`robosuite`: A Modular Simulation Framework and Benchmark for Robot Learning

Yuke Zhu[♣] Josiah Wong[♣] Ajay Mandlekar[♣] Roberto Martín-Martín[♣*]
Abhishek Joshi[◇] Kevin Lin [◇] Abhiram Maddukuri[◇]
Soroush Nasiriany[◇] Yifeng Zhu[◇†]

robosuite.ai

Abstract

`robosuite` is a simulation framework for robot learning powered by the MuJoCo physics engine. It offers a modular design for creating robotic tasks as well as a suite of benchmark environments for reproducible research. This paper discusses the key system modules and the benchmark environments of our new release `robosuite` v1.5. For the latest updates on `robosuite`, please visit our project website.

1 Introduction

We introduce `robosuite`, a modular simulation framework and benchmark for robot learning. This framework is powered by the MuJoCo physics engine [15], which performs fast physical simulation of contact dynamics. The overarching goal of this framework is to facilitate research and development of data-driven robotic algorithms and techniques. The development of this framework was initiated from the SURREAL project [3] on distributed reinforcement learning for robot manipulation, and is now part of the broader Advancing Robot Intelligence through Simulated Environments (ARISE) Initiative, with the aim of lowering the barriers of entry for cutting-edge research at the intersection of AI and Robotics.

Data-driven algorithms [9], such as reinforcement learning [13, 7] and imitation learning [12], provide a powerful and generic tool in robotics. These learning paradigms, fueled by new advances in deep learning, have achieved some exciting successes in a variety of robot control problems. Nonetheless, the challenges of reproducibility and the limited accessibility of robot hardware have impaired research progress [5]. In recent years, advances in physics-based simulations and

[*]♣: founding members who initiate and lead this project
[†]◇: core members who make significant contributions (in alphabetical order)

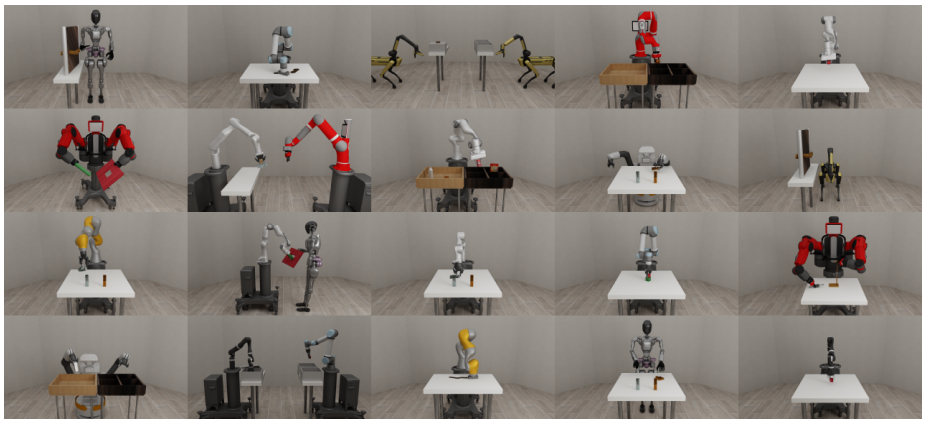


Figure 1: Procedurally generated robotic environments with `robosuite` APIs

graphics have led to a series of simulated platforms and toolkits [1, 14, 8, 2, 16] that have accelerated scientific progress on robotics and embodied AI. Through the `robosuite` project we aim to provide researchers with:

1. a modular design that offers great flexibility to create new robot simulation environments and tasks;
2. a high-quality implementation of robot controllers and off-the-shelf learning algorithms to lower the barriers to entry;
3. a standardized set of benchmark tasks for rigorous evaluation and algorithm development.

