampled and then applied via a post-load Y-stretch (see Section 4.2).	Also gated by `DIANA_RANDOMIZE_SIZE=1`; X is fixed to $0.30\,\mathrm{m}$, Y is sampled from `cloth_size_range` and applied via mesh span scaling before `loadSoftBody`.
Dynamics and cloth physics	A random row from `data/model_params.csv` is sampled when `dynamics_randomization=1`, affecting cloth and contact parameters such as `friction`, `geom_size`, `impratio`, `cone_type`, and MuJoCo `solimp`/`solref` terms.	Every reset re-samples the cloth setup `mass`, `frictionCoeff`, all spring stiffness terms (`springElasticStiffness`, `springDampingStiffness`, `springBendingStiffness`), and with `geom_size` and gravity.
Visual appearance	Material assignments (floor/table/cloth) are randomized. Procedural texture parameters are sampled. Robot link RGBA values are randomized per reset.	Table/floor/robot colors are randomized. Cloth meshes and textures are sampled from the configured asset set. Cloth tint can be randomized.
Camera and lighting	`train_camera_fovy` is sampled from `fovy_range`. Look-at target is jittered.	Per episode, camera FOV, eye jitter, look-at jitter, and light direction/color are sampled.
Temporal and image perturbation	The image-capture substep is sampled from a Gaussian distribution over substep indices. Optional Albumentations augmentations are applied before converting the observation to the grayscale policy input.	The same Gaussian substep sampling is used before converting the observation to the grayscale policy input. Albumentations are applied with the same configuration.

Table 4.5 shows that the DR strategies in both backends are structured similarly. The MuJoCo backend combines XML-level randomization of material and contact parameters with visual randomization and temporal observation perturbations [5]. This corresponds to static, episodic DR in the taxonomy of Muratore et al. [38]. The PyBullet implementation

retains the same high-level categories. When DR is disabled in PyBullet, a deterministic MuJoCo-like scene is generated. When DR is enabled, solver parameters are intentionally kept deterministic, while additional material and gravity randomization is applied in PyBullet. This keeps the policy interface consistent across backends, while preserving engine-specific deformation and contact behaviors for sim-to-sim transfer tests [40]. The BioIK controller parameters themselves stay fixed regardless of whether DR is on or off [6].

5. Approach

5.1. System overview

Based on the environment and controller architecture described in Chapter 4, this thesis adopts the visual-feedback policy–controller loop for dynamic cloth folding introduced by Hietala et al. This loop is used as a common interface for bimanual sim-to-sim evaluation across MuJoCo and PyBullet. The basic data flow is shown in Figure 5.1. At each policy step, the agent observes a rendered camera image I_t, the goal g, and, depending on the selected robot-observation mode, additional robot state or the previous action a_{t-1}. The policy then outputs a Cartesian end-effector displacement action a_t to the controller. The displacement action is integrated into the next Cartesian target by calculating $x_{t+1} = x_t + a_t$. Between policy steps, intermediate targets are smoothed with the first-order filter in Eq. 5.1 [5].

$$x_{d,k+1} = \lambda \cdot x_{t+1} + (1 - \lambda) \cdot x_{d,k} \tag{5.1}$$

$x_{d,k}$ is the current intermediate target and x_{t+1} is the next target requested by the policy. λ sets how quickly the intermediate target moves toward it. These intermediate targets are tracked at the faster control rate, which keeps the motion smooth. Compared to the reference implementation by Hietala et al., this thesis converts each Cartesian target into joint-space references using BioIK through a ROS service interface. Both backends go through the same IK call path, so the policy sees the same high-level interface in either case [5].

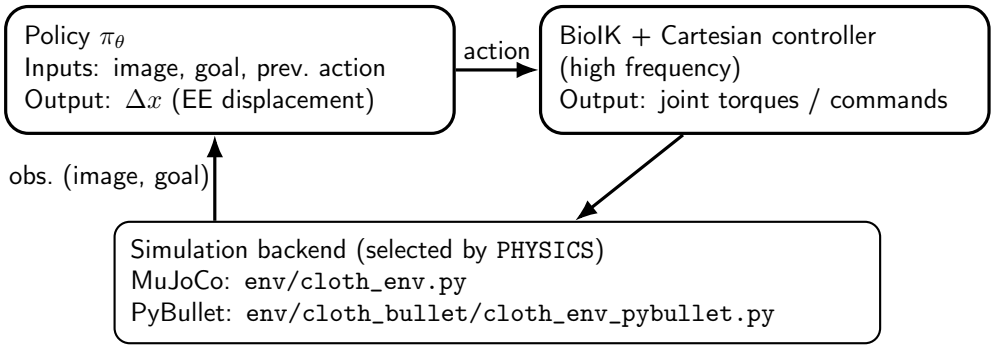


Figure 5.1.: System overview (adapted from the policy–controller loop in Hietala et al., Fig. 2) [5].

The individual components and design decisions are described in detail below, following the architecture shown in Figure 5.1.

5.1.1. Task definition and success criteria

The task definition by Hietala et al. serves as the basis for adaptations and extensions in this section [5]. In their task formulation, success is defined as the mean corner-to-goal distance d_{sum} metric in Equation (5.2) being below a threshold δ at the end of the episode [5].

$$d_{\text{sum}} = \frac{1}{C} \sum_{i=0}^{C-1} \|p_i - g_i\|_2 \tag{5.2}$$

Where p_i is the position of an evaluated cloth point, g_i is the corresponding target position, and C is the number of evaluated points (so d_{sum} is the mean of these distances) [5].

Figure 4.8 visualizes the cloth keypoint indexing $S\{i\}_\{j\}$ used in this thesis. The desired folded configuration is defined by a list of geometric constraints $\{(o_k, t_k)\}_{k=1}^{K}$ between origin sites o_k and target sites t_k. During reset, the desired goal vector g is formed by concatenating all target-site positions in the internal frame I. Only for constraints that define `noise_directions`, an episode-wide noise term is added. For each step t, the achieved goal vector a_t^g is defined as the concatenation of the current origin-site positions in the same internal frame I. In this thesis, two fold-direction variants are provided: `sideways` (forward) and `sideways_backward` (backward). Each variant uses six primary constraints and eight intermediate edge constraints, i.e., 14 constraints in total.

Let $p_t^k \in \mathbb{R}^3$ and $g^k \in \mathbb{R}^3$ be the achieved and desired positions of the active success corners $k \in \mathcal{C}$, and let δ be the `success_distance`. Then `is_success` is true if and only if the condition in equation (5.3) is satisfied.

$$\forall k \in \mathcal{C} : \ \|p_t^k - g^k\|_2 < \delta \tag{5.3}$$

In Hietala et al., success is evaluated at the end of the episode by checking whether the mean distance $d_{\text{sum}} < \delta$ [5]. In this thesis, episode termination uses the counter `episode_ee_close_steps` together with `success_distance` and `max_close_steps`. The counter is incremented as long as `is_success` is true, i.e., all grasped corners selected by `done_success_keys` are within `success_distance` (see Equation (5.3)). The episode ends as soon as it reaches `max_close_steps`. If counting has started and `is_success` becomes false, both backends set a failure state (`_left_close_range=True`) and from then on return only `fail_reward`. This prevents short, transient threshold crossings from being counted as stable folds.

Both implemented backends report per-corner distances as `corner_0..corner_3` and their sum as `corner_sum_error`. In the PyBullet backend, the same sum is additionally exposed as `dsum`. The reference implementation links `is_success` to a reward threshold. This thesis decouples success from reward, so success is purely geometric.

5.1.2. Controller and BioIK integration

Control architecture

In both backend implementations, the per-action Cartesian target `desired_pos_step_W` is updated once per policy action during an environment step update. The control target

`desired_pos_ctrl_W` is updated between physics substeps by repeatedly applying equation (5.1) [6]. The reference implementation, on the other hand, updates the filtered Cartesian target directly and passes it to the OSC controller [5].

The `osc` branch of the Diana repository contains an earlier implementation of the Diana scene with an OSC controller [74]. That version was useful as a stepping stone toward the later BioIK integration. However, the movement in the training is not robust enough, due to collisions with the yellow posts and between the arms in the bimanual version. BioIK is used to avoid these collisions.

In the first substep of a policy step, the BioIK solve is attempted, which is gated by a periodic update schedule (`DIANA_BIOIK_IK_UPDATE_EVERY_N_STEPS`). BioIK is used to compute a joint-space posture reference q_{ref} for the current Cartesian target. The shared hybrid operational-space torque controller then uses q_{ref} as a null-space regularization target. If no solution is found, the previous q_{ref} is retained and pursued further. Redundant solves are skipped if the filtered target has changed by less than $10^{-4}\,\mathrm{m}$. At each physics substep, a slew-rate limit and absolute saturation are applied to the calculated joint torques. In MuJoCo, the controller does not write directly into `mjData.qfrc_bias`. That field already contains MuJoCo's bias forces for the current state, i.e., the gravity and internal forces. Instead, after `mj_step1` these values are copied into `mjData.qfrc_applied`, which is the array for externally applied generalized forces. The per-arm controller torque is then added there before `mj_step2`. In PyBullet, the controller can calculate a bias torque with `calculateInverseDynamics` using zero joint velocities and accelerations [6].

Table 5.1 lists the BioIK and hybrid-OSC parameters, which have different default values in the two backends.

Parameter	Environment variable	MuJoCo default	PyBullet default
Minimal-displacement weight	`DIANA_BIOIK_MIN_DISP_WEIGHT`	0.5	0.03
Joint-regularization weight	`DIANA_BIOIK_JOINT_REG_WEIGHT`	0.5	1.0
Task-space stiffness	`DIANA_BIOIK_OSC_KP_POS`	400.0	350.0
Task-space damping ratio	`DIANA_BIOIK_OSC_DAMPING_RATIO`	1.0	0.8

Table 5.1.: BioIK and hybrid-OSC parameter defaults that differ between backends [6].

Robot kinematics

A Cartesian tracking point `ee_site_name` is defined for each arm (a MuJoCo site or a PyBullet link), while BioIK is solved for the corresponding link-7 body `link7_name`. Let p_W^{ee} be a desired end-effector target in world frame W, p_W^{link7} the corresponding link-7 position goal, R_W^{link7} the rotation matrix that maps vectors from the link-7 frame to the world frame W, and $r_{\mathrm{link7}\to\mathrm{ee}}^{\mathrm{link7}}$ the link-7 to EE offset in the link-7 frame.

The goal is now to convert the desired end-effector target into the link-7 position goal, which is shown in the steps in Figure 5.2.

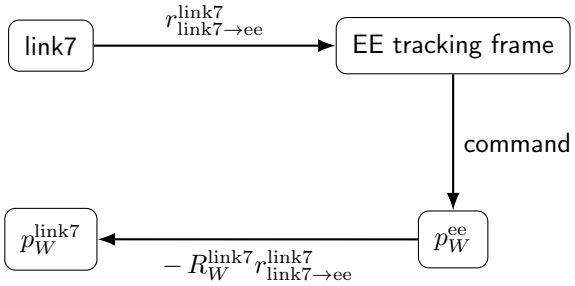


Figure 5.2.: Mapping from an end-effector target p^{ee}_W (controller target) to a link-7 target p^{link7}_W (BioIK goal) using link7-to-EE offset and Rotation Matrix R^{link7}_W [6].

First, the BioIK implementation caches the offset. Then, the following equation is applied to calculate the link-7 target:

$$p^{\text{link7}}_W = p^{\text{ee}}_W - R^{\text{link7}}_W r^{\text{link7}}_{\text{link7}\to\text{ee}} \tag{5.4}$$

In the bimanual collision check, the BioIK seed state is additionally extended by the joint positions of the other arm. This ensures that the IK query reflects the complete robot configuration. If the BioIK solve fails, the previous q_{ref} is retained. In bimanual operation, `GOAL_IN_COLLISION (-31)` may still be accepted as an approximate solution. After a successful solve, q_{ref} is stabilized by per-step joint-delta clipping and an exponential moving average before being used by the torque controller [6].

5.2. Observation Space Design

The goal-conditioned observation interface explained in Section 2.1.2 is used by both simulation backends. A central output after a step of both backends is a `gym.spaces.Dict` with the keys `image`, `robot_observation`, `observation`, `achieved_goal`, and `desired_goal`. Here, `desired_goal` (g) and `achieved_goal` ($a^g_t = \phi(s_t)$) follow the goal-conditioned formulation from Section 2.1.2 and result from the task constraints from Section 5.1.1.

A policy (agent) in rollouts is conditioned on `image` and `desired_goal`, but can additionally use `robot_observation`. These are listed in Table 5.2 with their tensor sizes. During optimization, the SAC critic receives an action, a goal, and a low-dimensional cloth state under `observation`, while the policy input remains image-based (conditioned on `image` and `desired_goal`, optionally extended by `robot_observation`). Thus, the two backends follow the observation asymmetry of Hietala et al., in which the replay buffer and Q-functions have access to the cloth-point state during training, as this information is not available during sim-to-real transfer on real hardware [5].

The cloth site positions in `desired_goal`, `achieved_goal`, and the position component of `observation` are specified relative to the internal frame I by subtracting `relative_origin`.

At each episode reset, `relative_origin` is set to the end-effector position. The extended state description of the critic concatenates the cloth site positions in frame I and the cloth site linear velocities in world frame W. If domain randomization is enabled in the MuJoCo backend, a numerical model parameter vector is additionally appended to `observation` [7]. Here, `robot_observation` is not a joint vector. It is a 9D block per arm, while `observation` stores cloth-site positions and velocities, and `desired_goal`/`achieved_goal` store the 3D target or current positions of the task constraints.

Component	Reference single-arm	This thesis (bimanual)
`image`	$F \cdot (W \cdot H)$	$F \cdot (W \cdot H)$
`robot_observation`	9 per arm	2 arm blocks · 9
`observation` (cloth pos_I + vel_W)	9 cloth sites · $(3+3)$	17 cloth sites · $(3+3)$
`desired_goal`/`achieved_goal`	6 constraints · 3	14 constraints · 3

Table 5.2.: Observation tensor sizes from the reference implementation and both Diana backends. F is `frame_stack_size` and $W = H$ is `image_size` [7].

5.2.1. Visual Feedback: Image-based Observations

The `image` key holds a flattened frame stack. `frame_stack_size` determines how many frames are stacked and `frame_stack` keeps the actual frames. At episode reset, the first rendered policy image is repeated `frame_stack_size` times to initialize that stack. During each environment step, at most one new image is appended to that stack. The image is captured at a physics substep index. This index is generated from a clipped Gaussian with the parameters `image_obs_noise_mean` and `image_obs_noise_std`. As the timing of the captured image is randomized within an environment step, it creates a temporal delay in the observation. This implements the motivation of Hietala et al. to randomize the temporal delays in the observations in order to support the sim-to-real transfer [5, 6].

In MuJoCo, the training camera is rendered with the configured buffer size and then center-cropped to `image_size`×`image_size`. In PyBullet, the same rule is implemented for MuJoCo parity before the policy image is generated.

After cropping, both backends convert the image to grayscale, normalize the pixel values to $[0, 1]$ and flatten the result. If `albumentations_randomization` is enabled, RGB augmentations are applied before grayscale conversion [6, 52]. In the reference method, the policy receives a 100×100 grayscale image as an observation for dynamic cloth folding [5].

5.2.2. Robot Proprioception: End-Effector (ee) vs. Control (ctrl) States

The state of the robot can be optionally included in the policy observation under the key `robot_observation`. It can be configured via `-robot-observation {ee,ctrl,none}` and is listed in Table 5.3. In bimanual mode (`DIANA_DUAL_CONTROL=1`), both backends concatenate one 9D block per arm into a fixed 18D vector in left–right order. In `ee` mode, each arm block concatenates the end-effector position in the internal frame I, the end-effector linear velocity, and the current controller target position, also expressed in frame

I. In `ctrl` mode, each arm block concatenates the previous raw action and six zeros. If `DIANA_DUAL_ACTION_INDEPENDENT` is disabled (coupled 3D actions), the same previous action is repeated for both arms. If `DIANA_DUAL_ACTION_INDEPENDENT` is enabled (independent 6D actions), the previous action is split into left and right 3D parts. Once per environment step, the stored `previous_raw_action` is updated and clipped to the action space bounds $[-1, 1]$ [6]. As in Hietala et al., the previous-action input a_{t-1} is used to add temporal context to an image-based policy [5]. In `none` mode, `robot_observation` is set to zeros while keeping the same tensor shape [6].

Mode	Per-arm contents	Dim.
`none`	all zeros	9
`ctrl`	previous raw action $a_{t-1} \in \mathbb{R}^3$ + zero padding	9
`ee`	$p_I^{\text{ee}} \in \mathbb{R}^3$, $\dot{p}^{\text{ee}} \in \mathbb{R}^3$, controller target $x_I^{\text{ctrl}} \in \mathbb{R}^3$	9

Table 5.3.: Robot observation variants in the bimanual Diana environment [6].

5.3. Action Space Configuration

The policy-controller loop described in Section 5.1 uses a normalized displacement action a_t with numerical scaling. The policy action is interpreted as the desired change in end-effector position [5]. Both Diana backends provide a continuous Gym `Box` action space with per-axis bounds $[-1, 1]$. The action is clipped per coordinate to the bounds and scaled by `output_max` to obtain a Cartesian displacement in meters.

$$a_t^{\text{clip}} = \text{clip}(a_t, -1, 1), \qquad \Delta x_t = \texttt{output_max} \cdot a_t^{\text{clip}}. \tag{5.5}$$

As in the Hietala et al. setup, an `output_max` of 0.03 m (per-axis scaling range $[-0.03, 0.03]$) is used as the default value for the PyBullet backend. In MuJoCo, `output_max` is reduced to ≈ 0.02 m (0.019 by default) for parity with PyBullet. With `DIANA_BIOIK_UPDATE_EVERY_SUBSTEPS=auto`, the controller interpolation block runs once per action step. With default timesteps, this yields `between_steps`=6 in PyBullet ($1/150$ s) and 10 in MuJoCo (0.01 s). So for `ctrl_filter`=0.03 the effective fractions are $1 - 0.97^6 = 0.167$ and $1 - 0.97^{10} = 0.263$, so MuJoCo applies about 57% more displacement per action at `output_max`=0.03. To match Bullet's effective displacement, MuJoCo uses `output_max` $\approx 0.03 \cdot (0.167/0.263) \approx 0.019$ as the default value.

In both backends, the clip-and-scale mapping is applied before updating the per-arm Cartesian step targets.

5.3.1. Dual-Arm Independent Action Control

As already stated the action dimensionality is selected by `DIANA_DUAL_ACTION_INDEPENDENT` (Table 4.1). When it is turned off, the action is $a_t \in [-1, 1]^3$, and the same displacement is applied to both arms. When it is turned on, the action is $a_t \in [-1, 1]^6$ and is split in left–right

order as $a_t = [a_t^{(l)}; a_t^{(r)}]$ with $a_t^{(l)}, a_t^{(r)} \in [-1, 1]^3$. Each arm receives a 3D displacement in meters via Eq. (5.5) in both modes. This keeps the shared 3D action layout aligned with the action layout of the reference implementation [5]. Table 5.4 summarizes both supported layouts.

`DIANA_DUAL_ACTION_INDEPENDENT`	Dim.	Action layout
0 (shared)	3	$a_t = [a_x, a_y, a_z]$, broadcast to both arms
1 (independent)	6	$a_t = [a_x^{(l)}, a_y^{(l)}, a_z^{(l)}, a_x^{(r)}, a_y^{(r)}, a_z^{(r)}]$

Table 5.4.: Implemented bimanual action layouts in the Diana MuJoCo and PyBullet environments [6].

5.4. Reward Engineering for Dynamic Folding

5.4.1. Sparse vs. Dense Rewards

The task reward implementation in this thesis follows the sparse-plus-dense idea of Hietala et al. for dynamic "sideways" folding. There, success at the end of the episode is evaluated based on the average corner-to-goal distance d_{sum} from Eq. (5.2) and compared to a threshold δ. Unsuccessful folds receive a sparse reward of -1. Successful folds receive a sparse reward of r_{succ}. As described in Section 5.1.1, a list of geometric constraints is checked against per-constraint distance thresholds to evaluate success. When dense shaping is enabled (`sparse_dense=1`), a normalized, distance-based shaping term is added to all samples regardless of success or failure [5].

The two backends also use the structure of the constraint-based helper. For each constraint $k \in \{1, \ldots, K\}$ from Section 5.1.1, the helper measures $d_k = \|a_t^{g,k} - g^k\|_2$, the distance between where the cloth actually is and where it should be. With 14 constraints instead of 6, more distances feed into the shaping term in Eq. (5.6). δ_k is the distance threshold for constraint k [5] and Eq. (5.6) gives the per-step task reward r_t^{task}.

$$r_t^{\text{task}} = r_{\text{base}} + [\texttt{sparse_dense} = 1] \cdot r_{\text{extra}} \cdot \frac{1}{K} \sum_{k=1}^{K} \left(1 - \frac{d_k}{\delta_k}\right), \tag{5.6}$$

where $r_{\text{base}} = r_{\text{fail}}$ if $\exists k : \; d_k \geq \delta_k$, and $r_{\text{base}} = r_{\text{succ}}$ otherwise.

