Evolution Gym: A Large-Scale Benchmark for Evolving Soft Robots -Supplementary-

Jagdeep Singh Bhatia
MIT CSAIL
jagdeep@mit.edu

Holly Jackson
MIT CSAIL
hjackson@mit.edu

Yunsheng Tian
MIT CSAIL
yunsheng@csail.mit.edu

Jie Xu
MIT CSAIL
jiex@csail.mit.edu

Wojciech Matusik
MIT CSAIL
wojciech@csail.mit.edu

A Simulation engine

In this section we describe Evolution Gym's simulator in detail. We describe the simulator's representation of objects, the dynamics of the underlying simulator, our implementation of contact forces, and other techniques we use to improve the quality of Evolution Gym. Finally, we present an analysis of how our simulation scales with the number of voxels, followed by some of the hyperparamters in our implementation.

A.1 Representation

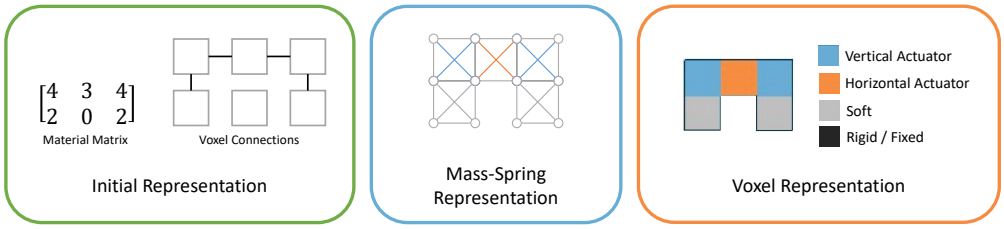

Figure 1: Representation of Simulation Objects

The simulation represents objects and their environment as a 2D mass-spring system in a grid-like layout, where objects are initialized as a set of non-overlapping, connected voxels.

More specifically, any objects loaded into the simulation, including the robot, can be represented as a material matrix and a set of connections between adjacent cells as seen in the green panel of Figure 1. The entries of the material matrix are integers corresponding to a voxel type from the set {Empty, Rigid (black), Soft (gray), Horizontal Actuator (orange), Vertical Actuator (blue), Fixed (black)}. In particular, actuator cells are specific to the robot object and fixed cells are only used in non-robot objects. Each voxel can be connected to each of its adjacent voxels, which determines the identity of objects in the simulation. In our work, we connect all pairs of adjacent voxels.

The simulation converts all objects into a set of point masses and springs by turning each voxel into a cross-braced square as shown in the blue panel of Figure 1. Note that some voxels share the same point masses and springs. All point masses in the simulation have the same mass and the equilibrium lengths of axis-aligned and diagonal springs are constants for simplicity. However,

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

the spring constants assigned vary based on voxel material-type – with ties broken in favor of the more-rigid spring. Please see section A.6 for more details on simulation hyperparameters.

A.2 Simulation Dynamics

Let y_n be the state of the simulation at time step n. We can view y_n as a length $2 \times 2N$ vector of 2D positions and velocities, where N is the number of point masses in the simulation. We use symplectic RK-4 integration to compute y_{n+1} by taking into account gravitational, contact, viscous drag, and spring forces.

In particular, spring forces are modelled by the following dynamics equations:

$$\mathbf{f}_{int} + \mathbf{f}_{ext} = M\ddot{\mathbf{x}}$$
$$\mathbf{f}_{int}^i = \sum_j k_j(l_j - \bar{l}_j)\mathbf{e}_j^i, \;\; j \in \text{springs associated with vertex } i$$

where $\mathbf{x}$ is the positions of all vertices in the mass-spring system, M is the mass matrix of system, $\mathbf{f}_{int}$ is the internal spring forces exerted on vertices, $\mathbf{f}_{ext}$ is the external forces produced by gravity and contacts, k_j is the spring constant in Hooke's law, l_j and $\bar{l}_j$ are the lengths of spring j in current shape and in rest shape respectively, and $\mathbf{e}_j^i$ is the unit direction of the spring j relative to vertex i.

For our simulator, we use 50Hz as the control frequency, while using 1500Hz as the simulation sub-step frequency. This is achieved by running 30 sub-steps of the simulation for each control input. We apply such sub-step scheme to improve the stability of the simulation.

A.3 Collision detection and contact forces

We use bounding box trees for collision detection, made up of the voxels on the surface of each object [2]. Exploiting the grid-like nature of voxels, the tree for each object is only computed at initialization. However, the bounding box at each node of the tree is recomputed at each time step.

To resolve collisions between cells, penalty-based contact forces and frictional forces are computed proportionally to the depth of penetration of the corresponding cells in contact. These forces are applied on voxel vertices in the normal and tangential directions of the contact respectively.

It is important to note that contact forces are pre-computed before each RK-4 step and are considered constant by the time step integrator. We chose to do this because computing contact forces is often computationally expensive.

A.4 Other techniques

We use several other techniques to improve the quality of the simulation engine and its interactions with the control optimization.

Strain limiting
Strain limiting is implemented to prevent the self-folding of objects in the simulation. If any spring grows or shrinks by more than 25%, the simulation will reposition the masses connected by the spring to reduce the strain. The springs of rigid cells have a more aggressive threshold for strain limiting at 3% compression/expansion. Note that strain limiting does not impose a hard limit on springs and can still be overcome by very strong actuations from the robot.

Self-folding
Even with strain limiting and collision detection, it is still possible that the robot in the simulation can fold in on itself. In order to combat this, we have a reliable check for whether the robot object is self-folding: we check whether the number of colliding, non-adjacent pairs of voxels on the surface of the robot is more than the number of voxels on the surface of the robot. In the each of our environments described in Section B, we penalize the robot with a one time reward of -3 and reset the environment any time self-folding is detected.

Delayed actuations
Each time the environment steps, the robot's controller provides a single actuation value for each of the robot's actuators. In order to move the actuators, the simulation changes the equilibrium lengths of the actuator's springs. However, rather than setting the goal equilibrium length right away, the

simulation sets the equilibrium length to be the weighted average of the goal and the spring's current length by parameters α and $1 - \alpha$, respectively, for $\alpha << 0.5$. This has the benefit of requiring the robot's controller to favor longer smoother motions – which are more in line with how a real actuator might behave – rather than short abrupt ones.

A.5 Scalability and speed analysis

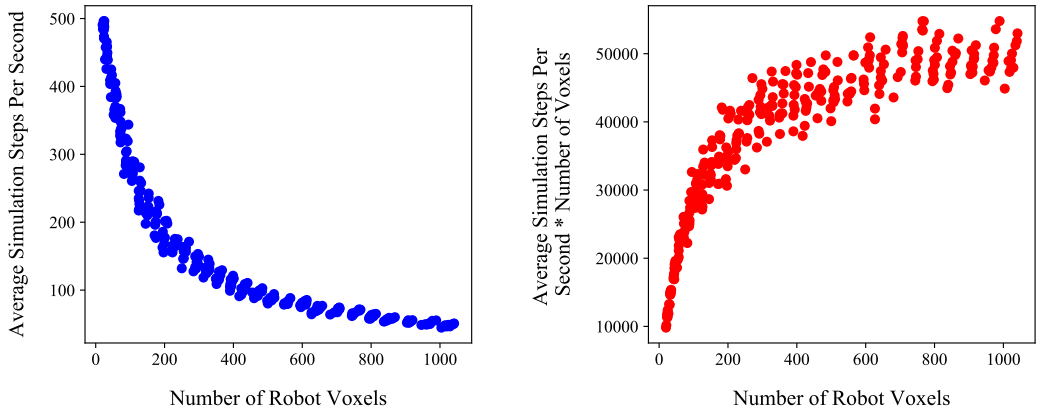

Figure 2: Simulating robots with varying voxel counts on a single core of an Intel Xeon CPU @ 2.80GHz. We graph the average number of simulation steps per second (left) and the average number of simulation steps per second times the number of voxels (right).

As co-design algorithms become more advanced, we should expect them to search over larger design spaces. Therefore, we believe that the ability of our simulation to scale well as the number of voxels increases is important. In this sub-section we test the performance of our simulator at varying numbers of voxels and present the result in Figure 2.