Our new release of `robosuite` v1.5 contains ten robot models, nine gripper models, four base models, six body part controller modes, and nine standardized tasks. The body part controllers span the most frequently used action spaces (joint space and Cartesian space) and have been tested and used in previous projects [11]. We use a composite controller interface to compose different body part controllers. `robosuite` also offers a modular design of APIs for building new environments with procedural generation. We highlight these primary features below:

1. standardized tasks: a set of standardized manipulation tasks of large diversity and varying complexity and RL benchmarking results for reproducible research;
2. procedural generation: modular APIs for programmatically creating new environments and new tasks as a combinations of robot models, arenas, and parameterized 3D objects;
3. realistic composite robot controllers: a high-level composite controller that orchestrates a selection of body part controller implementations to command the robots in joint space and Cartesian space, in position, velocity

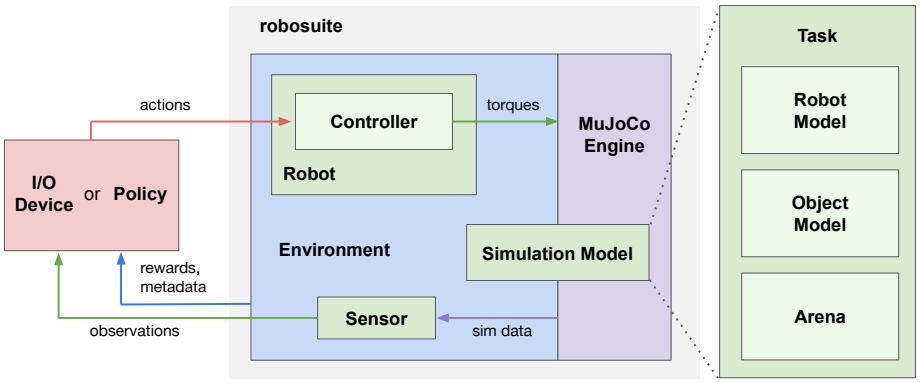


Figure 2: System diagram of `robosuite` modules. An actor (e.g. a Policy or a human using an I/O Device) generates actions commands and pass them to the `robosuite` Environment. The action is transformed by the controller of the robot into torque commands and executed by the MuJoCo physics engine. The result of the execution is observed via Sensors, that provide observations to the actors, together with reward and metadata from the Environment. The Environment is the result of instantiating a Task model composed of a Robot Model, an Arena, and, possibly, some objects defined by Object Models.

or torque, including inverse kinematics control, and operational space control;

4. multi-modal sensors: heterogeneous types of sensory signals, including low-level physical states, RGB cameras, depth maps, segmentation masks, and proprioception;

5. human demonstrations: utilities for collecting human demonstrations (with keyboard, 3D mouse and GUI with cursor devices), replaying demonstration datasets, and leveraging demonstration data for learning.

In the rest of the manuscript, we will describe the overall design of the simulation framework and the key system modules. We will also describe the benchmark tasks in the v1.0 `robosuite` release and benchmarking results of the most relevant state-of-the-art data-driven algorithms on these tasks.

2 System Modules

In this section we describe the overall system design of `robosuite`. `robosuite` offers two main APIs: 1) **Modeling APIs** to describe and define simulation environments and tasks in a modular and programmatic fashion, and 2) **Simulation APIs** to wrap around the physics engine and provide an interface for external actors (i.e. a **Policy** or an **I/O Device**) to execute actions, and receive observations and rewards. The Modeling APIs are used to generate a

Simulation Model that can be instantiated by the MuJoCo engine [15] to create a simulation runtime, called **Environment**. The Environment generates observations through the **Sensors**, such as cameras and robot proprioception, and receives action commands from policies or I/O devices that are transformed from the original action space (e.g. joint velocities, or Cartesian positions) into torque commands by the **Controllers** of the **Robots**. The diagram above illustrates the key components in our framework and their relationships.