The dense shaping term is added to every sample, even failed ones. That way the policy still gets a useful signal from the constraint distances before it does a successful fold.

5.4.2. Implementation Details

In the Diana backends, the total environmental reward is formed as the sum of the task reward r_t^{task} and a fold-quality shaping term r_t^{fold}. Site pairs are used to calculate the fold-quality shaping terms in order to measure the center alignment along the fold line. In practice, this gives a dense reward signal based on how closely the cloth lines up with the fold axis. For

backwards folding, the Diana backends use the site pair (S4_0, S4_8). Forward folding uses the reversed pair (S4_8, S4_0) accordingly.

Let $\mathcal{P}$ be the set of fold-quality site pairs used in an episode. Let d_p^{mid} be the Euclidean distance between the two mid-edge sites of a pair p in the internal frame I. Equation (5.7) yields the raw fold-quality shaping reward, where $d_{\text{ref}} = \texttt{cloth_size}$ is used as a reference distance to normalize the midpoint distance by the cloth size.

$$r_t^{\text{fold,raw}} = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \max\left(0,\ 1 - \frac{d_p^{\text{mid}}}{d_{\text{ref}}}\right). \tag{5.7}$$

There is one extra heuristic before this midpoint reward is used. Otherwise, the midpoint could look good while the actually grasped corners are still too far from their targets corners. The raw fold-quality reward is multiplied by a distance gate, which does not use all task constraints. It only looks at the two grasped corners which are stored in done_success_keys. Let $\mathcal{C}$ be this two-corner set, and let d_c be the Euclidean distance from grasped corner c to its target corners. The gate threshold is $d_{\text{ref}}^{\text{gate}} = \texttt{success_distance} \cdot \texttt{FOLD_QUALITY_GATE_DISTANCE_SCALE}$, with SCALE=2.0. Equation (5.8) defines the gate.

$$g_t = \min_{c \in \mathcal{C}} \max\left(0,\ 1 - \frac{d_c}{d_{\text{ref}}^{\text{gate}}}\right). \tag{5.8}$$

In practice, the minimum in Eq. (5.8) means the worse of the two grasped corners sets the gate. If one corner is still far away, the fold-quality bonus is reduced strongly. The effective fold-quality reward is then $r_t^{\text{fold}} = r_t^{\text{fold,raw}} \cdot g_t$. Equation (5.9) shows how the total per-step reward is calculated.

$$r_t = r_t^{\text{task}} + w_{\text{fold}} \cdot r_t^{\text{fold}}, \tag{5.9}$$

where w_{fold} is implemented as a constant FOLD_QUALITY_WEIGHT=3.0.

When the episode ends (see Section 5.1.1), both backends add a one-time DONE_BONUS of 2.0. Both backends log task_reward, fold_quality, fold_quality_raw, fold_quality_gate, original_constraint_error, and total_constraint_error in the per-step info dictionary for sim-to-sim comparability and debugging.

5.5. Learning and Training Procedure

All three environment setups train goal-conditioned policies for the dynamic cloth-folding task. The training recipe is Soft Actor-Critic (SAC) with Hindsight Experience Replay (HER). In this thesis codebase, the single entry point train.py selects the simulation backend for the Diana environments via the environment variable PHYSICS. The environments are wrapped with rlkit.envs.wrappers.NormalizedBoxEnv before data collection begins. This wrapper normalizes the action interface to the range $[-1, 1]$, and it can optionally normalize observations and scale rewards (default: no observation normalization, reward scale 1.0) [75].

In MuJoCo, domain randomization is applied using DomainRandomizationWrapper, which is described in Section 4.4. Robosuite is an open-source robotics simulation framework based

on MuJoCo that provides environments, robot models, and wrappers. For PyBullet, backend-specific defaults are defined in `bullet_model_kwargs.py` and replace the MuJoCo defaults from `mujoco_model_kwargs.py` in the training configuration when `PHYSICS=bullet`.

During training, rollouts are collected using the vectorized collector (`VectorizedKeyPathCollector` on `SubprocVecEnv`) by default. For the PyBullet environment, a separate LenientKeyPathCollector was implemented for GUI support, which is only intended for debugging and small runs, as it does not run in parallel. Using `rlkit.samplers.eval_suite`, a current policy is regularly evaluated and the fold metrics such as `success_rate` are logged. These checkpoints can be resumed with `-resume-dir` and `-resume-epoch` or initialized from another run with `-init-from`. The reference implementation links the number of exploration steps and gradient steps to a single argument, `-train-steps`. This has been adjusted so that exploration steps (`num_expl_steps_per_train_loop`) and gradient updates (`num_trains_per_train_loop`) can be configured separately.

5.5.1. SAC + HER Instantiation for Cloth Folding

The SAC algorithm for continuous actions presented by Haarnoja et al. is used as the base algorithm. The temperature (entropy) parameters can be automatically adjusted using the constrained formulation by Haarnoja et al. [23]. Within `train.py:experiment`, the thesis training code instantiates two Q-functions and corresponding target networks (`qf1`, `qf2`, `target_qf1`, `target_qf2`). In addition, a stochastic Tanh policy (`TanhScriptPolicy`) is created. The target entropy to $-|A|$ uses `-np.prod(action_space.shape)` and is applied to the SAC trainer. This follows the entropy target heuristic proposed by Haarnoja et al. As mentioned in Section 2.1.4, HER is used because of the sparse goal-reaching signals. It is used when goals are relabeled with achieved goals. The future strategy is defined by Andrychowicz et al. as sampling objectives from subsequent states within the same episode [21]. The 3 environment setups implement HER with `FutureObsDictRelabelingBuffer`, which is compatible with Open AI Gym, and by wrapping the SAC trainer with `ClothSacHERTrainer`. The replay buffer is initialized from the observation-space dictionary `observation_space.spaces` and receives the keys `observation_key`, `desired_goal_key`, and `achieved_goal_key`. It also has the internal keys `image` and `robot_observation`. After relabeling, the replay buffer recalculates rewards using `task_reward_function` and returns them as the minibatch rewards. The relabeling ratio can be configured with `-her-percent` (default 0.8), which is converted to `fraction_goals_rollout_goals = 1 - her_percent`.

An important change for the bimanual implementations was the modification of `reward_calculation`. In the earlier version, `fail_reward` was reapplied at the end of the helper, which cleared the dense distance term for unsuccessful samples. The HER path had a similar issue, as relabeled future-goal samples were supposed to retain the plain success reward instead of receiving additional dense shaping. After the correction, failed transitions still retained useful distance information and HER-relabeled samples received the intended goal reward. This made the reward signal more consistent during training and especially at the beginning of the training phase, when most transitions were still failures, useful information about the distance to the goal was provided.

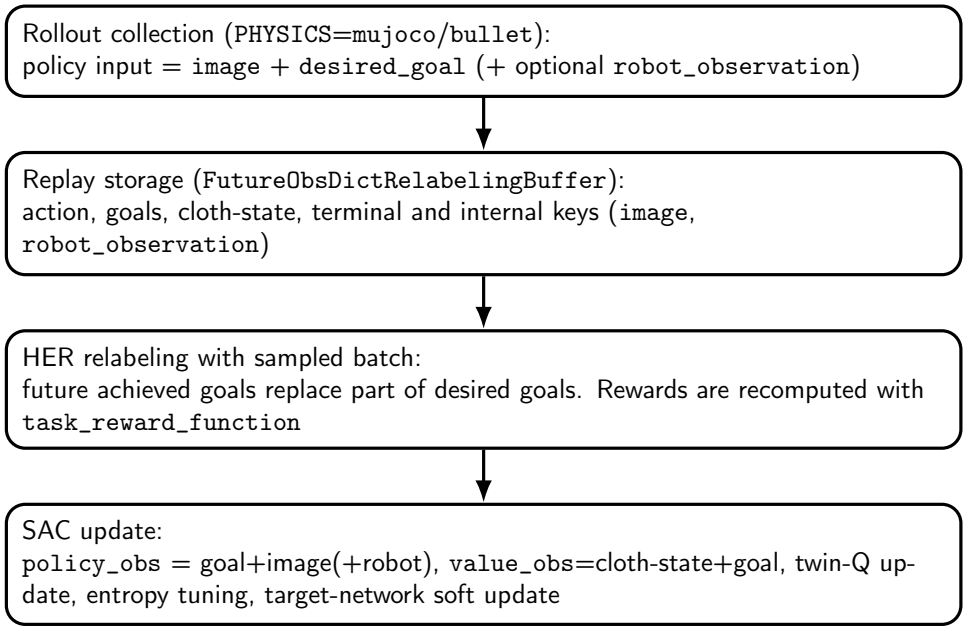


Figure 5.3.: SAC+HER data flow Implemented for three environment setups [7].

The replay buffer then stores the image observations and, optionally, the robot states. The input for the actor is formed as `policy_obs` = [`images`; `robot_observations` (optional); `resampled_goals`]. The input for the critic is formed as `value_obs` = [`observations`; `resampled_goals`]. Unlike in the reference implementation, `is_success` is not derived from a reward threshold (`reward > fail_reward`), but is calculated geometrically from the distances of the corners to the target positions. The `done` signal is triggered only if the corners remain within a defined `success_distance` for a certain number of steps. Both backends can additionally terminate early via their unhealthy-step watchdogs.

5.5.2. Network Architecture and Input Fusion

All three environment setups use the same network architecture. The policy uses the decomposition into three mappings by Matas et al., which was also adopted by Hietala et al. [5, 8]. In summary, the input image is first converted into a latent representation L by a CNN-based image encoder h_1. Then, a fully connected policy head h_2 maps the combination of L, the previous action a_{t-1}, and the goal g to the next action a_t. Finally, an auxiliary head h_3 maps the latent representation L to estimated pixel coordinates of the corners of the cloth in the image plane. These three mappings are described in Section 2.4.4 with more context.

Policy network

During training, the Path Collector concatenates the observation dictionary into a flat input vector for the policy. For the policy input x_π, the concatenation order is [`image`; `robot_observation`; `desired_goal`] [7].

The first part of the input consists of the elements from `image` and has the length $F \cdot W \cdot H$. Here, F is the `frame_stack_size` and $W = H = 100$ are the width and height of the policy image. This part is separated from the remaining input elements and reshaped into the form $(F,\ H,\ W)$. Four conv layers in `ScriptPolicy` process this, each with 3×3 kernels, 32 channels, stride 2, and ReLU. The output is flattened into a vector with size F_{conv}.

The rest of x_π, containing `robot_observation` and `desired_goal` values, gets concatenated with the flattened features to give a vector of dimension $F_{\text{conv}} + d_{\text{added}}$. This vector is fed into the fully connected stream and processed by four hidden layers, each with 256 units and ReLU activation. The last hidden state is finally projected by a linear layer onto the action means $\mu \in \mathbb{R}^{|A|}$. Separately, a linear head is now trained that produces a learned $\log \sigma$ vector, which is clamped to $[-20,\ 2]$. From these, the actions are sampled via the `TanhNormal` distribution. Finally, a $\tanh$ saturation is applied to the sampled value on a Gaussian distribution, thus keeping the actions in $[-1, 1]$ [7, 23].

Auxiliary corner prediction

The flattened convolutional features from Section 5.5.2 are fed in parallel into an auxiliary network. This consists of two hidden layers (256 and 8 units, ReLU) followed by a sigmoid output layer of dimension 9. During training, the first 8 outputs are compared with the ground-truth corner pixel coordinates from the simulation. This results in a mean-squared-error loss, which is weighted with `corner_prediction_loss_coef` $= 0.001$ and added to the policy loss. This is intended to encourage the CNN to learn relevant features for the perception of the cloth. As described in Section 2.4.4, the auxiliary outputs are only used during training [5].

Q-function network

The `ConcatMlp` (Concatenated Multi-Layer Perceptron) networks are used for the two Q-functions and their soft-updated targets. Each Q-function accepts a concatenated input vector x_Q, which consists of `observation`, `desired_goal`, and the action. Here, `observation` is the low-dimensional state of the cloth. By default, the Q-networks use five hidden layers, each with 512 units and ReLU activations, as well as a single scalar output [7].

Dimension changes in bimanual mode

The same code for the networks is used for both the single-arm and bimanual setups. This is possible because the network classes are parameterized by the input and output sizes. Table 5.5 compares the dimensions of the input vectors for the single-arm reference implementation and the bimanual Diana implementation.

Component	Reference	Bimanual	Reason
`observation` (cloth)	54	102	17 cloth sites · $(3+3)$
`desired_goal`	18	42	14 constraints · 3
`robot_observation`	9	18	2 arm blocks · 9
`added_fc_input`	27	60	$d_{\text{goal}} + d_{\text{robot}}$
`value_input_size`	75	147/150	$d_{\text{obs}} + d_{\text{goal}} + d_{\text{action}}$
Action dim.	3	3 or 6	shared/independent (Table 5.4)

Table 5.5.: Input dimension changes from single-arm reference to bimanual Diana setup [7].

The three environment setups have the same convolutional encoder architecture with the same number of hidden layers and the same hyperparameters. The same image pipeline is used, so that the CNN input size remains at $F \cdot 100 \cdot 100$. Only the fully connected layers after the CNN are affected by the increased dimensions of the auxiliary inputs [7]. Figure 5.4 summarizes the complete forward pass through the policy network. The figure shows how the flat input vector is split, passed through the CNN encoder, and then fed into the two parallel fully connected streams.

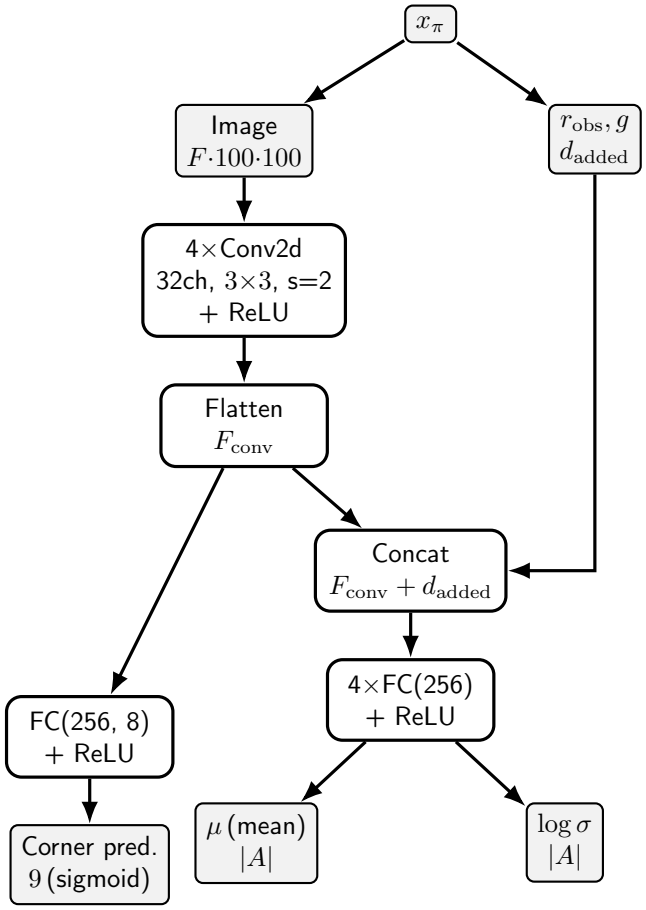


Figure 5.4.: Policy network forward pass (`ScriptPolicy`) [7, 8].

5.5.3. Training Hyperparameters and Evaluation Schedule

This section summarizes the training hyperparameters used in both Diana backends. These are compared with the reference implementation by Hietala et al.

Training loop structure

In all three environment setups, the training run consists of `num_epochs` epochs. Each epoch consists of `num_train_loops_per_epoch` cycles. In each cycle, `num_expl_steps_per_train_loop` steps are collected in the environment using the current stochastic policy with multiple parallel workers. The collected transitions are added to the replay buffer, and then `num_trains_per_train_loop` gradient updates are performed on minibatches from the buffer. After each epoch, the evaluation suite is performed [7].

As noted in Section 5.5, the reference code uses a single argument `-train-steps` for both the number of exploration steps and the number of gradient updates per cycle (both default 1000) [7]. In this thesis, the number of exploration steps remains at 1000, while the

gradient updates per cycle are set to 250 with `-num-updates`. This reduces the training time per cycle without significantly affecting learning progress in practice. The same structure with 100 epochs and 20 cycles per epoch is maintained in both Diana backends [6]. With `SubprocVecEnv`, 4 parallel environment workers are used for data collection, compared to the default of 1 process in the reference code [7].

As described in Section 5.5.1, the SAC implementation by Haarnoja et al. is used as the base algorithm [7, 23]. The hyperparameters used for this are summarized in Table 5.6. For the bimanual setup, the replay buffer size was increased from 10^5 to 10^6 to account for the larger observation dimension.

Parameter	Reference code	Diana (this thesis)
Epochs	100	100
Cycles per epoch	20	20
Exploration steps per cycle	1000	1000
Gradient updates per cycle	1000	250
Minibatch size	256	256
Parallel workers	1	4
Policy / Q-function learning rate	3×10^{-4}	3×10^{-4}
Discount γ	0.99	0.99
Soft target τ	0.005	0.005
Reward scale	1	1
Replay buffer size	10^5	10^6
HER relabeling ratio	80%	80%
Corner prediction loss coef.	0.001	0.001
Episode length (policy steps)	50	50
`success_distance`	0.05 m	0.05 m
`max_close_steps`	10	15

Table 5.6.: Training hyperparameters in the reference implementation by Hietala et al. and in the Diana backends [7].

Evaluation schedule

After each epoch, the current deterministic policy is evaluated using a fixed set of tests [7]. This involves 20 evaluation runs, in which the `SuccessRateTest` measures the success rate of the policy. An evaluation consists of up to `max_path_length` = 50 or until the end of the episode. After evaluation, the code records whether the rollout was successful. It also stores the per-corner distances to the target and their summed error.

In addition to regular epoch evaluations, a sanity check suite was used during development to detect faulty configurations before longer training runs were started. These checks are started by the script `sanity_test.sh`. They include short hold and move rollouts, smoke training runs, single-episode training runs, and checkpoint evaluations in both backends. For the move rollouts, a new `demos_30cm_backward.csv` was created from the `demos.csv` by Hietala et al., in which the folding movement was scaled for a 30 cm size clothing. The

movements were also split into roughly equal segments and adapted for backward folding. These checks make it easier to catch controller changes, environment tweaks, or backend defaults that quietly break stable behavior.

All three environment setups save complete policy checkpoints after each epoch (`save_policy_every_epoch = 1`) [7]. These checkpoints can be used to resume training, initialize new runs, or for evaluation between backends, as described in Section 5.5.

6. Evaluation

This chapter evaluates the training progress and the sim-to-sim transfer between the MuJoCo and PyBullet versions of the bimanual cloth folding task. The focus is on the objectives from Chapter 1 using the dual-arm Diana setup from Chapter 4.

6.1. Evaluation Goals

The goal is to answer the research questions from Section 1.3.1. These can be grouped into five objectives:

1. Measuring the in-engine performance.
2. Measuring the cross-engine transferability without retraining.
3. Testing the influence of training with domain randomization on cross-engine robustness.
4. Isolating the effects of design decisions on in-engine and cross-engine performance.
5. Testing performance on unseen setup configurations.

The various design choices are:

- **Robot observation mode (`ee` vs. `ctrl`):** Influence of end-effector state feedback or previous-action feedback (Table 5.3).
- **Action dimensionality (3D vs. 6D):** Whether coupled actions (same displacement for both arms) or independent actions (separate 3D displacement per arm, Table 5.4) lead to different results.
- **Cloth size randomization (RCS):** How changing the Y-dimension $[0.30, 0.40]\,\mathrm{m}$ of the cloth affects generalization (Section 4.4). Additionally, whether DR and RCS together lead to better generalization.

6.2. Experimental Protocol

6.2.1. Training Setup

Each training run was performed using the hyperparameters from Table 5.6 and the SAC with the HER algorithm from Section 5.5.1. The folding direction is set to `backward` for all runs (Section 5.1.1), with both arms always active. Further variations of the design decisions are described in Section 4.1.2. Both backends share the BioIK-based controller path described in Section 5.1.2. The policy still gets the same 100×100 grayscale image as in the original method [5]. Section 5.5.2 covers the network architecture and how inputs get fused.

Table 6.1.: Factors of the experimental matrix.

Factor	Levels
Physics backend	MuJoCo, PyBullet
Robot observation	`ee`, `ctrl`
Bimanual action layout	shared 3D, independent 6D
Domain randomization	off, on
Cloth-size randomization	off, on

6.2.2. Evaluation During Training

After each epoch, the deterministic policy is evaluated over 20 rollouts using the `SuccessRateTest`, as described in Section 5.5.3. In this process, metrics such as `success_rate`, the distances of the four corners (`corner_0` ... `corner_3`), and the sum of corner errors (`corner_sum_error`) are recorded [7]. The metrics collected per epoch are averaged across the 20 evaluation rollouts. During all training runs, the auxiliary real-corner evaluation head was deactivated with `skip_real_corner_eval=true`.