Specifically, our experiment is as follows: For each $n \in [5, 35]$, we sample 10, $(n \times n)$ robots and evaluate the performance of our simulator when simulating these robots for 10 seconds and with random actions. Robots are simulated in a simple environment with flat terrain consisting of 100 voxels. Simulations are run on a single core of an Intel Xeon CPU @ 2.80GHz.

From these experiments, we compare the `# of non-empty voxels` against `average number of simulation steps per second (assps)`. We also compute `assps` $\times$ `# of non-empty voxels` as another metric.

These results allow us to compare our simulator to others. For instance, our simulation speed is slower than most rigid-body simulations, like MuJoCo [10]. This is because simulation speed is very dependent on the type of simulation, and there are many more degrees of freedom in soft body simulation like ours compared to a standard rigid body simulation.

Another insightful comparison might be with soft, 3D FEM-based simulations. For example, in Du et al. [1], one simulation step consumes on the order of several seconds to minutes and the PPO training for a single robot design takes on the order of hours to converge. By comparison, in our framework, a single robot can be simulated with hundreds of steps per second and trained in parallel on a 4-core machine in a matter of minutes. This highlights the advantage of our 2D mass-spring approach.

Finally, the best comparison of our work would be other 2D voxel-based simulations, of which few exist. For example, our speed is comparable to Medvet et al. [5] as we both simulate objects in 2D and adopt relatively standard implementations for mass-spring systems. However, the main contribution of Medvet et al. is its simulation; our main advantages over Medvet et al 2020 are (1) a comprehensive set of well-designed tasks with various difficulty levels to provide the first comparison platform for evaluating co-design algorithms, (2) our implementation of state-of-the-art algorithms to establish a baseline for co-design, and (3) our ability to interface with standard python ML libraries through python bindings for our physics simulation.

A.6 Hyperparameters

Table 1: Values of Simulation hyperparameters

parameter name	value
point mass	1.0
viscous drag	0.1
gravity	110
contact stiffness coefficient	$2.1 \cdot 10^7$
collision penetration depth additive	$5 \cdot 10^{-3}$
coefficient of friction	0.1
frictional penalty factor	0.5
friction multiplier	$2.4 \cdot 10^3$
rigid main spring const	$\frac{3 \cdot 10^8}{3.5}$
rigid structural spring const	$\frac{3 \cdot 10^8}{7}$
soft main spring const	$\frac{3 \cdot 10^8}{5}$
soft structural spring const	$\frac{3 \cdot 10^8}{10}$
actuator main spring const	$\frac{3 \cdot 10^8}{6}$
actuator structural spring const	$\frac{3 \cdot 10^8}{24}$

In this section we describe the significance of all hyperparameters on the simulation.

We start with some general hyperparameters. The *point mass* constant describes the mass of all points. The *viscous drag* constant is a multiplier used to adjust the strength of the viscous drag force. Increasing this constant makes the particles move as though they are traveling through a more viscous fluid. The *gravity* constant describes the magnitude of the force of gravity in the simulation.

We continue with hyperparameters important to contact forces. The *contact stiffness coefficient* is a multiplier used to adjust the strength of the normal contact force between objects. The *collision penetration depth additive* is a constant added to the penetration depth of collision when contact forces are computed. *Coefficient of friction* and *frictional penalty factor* correspond to the coefficients of static and dynamic friction, respectively, while *friction multiplier* is a constant used to adjust the strength of the tangential contact force between objects.

Finally, we describe the hyperparameters which control spring rigidity. Increasing any of these constants makes the corresponding springs more rigid, and decreasing them has the opposite effect. Recall that in the simulation each voxel is a cross-braced square. *Main* spring constants describe the rigidity of springs around the square edges of a voxel while *structural* spring constants describe the rigidity of springs on the cross-brace. *Rigid, soft, and actuator* spring constants correspond to springs on rigid, soft, and actuator cells, respectively.

B Full benchmark suite

We have implemented a total of 32 tasks for Evolution Gym. Below, we describe in detail the reward and observation of each of the environments we have implemented.

For reference, the names of the 10 benchmark tasks are `Walker-v0`, `BridgeWalker-v0`, `UpStepper-v0`, `Traverser-v0`, `Climber-v0`, `Carrier-v0`, `Thrower-v0`, `Catcher-v0`, `Lifter-v0`, `BeamSlider-v0`.

B.1 Notation

We start by describing some notation that we will use in the following sections.

Position
Let p^o be a vector of length 2 that represents the position of the center of mass of an object o in the simulation at time t. p^o_x and p^o_y denote the x and y components of this vector, respectively. p^o is computed by averaging the positions of all the point-masses that make up object o at time t.

Velocity
Similarly, let v^o be a vector of length 2 that represents the velocity of the center of mass of an object o in the simulation at time t. v_x^o and v_y^o denote the x and y components of this vector, respectively. v^o is computed by averaging the velocities of all the point-masses that make up object o at time t.

Orientation
Similarly, let θ^o be a vector of length 1 that represents the orientation of an object called o in the simulation at time t. Let p_i be the position of point mass i of object o. We compute θ^o by averaging over all i the angle between the vector $p_i - p^o$ at time t and time 0. This average is a weighted average weighted by $||p_i - p^o||$ at time 0.

Special observations
Let c^o be a vector of length $2n$ that describes the positions of all n point masses of object o relative to p^o. We compute c^o by first obtaining the $2 \times n$ matrix of positions of all the point masses of object o, subtracting p^o from each column, and reshaping as desired.

Let $h_b^o(d)$ be a vector of length $(2d + 1)$ that describes elevation information around the robot below its center of mass. More specifically, for some integer $x \leq d$, the corresponding entry in vector $h_b^o(d)$ will be the highest point of the terrain which is less than p_y^o between a range of $[x, x + 1]$ voxels from p_x^o in the x-direction.

Let $h_a^o(d)$ be a vector of length $(2d + 1)$ that describes elevation information around the robot above its center of mass. More specifically, for some integer $x \leq d$, the corresponding entry in vector $h_a^o(d)$ will be the lowest point of the terrain which is greater than p_y^o between a range of $[x, x + 1]$ voxels from p_x^o in the x-direction.

B.2 Walking tasks

B.2.1 Walker-v0

Figure 3: `Walker-v0`

In this task the robot walks as far as possible on flat terrain. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+2}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r, c^r$$

with lengths 2 and n, respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction.

This environment runs for 500 steps. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

B.2.2 BridgeWalker-0

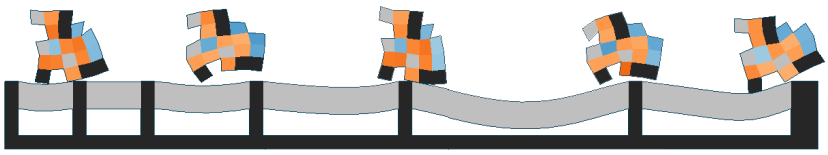

Figure 4: `BridgeWalker-v0`

In this task the robot walks as far as possible on a soft rope-bridge. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+3}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r$$

with lengths 2, 1, and n, respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

This environment runs for 500 steps.

B.2.3 BidirectionalWalker-v0

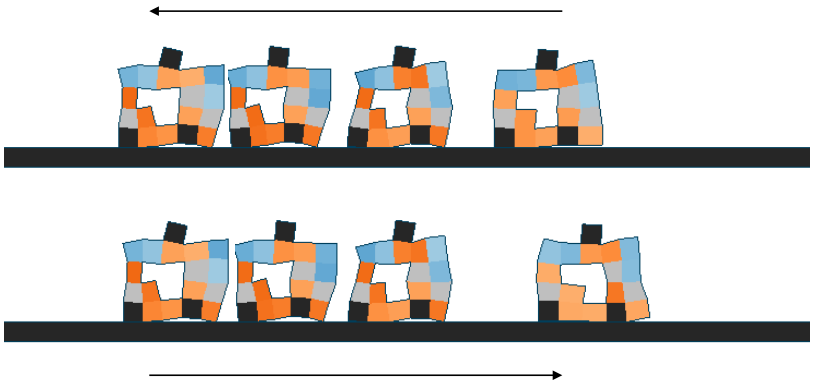

Figure 5: `BidirectionalWalker-v0`

In this task the robot walks bidirectionally. This task is **medium**.

