A Implementation Details

A.1 Objects

DaXBench models a variety of deformable objects of different dimensionalities, ranging from 0-dimensional liquid, 1-dimensional rope, to 2-dimensional cloth. The underlying deformation and the dynamics vary distinctly for each dimension. In general, the lower the dimension is, the higher the degrees of freedom (DoFs) of the system, thus the more complex modeling that system becomes. We use two different representations to model the deformable objects to balance the modeling capacity with its complexity: water and rope are modeled using Material Point Method (MPM), while the cloth is modeled using a mass-spring system.

Modelling Water and Rope using MPM Material Point Method (MPM) demonstrates strong modeling capacity and flexibility. MPM models the deformable objects as a set of particles; each particle is represented by its position and some properties such as mass. The dynamics and the deformation of the object are computed by simulating the interactions among all individual particles, conforming to some pre-defined physics rules. This Particle-level representation and the dynamics simulation allow MPM to model the object under severe deformation. We, therefore, choose to use MPM to model water and rope that can undergo severe deformation, for they can exploit the model capacity of MPM to its full potential.

Modelling Cloth using Mass-Spring System Cloth only undergoes limited deformation as it has to conform to the canonical shape of the initial cloth through manipulation. This limited deformable does not require the strong modeling capacity of MPM, which is computationally expensive. Therefore, rather than modeling the deformation on the Particle-level, we choose to model the cloth using the mass-spring system, which is a simpler model that uses the physical constraints as a prior.

A.2 Code Example

```
import numpy as np
from core.envs.registration import env_functions

env = env_functions["fold_cloth3"](batch_size=32, seed=1)
obs, state = env.reset(env.simulator.key_global)

actions = np.random.uniform(0, 1, (env.batch_size, env.action_size))
obs, reward, done, info = env.step_diff(actions, state)
```

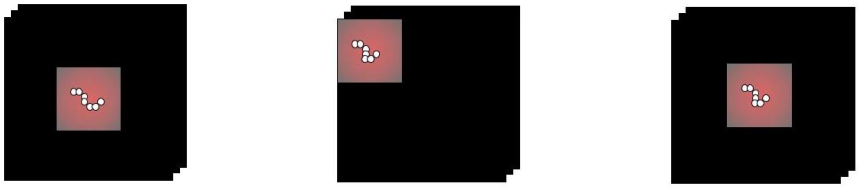
Listing 1: Example of Running Cloth Environment

```
import jax
import numpy as np
from core.envs.registration import env_functions

env = env_functions["fold_cloth3"](batch_size=32, aux_reward=True)
obs, state = env.reset(env.simulator.key_global)
actions = np.random.uniform(0, 1, (env.batch_size, env.action_size))

@jax.jit
@jax.grad
def compute_grad(actions, state):
    obs, reward, done, info = env.step_diff(actions, state)
    return -reward.sum()

print("action gradients", compute_grad(actions, state))
```

Listing 2: Example of Computing Action Gradients via DaX

A.3 Jax to parallelize

One major limitation of the existing simulators is the lack of support for highly paralleled sampling. The key challenges for Deformable Object Manipulation are the complex governing equations and



(a) Detect the active region.

(b) Update the deformation.

(c) Shift back the update active region.

Figure 6: We demonstrate how to use the active region to reduce the computation complexity for the simulation. Taking rope for example, the original environment, denoted as the black background, is of size $128 \times 128 \times 128$, while the active region, where the region that will be actively changed due to the current deformation highlighted in red, is of size $32 \times 6 \times 32$, as illustrated in (a). To avoid updating all the particles in the environment, in (b), we shift the active region to the top corner, the *origin* of the environment, and update the dynamic and observation. Finally, in (c), we shift the updated active region back to align its original coordinates.

the enormous state space. The complex governing equations invalidate most of the analytical approaches since solving the complex governing equations is intractable. On the other hand, the success of the sampling-based methods hinges on the reasonable coverage of the state space. The lack of support for highly paralleled sampling to better cover the state space is a major impediment to the works on Deformable Object Manipulation. DaXBench fills in this gap by implementing its simulator using JAX, which supports highly paralleled sampling with great automatic differentiation. Using our implementation, we can finish an iteration (both forward and backward pass) for 128 rollouts with 80 timesteps on a server with 4 2080-Ti GPUs in 3 seconds.

A.4 Saving Memory

Lazy Dynamic Update DaXBench optimizes the time and memory consumption of MPM by only updating the dynamic of a small region affected by the manipulation at each step. MPM models and updates the state and dynamics on the Particle-level; its time and memory consumption is proportional to the size of the environment, which can be arbitrarily large. Our insight is that, though the deformable object can potentially interact with the entire environment, only a small region will be affected at each time. This inspires DaXBench to optimize the time and memory consumption of running MPM by exclusively considering the *active region* during the interactions. Using this trick, DaXBench effectively reduces the computation time by two orders of magnitude on average. Taking the rope for example, the original environment is of size $128 \times 128 \times 128$. However, a grid of size $32 \times 6 \times 32$ is sufficient to cover the region of interaction between the environment and the rope. Therefore, we first detect the *active region* of size $32 \times 6 \times 32$, then we perform a linear transformation to shift the *active region* to the *origin*, perform the update due to deformation and interaction, finally we shift the updated *active region* to align with its original coordinates.