6.2.3. Cross-Engine Evaluation Protocol

In cross-engine evaluation, a trained policy checkpoint is loaded into the other physics backend. The same deterministic policy weights are used without retraining or fine-tuning. Both backends have the same observation dict, action bounds, and goal interface (Section 5.2, Section 5.3), so checkpoints load without any conversion. That follows the protocol from Mohammed et al., where cross-engine evaluation keeps the task interface unchanged [40]. In-engine results come from the normal training logs. Cross-engine results are generated with `test/eval_policy.py`. There, the default setup configuration can be loaded with a checkpoint, with the option to switch between configurations and physics backends.

6.3. Experimental Conditions

In summary, five binary factors are combined: physics backend, action dimensionality, robot observation mode, domain randomization, and randomized cloth size (RCS). These binary factors are shown in Table 6.1. The completed training runs are shown in Table 6.2.

Table 6.2.: Completed training runs.

Backend	DoF	Obs	DR	RCS
MuJoCo	3D	ee	0	0
MuJoCo	3D	ee	1	1
MuJoCo	3D	ctrl	0	0
MuJoCo	3D	ctrl	1	0
MuJoCo	6D	ee	0	0
MuJoCo	6D	ee	1	0
MuJoCo	6D	ee	1	1
MuJoCo	6D	ctrl	0	0
MuJoCo	6D	ctrl	1	1
Bullet	3D	ee	0	0
Bullet	3D	ee	1	1
Bullet	3D	ctrl	0	0
Bullet	3D	ctrl	1	1
Bullet	6D	ee	0	0
Bullet	6D	ee	1	0
Bullet	6D	ee	1	1
Bullet	6D	ctrl	0	0
Bullet	6D	ctrl	1	1

A complete matrix would have $2^5 = 32$ cells. Table 6.2 summarizes the 18 run configurations. The baseline is the training condition $DR = 0$ and $RCS = 0$, while using ee observations. The randomization parameters are listed in Table 4.5. Section 4.2 covers how cloth scaling works.

6.4. Metrics

6.4.1. Success Rate

A rollout is successful if `is_success` evaluates to true at least once during the episode. For this to happen, the two grasped corners selected must lie within the `success_distance` of their target positions (Equation (5.3)). Termination occurs after 15 consecutive successful steps, but success is counted as early as the first qualifying step. This metric was also used in the paper from Hietala et al. for evaluating the policies [5]. The metric is logged as `suite/randomized_cloth_success_rate` during training.

6.4.2. Corner Sum Error

This is the sum of the Euclidean distances from each evaluated corner of the cloth to its corresponding target position at the end of the rollout. In both backends, the metric is calculated identically at the end of a rollout by summing the four per-corner distances from the task definition (Section 5.1.1). In the paper by Hietala et al., the corresponding geometric

evaluation quantity is the fold distance error d_{sum}, which sums two point-to-goal distances rather than four corner distances [5]. In this thesis, the four per-corner distances are logged instead. This metric is therefore only an approximate proxy for comparison with the results of Hietala et al. and is logged as `suite/randomized_cloth_corner_sum_error` during the training

6.4.3. Final Constraint Error

This metric sums all 14 constraint errors for the `sideways_backward` task at the final rollout step. It is used to compare performance between the two backends. It provides the most direct task-specific measure of how well the final cloth state satisfies all geometric constraints. In the training log, the epoch-wise mean over rollout end states is reported under the name `exploration/env_infos/final/total_constraint_error Mean`.

6.4.4. Episode Return

Episode return is the sum of the rewards across all steps of a rollout. The reward is defined in Section 5.4. It consists of the task reward r_t^{task} (Equation (5.6)) and the fold-quality shaping term r_t^{fold} (Equation (5.9)). It is more of a secondary diagnostic during the training, as the reward contains shaped terms and is therefore less direct than the geometric task metrics. The training log reports the mean episode return of the exploration rollouts under the name `exploration/Average Returns`.

6.5. Results

6.5.1. Learning curves per engine and setting

In robot-observation mode *ee*, both backends achieve in-engine success. However, the geometric error and the stability of the learning curves vary depending on the engine and settings. With the setting `6D ee 0 0`, the graphs in Figure 6.1 show a comparison of the 6D baseline. PyBullet achieves and maintains a stable success rate of 1.0 early on and steadily reduces the corner-sum error and the final-constraint error. MuJoCo also achieves a success rate of 1.0, but experiences repeated drops or unstable phases in the success rate, corner-sum error, and final-constraint error in the late epochs.

The 3D baseline (`3D ee 0 0`) in Figure 6.2 also reaches a success rate of 1.0 quickly in both backends. However, PyBullet reduces the geometric error further, while MuJoCo remains at a final constraint error of about 0.8. This is consistent with visual inspection of the recorded rollouts and captured images. In this setting, the MuJoCo policy often does not lift the grasped corners sufficiently and instead pulls them almost directly toward the target positions, so the fold is not fully smoothed out. In the runs inspected here, final constraint errors around 0.5 or below were generally associated with policies that lifted the corners more clearly and reduced wrinkles, resulting in a better final outcome.

With randomization, the engine gap becomes more setting-dependent. With the settings `3D ee 1 1` (with domain $DR = 1$ and cloth-size randomization $RCS = 1$), MuJoCo achieved

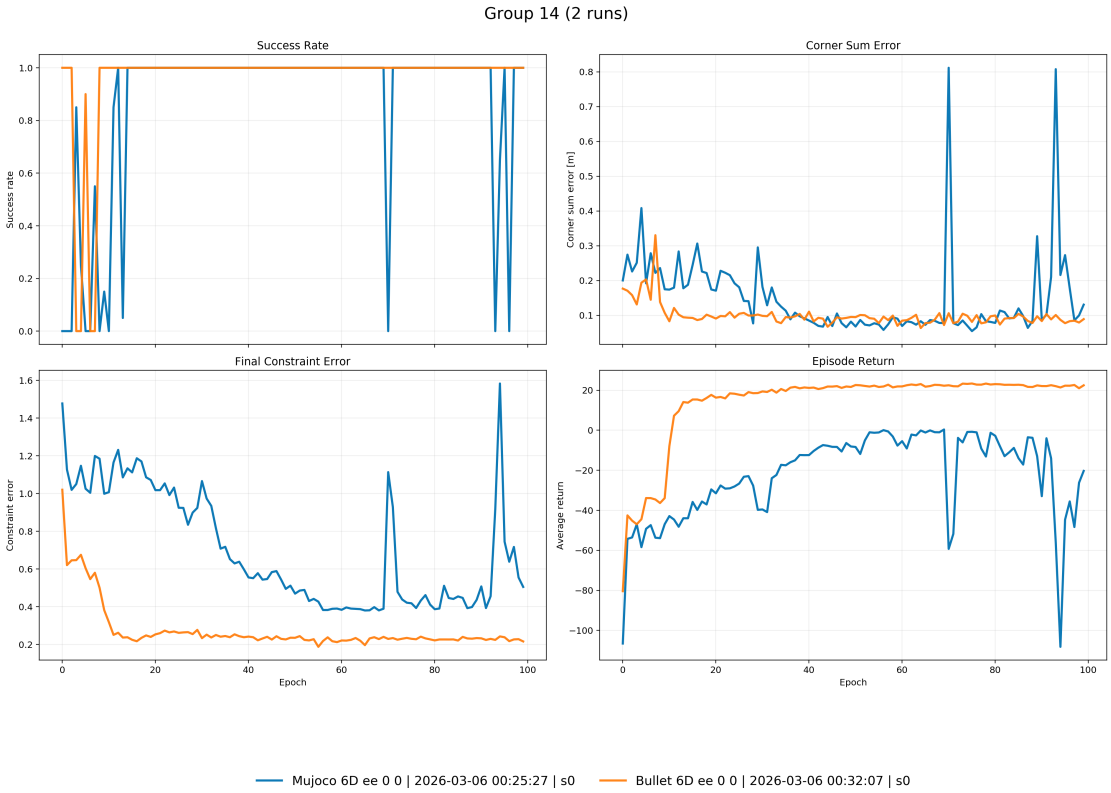


Figure 6.1.: Learning curves for the shared baseline setting `6D ee` with $DR = 0$ and $RCS = 0$.

a lower geometric error than PyBullet (Figure 6.3). Although MuJoCo's success rate curve also shows dips and the reward is smaller than in PyBullet, the corner-sum error and the final constraint error at the end of the training phase are lower than in PyBullet.

Compared with the corresponding 3D settings, the 6D settings show less stable learning curves in the matched runs considered here. For `6D ee 1 1` with $DR = 1$ and $RCS = 1$, both engines learn more slowly than in the non-randomized 6D baseline (Figure 6.4). MuJoCo exhibits greater oscillations until late training, while PyBullet ends up with the lower final constraint error. For these matched settings, the geometric metrics are therefore more informative than success rate alone.

The PyBullet control run `3D ctrl 1 1` is reported separately in Appendix B Figure .10 shows that the run still reaches a final success rate of 1.0 and ends at a CSE of 0.040 and an FCE of 0.148. It performs better than the PyBullet `3D ctrl 0 0` run, which finishes at 0.075 CSE and 0.420 FCE.

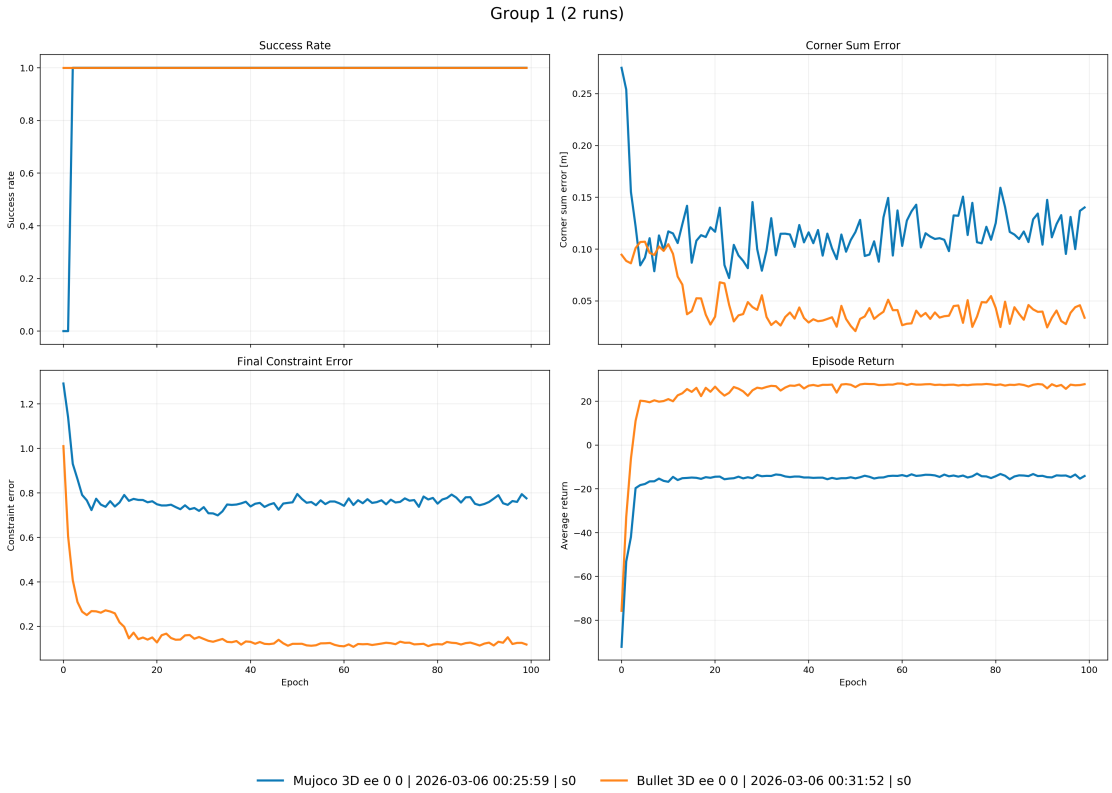


Figure 6.2.: Learning curves for the shared baseline setting `3D ee` with $DR = 0$ and $RCS = 0$.

6.5.2. In-engine evaluation

Figure 6.5 summarizes the final-epoch in-engine results for the matched settings that are available in both backends. All shared `ee` settings, the `3D ctrl 0 0` baseline and the PyBullet `3D ctrl 1 1` run achieve a success rate of 1.0 at the final checkpoint in their own engine. The `6D ctrl` settings are less stable. MuJoCo fails in `6D ctrl 1 1`, while PyBullet fails in `6D ctrl 0 0`. For the successful settings, differences are therefore primarily evident in the geometric metrics.

The trajectories and visual results for `3D ee 1 1` and `6D ee 1 1` are shown in Figures 6.6, 6.7, 6.8, and 6.9. The remaining trajectories and visual results for the other pairs can be found in Appendix A. In the composite images, the upper left panel shows the actual end-effector trajectory during the rollout. The other three panels show the state of the cloth as viewed by the corner camera at step 10, step 15, and the final step.

Policies trained in PyBullet achieve better results than MuJoCo policies for the non-randomized baseline in both action layouts. In `3D ee 0 0`, the final corner-sum error (CSE) is 0.140 for MuJoCo and 0.034 for PyBullet. The final constraint error (FCE) is 0.776 in MuJoCo and 0.118 in PyBullet. In `6D ee 0 0`, CSE is 0.131 in MuJoCo and 0.089 in PyBullet,

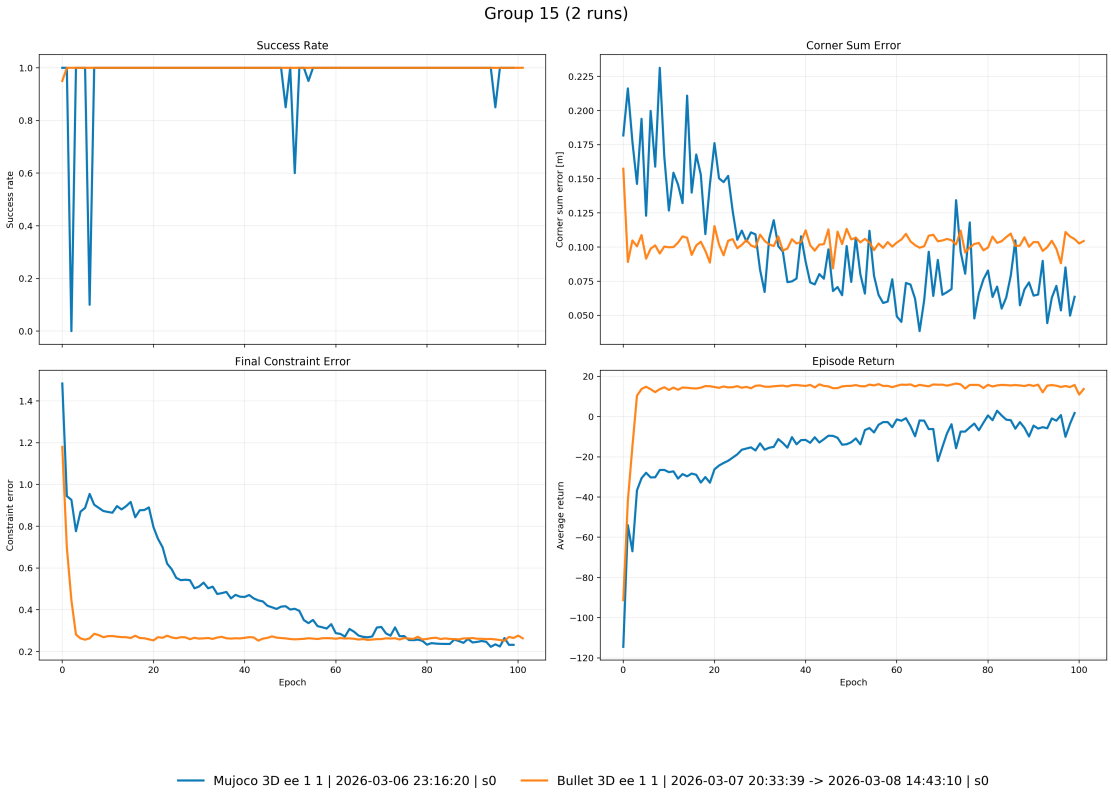


Figure 6.3.: Learning curves for the shared randomized setting `3D ee` with $DR = 1$ and $RCS = 1$.

while FCE is 0.505 in MuJoCo and 0.216 in PyBullet.

For $DR = 1$ and $RCS = 1$, the results are mixed and the differences in the geometric metrics are smaller. In `3D ee 1 1`, MuJoCo achieves a lower CSE (0.064) than PyBullet (0.104) and a lower FCE (0.232) than PyBullet (0.263). In `6D ee 1 1`, PyBullet achieves a lower CSE (0.075) than MuJoCo (0.081) and a lower FCE (0.241) than MuJoCo (0.351). Randomization reduces the differences in the geometric metrics. The `ctrl` observation mode behaves differently depending on the engine and whether actions are 3D or 6D. In the 3D baseline, both engines end with a success rate of 1.0 and MuJoCo benefits from `ctrl`. MuJoCo achieves the lower CSE (0.034 vs. 0.075), while PyBullet attains the slightly lower FCE (0.420 vs. 0.454). Relative to the corresponding `ee` baseline, `ctrl` improves both geometric metrics in MuJoCo, but degrades them in PyBullet.

For the 6D baseline, the observation-mode effect is more asymmetric. MuJoCo `6D ctrl 0 0` reaches a success rate of 1.0 with a CSE of 0.058 and an FCE of 0.514, which is close to the MuJoCo `6D ee 0 0` result. In contrast, PyBullet `6D ctrl 0 0` ends with a success rate of 0.0, a CSE of 0.148, and an FCE of 0.411. This PyBullet run nonetheless reaches a successful checkpoint earlier in training: at epoch 26, it attains a success rate of 1.0 with a

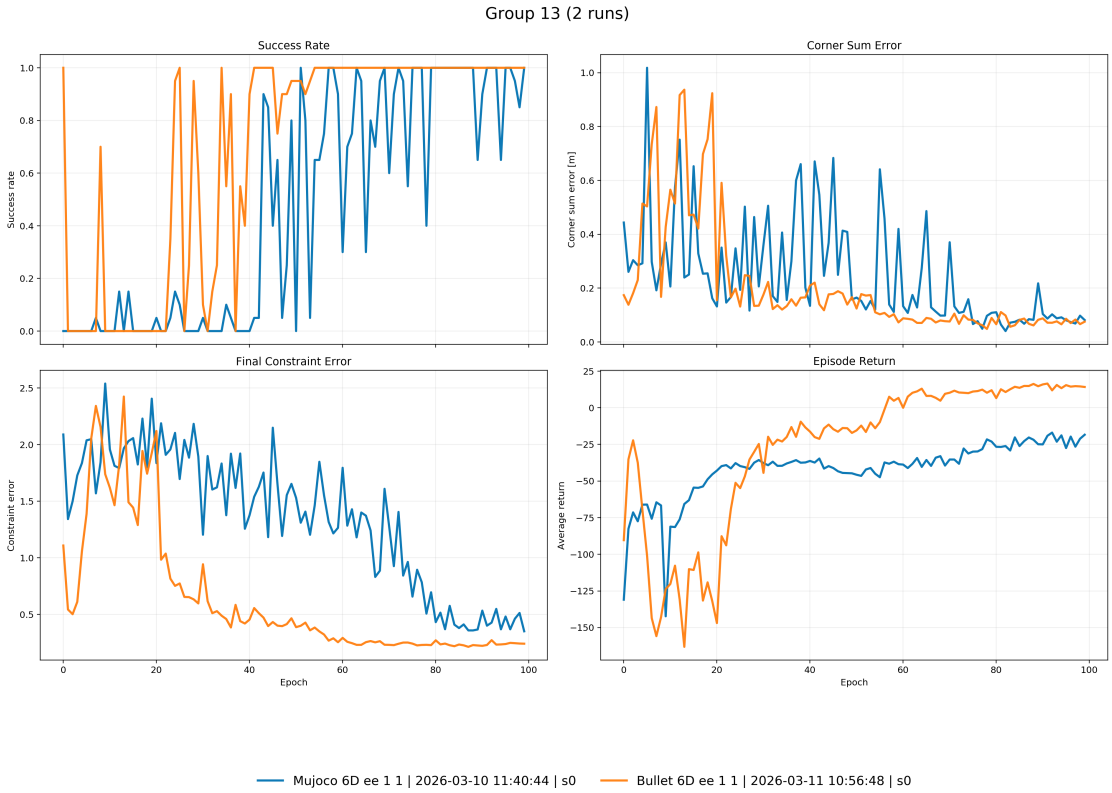


Figure 6.4.: Learning curves for the shared randomized setting `6D ee` with $DR = 1$ and $RCS = 1$.

CSE of 0.095 and an FCE of 0.273, before deteriorating later.