Let the robot object be r. Let g_x be a goal x-position that is randomized and changes throughout the task. There is also a counter c which counts how many times the goal has changed. The observation space has dimension $\mathcal{S} \in R^{n+5}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, c^r,\, c,\, g_x,\, g_x - p_x^r$$

with lengths 2, n, 1, 1, and 1, respectively. The reward R is

$$R = -\Delta |g_x - p_x^r|$$

which rewards the robot for moving towards the goal in the x-direction.

This environment runs for 1000 steps.

B.3 Object manipulation tasks

B.3.1 Carrier-v0

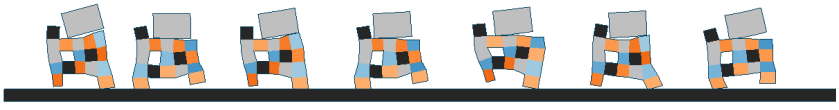

Figure 6: `Carrier-v0`

In this task the robot catches a box initialized above it and carries it as far as possible. This task is **easy**.

Let the robot object be r and the box object the robot is trying to carry be b. The observation space has dimension $\mathcal{S} \in R^{n+6}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^b,\ p^b - p^r,\ v^r,\ \ c^r$$

with lengths 2, n, 2, and 2, respectively. The reward $R = R_1 + R_2$ is the sum of several components.

$$R_1 = 0.5 \cdot \Delta p_x^r + 0.5 \cdot \Delta p_x^b$$

which rewards the robot and box for moving in the positive x-direction.

$$R_2 = \begin{cases} 0 & \text{if } p_y^b \geq t_y \\ 10 \cdot \Delta p_y^b & \text{otherwise} \end{cases}$$

which penalizes the robot for dropping the box below a threshold height t_y.

This environment runs for 500 steps. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

B.3.2 Carrier-v1

Figure 7: `Carrier-v1`

In this task the robot carries a box to a table and places the box on the table. This task is **hard**.

Let the robot object be r and the box object the robot is trying to carry be b. We achieve the described behavior by setting a goal x-position - g_x^r and g_x^b - for the robot and box, respectively. The observation space has dimension $\mathcal{S} \in R^{n+6}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^b,\ p^b - p^r,\ v^r,\ \ c^r$$

with lengths 2, n, 2, and 2, respectively. The reward $R = R_1 + R_2 + R_3$ is the sum of several components.

$$R_1 = -2 \cdot \Delta |g_x^b - p_x^b|$$

which rewards the box for moving to its goal in the x-direction.

$$R_2 = -\Delta |g_x^r - p_x^r|$$

which rewards the box for moving to its goal in the x-direction.

$$R_3 = \begin{cases} 0 & \text{if } p_y^b \geq t_y \\ 10 \cdot \Delta p_y^b & \text{otherwise} \end{cases}$$

which penalizes the robot for dropping the box below a threshold height t_y. Note that in this task t_y is not constant, and varies with the elevation of the terrain.

This environment runs for 1000 steps.

B.3.3 Pusher-v0

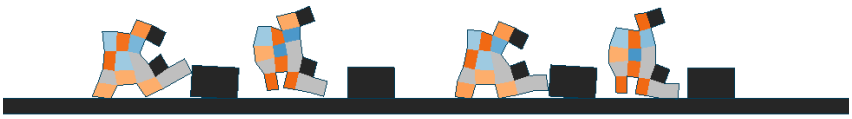

Figure 8: `Pusher-v0`

In this task the robot pushes a box initialized in front of it. This task is **easy**.

Let the robot object be r and the box object the robot is trying to push be b. The observation space has dimension $\mathcal{S} \in R^{n+6}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^b,\ p^b - p^r,\ v^r,\ \ c^r$$

with lengths 2, n, 2, and 2, respectively. The reward $R = R_1 + R_2$ is the sum of several components.

$$R_1 = 0.5 \cdot \Delta p_x^r + 0.75 \cdot \Delta p_x^b$$

which rewards the robot and box for moving in the positive x-direction.

$$R_2 = -\Delta |p_x^b - p_x^r|$$

which penalizes the robot and box for separating in the x-direction.

This environment runs for 500 steps. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

B.3.4 Pusher-v1

Figure 9: `Pusher-v1`

In this task the robot pushes/drags a box initialized behind it in the forward direction. This task is **medium**.

Let the robot object be r and the box object the robot is trying to push be b. The observation space has dimension $\mathcal{S} \in R^{n+6}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^b,\ p^b - p^r,\ v^r,\ \ c^r$$

with lengths 2, n, 2, and 2, respectively. The reward $R = R_1 + R_2$ is the sum of several components.

$$R_1 = 0.5 \cdot \Delta p_x^r + 0.75 \cdot \Delta p_x^b$$

which rewards the robot and box for moving in the positive x-direction.

$$R_2 = -\Delta |p_x^b - p_x^r|$$

which penalizes the robot and box for separating in the x-direction.

This environment runs for 600 steps. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

B.3.5 Thrower-v0

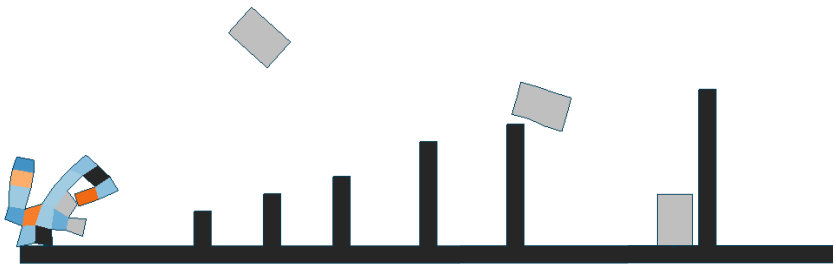

Figure 10: `Thrower-v0`

In this task the robot throws a box initialized on top of it. This task is **medium**.

Let the robot object be r and the box object the robot is trying to throw be b. The observation space has dimension $\mathcal{S} \in R^{n+6}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^b,\ p^b - p^r,\ v^r,\ \ c^r$$

with lengths 2, n, 2, and 2, respectively. The reward $R = R_1 + R_2$ is the sum of several components.

$$R_1 = \Delta p_x^b$$

which rewards the box for moving in the positive x-direction.

$$R_2 = \begin{cases} -\Delta p_x^r & \text{if } p_x^r \geq 0 \\ \Delta p_x^r & \text{otherwise} \end{cases}$$

which penalizes the robot for moving too far from $x = 0$ when throwing the box.

This environment runs for 300 steps.

B.3.6 Catcher-v0

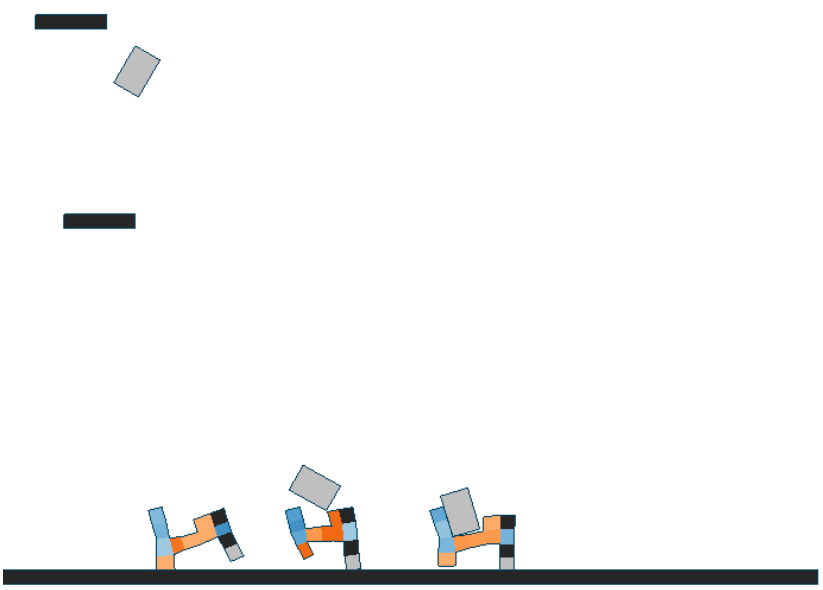

Figure 11: `Catcher-v0`

In this task the robot catches a fast-moving, rotating box. This task is **hard**.