A simulation model is defined by a **Task** object, which encapsulates three essential constituents of robotic simulation: **RobotModel**(s), Object Model(s), and an **Arena**. A task may contain one or more robots, zero to many objects, and a single arena. The **RobotModel** object loads models of robots and their corresponding **GripperModel**(s) and **RobotBaseModel** from provided XML files. The Object Model defined by **MujocoObject** can be either loaded from 3D object assets or procedurally generated with programmatic APIs. The **Arena** object defines the workspace of the robot, including the environment fixtures, such as a tabletop, and their placements. The **Task** class recombines these constituents into a single XML object in MuJoCo's **MJCF modeling language**. The generated MJCF object is passed to the MuJoCo engine through the **mujoco** library that instantiates and initializes the simulation. The result of this instantiation is a MuJoCo runtime simulation object (the **MjSim** object) that contains the state of the simulator, and that will be interfaced via our Simulation APIs.

The **Environment** object (see Section 2.1) provides **OpenAI Gym**-style APIs for external inputs to interface with the simulation. External inputs correspond to the action commands used to control the **Robot**(s) their grippers and their bases (see Section 2.2), where the action spaces are specific to the **Controller**(s) used by the robots (see Section 2.3). For instance, for a joint-space position controller for a robot arm, the action space corresponds to desired position of each joint of the robot (dimensionality=number of degrees of freedom of the robot), and for an operational space controller, the action space corresponds to either desired 3D Cartesian position or desired full 6D Cartesian pose for the end-effector. These action commands can either be automatically generated by an algorithm (such as a deep neural network policy) or come from an I/O **Device**(s) for human teleoperation, such as the keyboard or a 3D mouse (see Section 2.6). The controller of the robot is responsible for interpreting these action commands and transforming them into the low-level torques passing to the underlying physics engine (MuJoCo), which performs internal computations to determine the next state of the simulation. **Sensor**(s) retrieve information from the **MjSim** object and generate observations that correspond to the physical signals that the robots receive as response to their actions (see Section 2.5). Our framework supports multiple sensing modalities, such as RGB-D cameras, force-torque measurements, and proprioceptive data, allowing multimodal solutions to be developed. In addition to these sensory data, environments also provide additional information about the task progress and success conditions, including reward (for reinforcement learning) and other meta-data (e.g. task completion). In the following, we describe in detail the individual components

of robosuite.

2.1 Environments

Environments provide the main APIs for external/user code to interact with the simulator and perform tasks. Each environment corresponds to a robotic task and provides a standard interface for an agent to interact with the environment.

Environment(s) are created by calling `make` with the name of the task (see Section 3.1 for a suite of standardized tasks provided in `robosuite`) and with a set of arguments that configure environment properties such as whether on-screen (`has_renderer`) or off-screen rendering (`has_offscreen_renderer`) is used, whether the observation space includes physical states (`use_object_obs`) or image (`use_camera_obs`) observations, the control frequency (`control_freq`), the episode horizon (`horizon`), and whether to shape rewards or use a sparse one (`reward_shaping`).

Environments have a modular structure, making it easy to construct new ones with different robot arms (`robots`), grippers (`gripper_types`), and controllers (`controller_configs`). Every environment owns its own MJCF model that sets up the MuJoCo physics simulation by loading the robots, the workspace, and the objects into the simulator appropriately. This MuJoCo simulation model is programmatically instantiated in the `_load_model` function of each environment, by creating an instance of the **Task** class.

Each **Task** class instance owns an **Arena** model, a list of **Robot** model instances, and a list of **Object** model instances. These are `robosuite` classes that introduce a useful abstraction in order to make designing scenes in MuJoCo easy. Every **Arena** is based off of an XML that defines the workspace (for example, table or bins) and camera locations. Every **Robot** is a MuJoCo model of each type of robot arm (e.g., Sawyer, Panda, etc.). Every **Object** model corresponds to a physical object loaded into the simulation (e.g., cube, pot with handles, etc.).

Each **Task** class instance also takes a `placement_initializer` as input. The `placement_initializer` determines the start state distribution for the environment by sampling a set of valid, non-colliding placements for all of the objects in the scene at the start of each episode (e.g., when `env.reset()` is called).