Together with `6D ctrl 1 1`, the 6D `ctrl` mode is therefore the least stable family across the evaluated settings. PyBullet succeeds for `6D ctrl 1 1` with a success rate of 1.0, a CSE of 0.060, and an FCE of 0.179, while MuJoCo fails with a success rate of 0.0, a CSE of 0.179, and an FCE of 1.867. The opposite pattern appears for `6D ctrl 0 0`, where MuJoCo succeeds but PyBullet fails at the final epoch.

6.5.3. Cross-engine evaluation

For the cross-engine evaluation, one checkpoint is selected for each `ee` setting that was completed in both backends. The selected checkpoint is chosen based on the following order:

1. highest `suite/randomized_cloth_success_rate`,
2. lowest `exploration/env_infos/final/total_constraint_error Mean`,
3. lowest `suite/randomized_cloth_corner_sum_error`,

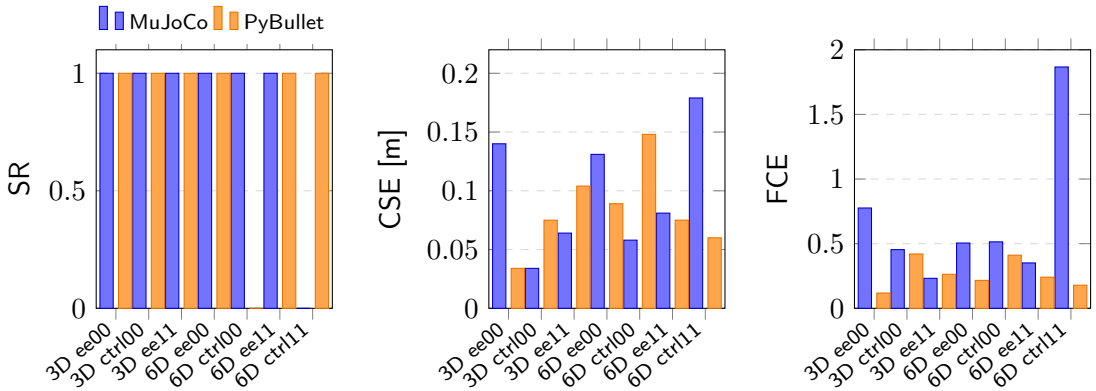


Figure 6.5.: Final-epoch in-engine metrics for matched settings in both backends. SR(success rate). CSE(corner-sum error). FCE(final constraint error).

Table 6.3.: Selected checkpoints for cross-engine evaluation.

Setting	MuJoCo checkpoint	PyBullet checkpoint
`3D ee 0 0`	epoch 33	epoch 62
`3D ee 1 1`	epoch 94	epoch 97
`6D ee 0 0`	epoch 65	epoch 55
`6D ee 1 1`	epoch 99	epoch 87

This selects checkpoints at epochs that can already reliably solve the task, as shown in table 6.3. MuJoCo checkpoints are evaluated in PyBullet and vice versa, without retraining.

The success rate of each checkpoint is 1.0 in its own backend, while the target backend retains the same settings for DR, RCS, and the 3D vs. 6D action layout. Only the physics backend is changed. The selected checkpoints are evaluated for 20 rollouts in the other engine. Table 6.4 and Figure 6.10 summarize the resulting transfer performance.

Three of the four policies trained on MuJoCo transfer successfully to PyBullet: `3D ee 0 0`, `6D ee 0 0`, and `6D ee 1 1` all achieve a success rate of 1.0 in PyBullet. The randomized 6D policy `6D ee 1 1` achieves the lowest transferred FCE of 0.302 and a CSE of 0.109. Figure 6.11 shows this best performing MuJoCo→PyBullet transfer visually. The policy was able to adjust the trajectory to have a steeper slope compared to Figure 6.8 and fold the clothing. The non-randomized 6D baseline `6D ee 0 0` also achieves a low FCE of 0.389 and a CSE of 0.124. As the last of the three successful MuJoCo→PyBullet transfers, the non-randomized 3D baseline `3D ee 0 0` achieves a success rate of 1.0, but with a significantly higher FCE of 0.486 despite a low CSE of 0.100. Only the randomized 3D policy `3D ee 1 1` fails to transfer successfully from MuJoCo to PyBullet, with an FCE of 1.230 and a CSE of 0.387.

None of the PyBullet-trained policies transfer successfully to MuJoCo. However, there are differences in geometric error between the settings. The lowest transferred error is from `6D ee 0 0` with an FCE of 0.746 and a CSE of 0.196. The two 3D settings remain significantly

Table 6.4.: Cross-engine evaluation results for the selected checkpoints.

Setting	Direction	SR	CSE [m]	FCE
3D ee 0 0	M→B	1.00	0.100	0.486
3D ee 0 0	B→M	0.00	0.423	1.135
3D ee 1 1	M→B	0.00	0.387	1.230
3D ee 1 1	B→M	0.00	0.473	1.740
6D ee 0 0	M→B	1.00	0.124	0.389
6D ee 0 0	B→M	0.00	0.196	0.746
6D ee 1 1	M→B	1.00	0.109	0.302
6D ee 1 1	B→M	0.00	0.297	2.410

Table 6.5.: Generalization to an unseen colors16 cloth texture in each policy's native engine. Each cell reports SR / CSE / FCE.

Policy	25 cm	40 cm	High fric.	Low fric.
3D ee 0 0 (M)	1.00 / 0.028 / 0.441	1.00 / 0.085 / 0.672	1.00 / 0.038 / 0.507	1.00 / 0.092 / 0.650
3D ee 0 0 (B)	1.00 / 0.027 / 0.098	0.65 / 0.183 / 0.421	1.00 / 0.030 / 0.132	1.00 / 0.030 / 0.103
3D ee 1 1 (M)	1.00 / 0.023 / 0.138	1.00 / 0.058 / 0.118	1.00 / 0.035 / 0.097	1.00 / 0.038 / 0.111
3D ee 1 1 (B)	1.00 / 0.102 / 0.320	1.00 / 0.114 / 0.342	1.00 / 0.102 / 0.288	1.00 / 0.112 / 0.316
6D ee 0 0 (M)	1.00 / 0.053 / 0.279	1.00 / 0.114 / 0.606	1.00 / 0.075 / 0.560	1.00 / 0.082 / 0.544
6D ee 0 0 (B)	1.00 / 0.074 / 0.179	1.00 / 0.127 / 0.300	1.00 / 0.077 / 0.201	1.00 / 0.091 / 0.241
6D ee 1 1 (M)	1.00 / 0.102 / 0.340	1.00 / 0.067 / 0.348	1.00 / 0.092 / 0.353	1.00 / 0.096 / 0.356
6D ee 1 1 (B)	1.00 / 0.100 / 0.265	1.00 / 0.065 / 0.213	1.00 / 0.078 / 0.208	1.00 / 0.080 / 0.219

higher, with an FCE of over 1.1. The randomized 6D policy 6D ee 1 1 achieves a CSE of 0.297, but with the highest FCE of 2.410. Figure 6.12 shows what goes wrong. The fold looks reasonable, but it doesn't satisfy the constraints.

In summary, this shows that transferring from PyBullet to MuJoCo is significantly more difficult for the evaluated settings than transferring from MuJoCo to PyBullet.

6.5.4. Generalization to unseen cloth configurations

Each checkpoint from Table 6.3 was evaluated in its own engine with DR=0 on an unseen colors16.png cloth texture from PyBullet data. Four controlled test conditions were used in the evaluation. The cloth sizes were set to $25\,\mathrm{cm}$ and $40\,\mathrm{cm}$, and the cloth friction was varied across two levels. For High fric., the friction was increased from 1.1 to 1.4 in PyBullet and from 4.736 to 6.0 in MuJoCo. For Low fric., the friction was reduced from 1.1 to 0.8 in PyBullet and from 4.736 to 3.0 in MuJoCo. Table 6.5 lists the success rate (SR), corner sum error (CSE), and final constraint error (FCE) for these runs.

31 of the 32 policy-condition pairs achieved a success rate of 1.0. The only significant drop in performance is observed for PyBullet 3D ee 0 0 on the $40\,\mathrm{cm}$ cloth, where the success rate drops to 0.65.

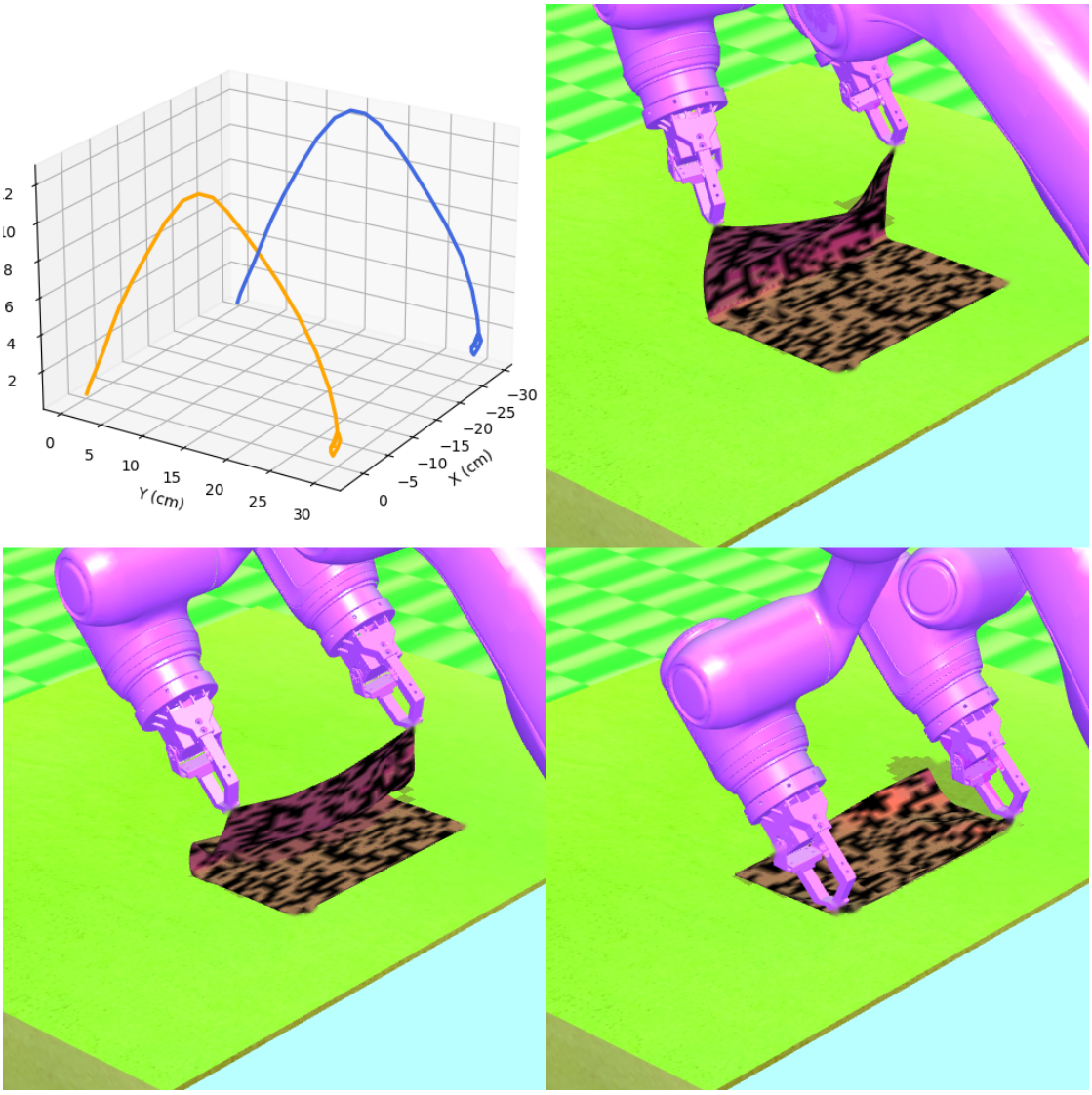


Figure 6.6.: MuJoCo 3d ee 11, epoch 94. (Lowest FCE among MuJoCo configurations.)

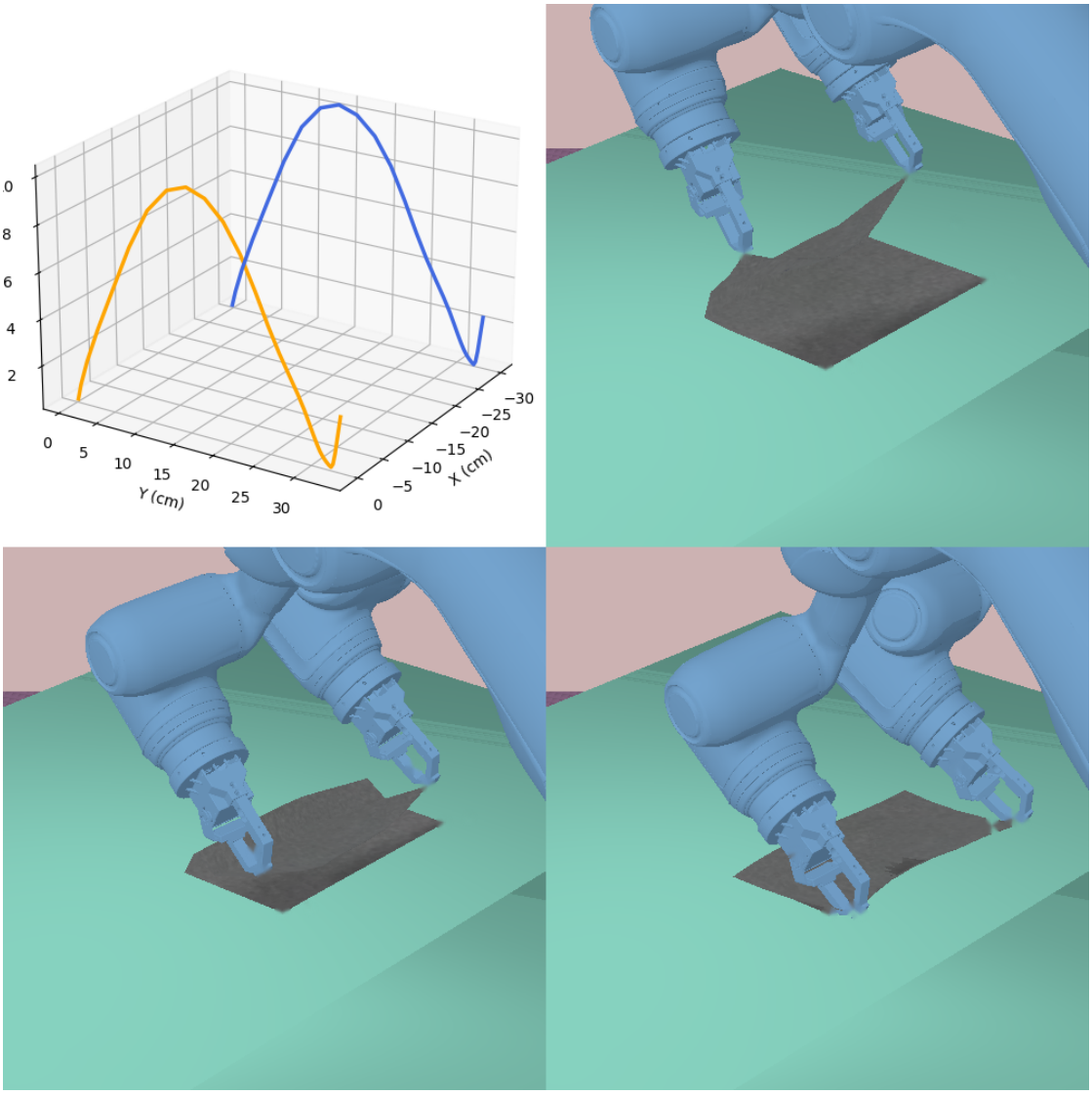


Figure 6.7.: PyBullet 3d ee 11, epoch 97.

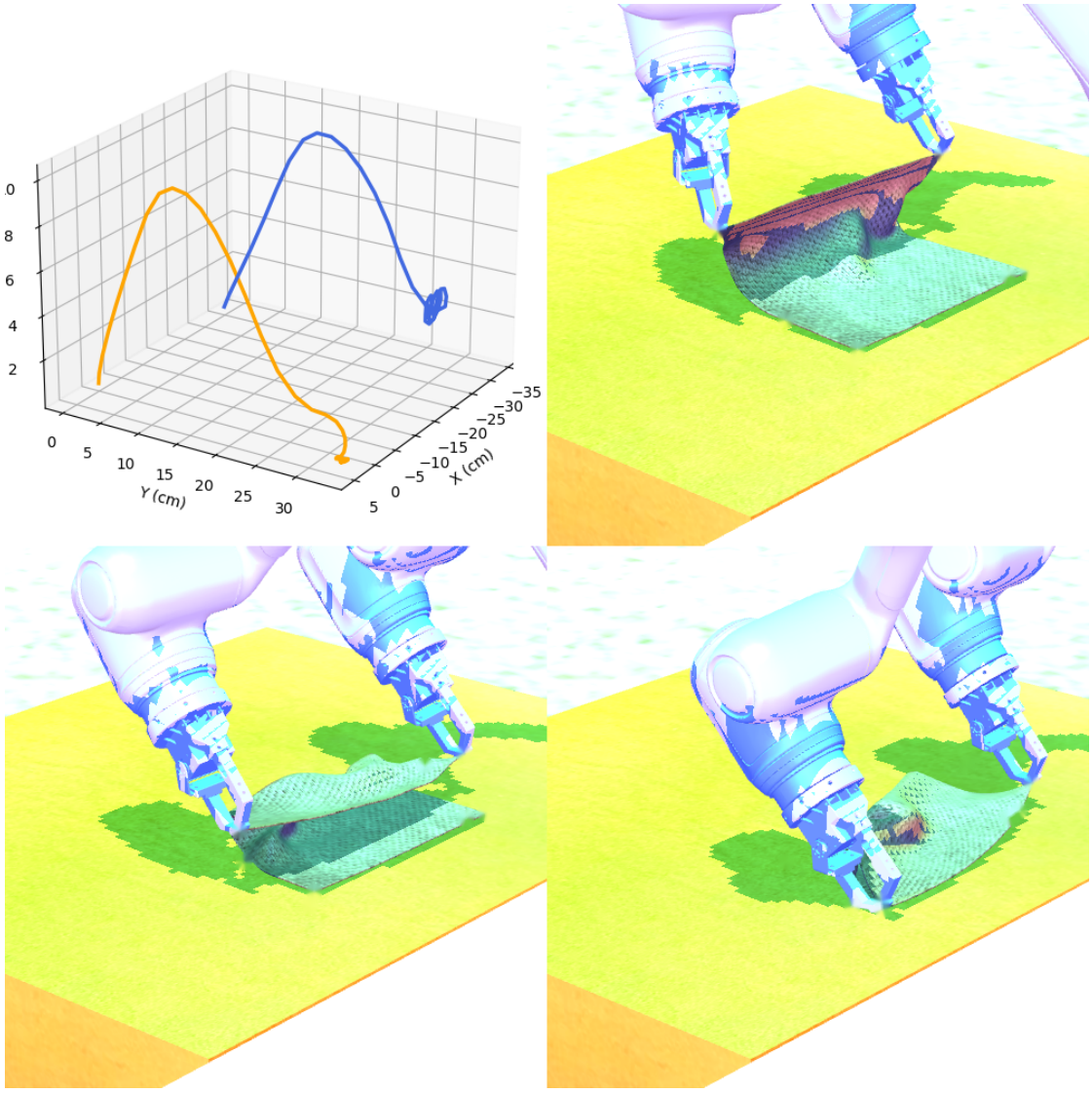


Figure 6.8.: MuJoCo 6d ee 11, epoch 99.