Let the robot object be r and the box object the robot is trying to throw be b. The observation space has dimension $\mathcal{S} \in R^{n+7}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$p^b - p^r,\ v^r,\ v^b,\ \theta^b,\ c^r$$

with lengths 2, 2, 2, 1, and n, respectively. The reward $R = R_1 + R_2$ is the sum of several components.

$$R_1 = -\Delta |p_x^b - p_x^r|$$

which rewards the robot for moving to the box in the x-direction.

$$R_2 = \begin{cases} 0 & \text{if } p_y^b \geq t_y \\ 10 \cdot \Delta p_y^b & \text{otherwise} \end{cases}$$

which penalizes the robot for dropping the box below a threshold height t_y.

This environment runs for 400 steps.

B.3.7 BeamToppler-v0

Figure 12: `BeamToppler-v0`

In this task the robot knocks over a beam sitting on two pegs from underneath. This task is **easy**.

Let the robot object be r and the beam object the robot is trying to topple be b. The observation space has dimension $\mathcal{S} \in R^{n+7}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$p^b - p^r,\ v^r,\ v^b,\ \theta^b,\ c^r$$

with lengths 2, 2, 2, 1, and n, respectively. The reward $R = R_1 + R_2 + R_3$ is the sum of several components.

$$R_1 = -\Delta|p_x^b - p_x^r|$$

which rewards the robot for moving to the beam in the x-direction.

$$R_2 = |\Delta p_x^b| + 3 \cdot |\Delta p_y^b|$$

which rewards the robot for moving the beam.

$$R_3 = -\Delta p_y^b$$

which rewards the robot for making the beam fall.

This environment runs for 1000 steps. The robot also receives a one-time reward of 1 for completing the task.

B.3.8 BeamSlider-v0

Figure 13: `BeamSlider-v0`

In this task the robot slides a beam across a line of pegs from underneath. This task is **hard**.

Let the robot object be r and the beam object the robot is trying to slide be b. The observation space has dimension $\mathcal{S} \in R^{n+7}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$p^b - p^r,\ v^r,\ v^b,\ \theta^b,\ c^r$$

with lengths 2, 2, 2, 1, and n, respectively. The reward $R = R_1 + R_2$ is the sum of several components.

$$R_1 = -\Delta|p_x^b - p_x^r|$$

which rewards the robot for moving to the beam in the x-direction.

$$R_2 = \Delta p_x^b$$

which rewards the robot for moving the beam in the positive x-direction.

This environment runs for 1000 steps.

B.3.9 Lifter-v0

Figure 14: `Lifter-v0`

In this task the robot lifts a box from out of a hole. This task is **hard**.

Let the robot object be r and the box object the robot is trying to lift be b. The observation space has dimension $\mathcal{S} \in R^{n+7}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$p^b - p^r,\ v^r,\ v^b,\ \theta^b,\ c^r$$

with lengths 2, 2, 2, 1, and n, respectively. The reward $R = R_1 + R_2 + R_3$ is the sum of several components.

$$R_1 = 10 \cdot \Delta p_y^b$$

which rewards the robot for moving the beam in the positive y-direction.

$$R_2 = -10 \cdot \Delta |g_x - p_x^b|$$

which penalizes the robot for moving the box away from a goal x-position, g_x. This ensures that the robot lifts the box straight up.

$$R_3 = \begin{cases} 0 & \text{if } p_y^r \geq t_y \\ 20 \cdot \Delta p_y^r & \text{otherwise} \end{cases}$$

which penalizes the robot for falling below a threshold height t_y (at which point the robot has fallen into the hole).

This environment runs for 300 steps.

B.4 Climbing tasks

B.4.1 Climber-v0

Figure 15: `Climber-v0`

In this task the robot climbs as high as possible through a flat, vertical channel. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+2}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r, c^r$$

with lengths 2 and n, respectively. The reward R is

$$R = \Delta p_y^r$$

which rewards the robot for moving in the positive y-direction.

This environment runs for 400 steps. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

B.4.2 Climber-v1

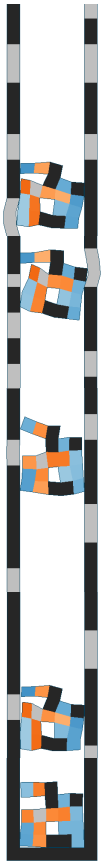

Figure 16: `Climber-v1`

In this task the robot climbs as high as possible through a vertical channel made of mixed rigid and soft materials. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+2}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r, c^r$$

with lengths 2 and n, respectively. The reward R is

$$R = \Delta p_y^r$$

which rewards the robot for moving in the positive y-direction.

This environment runs for 600 steps. The robot also receives a one-time reward of 1 for reaching the end of the terrain.

B.4.3 Climber-v2

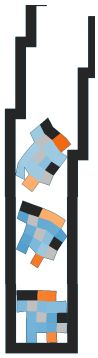

Figure 17: `Climber-v2`

In this task the robot climbs as high as possible through a narrow stepwise channel. This task is **hard**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+10}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r, \theta^r, c^r, h_a^r(3)$$

with lengths 2, 1, n, and 7 respectively. The reward R is

$$R = \Delta p_y^r + 0.2 \cdot \Delta p_x^r$$

which rewards the robot for moving in the positive y-direction and positive x-direction.

This environment runs for 1000 steps.

B.5 Forward locomotion tasks

B.5.1 UpStepper-v0

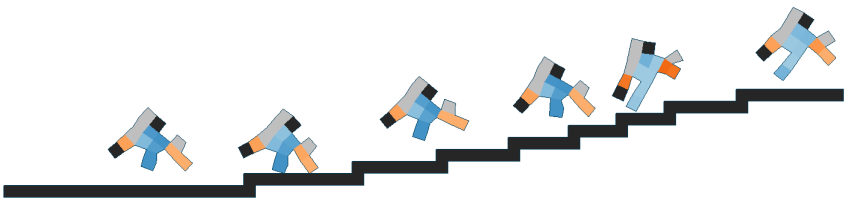

Figure 18: `UpStepper-v0`

In this task the robot climbs up stairs of varying lengths. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r, \theta^r, c^r, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward of 2 for reaching the end of the terrain, and a one-time penalty of -3 for rotating more than 75 degrees from its originally orientation in either direction (after which the environment resets).

This environment runs for 600 steps.

B.5.2 DownStepper-v0

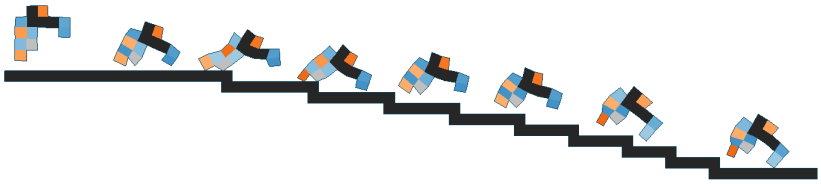

Figure 19: `DownStepper-v0`

In this task the robot climbs down stairs of varying lengths. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward of 2 for reaching the end of the terrain, and a one-time penalty of -3 for rotating more than 90 degrees from its originally orientation in either direction (after which the environment resets).

This environment runs for 500 steps.

B.5.3 ObstacleTraverser-v0

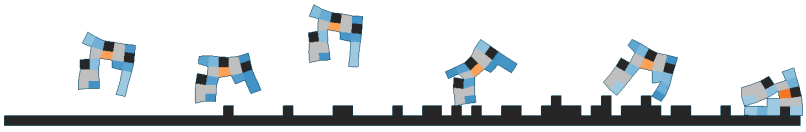

Figure 20: `ObstacleTraverser-v0`

In this task the robot walks across terrain that gets increasingly more bumpy. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward of 2 for reaching the end of the terrain, and a one-time penalty of -3 for rotating more than 90 degrees from its originally orientation in either direction (after which the environment resets).