2.2 Robots

Robots are a key component in `robosuite`, serving as the embodiment of the agent that interacts within the environment. `robosuite` captures this level of abstraction with the **Robot**-based classes, with support for both single-armed and bimanual variations, as well as robots with mobile manipulation capabilities, including both legged and wheeled variants. In turn, the Robot class is defined by a **RobotModel**, **GripperModel**(s) (with no gripper being represented by a

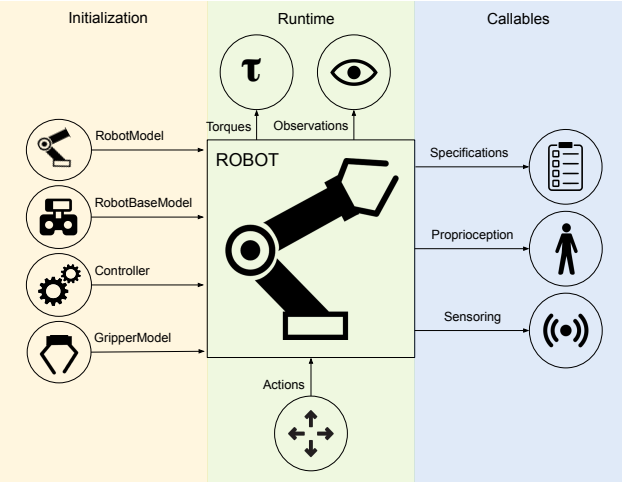


Figure 3: Overview of the Robot module's structure and usage. A **Robot** is initialized with appropriate models and controller, interacts with the environment during runtime, and can be accessed to retrieve relevant state information at any point during the simulation.

dummy class), **RobotBaseModel**, and **Controller**(s). The high-level features of `robosuite`'s robots are described as follows:

- **Diverse and Realistic Models:** the current version of `robosuite` provides models for 10 commercially-available robots (including the **humanoid** GR1 Robot), 9 grippers (including the Inspire **dexterous hand** model), 4 bases (including the Omron wheeled mobile base), and 6 body-part controllers, with model properties either taken directly from official product documentation or raw spec sheets. We also provide an extension package from the `robosuite-models` repository which currently includes additional 8 robots, 8 grippers, and 3 bases. This extension package must be installed separately and it is actively maintained.

- **Modularized Support:** Robots are designed to be plug-and-play—any combinations of robots, grippers, bases, and controllers can be used, assuming the given environment is intended for the desired robot configuration. Because each robot is assigned a unique ID number, multiple instances of identical robots can be instantiated within the simulation without error.

- **Self-Enclosed Abstraction:** For a given task and environment, any information relevant to the specific robot instance can be found within the properties and methods within that instance. This means that each robot is responsible for directly setting its initial state within the simulation at

the start of each episode, and also directly controls the robot in simulation via torques outputted by its controller's transformed actions.

`robosuite` currently supports 10 commercially-available robot models. We briefly describe each individual model along with its features below:

Panda is a 7-DoF and relatively new robot model produced by Franka Emika, and boasts high positional accuracy and repeatability. A common choice for both simulated and real-robot research, we provide a substantial set of benchmarking experiments using this robot. The default gripper for this robot is the **PandaGripper**, a parallel-jaw gripper equipped with two small finger pads, that comes shipped with the robot arm.

Sawyer is Rethink Robotic's 7-DoF single-arm robot, which also features an additional 8th joint (inactive and disabled by default in `robosuite`) for swiveling its display monitor. Along with Panda, Sawyer serves as the second testing robot for our set of benchmarking experiments. Sawyer's default **RethinkGripper** model is a parallel-jaw gripper with long fingers and useful for grasping a variety of objects.

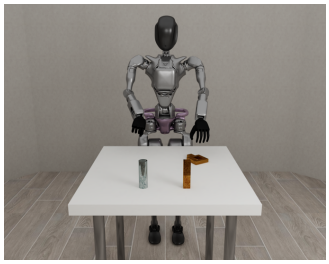
IIWA is one of KUKA's industrial-grade 7-DoF robots, and is equipped with the strongest actuators of the group, with its per-joint torque limits exceeding nearly all the other models in `robosuite` by over twofold! By default, IIWA is equipped with the **Robotiq140Gripper**, **Robotiq's 140mm variation** of their multi-purpose two finger gripper models.