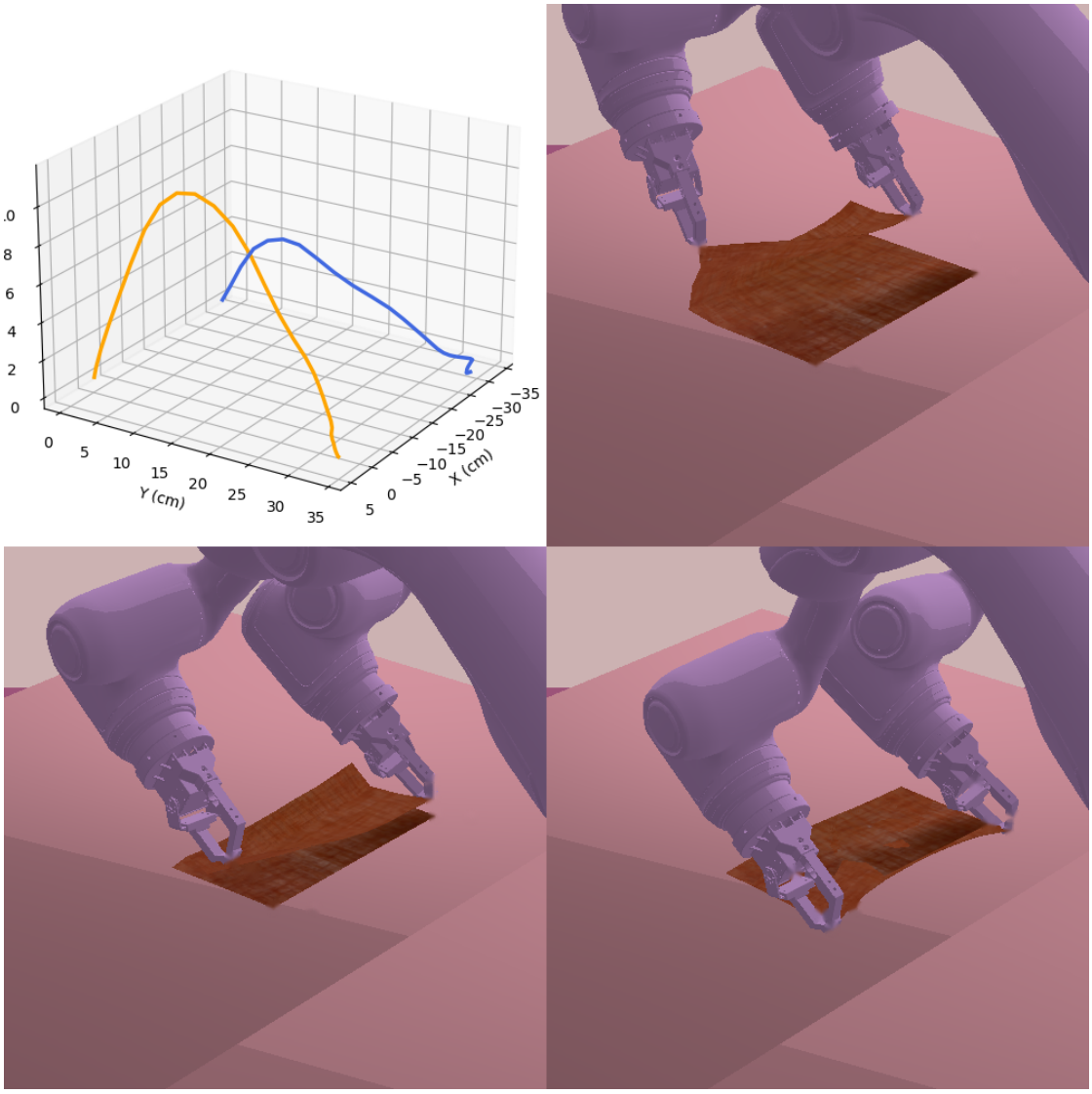


Figure 6.9.: PyBullet 6d ee 11, epoch 87.

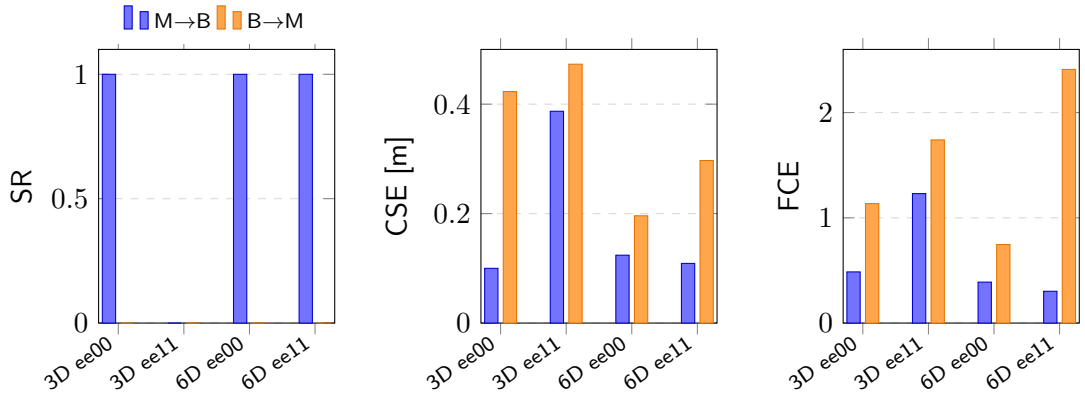


Figure 6.10.: Cross-engine evaluation results for the selected checkpoints. Left: success rate (SR). Middle: corner-sum error (CSE). Right: final constraint error (FCE).

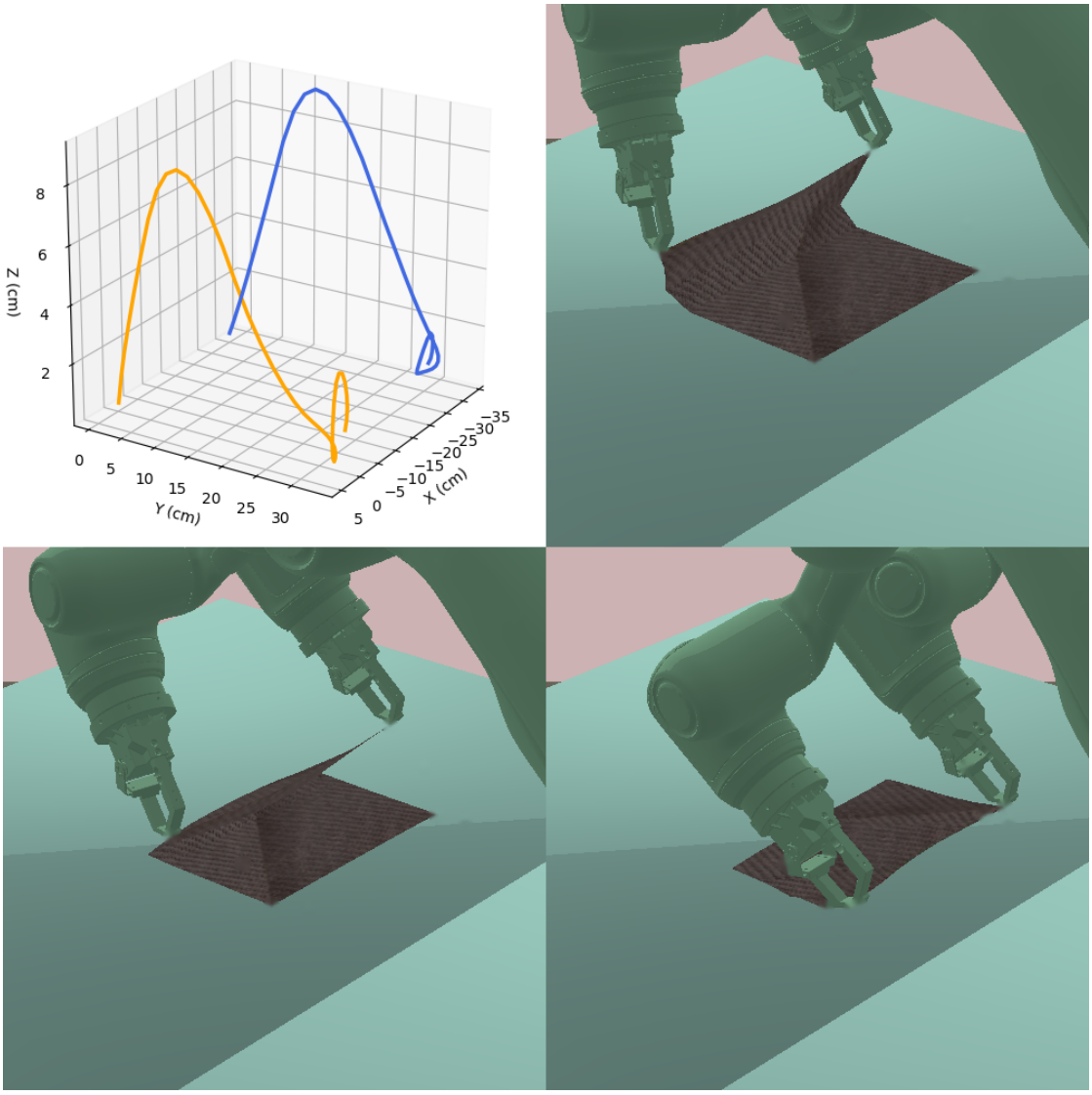


Figure 6.11.: MuJoCo 6d ee 11 in PyBullet, epoch 99.

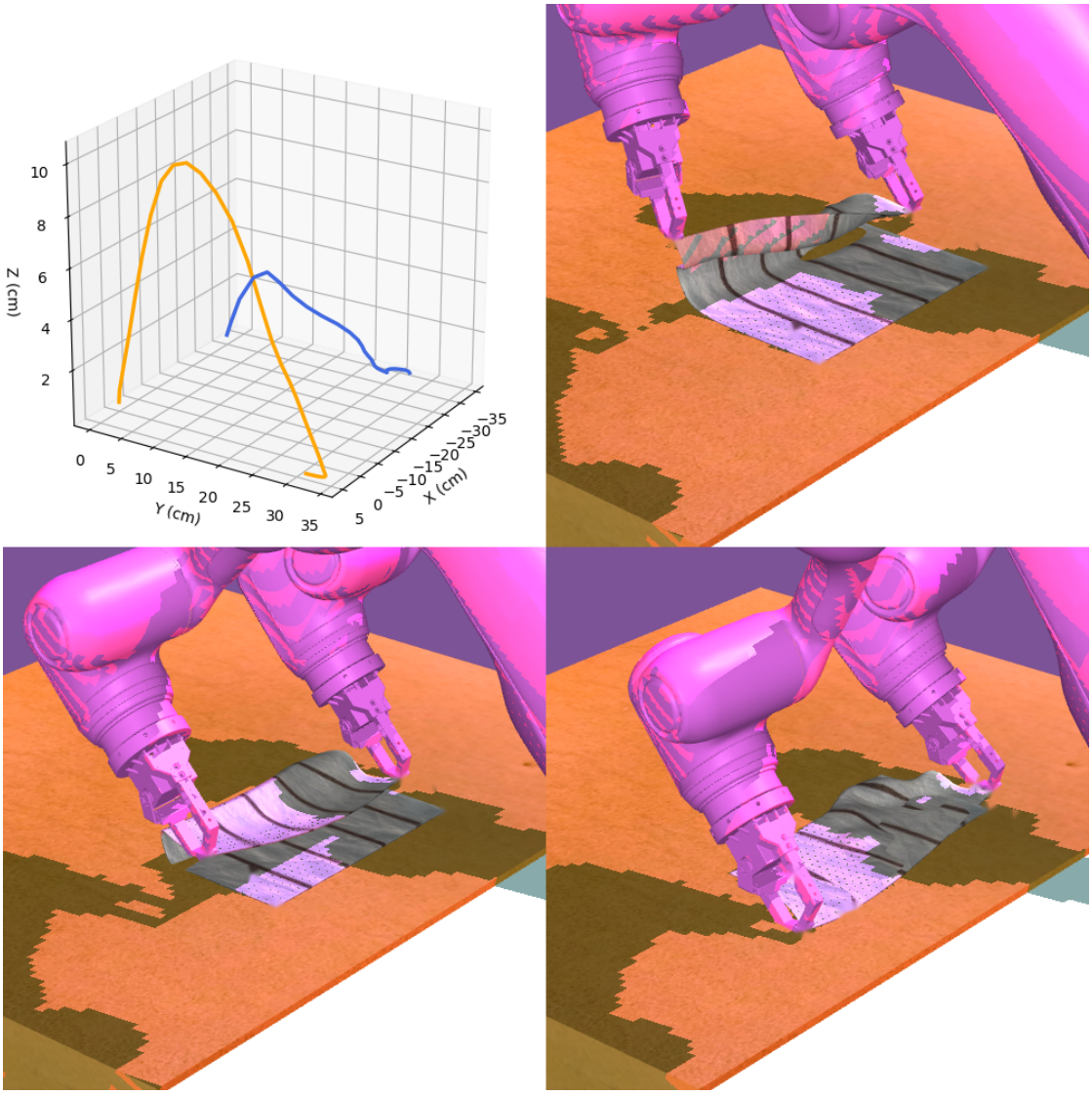


Figure 6.12.: PyBullet 6d ee 11 in MuJoCo, epoch 87.

7. Discussion

The results from Chapter 6 are now interpreted based on the research questions formulated in Section 1.3.1.

7.1. In-Engine Bimanual Dynamic Folding

Section 6.5 shows that visual-feedback policies for dynamic cloth folding can be trained and executed in a bimanual setup. The first research question can therefore be answered positively. This type of visual-feedback policy is suitable for bimanual dynamic cloth folding. All evaluated ee-mode configurations achieved an in-engine success rate of 1.0 (Figure 6.5). At the same time, the results also show that success rate alone is too rough to judge fold quality in this task. Success is triggered once the selected corners remain within `success_distance`=0.05 m for consecutive steps, whereas the fold itself is defined by 14 geometric constraints. For that reason, the final constraint error FCE is the more informative metric when comparing the quality of successful folds. In MuJoCo, `3D ee 0 0` reaches the highest FCE among the evaluated ee configurations (0.776), while `3D ee 1 1` achieves the lowest value (0.232). FCE values around 0.5 or below are usually associated with visibly cleaner folds and smoother trajectories, as illustrated by Figure 6.6.

PyBullet baselines consistently outperform MuJoCo baselines on both metrics (CSE and FCE) when domain randomization is disabled. This difference is more plausibly attributed to backend-specific cloth and contact dynamics than to the policy or controller interface, which is shared across both engines. Visually, the PyBullet cloth often remains flatter, whereas MuJoCo shows more residual wrinkling and a less consistent fold trajectory. In the randomized PyBullet runs, a different failure mode becomes visible. The cloth sometimes folds onto itself and the two grasped corners are pulled apart again after the main folding motion (see Figure 6.9).

Both backends use the same BioIK controller (Section 5.1.2), reward function (Section 5.4), and observation/action spaces (Section 5.2, Section 5.3). So the differences probably come down to how each engine handles deformable bodies. MuJoCo models cloth as a composite grid with tendon-based spring-damper connections (Paragraph 2.3.2). PyBullet uses triangulated soft bodies loaded via `loadSoftBody` with its own stiffness, damping, friction, and collision settings (Paragraph 2.3.2).

The contact and friction models differ too. MuJoCo uses `solimp`/`solref` constraint-based contacts with per-geometry friction. PyBullet has its own solver with separate friction and collision-margin settings (Section 4.2). Different engine dynamics push each policy toward different locally optimal trajectories.

With domain and cloth-size randomization turned on ($DR = 1$, $RCS = 1$), the FCE gap

shrinks. In 3D ee 1 1, MuJoCo hits 0.232 FCE versus 0.263 for PyBullet. In 6D ee 1 1, PyBullet does a bit better (0.241 vs. 0.351). This suggests the randomized policy isn't as tied to one set of physics parameters and learns something more general.

7.2. Cross-Engine Transfer

The most striking result is that none of the policies trained using PyBullet could be successfully transferred to MuJoCo. Although the policy with 6D ee 1 1 demonstrates a visually plausible folding process in MuJoCo (see Figure 6.12), this is not reflected in the metrics. One plausible explanation is that the transferred trajectory captures part of the raw folding motion but fails to satisfy MuJoCo's geometric constraints at the end of the episode. Since all PyBullet→MuJoCo transfers fail, the present results do not support the claim that $DR = 1$ and $RCS = 1$ improve transfer success in that direction. Three out of four MuJoCo-trained policies transfer to PyBullet with a success rate of 1.0 and an FCE below 0.5. This confirms the finding reported by Mohammed et al. that the direction of transfer plays a role depending on the task and learning algorithm [40]. The most plausible interpretation of the asymmetry is that PyBullet-trained policies are more specialized to PyBullet's cloth-contact dynamics, whereas MuJoCo-trained policies tolerate a wider range of deformations at the cost of less precise in-engine folds. That trade-off is consistent with the observation that MuJoCo policies often have higher in-engine FCE values, yet transfer more successfully to PyBullet. This also matches the trend summarized by Kaup et al. [14], where policies trained in MuJoCo transferred more reliably across simulators than policies trained in PyBullet.

The clearest advantage of 6D actions appears in the MuJoCo→PyBullet direction. There, both 6D policies outperform the corresponding 3D transfers in terms of FCE. This suggests that independent arm control gives the policy more freedom to adapt the folding trajectory to backend-specific deformations.

7.3. The Role of Domain Randomization

Domain randomization and cloth-size randomization have varying effects depending on the engine, action layout, and observation mode. The separate effect of switching from ee to ctrl is covered in the in-engine comparison above and interpreted further in Section 7.4. In the paired 3D ee setting, the combination of $DR = 1$ and $RCS = 1$ improves MuJoCo substantially, reducing the final CSE from 0.140 to 0.064 and the FCE from 0.776 to 0.232. In PyBullet, the same change has the opposite effect, increasing the CSE from 0.034 to 0.104 and the FCE from 0.118 to 0.263.

For 6D ee, the randomized setting again narrows the engine gap but does not provide a uniform gain. In MuJoCo, 6D ee 1 1 improves over 6D ee 0 0, reducing the CSE from 0.131 to 0.081 and the FCE from 0.505 to 0.351. In PyBullet, the same comparison improves the CSE slightly (0.089 to 0.075) but worsens the FCE slightly (0.216 to 0.241). The effect of randomization is therefore engine-dependent even within the same action layout. The PyBullet 3D ctrl 1 1 run also suggests that randomization can help the simpler control

setting in PyBullet. Relative to PyBullet `3D ctrl 0 0`, the final CSE drops from 0.075 to 0.040 and the FCE from 0.420 to 0.148.

The strongest interaction appears in the `ctrl` runs. For `6D ctrl`, the randomized `1 1` run fails in MuJoCo but succeeds in PyBullet. The non-randomized `0 0` run shows the opposite final outcome, that MuJoCo succeeds and PyBullet fails at the last epoch. This suggests that DR and RCS interact more strongly with the backend in the more difficult independent-action setting (6D) than in the simpler shared-action setting (3D).

The `3D ee 1 1` setting is the only MuJoCo-trained configuration that cannot be successfully transferred to PyBullet. The non-randomized `3D ee 0 0` transfers with an FCE of 0.486, while the randomized `3D ee 1 1` fails with an FCE of 1.230. One possible explanation is that the combination of shared 3D actions and randomization produces a policy that is robust within MuJoCo's randomized training distribution but remains too specialized to transfer to PyBullet. By contrast, `6D ee 1 1` achieves the best MuJoCo→PyBullet transfer result. The learning curves in Section 6 also show that the randomized 6D policies in MuJoCo exhibit more fluctuations and slower progress than the non-randomized 6D policies, which is consistent with a robustness-versus-sample-efficiency trade-off. Figure 6.4 compared to Figure 6.1 makes this particularly clear. This is consistent with the general observation that training on a broader distribution of environments entails a trade-off between sample efficiency and robustness [38].

7.4. Robot Observation Modes

The `ee` mode provides end-effector position, velocity, and controller target in the internal frame. The `ctrl` mode provides only the previous raw action padded with zeros.

As mentioned earlier, all evaluated `ee` mode variants achieved a success rate of 1.0 in both backends. In MuJoCo, the `6D ctrl 1 1` configuration achieved the worst results with a success rate of 0.0 and an FCE of 1.867 (Figure 6.5). The same configuration achieves a success rate of 1.0 in PyBullet with an FCE of 0.179.

In the 3D baseline, MuJoCo benefits from `ctrl` (FCE 0.776 → 0.454). The PyBullet 3D baseline shows the opposite (FCE 0.118 → 0.420). However, the additional PyBullet `3D ctrl 1 1` run reaches an FCE of 0.148, so `ctrl` can still work well in the simpler shared-action setting when DR and RCS are enabled.

In the 6D baseline, `ctrl` is less reliable than `ee`. In MuJoCo, `6D ctrl 0 0` and `6D ee 0 0` remain similar in FCE (0.514 vs. 0.505), but PyBullet collapses in `6D ctrl 0 0` during the last epoch.

Taken together, this suggests that `ctrl` observations are less consistent than `ee` observations across the currently available runs, a finding supported in particular by the MuJoCo failure in `6D ctrl 1 1`. The most plausible reason is that `ctrl` only provides the previous action, not the current robot state. When the realized motion differs from the intended motion because of controller dynamics, contacts, or randomization, the policy must infer that mismatch indirectly from the image alone. The `ee` mode is more robust because it adds the current end-effector position, velocity, and controller target as explicit state information.

7.5. 3D vs. 6D Action Layouts

With the exception of the `6D ctrl 1 1` MuJoCo failure, the in-engine results do not show a clear advantage of one layout over the other. The differences between randomized and non-randomized runs are also not clear, although the randomized 6D runs tend to yield slightly better or comparable results. However, a clearer picture emerges when it comes to cross-engine transfer. The MuJoCo-to-PyBullet transfers using 6D policies achieve lower FCE values than the corresponding 3D transfers in both compared settings (0.389 vs. 0.486 for `ee 0 0`, and 0.302 vs. 1.230 for `ee 1 1`). The left and right arm trajectories are also different, suggesting better adaptability due to independent arm control (see Figure 6.8). Under gravity, the fabric stretches in the middle of the grasped corners, which can be better compensated for with a 6D layout. In this setup, the additional dimensionality of the 6D action therefore appears worthwhile for cross-engine robustness.