This environment runs for 1000 steps.

B.5.4 ObstacleTraverser-v1

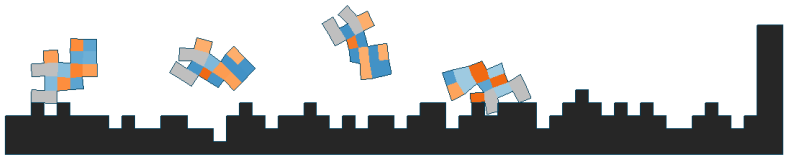

Figure 21: `ObstacleTraverser-v1`

In this task the robot walks through very bumpy terrain. This task is **hard**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward of 2 for reaching the end of the terrain (after which the environment resets).

This environment runs for 1000 steps.

B.5.5 Hurdler-v0

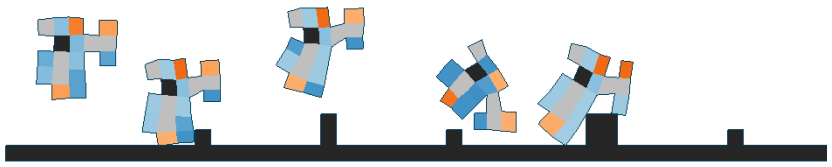

Figure 22: `Hurdler-v0`

In this task the robot walks across terrain with tall obstacles. This task is **hard**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time penalty of -3 for rotating more than 90 degrees from its originally orientation in either direction (after which the environment resets).

This environment runs for 1000 steps.

B.5.6 PlatformJumper-v0

Figure 23: `PlatformJumper-v0`

In this task the robot traverses a series of floating platforms at different heights. This task is **hard**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time penalty of -3 for rotating more than 90 degrees from its originally orientation in either direction or for falling off the platforms (after which the environment resets).

This environment runs for 1000 steps.

B.5.7 GapJumper-v0

Figure 24: `GapJumper-v0`

In this task the robot traverses a series of spaced-out floating platforms all at the same height. This task is **hard**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time penalty of -3 for falling off the platforms (after which the environment resets).

This environment runs for 1000 steps.

B.5.8 Traverser-v0

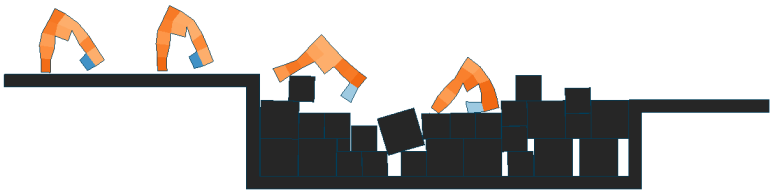

Figure 25: `Traverser-v0`

In this task the robot traverses a pit of rigid blocks to get to the other side without sinking into the pit. This task is **hard**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+14}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, \theta^r,\, c^r,\, h_b^r(5)$$

with lengths 2, 1, n, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward off 2 for reaching the end of the terrain (after which the environment resets).

This environment runs for 1000 steps.

B.5.9 CaveCrawler-v0

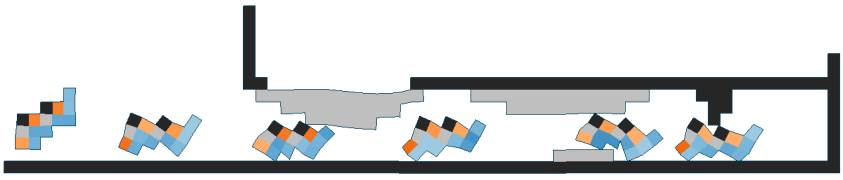

Figure 26: `CaveCrawler-v0`

In this task the robot squeezes through caves and low-hanging obstacles. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+24}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, c^r,\, h_b^r(5),\, h_a^r(5)$$

with lengths 2, n, 11, and 11 respectively. The reward R is

$$R = \Delta p_x^r$$

which rewards the robot for moving in the positive x-direction. The robot also receives a one-time reward off 1 for reaching the end of the terrain (after which the environment resets).

This environment runs for 1000 steps.

B.6 Shape change tasks

B.6.1 AreaMaximizer-v0

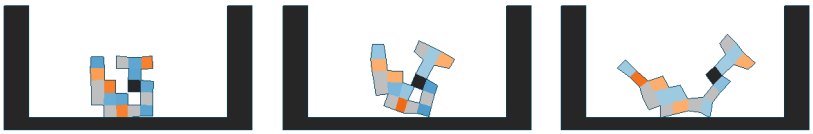

Figure 27: `AreaMaximizer-v0`

In this task the robot grows to occupy the largest possible surface area. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^n$, where n is the number of point masses in object r, and is simply the vector

$$c^r$$

with length n. Let a^r be the area of the convex hull formed by the point masses of r. The reward R is

$$R = \Delta a^r$$

which rewards the robot for growing.

This environment runs for 600 steps.

B.6.2 AreaMinimizer-v0

Figure 28: `AreaMinimizer-v0`

In this task the robot shrinks to occupy the smallest possible surface area. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^n$, where n is the number of point masses in object r, and is simply the vector

$$c^r$$

with length n. Let a^r be the area of the convex hull formed by the point masses of r. The reward R is

$$R = -\Delta a^r$$

which rewards the robot for shrinking.

This environment runs for 600 steps.

B.6.3 WingspanMaximizer-v0

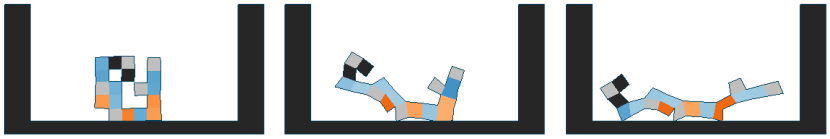

Figure 29: `WingspanMaximizer-v0`

In this task the robot grows to be as wide as possible. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^n$, where n is the number of point masses in object r, and is simply the vector

$$c^r$$

with length n. Let p^i be the vector representing the position of point mass i in r. The reward R is

$$R = \Delta\left[\max_i p_x^i - \min_i p_x^i\right]$$

which rewards the robot for growing in the x-direction.

This environment runs for 600 steps.

B.6.4 HeightMaximizer-v0

Figure 30: `HeightMaximizer-v0`

In this task the robot grows to be as tall as possible. This task is **medium**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^n$, where n is the number of point masses in object r, and is simply the vector

$$c^r$$

with length n. Let p^i be the vector representing the position of point mass i in r. The reward R is

$$R = \Delta\left[\max_i p_y^i - \min_i p_y^i\right]$$

which rewards the robot for growing in the y-direction.

This environment runs for 500 steps.

B.7 Miscellaneous tasks

B.7.1 Flipper-v0

Figure 31: `Flipper-v0`

In this task the robot flips counter-clockwise as many times as possible on flat terrain. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+1}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$\theta^r,\, c^r$$

with lengths 1 and n respectively. The reward R is

$$R = \Delta\theta^r$$

which rewards the robot for rotating counter-clockwise.

This environment runs for 600 steps.

B.7.2 Jumper-v0

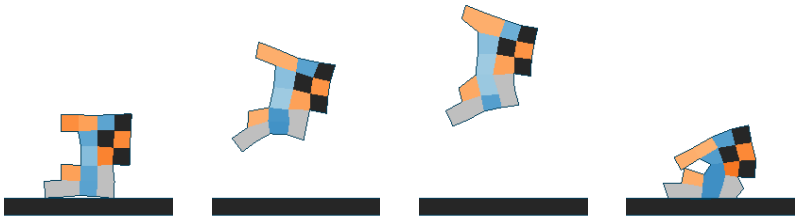

Figure 32: `Jumper-v0`

In this task the robot jumps as high as possible in place on flat terrain. This task is **easy**.

Let the robot object be r. The observation space has dimension $\mathcal{S} \in R^{n+7}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$v^r,\, c^r,\, h_b^r(2)$$

with lengths 2, n, and 5 respectively. The reward R is

$$R = 10 \cdot \Delta p_y^r - 5 \cdot |\Delta p_x^r|$$

which rewards the robot for moving in the positive y-direction and penalizes the robot for any motion in the x-direction.