Checkpointing method DaXBench only stores the values of a few states (i.e. checkpoints), and re-evaluates the sub-step gradients online during the back-propagation to reduce the memory consumption for long horizon problems. The checkpointing method has been proven useful in saving the memory consumption for differentiable simulations in many prior works (Qiao et al. 2021; Chen et al. 2016; Griewank & Walther 2000). To make the simulator end-to-end differentiable, every value in the neural network that is computed during the forward pass has to be saved, as they are needed to compute the gradient during the backward pass. The computation complexity scales up quickly w.r.t. the length of the task horizon. This is a significant problem for our simulator for high DoFs deformable object manipulation, as the memory required for each timestep's update is enormous compared to its rigid body counterpart. To overcome the large memory consumption issue, DaXBench trade-offs the computation complexity slightly for large memory consumption improvement: DaXBench only stores the values of a few states (i.e. checkpoints) at the forward pass; during the backward propagation, DaXBench re-compute segments of the forward values starting from every stored state in the reverse order sequentially for the overall gradient. With this trick, DaXBench

Task Type	Task	PPO	APG	SHAC
High-level	Fold Cloth 1	**0.40±0.13**	0.36±0.06	0.34±0.07
	Fold Cloth 3	0.21±0.11	0.19±0.09	**0.22±0.22**
	Fold T-shirt	**0.61±0.08**	0.40±0.05	0.44±0.09
	Unfold Cloth 1	**0.72±0.01**	0.42±0.02	0.50±0.03
	Unfold Cloth 3	**0.56±0.00**	0.39±0.02	0.48±0.03
	Rope Push	**0.75±0.02**	**0.72±0.02**	**0.75±0.01**
Low-level	Wipe Rope	0.25±0.10	**0.83±0.01**	0.66±0.03
	Pour Water	0.27±0.02	**0.27±0.02**	**0.28±0.00**
	Pour Soup	0.27±0.08	0.27±0.00	0.32±0.13

Table 4: Reinforcement Learning Methods Task Results. We report the mean and standard error over 20 random seeds for each entry.

now doubles the computation complexity, but saves an arbitrary amount of memory, depending on the intervals of the states saved.

A.5 Discontinuous Gradient

As an end-to-end differentiable simulator, the effectiveness of DaXBench rests on the accuracy of the gradient computed. We found that for deformable object manipulation, the action not in contact with the object will cause the gradient to be discontinuous, which causes the estimate of the gradient after this action to suffer from high variance and poor accuracy in the backward propagation. To mitigate this problem, for every action input by the agent or the human user, DaXBench will perform local adjustment such that the action will always make the gripper in contact with the deformable objects. This ensures that our gradient is end-to-end continuous therefore improving the accuracy of the estimated gradient.

B Sim-2-Real Gap

To verify that the dynamics of our simulated tasks have high fidelity and a small Sim2Real gap, we carry out a real robot experiment. We deploy CEM-MPC to the PushRope task on a Kinova Gen3 robot. CEM-MPC uses our DaX simulator as the predictive model. Given a state, CEM-MPC plans the next best *push* action (x, y, z, x', y', z'). The state is estimated from a point cloud image, and the Cartesian space *push* action is transformed to the joint space trajectory via an inverse kinematics module. Note we are not verifying the sim2real gap caused by the inaccuracy of state estimation or kinematics; we are interested in whether the dynamics of DaX can correctly guide the CEM-MPC to succeed in the task. The experiment is included in our supplementary video.

C RL Algorithms Numerical Experiment Results

We report the numerical experiment performance for the RL algorithms in Table 4. This is to supplement the learning curve in the main text. These numerical results report the final performance of the learning curves in Figure 3.

		Imitation Learning			
Task Type	Task	Transporter	Transporter-RGB	ILD	Expert
High-level	Fold-Cloth-1	0.19±0.04	0.86±0.07	0.76±0.04	0.91±0.00
	Fold-Cloth-3	0.40±0.05	0.56±0.25	0.82±0.05	0.89±0.00
	Fold-T-shirt	0.46±0.02	0.83±0.05	0.59±0.11	0.85±0.00
	Unfold-Cloth-1	0.48±0.01	0.76±0.06	0.64±0.03	0.87±0.00
	Unfold-Cloth-3	0.30±0.00	0.70±0.06	0.52±0.02	0.87±0.00
	Push-Rope	0.70±0.00	0.86±0.09	0.76±0.02	0.93±0.00
Low-level	Whip-Rope	—	—	0.70±0.06	1.00± 0.00
	Pour-Water	—	—	0.32±0.03	0.91± 0.06
	Pour-Soup	—	—	0.42±0.12	0.85± 0.00

Table 5: **Task performance for the planning and imitation learning methods.** We report the mean and standard error for the policy/control sequences evaluated under 20 seeds.

D TRANSPORTER WITH IMAGE AS INPUT

We reported the performance of two different versions of Transporter in Table 5: Transporter, which uses 3D particles and projects them onto an image to form a height map; and Transporter-RGB, which adds RGB channels and projects the particles onto the image plane, giving it white color with a black background. We have found that an explicit process of rendering particles into RGB images significantly enhances the performance of Transporter.