7.6. Generalization to Unseen Cloth Configurations

Table 6.5 shows the results of the generalization tests on an unseen cloth texture (`colors16.png`), different fabric sizes ($25\,\mathrm{cm}$ and $40\,\mathrm{cm}$), and different friction coefficients. 31 out of 32 policy-condition combinations achieved a success rate of 1.0, with most FCE values below 0.5. The only outlier is the PyBullet `3D ee 0 0` configuration on a $40\,\mathrm{cm}$ fabric, which achieves a success rate of 0.65 and an FCE of 0.421. Even baselines without domain randomization ($DR = 0$, $RCS = 0$) generalize well to unknown textures, varying friction, and different fabric sizes. One reason for this could be the 100×100 grayscale image training camera compression, which reduces the visual differences between textures [5]. Policies trained with domain randomization do not improve every individual comparison, but they do tend to reduce FCE in several of the MuJoCo tests and yield a lower average FCE across all unseen-condition evaluations. Across all policies, changes in cloth size have a clearer effect than the tested friction variations, whose impact on FCE remains comparatively small.

7.7. Limitations

Only 18 of the possible 32 configurations (Table 6.2) were trained, which limits the possibility of pairwise comparisons. Since it became evident early that the `ee` mode is more robust, fewer `ctrl` mode configurations were trained and compared across backends. This incomplete matrix also makes it difficult to separate the effects of DR and RCS cleanly in every setting.

Success rate is also a limited metric in this setup. The threshold `success_distance`=0.05 m is fairly rough, as already mentioned. Success is determined from selected corner distances rather than from all 14 fold constraints. As a result, success rate is less informative than the FCE metric for judging final fold quality.

The generalization tests and evaluations do not show how policies fold cloth at different widths, but only on square or rectangular shapes.

Each training run was performed only once with a single seed, which introduces the possibility of seed-dependent variance in the results.

The PyBullet physics parameters were adjusted manually, as described in Section 4.2.2. This was based primarily on visual feedback from demo trajectories. The DR distributions (Table 4.5) were designed for in-engine variability and material diversity, not specifically for cross-engine transferability. It appears that the variations within the DR distribution were not chosen to be sufficiently large.

8. Conclusion

Many studies focus on the training of single-arm robots. This thesis outlines the steps required to transition from an existing single-arm setup to a bimanual setup. Step one is to replace the simulation environment with a bimanual robot configuration. Here, an `xacro` robot description was used, which was converted to `urdf` and then finally to `MJCF` (Section 4.3). For MuJoCo, the resulting `arena_diana.xml` was adapted to keep the cloth setup and most of the domain-randomization logic largely unchanged. The end-effector and gripper control-point positions must be defined. Central environment functions such as joint reset, stepping, observation generation, reward evaluation, trajectory logging, and workspace handling must be extended from one arm to two. Camera poses must be chosen so that as much of the cloth as possible is visible.

Step two is the implementation of a suitable controller that accounts for collisions with environmental objects such as posts, the table, and between the arms themselves. The workspace should be configured so that as few constraints as possible need to be handled at once, in order to ensure stability during movement (Section 5.1.2). A hybrid BioIK-based controller solution was implemented here. Cartesian targets are clipped to the workspace bounds, while BioIK solves joint targets with collision checking. The height of `additional_table_link` was reduced so that BioIK can still find collision-free solutions close to the table. A test script `sanity_test.sh` was very helpful for the BioIK integration. Using hold tests, in which the robot must maintain a pose, the static stability of the controller and the bias-torque compensation path were checked. Collision handling between robot components was also checked with the sanity tests. The move tests were then used to determine the precision of the robot's movements and to tune parameters. Smoke and episode tests were used regularly to verify the entire pipeline and identify performance improvements.

Step three is adapting the training setup, policy learning, and reward to the new environment. The observation and action spaces must be adapted to support two arms (Sections 5.2, 5.3). The reward function was adapted on a geometric basis and separated from the success signal (Section 5.4). The original implementation of SAC with HER did not work well. Lifting the cloth was inconsistent and never really settled into a reliable behavior. Section 5.5.1 goes through the SAC and HER changes that made the setup trainable.

The last step was tuning the training, physics, and DR parameters (Section 5.5.3, Table 4.2, Section 4.4, Table 4.5). Chapter 7 goes through the results and answers the research questions. The resulting trajectories and training curves in section 6.5 are similar to those of the single-arm reference by Hietala et al. (Hietala et al., 2022). With the enlarged cloth, it is now raised to approximately 10–15 cm instead of around 9 cm, as illustrated in Figure 6.6. In a 6D layout, the arms do not move symmetrically but react individually to the dynamics of the cloth, as shown in Figure 6.8.

9. Future Work

The training matrix (Table 6.2) covers 17 of 32 possible factor combinations. Both backends include baseline runs and several fully activated configurations, but many clean pairwise comparisons are still missing. This is especially true for `ctrl` mode and for separating the effects of DR and RCS. Adding those runs would make the individual factor effects easier to interpret. Training each configuration with multiple seeds would also show how much variance comes from the seed alone.

The static domain randomization in PyBullet was developed for in-engine cloth diversity, not for bridging the gap between MuJoCo and PyBullet (Section 7.3). This could be extended and adapted to better cover the target engine. An alternative would be to train both engines simultaneously by retrieving rollouts from MuJoCo and PyBullet within the same training run.

This thesis focused on sim-to-sim evaluation. The next step is to transfer the learned policies to a physical Diana7 robot. Hietala et al. have already demonstrated a successful sim-to-real transfer for the single-arm robot. For real-world deployment, it would make sense to use the MuJoCo training as a starting point, as it exhibits greater tolerance to physics mismatch.

For PyBullet, a ROS bridge such as `ros_pybullet_interface` could be used to create a common interface between PyBullet and ROS, which would also help with later transfer to real hardware [76]. Although movements have already been successfully synchronized via a prototype with the framework in PyBullet and ROS, this was not pursued further as it quickly went beyond the scope of this thesis. The existing BioIK service interface used in the simulation (Section 5.1.2) could likely be reused on real hardware, since it already operates through ROS.

Several extensions to a richer bimanual policy are possible. The task and site mapping assume a quadratic or rectangular cloth. Extending it to clothing like T-shirts would show whether the policy can generalize to more complex shapes.

The task could be made more complex through multi-stage folding, for example by first folding in half and then folding again. Grasping and regrasping within multi-stage folding could be addressed using the grasping strategy proposed by Fiedler et al., in which a fold is specifically created for grasping through controlled pressing [77].

The MuJoCo cloth uses a legacy `composite grid` model, which is deprecated in newer versions of MuJoCo (Section 4.2.1). It would be worth evaluating a switch to the `flex` deformable model [45]. That would not be a drop-in change, since the current site indexing and grasping mechanism depend on the composite-grid structure and would need to be adapted to the `flex` API.

In PyBullet, the cloth parameters were tuned manually, starting from replayed reference demonstrations and then adjusting the Diana setup by hand (Section 4.2.2). A systematic

parameter identification process, such as measuring cloth drop and drape behavior and adjusting the simulation parameters to match them, would make the tuning more reproducible and could reduce the cross-engine gap.

Ultimately, the results should also be compared with other bimanual approaches to evaluate the relative strength of the pipeline used here.

Bibliography

[1] E. Coumans and Y. Bai, *PyBullet Quickstart Guide*. Bullet Physics / bullet3 (GitHub repository), 2022. [Online]. Available: https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, ser. Adaptive Computation and Machine Learning. The MIT Press, Feb. 1998.

[3] S. Starke, N. Hendrich, and J. Zhang, "Memetic Evolution for Generic Full-Body Inverse Kinematics in Robotics and Animation," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 3, pp. 406–420, Jun. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8449979

[4] opencv, "Camera Calibration and 3D Reconstruction," 2026, published: OpenCV Documentation. [Online]. Available: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html

[5] J. Hietala, D. Blanco-Mulero, G. Alcan, and V. Kyrki, "Learning Visual Feedback Control for Dynamic Cloth Folding," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2022, pp. 1455–1462, arXiv:2109.04771 [cs]. [Online]. Available: http://arxiv.org/abs/2109.04771

[6] bruehl, "bruehldev/diana," 2026. [Online]. Available: https://github.com/bruehldev/diana

[7] J. Hietala, "hietalajulius/dynamic-cloth-folding," Jan. 2026, original-date: 2020-09-14T14:40:16Z. [Online]. Available: https://github.com/hietalajulius/dynamic-cloth-folding

[8] J. Matas, S. James, and A. J. Davison, "Sim-to-Real Reinforcement Learning for Deformable Object Manipulation," in *Proceedings of The 2nd Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, vol. 87. PMLR, 2018, pp. 734–743. [Online]. Available: https://proceedings.mlr.press/v87/matas18a.html

[9] J. Sanchez, J.-A. Corrales, B.-C. Bouzgarrou, and Y. Mezouar, "Robotic manipulation and sensing of deformable objects in domestic and industrial applications: a survey," *The International Journal of Robotics Research*, vol. 37, no. 7, pp. 688–716, Jun. 2018. [Online]. Available: https://journals.sagepub.com/doi/10.1177/0278364918779698

[10] A. Doumanoglou, J. Stria, G. Peleka, I. Mariolis, V. Petrik, A. Kargakos, L. Wagner, V. Hlavac, T.-K. Kim, and S. Malassiotis, "Folding Clothes Autonomously: A Complete

Pipeline," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1461–1478, Dec. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7589002/

[11] A. Edsinger and C. C. Kemp, "Two Arms Are Better Than One: A Behavior Based Control System for Assistive Bimanual Manipulation," in *Recent Progress in Robotics: Viable Robotic Service to Human: An Edition of the Selected Papers from the 13th International Conference on Advanced Robotics*, S. Lee, I. H. Suh, and M. S. Kim, Eds. Berlin, Heidelberg: Springer, 2008, pp. 345–355. [Online]. Available: https://doi.org/10.1007/978-3-540-76729-9_27

[12] R. L. Ahmad Shauri and K. Nonami, "Assembly manipulation of small objects by dual-arm manipulator," *Assembly Automation*, vol. 31, pp. 263–274, Aug. 2011.

[13] T. Weng, S. Bajracharya, Y. Wang, K. Agrawal, and D. Held, "FabricFlowNet: Bimanual Cloth Manipulation with a Flow-based Policy," Apr. 2022, arXiv:2111.05623 [cs]. [Online]. Available: http://arxiv.org/abs/2111.05623

[14] M. Kaup, C. Wolff, H. Hwang, J. Mayer, and E. Bruni, "A Review of Nine Physics Engines for Reinforcement Learning Research," Jul. 2024, arXiv:2407.08590 [cs] version: 1. [Online]. Available: http://arxiv.org/abs/2407.08590

[15] MuJoCo, "MuJoCo — Advanced Physics Simulation," 2026. [Online]. Available: https://mujoco.org/

[16] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement Learning in Robotics: A Survey," *Journal*, vol. 32, pp. 1238–1274, 2013. [Online]. Available: https://doi.org/10.1177/0278364913495721

[17] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1-2, pp. 99–134, May 1998. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S00043702 9800023X

[18] M. T. J. Spaan, "Partially Observable Markov Decision Processes," *Reinforcement Learning*, pp. 387–414, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-27645-3_12

[19] M. Lauri, D. Hsu, and J. Pajarinen, "Partially Observable Markov Decision Processes in Robotics: A Survey," *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 21–40, Feb. 2023, arXiv:2209.10342 [cs]. [Online]. Available: http://arxiv.org/abs/2209.10342

[20] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal Value Function Approximators," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 37, 2015, pp. 1312–1320. [Online]. Available: https://proceedings.mlr.press/v37/schaul15.html

[21] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight Experience Replay," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/hash/453fadbd8a1a3af50a9df4df899537b5-Abstract.html

[22] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," Aug. 2018, arXiv:1801.01290 [cs]. [Online]. Available: http://arxiv.org/abs/1801.01290

[23] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft Actor-Critic Algorithms and Applications," Jan. 2019, arXiv:1812.05905 [cs]. [Online]. Available: http://arxiv.org/abs/1812.05905

[24] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control | Guide books | ACM Digital Library*. Springer Publishing Company, Incorporated, Dec. 2008. [Online]. Available: https://dl.acm.org/doi/book/10.5555/1524151

[25] K. M. Lynch and F. C. Park, *Modern Robotics*. Cambridge University Press, May 2017, google-Books-ID: 5NzFDgAAQBAJ.

[26] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *IEEE Journal on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, Feb. 1987. [Online]. Available: http://ieeexplore.ieee.org/document/1087068/

[27] Y. Nakamura, *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley Publishing Company, 1991, google-Books-ID: hp4QAQAAMAAJ.

[28] T. Yoshikawa, "Manipulability of Robotic Mechanisms," *The International Journal of Robotics Research*, vol. 4, no. 2, pp. 3–9, Jun. 1985. [Online]. Available: https://journals.sagepub.com/doi/10.1177/027836498500400201

[29] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," 2009.

[30] J. M. O'Kane, *A gentle introduction to ROS*, version 2.1.1 (3e3d9c5), generated in november 20, 2014 ed. Columbia, SC: Jason M. O'Kane, 2014.

[31] TAMS-Group, "TAMS-Group/bio_ik_service," 2026. [Online]. Available: https://github.com/TAMS-Group/bio_ik_service

[32] ——, "bio_ik_msgs (in TAMS-Group/bio_ik_service)," 2026. [Online]. Available: https://github.com/TAMS-Group/bio_ik_service/tree/master/bio_ik_msgs

[33] C. Smith, Y. Karayiannidis, L. Nalpantidis, X. Gratal, P. Qi, D. V. Dimarogonas, and D. Kragic, "Dual arm manipulation—A survey," *Robotics and Autonomous Systems*, vol. 60, no. 10, pp. 1340–1353, Oct. 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S092188901200108X

[34] E. Coumans and Y. Bai, "PyBullet Quickstart Guide," 2018.

[35] MuJoCo, "Simulation - MuJoCo Documentation," 2026. [Online]. Available: https://mujoco.readthedocs.io/en/latest/programming/simulation.html

[36] D. Baraff and A. Witkin, "Large Steps in Cloth Simulation," 1998.

[37] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson, "Physically Based Deformable Models in Computer Graphics," *Computer Graphics Forum*, vol. 25, no. 4, pp. 809–836, Dec. 2006. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2006.01000.x

[38] F. Muratore, F. Ramos, G. Turk, W. Yu, M. Gienger, and J. Peters, "Robot Learning From Randomized Simulations: A Review," *Frontiers in Robotics and AI*, vol. 9, p. 799893, Apr. 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frobt.2022.799893/full

[39] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position Based Dynamics," in *Workshop in Virtual Reality Interactions and Physical Simulation (VRIPHYS)*, 2006. [Online]. Available: https://matthias-research.github.io/pages/publications/posBasedDyn.pdf

[40] A. P. Mohammed and M. Valdenegro-Toro, "Can Reinforcement Learning for Continuous Control Generalize Across Physics Engines?" Oct. 2020, arXiv:2010.14444 [cs]. [Online]. Available: http://arxiv.org/abs/2010.14444

[41] Wikipedia, "Convex function," Sep. 2025, page Version ID: 1312188777. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Convex_function&oldid=1312188777

[42] C. MuJoCo, "Computation - MuJoCo Documentation," 2026. [Online]. Available: https://mujoco.readthedocs.io/en/stable/computation/index.html

[43] Bullet Physics Project, *Bullet Physics SDK User Manual*. Bullet Physics, 2024. [Online]. Available: https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/Bullet_User_Manual.pdf

[44] btSequentialImpulseConstraint, "Bullet Collision Detection & Physics Library: btSequentialImpulseConstraintSolver Class Reference," 2026. [Online]. Available: https://pybullet.org/Bullet/BulletFull/classbtSequentialImpulseConstraintSolver.html

[45] M. MuJoCo, "Modeling - MuJoCo Documentation," 2026. [Online]. Available: https://mujoco.readthedocs.io/en/stable/modeling.html

[46] mujoco, "XML Reference - MuJoCo Documentation," 2026. [Online]. Available: https://mujoco.readthedocs.io/en/3.1.3/XMLreference.html

[47] btSoftBody, "Bullet Collision Detection & Physics Library: btSoftBody Class Reference," 2026. [Online]. Available: https://pybullet.org/Bullet/BulletFull/classbtSoftBody.html

[48] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, Australia, May 2018. [Online]. Available: https://xbpeng.github.io/projects/SimToReal/SimToReal_2018.pdf

[49] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," Mar. 2017, arXiv:1703.06907 [cs]. [Online]. Available: http://arxiv.org/abs/1703.06907

[50] Y. Bouteiller, S. Ramstedt, G. Beltrame, C. Pal, and J. Binas, "Reinforcement Learning with Random Delays," in *International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: https://openreview.net/pdf/744fcf663d9a7335f90ed1ec81d97b3661166e56.pdf

[51] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[52] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, "Albumentations: Fast and Flexible Image Augmentations," *Information*, vol. 11, no. 2, p. 125, 2020. [Online]. Available: https://www.mdpi.com/2078-2489/11/2/125

[53] C. Shorten and T. M. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *Journal of Big Data*, vol. 6, p. 60, 2019. [Online]. Available: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0197-0

[54] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[55] Stanford CS231n, "Convolutional Neural Networks (CNNs / ConvNets)," 2026, published: Course notes. [Online]. Available: https://cs231n.github.io/convolutional-networks/

[56] A. Longhini, M. C. Welle, Z. Erickson, and D. Kragic, "AdaFold: Adapting Folding Trajectories of Cloths via Feedback-loop Manipulation," Dec. 2024, arXiv:2403.06210 [cs]. [Online]. Available: http://arxiv.org/abs/2403.06210

[57] T. Lips, V.-L. D. Gusseme, and F. wyffels, "Learning Keypoints for Robotic Cloth Manipulation using Synthetic Data," May 2024, arXiv:2401.01734 [cs]. [Online]. Available: http://arxiv.org/abs/2401.01734

[58] A. Majumdar and V. Pacelli, "Fundamental Performance Limits for Sensor-Based Robot Control and Policy Learning," in *Robotics: Science and Systems XVIII*. Robotics: Science and Systems Foundation, Jun. 2022. [Online]. Available: http://www.roboticsproceedings.org/rss18/p036.pdf

[59] H. Ha and S. Song, "FlingBot: The Unreasonable Effectiveness of Dynamic Manipulation for Cloth Unfolding," in *Proceedings of the 5th Conference on Robot Learning*. PMLR, Jan. 2022, pp. 24–33. [Online]. Available: https://proceedings.mlr.press/v164/ha22a.html

[60] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, "Sim-to-Real via Sim-to-Sim: Data-Efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2019/papers/James_Sim-To-Real_via_Sim-To-Sim_Data-Efficient_Robotic_Grasping_via_Randomized-To-Canonical_Adaptation_Networks_CVPR_2019_paper.pdf

[61] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," Jun. 2016, arXiv:1606.01540 [cs]. [Online]. Available: http://arxiv.org/abs/1606.01540

[62] bio_ik, "TAMS-Group/bio_ik," Jan. 2026, original-date: 2017-09-22T05:40:17Z. [Online]. Available: https://github.com/TAMS-Group/bio_ik

[63] MoveIt, "Plugins | MoveIt," 2026. [Online]. Available: https://moveit.ai/documentation/plugins/

[64] J. Hietala, "hietalajulius/dynamic-cloth-folding," Jul. 2025, original-date: 2020-09-14T14:40:16Z. [Online]. Available: https://github.com/hietalajulius/dynamic-cloth-folding

[65] bruehldev, "bruehldev/dynamic-cloth-folding," Mar. 2026, original-date: 2025-09-21T16:46:24Z. [Online]. Available: https://github.com/bruehldev/dynamic-cloth-folding

[66] D. Blanco-Mulero, O. Barbany, G. Alcan, A. Colomé, C. Torras, and V. Kyrki, "Benchmarking the Sim-to-Real Gap in Cloth Manipulation," *IEEE Robotics and Automation Letters*, vol. 9, no. 3, pp. 2981–2988, Mar. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10417136/

[67] TAMS, "TAMS-Group/tams_diana_7," 2024. [Online]. Available: https://github.com/TAMS-Group/tams_diana_7

[68] ——, "TAMS-Group/libdiana7," 2024. [Online]. Available: https://github.com/TAMS-Group/libdiana7

[69] ——, "TAMS-Group/diana7_ros," 2022. [Online]. Available: https://github.com/TAMS-Group/diana7_ros

[70] "TAMS-Group/q_gripper," Sep. 2025, original-date: 2024-06-28T19:58:13Z. [Online]. Available: https://github.com/TAMS-Group/q_gripper

[71] "TAMS-Group/agile_gripper," Feb. 2025, original-date: 2025-02-21T13:18:04Z. [Online]. Available: https://github.com/TAMS-Group/agile_gripper

[72] "realsenseai/realsense-ros," Feb. 2026, original-date: 2016-02-23T23:52:58Z. [Online]. Available: https://github.com/realsenseai/realsense-ros

[73] rraetz, "rraetz/xacro_to_urdf:," 2025. [Online]. Available: https://gitee.com/resource_3/xacro_to_urdf

[74] "bruehldev/diana at osc." [Online]. Available: https://github.com/bruehldev/diana/tree/osc

[75] "rlkit/rlkit/envs/wrappers/normalized_box_env.py at master · rail-berkeley/rlkit." [Online]. Available: https://github.com/rail-berkeley/rlkit/blob/master/rlkit/envs/wrappers/normalized_box_env.py

[76] "ros-pybullet/ros_pybullet_interface," Dec. 2025, original-date: 2022-11-14T17:05:58Z. [Online]. Available: https://github.com/ros-pybullet/ros_pybullet_interface

[77] N. Fiedler, Y. Jonetzko, and J. Zhang, "A Multimodal Pipeline for Grasping Fabrics from Flat Surfaces with Tactile Slip and Fall Detection," in *2023 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Koh Samui, Thailand: IEEE, Dec. 2023, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10354620/

Appendices

A. Appendix A: Additional Native-Engine Rollouts for Selected Checkpoints

Figures .1, .2, .3, and .4 show the remaining native-engine rollout composites for the checkpoint set from Table 6.3. As in Chapter 6 explained, the first panel shows the recorded end-effector trajectory and the remaining panels show steps 10, 15, and the final step from the corner-camera view.