This environment runs for 500 steps.

B.7.3 Balancer-v0

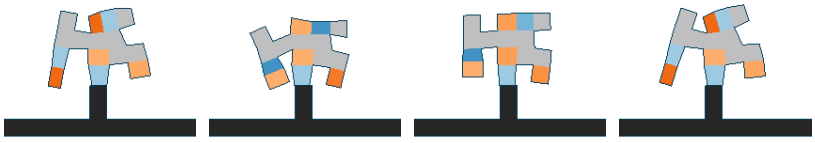

Figure 33: `Balancer-v0`

In this task the robot is initialized on top of a thin pole and balances on it. This task is **easy**.

Let the robot object be r. We achieve the described behavior by setting a goal position - g_x and g_y - for the robot located on top of the pole. The observation space has dimension $\mathcal{S} \in R^{n+2}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$g_x - p_x^r,\, g_y - p_y^r,\, c^r$$

with lengths 1, 1, and n respectively. The reward R is

$$R = -\Delta|g_x - p_x^b| - \Delta|g_y - p_y^b|$$

which rewards the robot for moving towards the goal in the x and y directions

This environment runs for 600 steps.

B.7.4 Balancer-v1

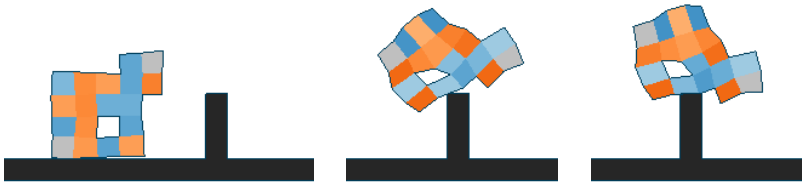

Figure 34: `Balancer-v1`

In this task the robot is initialized next to a thin pole. The robot jumps on the pole and balances on it. This task is **medium**.

Let the robot object be r. We achieve the described behavior by setting a goal position - g_x and g_y - for the robot located on top of the pole. The observation space has dimension $\mathcal{S} \in R^{n+2}$, where n is the number of point masses in object r, and is formed by concatenating vectors

$$g_x - p_x^r,\ g_y - p_y^r,\ c^r$$

with lengths 1, 1, and n respectively. The reward R is

$$R = -\Delta|g_x - p_x^b| - \Delta|g_y - p_y^b|$$

which rewards the robot for moving towards the goal in the x and y directions

This environment runs for 600 steps.

C Optimization methods

In this section, we extend the description of optimization methods that we describe in Section 4 of the main paper and also provide detailed pseudocode.

C.1 Genetic algorithm (GA)

We implement a simple GA using elitism selection and a simple mutation strategy to evolve the population of robot designs. The selection keeps the top $x\%$ of the robots from the current population as survivors and discards the rest, and the mutation can randomly change each voxel of the robot with certain probability. Crossover is not implemented for simplicity, but GA can potentially perform better with carefully designed crossover operators. See Algorithm 1 for more details in each generation of GA.

Algorithm 1 Genetic algorithm (per generation)

Inputs: History data S, population size p, current generation number N_{cur}, max generation number N_{max}.
Outputs: The proposed population of designs to evaluate $D_1, ..., D_p$.
Retrieve the population from the last generation $D_1^{'}, ..., D_p^{'}$ from S (sorted by reward).
Compute survival rate $x \leftarrow 0.6(1 - N_{cur}/N_{max})$
Compute number of survivors from the last generation $p_{sur} \leftarrow \max(2, \lceil px \rceil)$
for $i \leftarrow 1$ **to** p_{sur} **do**
  $D_i \leftarrow D_i^{'}$
for $i \leftarrow p_{sur} + 1$ **to** p **do**
  Randomly sample a parent $D_r^{'}$ from the survivors $D_1^{'}, ..., D_{p_{sur}}^{'}$
  Mutate $D_r^{'}$ to be D_i with 10% probability of changing each voxel

C.2 Bayesian optimization (BO)

BO tries to reduce the number of evaluations on expensive black-box functions by learning and utilizing a surrogate model. The surrogate model is learned, i.e. fitted by the history data, trying to map the inputs to their corresponding outputs, which mimics the real function evaluation. Instead of directly optimizing on the learned surrogate model, an acquisition function is constructed as the optimization objective in order to trade-off between exploitation and exploration of the surrogate model's prediction, which favors the uncertain region of the input space as well as the high-performing region. Next, the optimizer is applied on top of the acquisition function to search for the most promising input parameters to evaluate on the real function. Finally, the whole process repeats after the real evaluations are done and the results are added to the history data.

In our BO implementation, we use a Gaussian process as the surrogate model with a Matern 5/2 kernel [7], Expected Improvement (EI) [6] as the acquisition function, batch Thompson sampling [8] together with L-BFGS optimizer [3] to optimize the acquisition function in a batched manner. See Algorithm 2 for more details in each generation of BO.

Algorithm 2 Bayesian optimization (per generation)

Inputs: History data S, population size p.
Outputs: The proposed population of designs to evaluate $D_1, ..., D_p$.
Fit a Gaussian process model G that maps from designs D to reward r in dataset S
Build expected improvement acquisition function f based on prediction from G
Generate initial population $D_1^0, ..., D_p^0$ by random sampling
for $i \leftarrow 1$ **to** p **do**
  Optimize design D_i^0 to be D_i on acquisition function f by L-BFGS

C.3 CPPN-NEAT

As described in Section 4.1, in this method, the robot design is parameterized by a Compositional Pattern Producing Network (CPPN) and NeuroEvolution of Augmenting Topologies (NEAT) algorithm is used to evolve the structure of CPPNs by working as a genetic algorithm with specific mutation, crossover, and selection operators defined on network structures. We use a standard implementation of both CPPN and NEAT components, so please refer to the original NEAT paper for the theoretical illustrations of the optimization process, and the hyperparameters of NEAT are presented in Appendix D.

D Hyperparameters

In this section we describe hyperparameters used for each algorithm in our work. We break down this section by first describing general hyperparameters used in our co-design experiments. Next, we specify the hyperparameters specific to each design optimization algorithm. Finally, we describe the hyperparameters used in proximal policy optimization (PPO), our control optimization algorithm.

D.1 General hyperparameters

Table 2: Values of experiment hyperparameters

parameter name	value
population size	25
robot shape	(5×5), (5×7)
max evaluations	$250, 500, 750$
train iters	1000

In this section we describe some general hyperparameters used in all our co-design algorithms. Each of the co-design algorithms operates on a population of individuals, the size of which is specified by the parameter *population size.* These algorithms also work with a grid-like design space whose size

is specified by *robot shape*. For most tasks the (5×5) grid size is sufficient for the algorithms to find complex, interesting robots. The only exception is the *Lifter* task where we increase the size of the design space to a (5×7) grid, to accommodate a near-optimal robot we hand-designed. *Max evaluations* specifies how many unique robots are trained in each algorithm. We use this metric to compare algorithms (instead of, for instance, a maximum number of generations) because the Genetic algorithm trains a different number of unique robots per generation compared to the Bayesian optimization and CPPN-NEAT algorithms. *Max evaluations* varies between tasks as we use less evaluations to train algorithms whose performance converges faster. We do not train any algorithms more than 750 evaluations. Finally, we train each robot for *train iters* iterations using reinforcement learning (RL) in order to evaluate its performance on the task at hand. Note that the number of total RL steps will be the product of *train iters*, *num steps*, and *num processes* (from Section D.3) for a total of 512000 steps.

D.2 Design optimization hyperparameters

In this section we specify hyperparameters relevant to our design optimization algorithms.

D.2.1 Genetic algorithm (GA)

Table 3: Values of GA hyperparameters

parameter name	value
mutation rate	10%
survivor rate range	$[0.0, 0.6]$

The genetic algorithm only has two significant hyperparameters. The *mutation rate* is important for constructing offspring robots of an existing survivor robot. The *mutation rate* specifies the probability of mutating each voxel of the survivor robot's structure and the resulting structure after mutation becomes that of the offspring robot. The *survivor rate range* specifies how the percent of robots that survive each generation of the algorithm changes over time. The survivor rate starts at the maximum value in the range and decreases linearly to the minimum value.