Jaco is a popular sleek 7-DoF robot produced by Kinova Robotics and intended for human assistive applications. As such, it is relatively weak in terms of max torque capabilities. Jaco comes equipped with the **JacoThreeFingerGripper** by default, a three-pronged gripper with multi-jointed fingers.

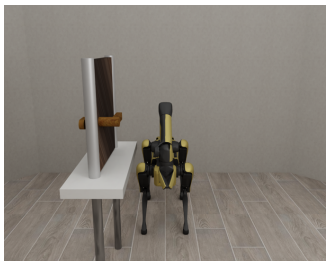
Kinova3 is Kinova's newest 7-DoF robot, with integrated sensor modules and interfaces designed for research-oriented applications. It is marginally stronger than its Jaco counterpart, and is equipped with the **Robotiq85Gripper**, **Robotiq's 85mm variation** of their multi-purpose two finger gripper models.

UR5e is Universal Robot's newest update to the UR5 line, and is a 6-DoF robot intended for collaborative applications. This newest model boasts an improved footprint and embedded force-torque sensor in its end effector. This arm also uses the **Robotiq85Gripper** by default in `robosuite`.

Baxter is an older but classic bimanual robot originally produced by Rethink Robotics but now owned by Co-Think Robotics, and is equipped with two 7-DoF arms as well as an addition joint for controlling its swiveling display screen (inactive and disabled by default in `robosuite`). Each arm can be controlled independently in, and is the single multi-armed model currently supported in `robosuite`. Each arm is equipped with a **RethinkGripper** by default.



GR1 is a 44-DoF humanoid robot produced by Fourier Intelligence. Standing 165 cm tall and weighing 55 kg, GR1 has the capability for locomotion, bimanual manipulation, and head movement. GR1 also features cutting-edge vision and sound sensors, enabling intuitive human-like interactions. Attached to each arm is a dexterous hand by default in `robosuite`.



Spot is a 12-DoF, agile, four-legged robot developed by Boston Dynamics. It is capable of navigating complex terrain, avoiding obstacles, and carrying up to 14 kg of payload capacity. In `robosuite`, the Spot robot is equipped with a 6-DoF arm by default.

TIAGo is a bimanual mobile manipulator robot developed by PAL Robotics. Key features include navigation, interactibility, and a modular design that allows for customization of end effectors, base drives, and computing capabilities. In `robosuite`, the TIAGo robot is equipped with two 7-DoF arms and has mobile base control.

2.3 Controllers

Controllers transform the high-level actions into low-level virtual motor commands that actuate the robots. The high-level actions are interpreted as reference signals for the controllers, i.e., desired configurations to be reached. Underlying all our robot models is a set of virtual motors actuated at each joint that execute given torques. The controllers will translate the reference signals into corresponding joint torque values to try to achieve that desired configuration. Starting from `robosuite` v1.5, we employ **composite** controllers. A composite controller takes in a high-level action vector and converts it into commands for each **body part** controller, where each body part's controller is responsible for translating supplied actions into joint torques for that specific part. This design enables modularity when controlling robots that can be decomposed into multiple body parts. For example, the Operational Space Controllers [6] interpret high-level actions as desired end-effector configurations, either positions (three degrees of freedom) or poses (six degrees of freedom), and compute the corresponding joint torques to move the robot's end-effector from its current pose to a desired one. Our controllers facilitate sim-to-real transferability, as torque-based controllers are common to most real-world existing robotic platforms such as Rethink Robotics Sawyer, Franka Panda, Kuka IIWA, and Kinova Jaco, and enables contact-rich manipulation with control of the interaction forces.