A.1. Selected `ctrl` checkpoints by lowest FCE

Each checkpoint was then reevaluated for 20 rollouts in its native engine with seed 0. Table .1 reports the native-engine success rate (SR), corner-sum error (CSE), and final constraint error (FCE).

A.2. Cross-engine evaluation of the selected `ctrl` checkpoints

The same three checkpoints were also evaluated in the opposite backend with the cross-engine evaluation tool for 20 rollouts. Table .2 reports the resulting success rate (SR), corner-sum error (CSE), and final constraint error (FCE).

B. Appendix B: Additional Training Curves

Figure .10 shows the training curves for the PyBullet `3D ctrl 1 1` run. This configuration uses domain randomization and cloth-size randomization ($DR = 1$, $RCS = 1$).

C. Usage of gKI

In this thesis, gKI was used for the following tasks:

1. Summarizing scientific sources cited in the thesis
2. Debugging and assistance with programming
3. Generating tests
4. Generating tools for extracting results and creating cloth in PyBullet
5. Reviewing code and written texts

Table .1.: Native-engine rerun results for the selected `ctrl` checkpoints.

Setting	Epoch	SR	CSE [m]	FCE
`3D ctrl 1 0` (M)	82	1.00	0.063	0.387
`3D ctrl 1 1` (B)	82	1.00	0.072	0.196
`6D ctrl 1 1` (B)	94	1.00	0.070	0.192

Table .2.: Cross-engine rerun results for the selected `ctrl` checkpoints.

Setting	Direction	Epoch	SR	CSE [m]	FCE
`3D ctrl 1 0`	M→B	82	0.85	0.119	0.554
`3D ctrl 1 1`	B→M	82	0.00	0.461	1.531
`6D ctrl 1 1`	B→M	94	0.00	0.594	2.639

6. Assisting with the creation of diagrams and formulas
7. DeepL Translator and DeepL Write for translation and writing assistance

All generated content was reviewed and adjusted as necessary.

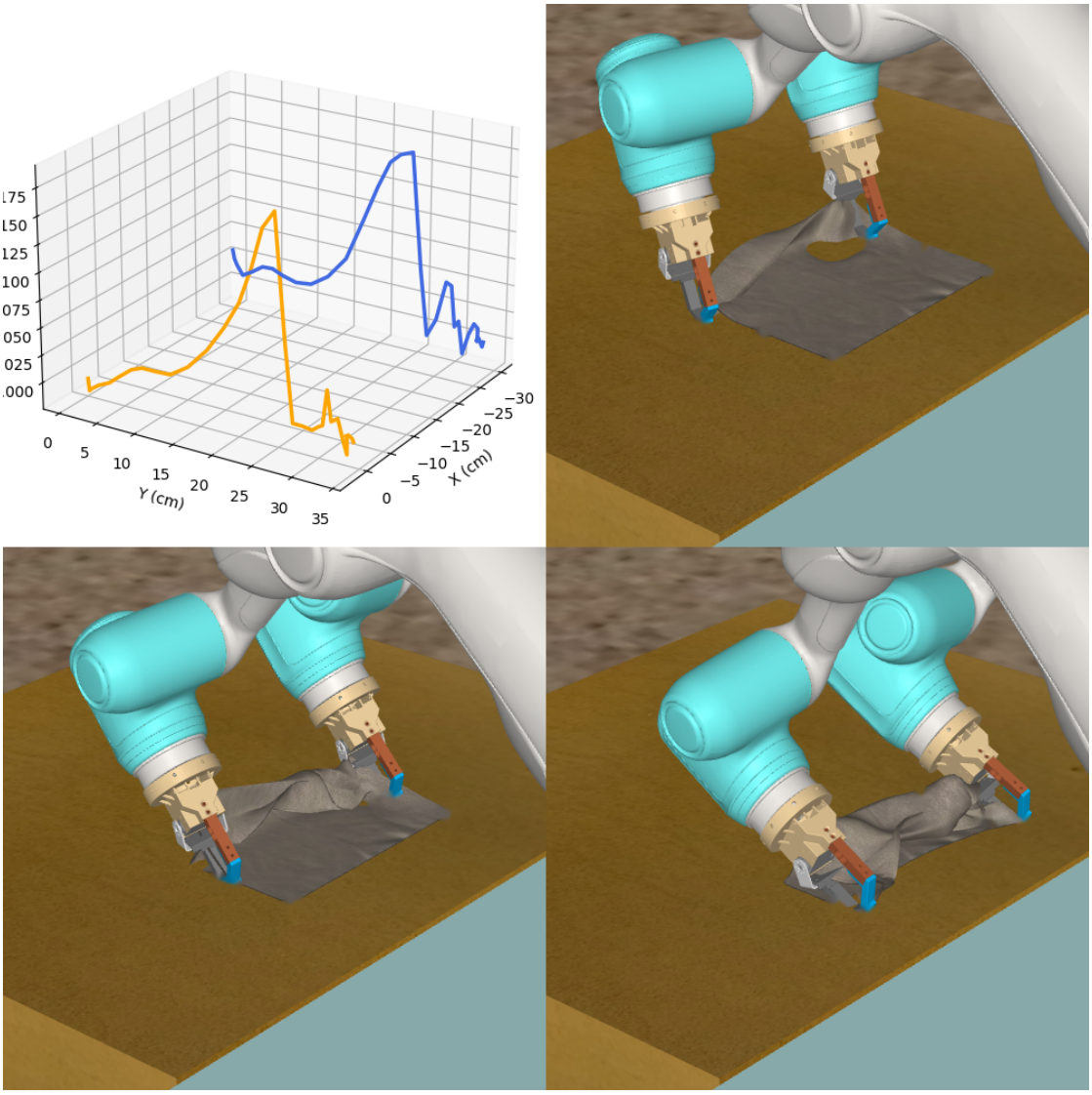


Figure .1.: MuJoCo 3d ee 00, epoch 33. (Worst FCE among all configurations.)

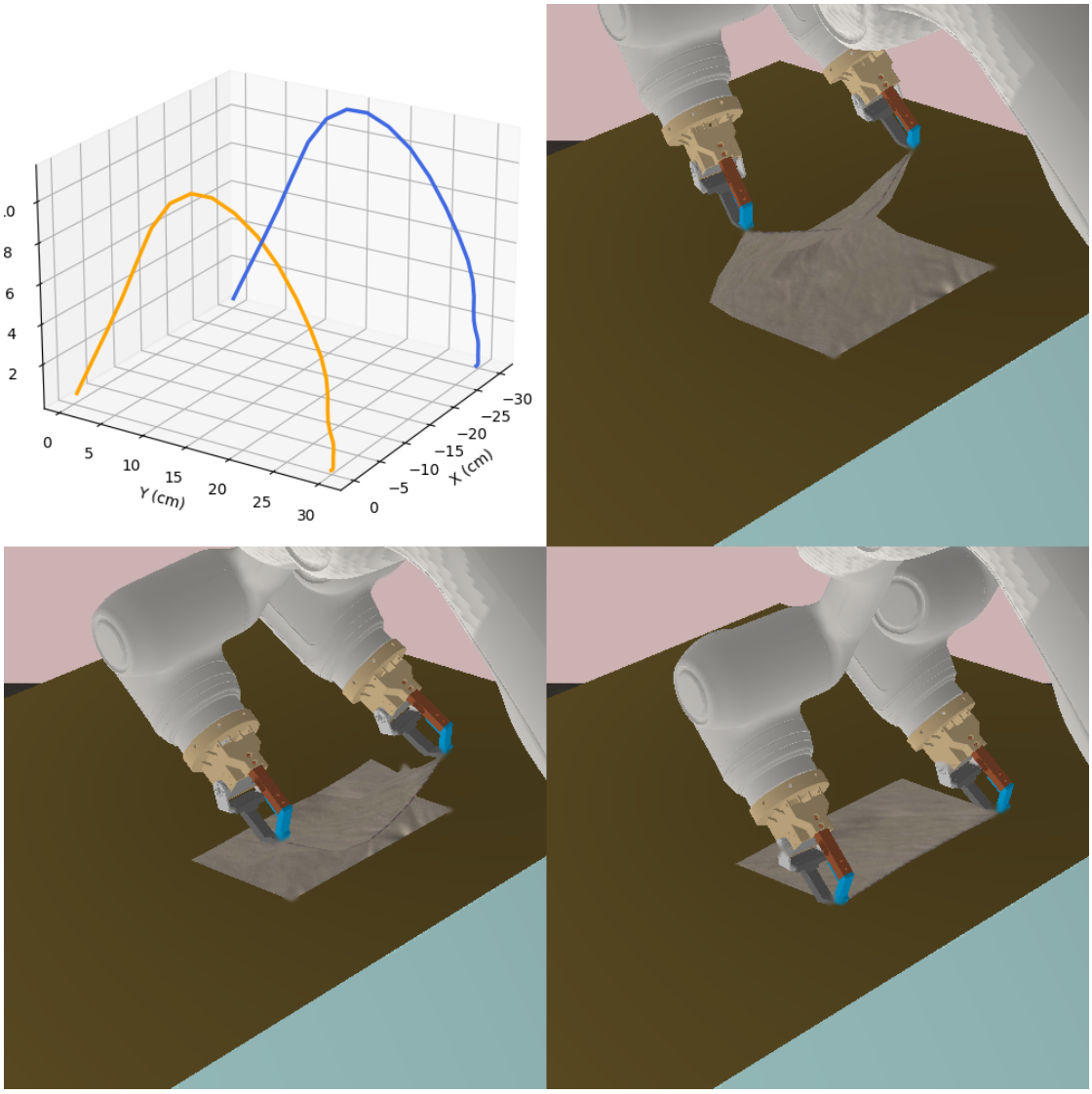


Figure .2.: PyBullet 3d ee 00, epoch 62.

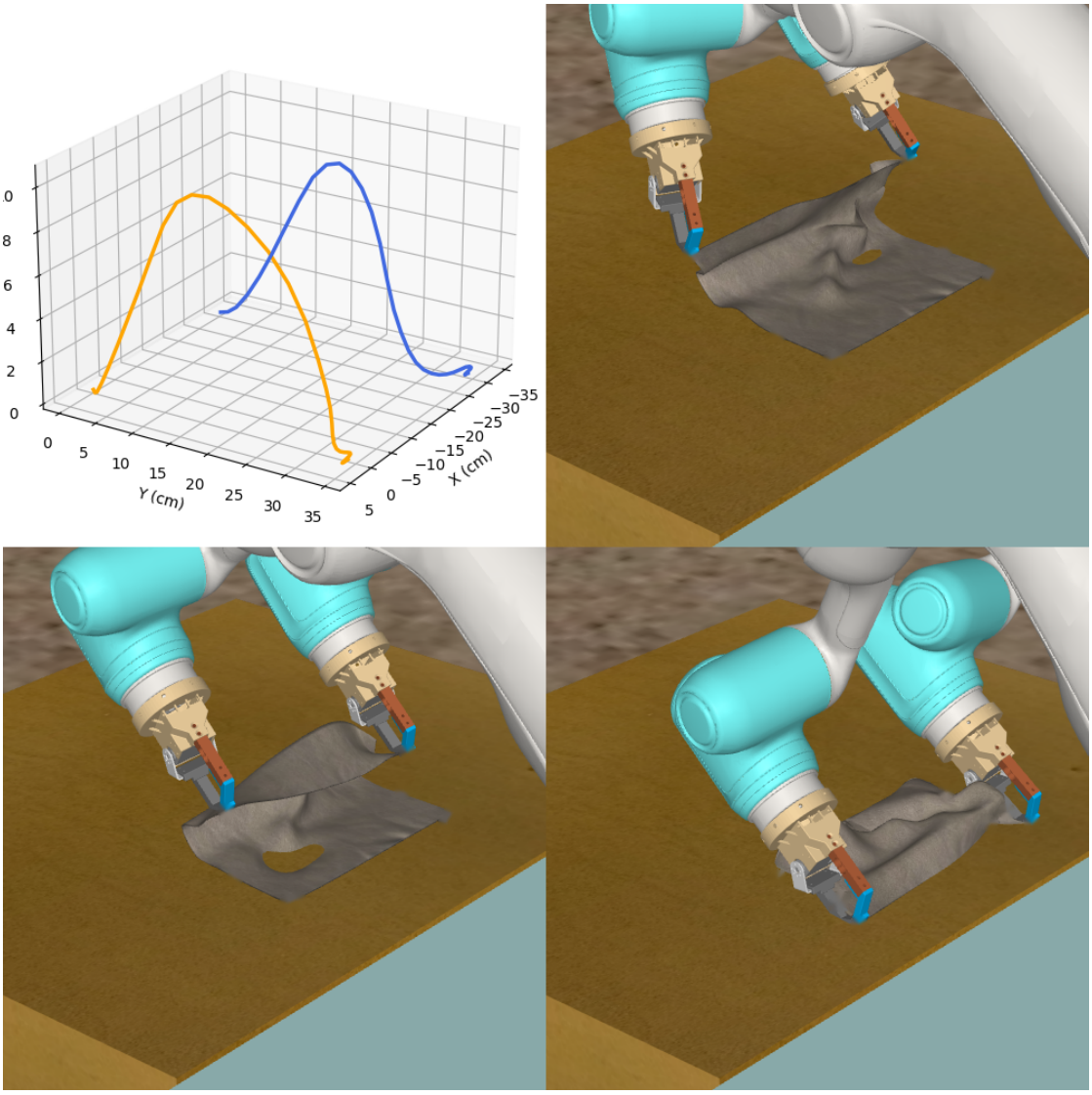


Figure .3.: MuJoCo 6d ee 00, epoch 65.

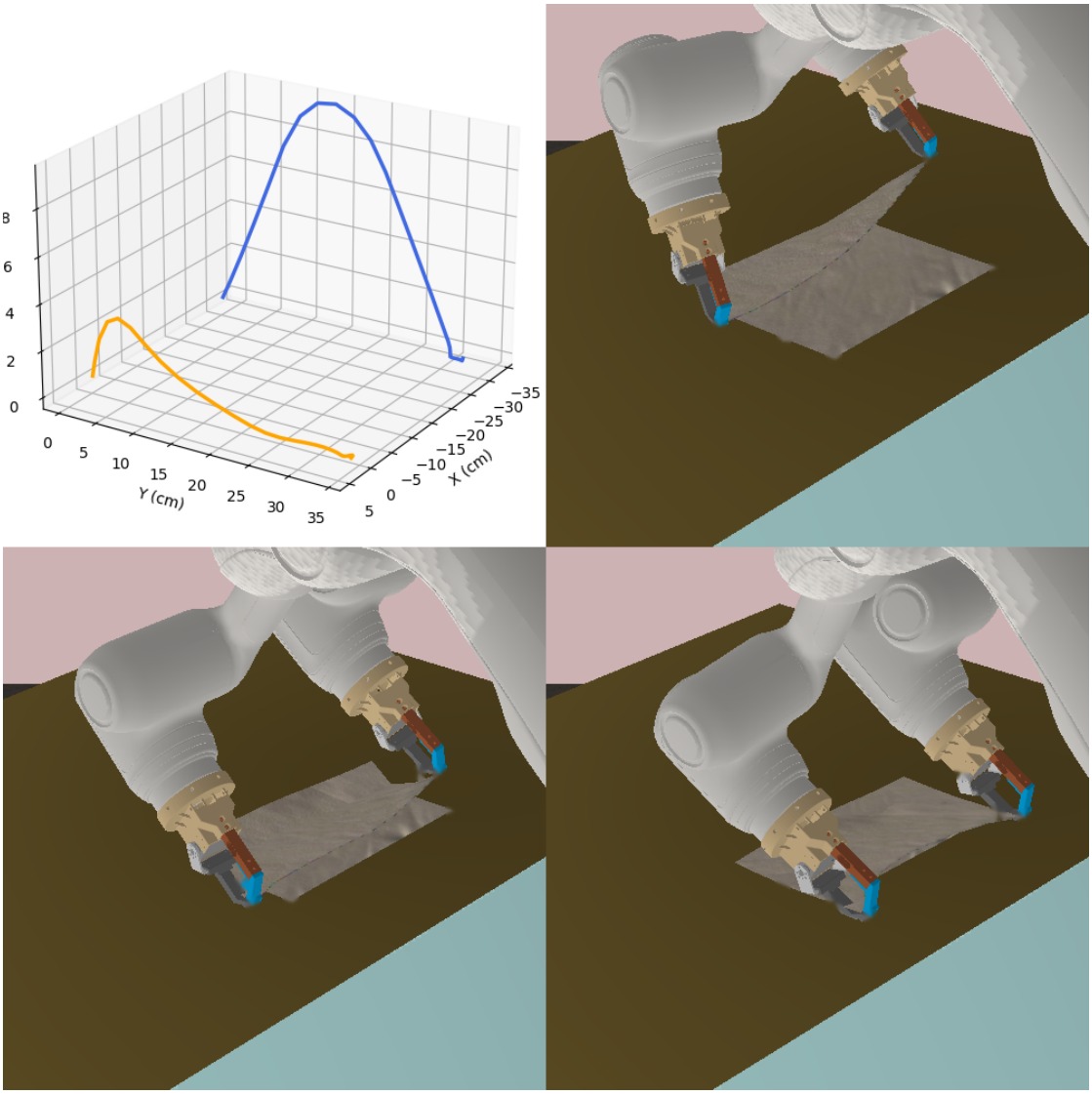


Figure .4.: PyBullet 6d ee 00, epoch 55.

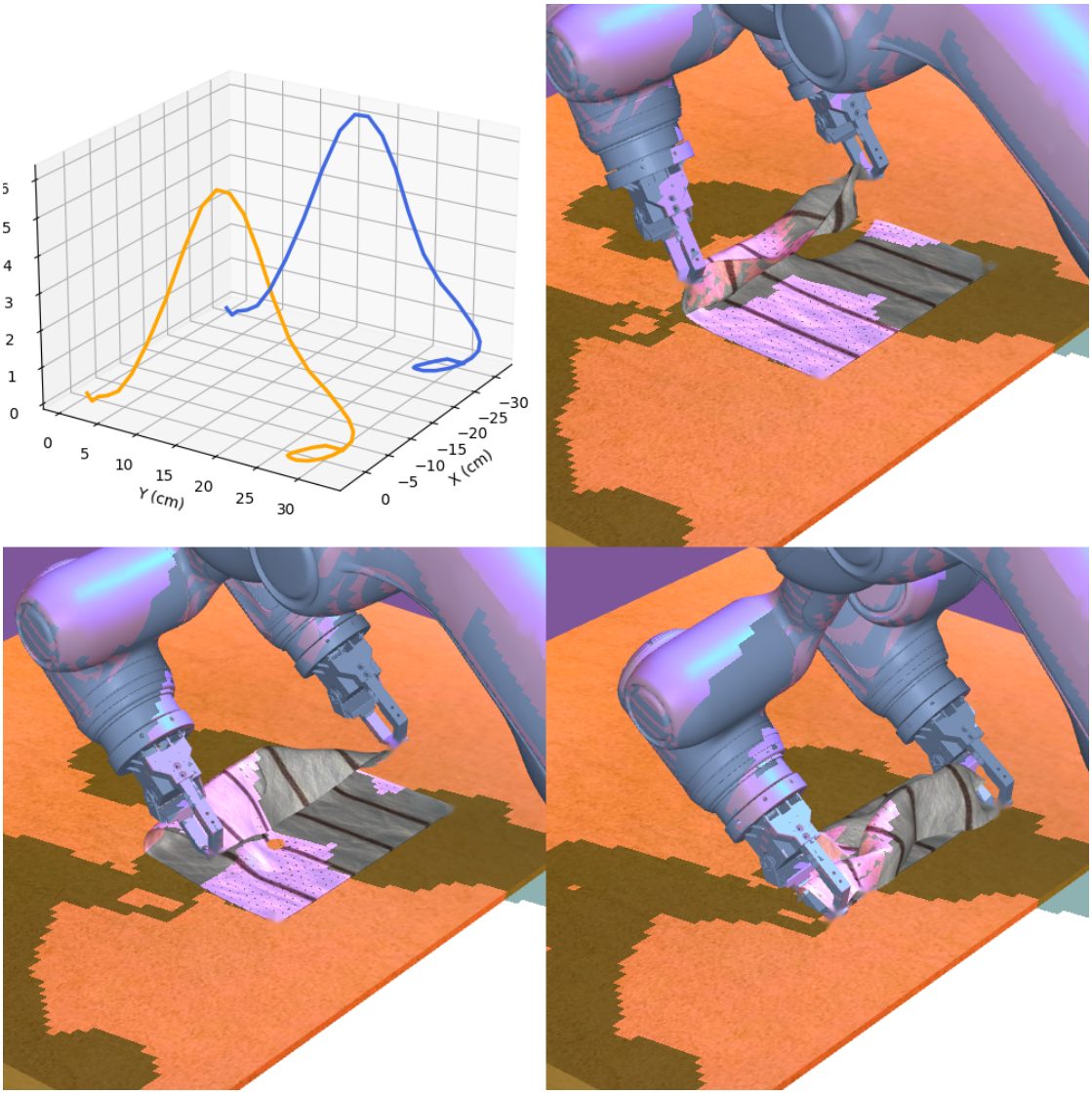


Figure .5.: MuJoCo 3d ctrl 10, epoch 82. (Lowest FCE within the requested MuJoCo `3D ctrl 1 0` setting.)