D.2.2 Bayesian optimization (BO)

Table 4: Values of BO hyperparameters

parameter name	value
kernel variance σ	1.0
kernel length scale l	$(1, ..., 1) \in \mathbb{R}^d$
optimizer max iterations	100
optimizer restarts	5

We use the default implementation from the `GPyOpt` package and we do not change any specific hyperparameters. Values of the most important hyperparameters are listed in Table 4. σ and l are the variance and the length scale of the Matern 5/2 kernel in the surrogate model, where d is the dimension of the input (number of voxels). The *optimizer max iterations* is the max number of iterations used to optimize the parameters of the surrogate model by the L-BFGS optimizer, and *optimizer restarts* specifies the number of restarts in the optimization.

D.2.3 CPPN-NEAT

The hyperparameters of CPPN-NEAT are listed in Table 5, whose interpretations can be found in the documentation of the neat-python package (`https://neat-python.readthedocs.io/en/latest/config_file.html`).

Table 5: Values of CPPN-NEAT hyperparameters

parameter name	value	parameter name	value
pop size	50	bias mutate rate	0.7
num inputs	3	bias mutate power	0.5
num hidden	1	bias max value	30.0
num outputs	5	bias min value	-30.0
initial connection	partial direct 0.5	response init mean	1.0
feed forward	True	response init stdev	0.0
compatibility threshold	3.0	response replace rate	0.0
compatibility disjoint coefficient	1.0	response mutate rate	0.0
compatibility weight coefficient	0.6	response mutate power	0.0
conn add prob	0.2	response max value	30.0
conn delete prob	0.2	response min value	-30.0
node add prob	0.2	weight max value	30
node delete prob	0.2	weight min value	-30
activation options	sigmoid	weight init mean	0.0
activation mutate rate	0.0	weight init stdev	1.0
aggregation options	sum	weight mutate rate	0.8
aggregation mutate rate	0.0	weight replace rate	0.1
bias init mean	0.0	weight mutate power	0.5
bias init stdev	1.0	enabled default	True
bias replace rate	0.1	enabled mutate rate	0.01

Table 6: Values of PPO hyperparameters

parameter name	value
use gae	True
learning rate	$2.5 \cdot 10^{-4}$
use linear learning rate decay	True
clip parameter	0.1
value loss coefficient	0.5
entropy coefficient	0.01
num steps	128
num processes	4
evaluation interval	50

D.3 Control optimization hyperparameters

In this section we specify hyperparameters relevant to our control optimization algorithm (PPO), which are listed in Table 6. The *use gae* indicates whether we apply Generalized Advantage Estimation (GAE) [9] during PPO. The *learning rate* is for the Adam optimizer [4] optimizing the actor and critic networks and we *use linear learning rate decay* throughout training. The *clip parameter*, *value loss coefficient*, *entropy coefficient* are easily explained in any brief reference manual on PPO. The *num steps* specifies the number of steps that each process samples in each iteration of PPO while the *num processes* indicates the number of processes we use for parallel sampling. The *evaluation interval* specifies the number of training iterations between evaluations. For all other parameters we use their default values.

E Complete experiment results

In the main paper, we highlight ten tasks in our environment benchmark suite and compare their performance using all three design optimization algorithms. Our full benchmark suite includes 32 tasks, described in detail in Section B. This sections details the evaluation results for all 32 tasks.

We evaluated the complete benchmark suite using the baseline co-design algorithm with the GA for design optimization and PPO for the control optimization. Figure 35 shows the reward curves for the

GA baseline algorithm on all 32 tasks. In our main benchmark suite of ten tasks (see Section 5 of the main text), we found the GA baseline algorithm outperformed the other baselines algorithms the majority of the time.

We highlight the design and control optimization results of a subset of six tasks in Figures 36 and 37: Balancer-v0, CaveCrawler-v0, PlatformJumper-v0, DownStepper-v0, BeamToppler-v0, and Hurdler-v0. In Figure 36, for each of the six tasks, we visualize the top four robots from three different generations. We also show the average reward these designs achieve.

Balancer-v0 requires the robot balance on top of a thin pole. Figure 36 reveals how the robot slowly evolves arm-like features which it actuates to maintain balance atop the pole – much like how humans stretch out their arms to balance.

CaveCrawler-v0 requires the robot slither under and between a number of low-hanging obstacles. By the last generation, the robots have converged on a small snake-like form which is short enough to clear rigid obstacles and is lined with horizontal actuators to allow slithering motion across the ground.

PlatformJumper-v0 requires the robot jump between floating platforms at different heights. Consequently, optimal designs contain many actuators oriented such that the robot can spring forward at will.

DownStepper-v0 requires the robot traverse down an uneven staircase. Optimal robots evolve towards a bipedal form – with a horizontal actuator on one foot and a vertical actuator on the other to promote seamless movement.

BeamToppler-v0 requires the robot knock over a beam resting on two pegs. Interestingly, two optimal designs survive the co-design optimization. Both designs evolve a hand-like mechanical arm to push the beam from underneath. But one design (#4 in generation 40, Figure 36) also evolves a hook-like gripper to push the beam off from above instead of from below.

In Figure 37 we show step-by-step sequences of the performance of six optimized designs.

A number of tasks perform very well. Optimal robots in Balancer-v0 actuate their arm-like counterweights to maintain their position atop the thin pole. Robots in DownStepper-v0 quickly run down the unevenly spaced stairs using their bipedal legs. Optimal robots in BeamToppler-v0 repeatedly actuate their hand-like mechanism to nudge the beam of its pegs.

Some environments have few successful robots. Most near-optimal designs produced in CaveCrawler-v0 are unable to clear the last rigid low-hanging obstacle in the cave. Although some, like the robot shown in Figure 37 are able to clear all sections.

Some tasks are more complicated than others and the baseline algorithms fails to evolve a fully successful robot. For example, the optimal robot in the PlatformJumper-v0 environment successfully lands on many platforms, but ultimately gets stuck in a gap between two platforms. In Hurdler-v0, the robot is able to clear many tall thin vertical obstacle, but its hook-like jumping design ultimately fails when it gets caught on a wider vertical barrier.

A visualization of all robots for the 32 benchmark tasks is included in the Supplementary Materials.