We include the following composite controller options as part of `robosuite`: `BASIC` and `WHOLE_BODY_IK`. The `BASIC` composite controller directly splits and passes down the high level action vector to the individual body part controllers that operate independently to control various parts of the robot, such as arms, torso, head, base, and legs. Each part can be assigned a specific body part controller type (e.g., `OSC_POSE` and `JOINT_POSITION`) depending on the desired

Controller Name and Options	Controller Type	Action Dimensions (Gripper Not Included)	Action Format
OSC_POSE impedance_mode = fixed	Operational Space Control (Position & Orientation)	6	$(p_x, p_y, p_z, r_x, r_y, r_z)$
OSC_POSE impedance_mode = variable_kp	Operational Space Control with variable stiffness (critically damped)	12	$(p_x, p_y, p_z, r_x, r_y, r_z)$ $(k^p_{px}, k^p_{py}, k^p_{pz}, k^p_{rx}, k^p_{ry}, k^p_{rz})$
OSC_POSE impedance_mode = variable	Operational Space Control with variable impedance	18	$(p_x, p_y, p_z, r_x, r_y, r_z)$ $(k^p_{px}, k^p_{py}, k^p_{pz}, k^p_{rx}, k^p_{ry}, k^p_{rz})$ $(k^d_{px}, k^d_{py}, k^d_{pz}, k^d_{rx}, k^d_{ry}, k^d_{rz})$
OSC_POSITION impedance_mode = fixed	Operational Space Control (Position only)	3	(p_x, p_y, p_z)
OSC_POSITION impedance_mode = variable_kp	Operational Space Control with variable stiffness (critically damped)	9	(p_x, p_y, p_z) $(k^p_{px}, k^p_{py}, k^p_{pz}, k^p_{rx}, k^p_{ry}, k^p_{rz})$
OSC_POSITION impedance_mode = variable	Operational Space Control with variable impedance	15	(p_x, p_y, p_z) $(k^p_{px}, k^p_{py}, k^p_{pz}, k^p_{rx}, k^p_{ry}, k^p_{rz})$ $(k^d_{px}, k^d_{py}, k^d_{pz}, k^d_{rx}, k^d_{ry}, k^d_{rz})$
IK_POSE	Inverse Kinematics Control (Position & Orientation)	7	$(p_x, p_y, p_z, q_x, q_y, q_z, q_w)$
JOINT_POSITION impedance_mode = fixed	Joint Position Control	n	n joint positions
JOINT_POSITION impedance_mode = variable_kp	Joint Position Control with variable stiffness (critically damped)	2n	n joint positions and k^p for each joint
JOINT_POSITION impedance_mode = variable	Joint Position Control with variable impedance	3n	n joint positions and (k^p, k^d) for each joint
JOINT_VELOCITY	Joint Velocity Control	n	n joint velocities
JOINT_TORQUE	Joint Torque Control	n	n joint torques

Table 1: Body Part Controller Configurations available in `robosuite`

control behavior for that part. For example, arms may use `OSC_POSE` for precise end-effector control, while the base may use `JOINT_VELOCITY` for movement across the ground. The `WHOLE_BODY_IK` composite controller assumes that the high level action vector contains end effector pose targets and uses an inverse kinematics solver to compute joint angles to reach those pose targets. Then, the composite controller passes the joint angles down to the corresponding `JOINT_POSITION` body part controllers. Finally, users can leverage custom third-party composite controllers, by subclassing the **CompositeController** class and implementing the custom methods. We provide an example of a third-party composite controller implementation, `WHOLE_BODY_MINK_IK` [17], in `robosuite`.

We include the following body part controller options as part of `robosuite`: `OSC_POSE`, `OSC_POSITION`, `JOINT_POSITION`, `JOINT_VELOCITY`, and `JOINT_TORQUE` (see Table 1). For `OSC_POSE`, `OSC_POSITION`, and `JOINT_POSITION`, we include three variants: First, the most common variant is to use a predefined and **fixed** set of impedance parameters (`impedance_mode = fixed`). In this case, the action space only includes the desired pose, position, or joint configuration. The second options is to control the **stiffness** of the actuation (`impedance_mode = variable_kp`), i.e., with how much force will the robot react to deviations to the desired configuration. This is controlled via the proportional parameters of the controller (k_p). The damping parameters (k_d) are automatically set to the values that lead to a critically damped system. The third variant allows full control of the **impedance** behavior (`impedance_mode = variable`), with the actions including both stiffness and damping parameters. These last two variants lead to extended action spaces that can control the stiffness and damping behavior of the controller in a variable manner over the course of an interaction, providing full control to the policy/solution over the contact and dynamic behavior of the robot.