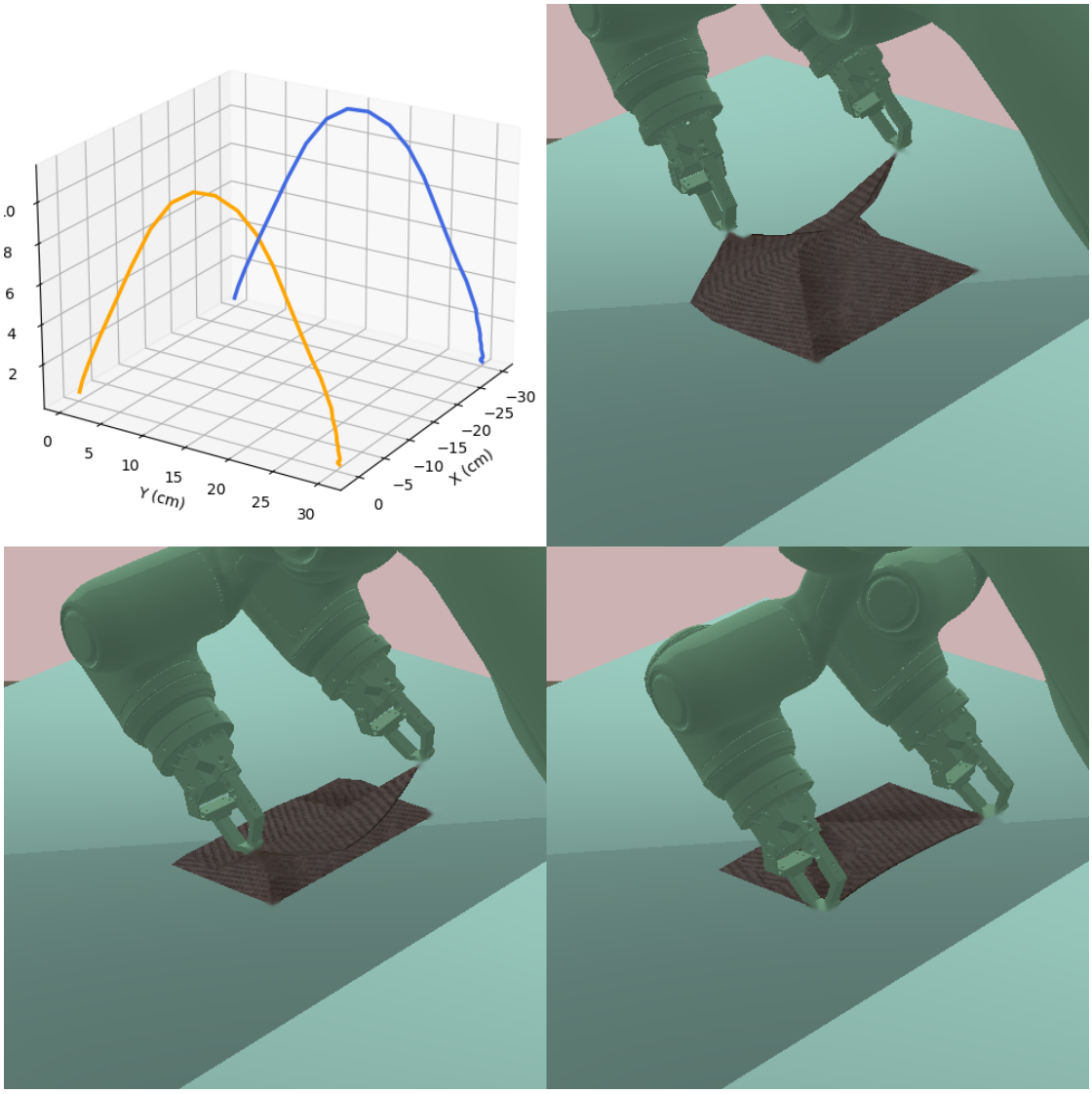


Figure .6.: PyBullet 3d ctrl 11, epoch 82. (Lowest FCE within the requested PyBullet `3D ctrl 1 1` setting.)

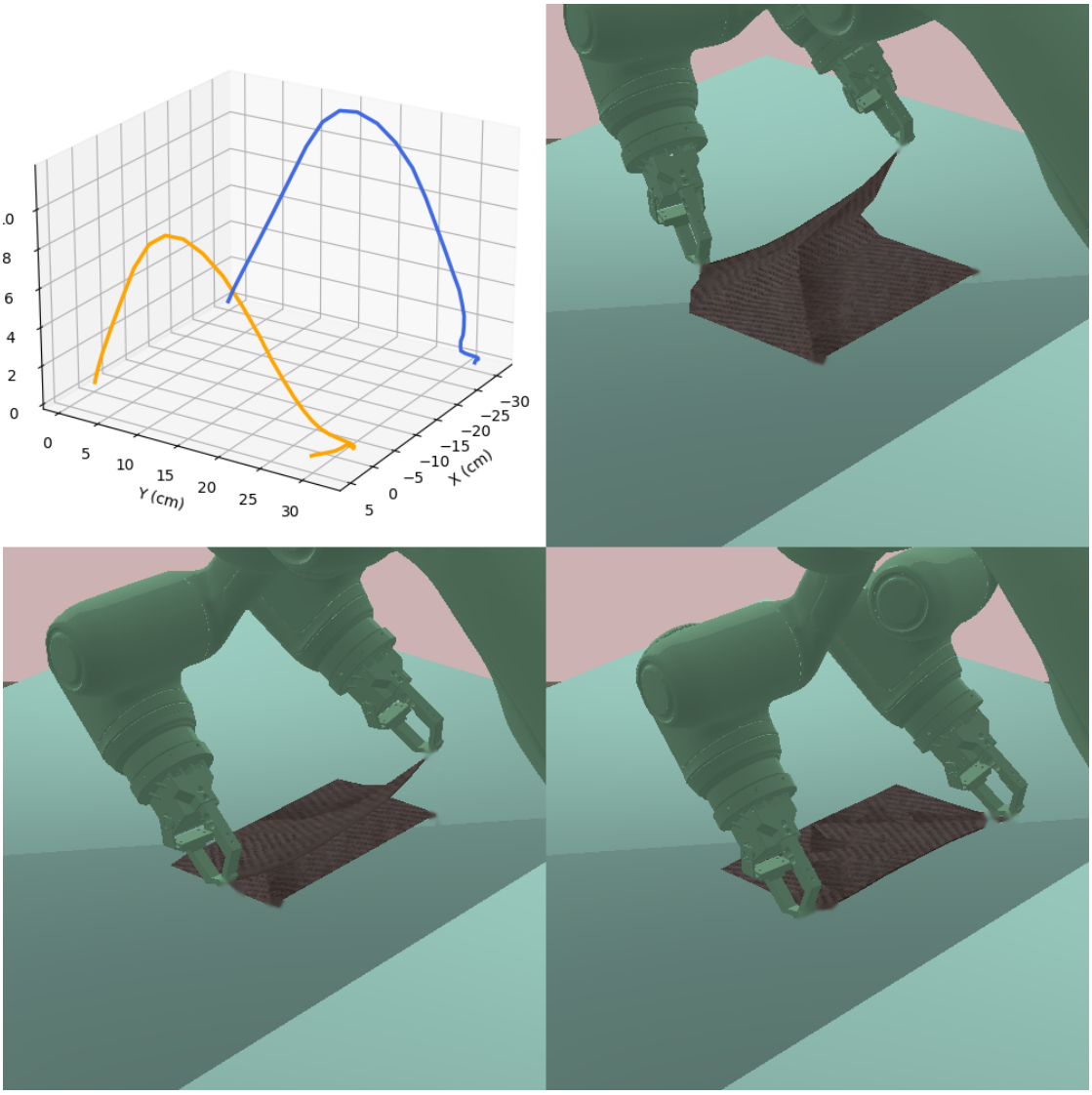


Figure .7.: PyBullet 6d ctrl 11, epoch 94. (Lowest FCE within the requested PyBullet `6D ctrl 1 1` setting.)

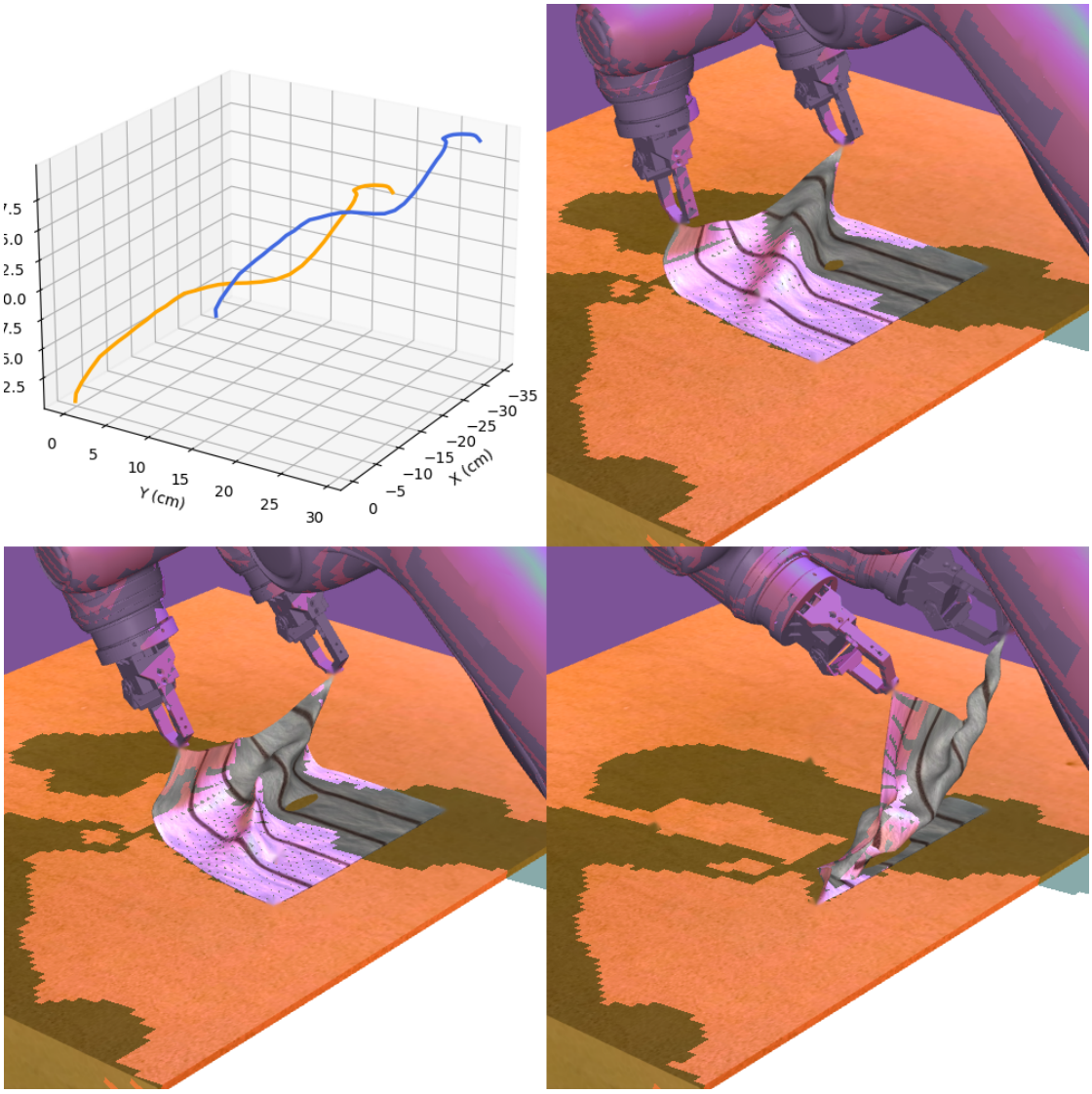


Figure .8.: PyBullet 3d ctrl 11 in MuJoCo, epoch 82.

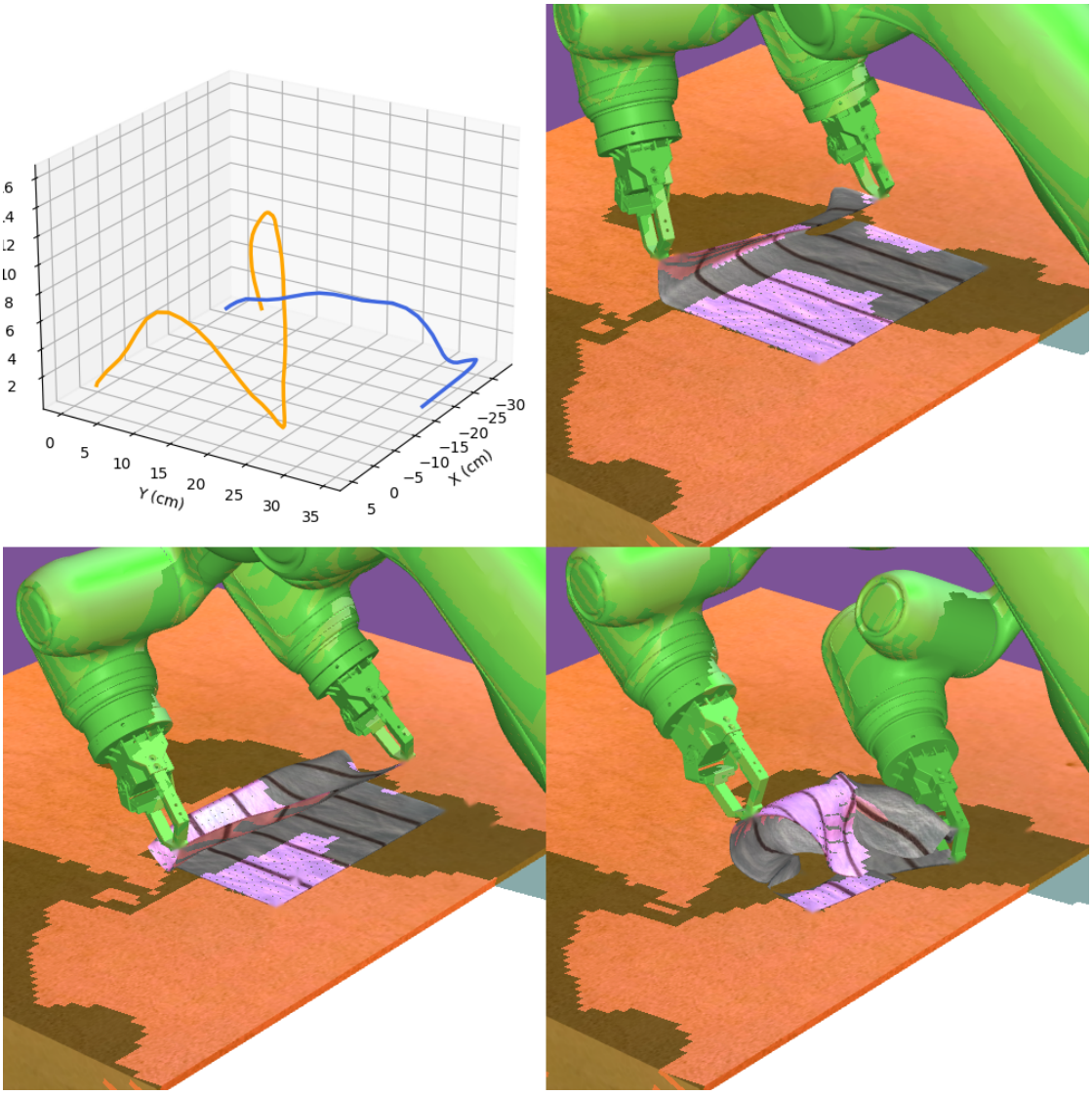


Figure .9.: PyBullet 6d ctrl 11 in MuJoCo, epoch 94.

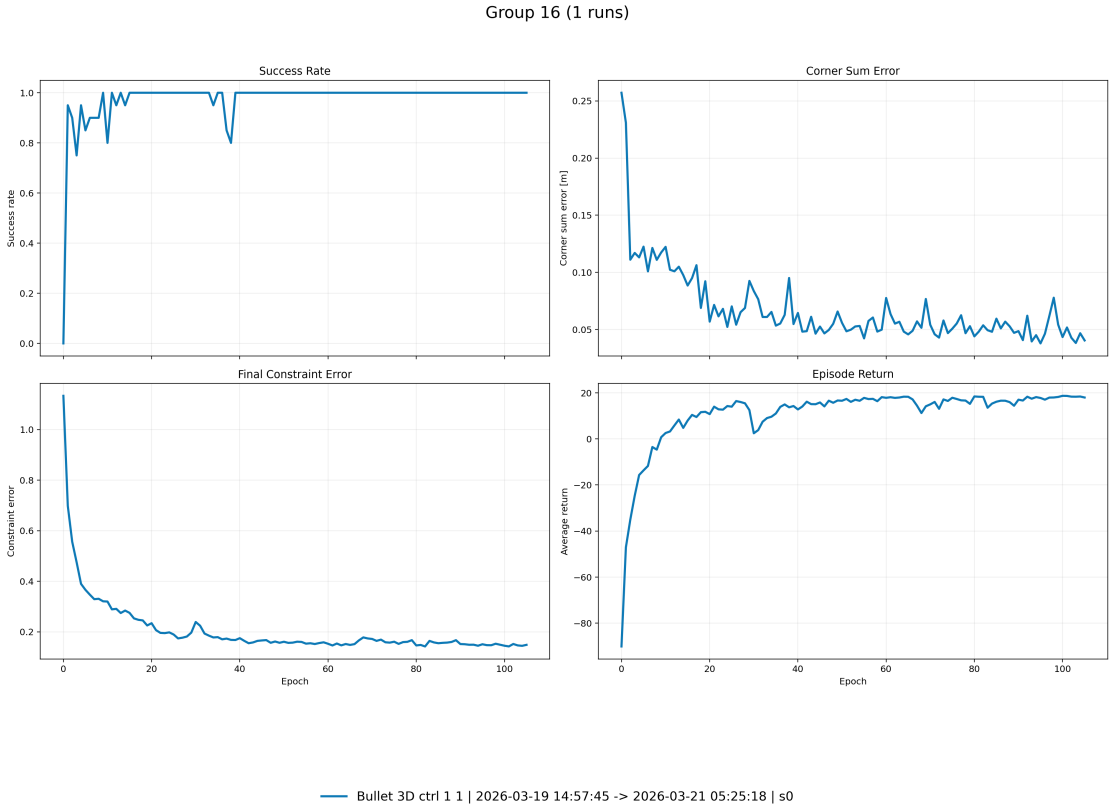


Figure .10.: Learning curves for the additional PyBullet-only setting `3D ctrl` with $DR = 1$ and $RCS = 1$.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe. Sofern im Zuge der Erstellung der vorliegenden Abschlussarbeit generative Künstliche Intelligenz (gKI) basierte elektronische Hilfsmittel verwendet wurden, versichere ich, dass meine eigene Leistung im Vordergrund stand und dass eine vollständige Dokumentation aller verwendeten Hilfsmittel gemäß der Guten Wissenschaftlichen Praxis vorliegt. Ich trage die Verantwortung für eventuell durch die gKI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.

Hamburg, den 22.03.2026



Daniel Brühl

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 22.03.2026



Daniel Brühl