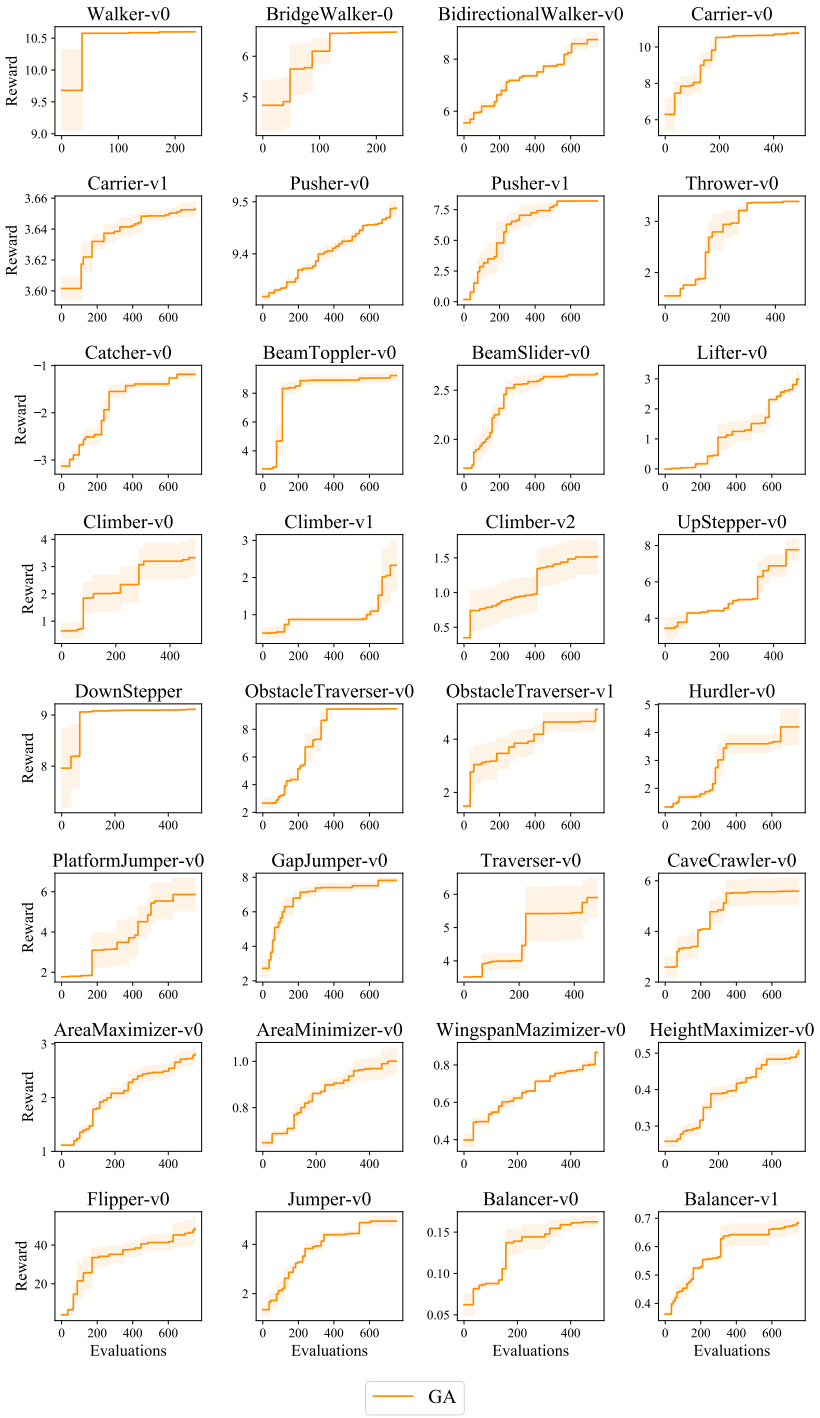

Figure 35: **Performance of GA baseline algorithm.** We plot the best performance of robots that the GA algorithm has evolved w.r.t. the number of evaluations on each task. All the curves are averaged over 3 different random seeds, and the variance is shown as a shaded region.

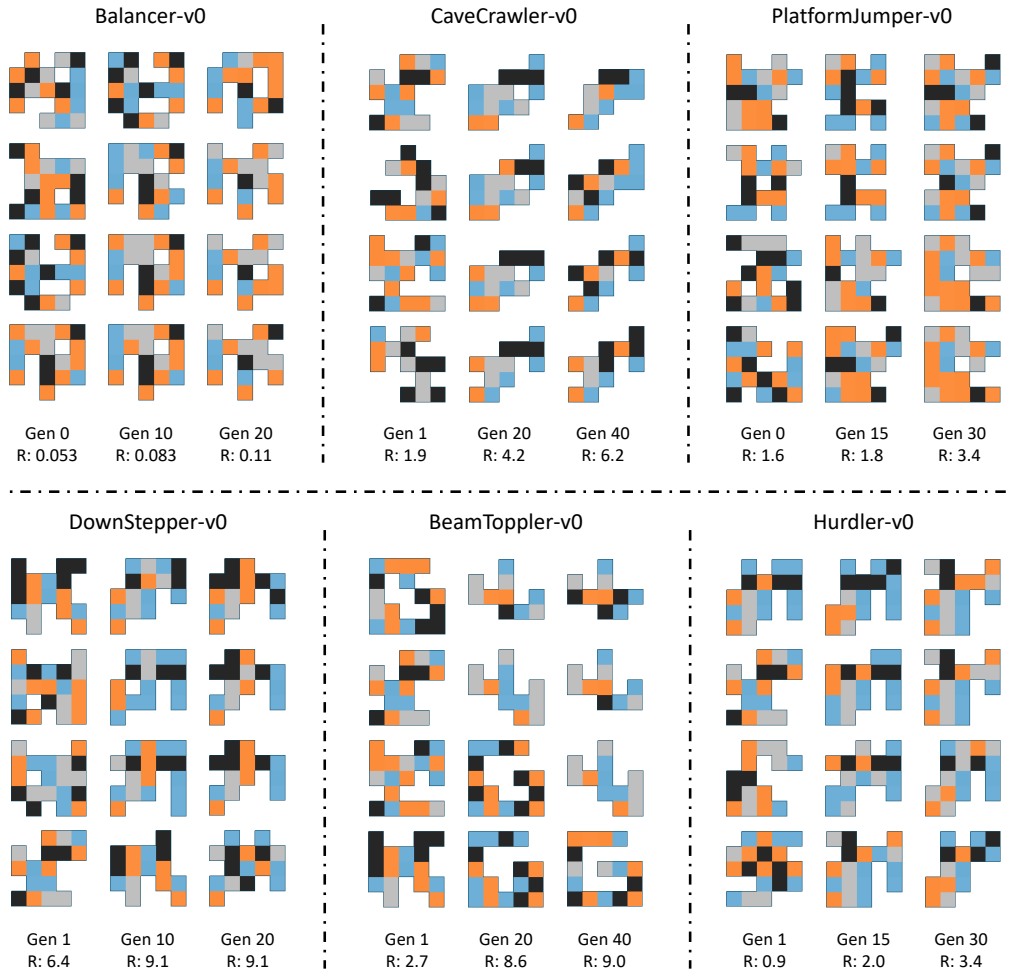

Figure 36: **Evolution of robot designs.** For each of the six selected tasks, we visualize the population in three different generations. Each column corresponds to one generation for which we show the four top performing robots along with their average reward.

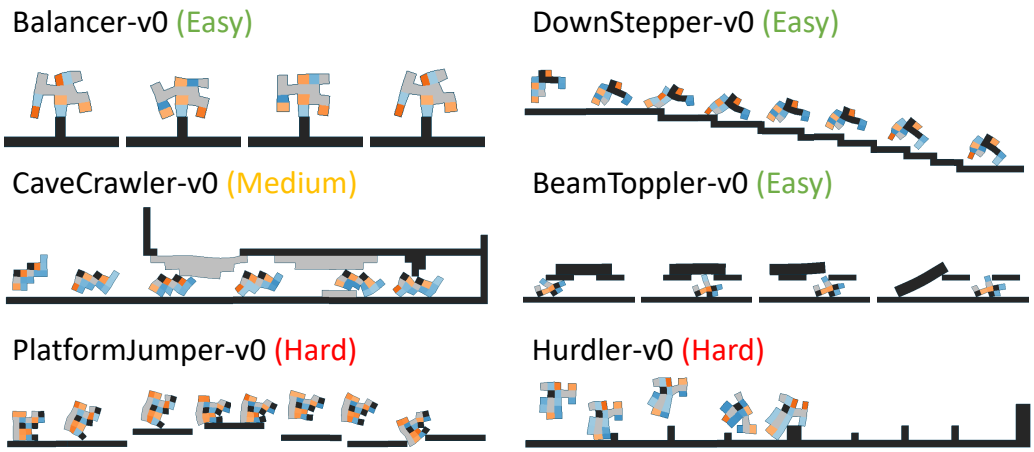

Figure 37: **Algorithm-optimized robots**. For each of the six selected tasks, we visualize a step-by-step sequence of a robot optimized by the algorithm.

References

[1] Tao Du, Kui Wu, Pingchuan Ma, Sebastien Wah, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. Diffpd: Differentiable projective dynamics with contact. *arXiv preprint arXiv:2101.05917*, 2021.

[2] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[3] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[5] Eric Medvet, Alberto Bartoli, Andrea De Lorenzo, and Giulio Fidel. Evolution of distributed neural controllers for voxel-based soft robots. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 112–120, 2020.

[6] Jonas Močkus. On bayesian methods for seeking the extremum. In *Optimization techniques IFIP technical conference*, pages 400–404. Springer, 1975.

[7] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[8] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A tutorial on thompson sampling. *arXiv preprint arXiv:1707.02038*, 2017.

[9] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[10] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.