For the `OSC_POSITION` variants, the robot will hold the initial orientation while trying to achieve the position given in the action. Variants controlling stiffness, or stiffness and damping can specify not only these parameters for the position but also for orientation. Therefore, the dimensionality of the action spaces with these controllers are 9 and 15 (row 6 and 7 in Table 1).

2.4 Objects

Objects, such as boxes and cans, are essential to building manipulation environments. We designed the **MujocoObject** interfaces to standardize and simplify the procedure for importing 3D models into the scene or procedurally generate new objects. MuJoCo defines models via the MJCF XML format. These MJCF files can either be stored as XML files on disk and loaded into simulator, called **MujocoXMLObject**, or be created on-the-fly by code prior to simulation, called **MujocoGeneratedObject**. Based on these two mechanisms of how MJCF models are created, we offer two main ways of creating your own object:

- Define an object in an MJCF XML file;
- Use procedural generation APIs to dynamically create an MJCF model.

In the former case, an object can be defined using MuJoCo's native **MJCF format** and can be loaded directly into the simulation using `robosuite`'s API. In the latter case, a complex object can be defined by sequentially composing a set of primitive geoms and defining their poses relative to the rest of the object. Additionally, `robosuite` supports custom texture definitions to be added to specific geoms defined. We refer the interested reader to the **HammerObject** class as a showcase example for procedurally-generated objects.

2.5 Sensors

The simulator generates virtual physical signals as response to a robot's interactions through **Sensors**. Virtual signals include images, force-torque measurements (from a force-torque sensor like the one included by default in the wrist of all **Gripper** models), pressure signals (e.g., from a sensor on the robot's finger or on the environment), etc. Sensors, except cameras and joint sensors, are accessed via the function `get_sensor_measurement`, by providing the name of the sensor.

Every robot joint provides information about its state, including position and velocity. In MuJoCo these are not measured by sensors, but resolved and set by the simulator as the result of the actuation forces. Therefore, they are not accessed through the common `get_sensor_measurement` function but as properties of the **Robot** simulation API, for instance, `_joint_positions` and `_joint_velocities`.

Cameras bundle a name to a set of properties to render images of the environment such as the pose and pointing direction, field of view, and resolution. Inheriting from MuJoCo, cameras are defined in the **RobotModel** and **Arena** models and can be attached to any body. Images, as they would be generated from the cameras, are not accessed through `get_sensor_measurement` but via the renderer (e.g., OpenGL-based **MjViewer** or **PyGame**). In a common user pipeline, images are not queried directly; we specify one or several cameras we want to use images from when we create the environment, and the images are generated and appended automatically to the observation dictionary.

2.6 I/O Devices

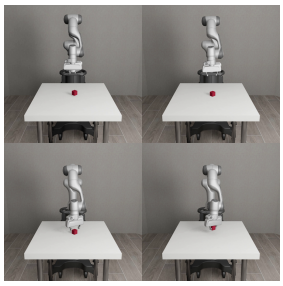
Devices define the external controllers a user can teleoperate robots in real-time. A **Device** object reads user input from I/O devices, such as a **Keyboard**, **SpaceMouse** or **MJGUI**, and parse it into control commands sent to the robots. The **MJGUI** device uses MuJoCo's built in GUI and users' cursor to drag and drop certain pre-defined robot controller targets. The **SpaceMouse** from 3Dconnexion is a 3D mouse that we used extensively for collecting demonstrations [18, 10] and debugging task designs. More generally, we support any interface that implements the **Device** abstract class. To support your

own custom device, simply subclass this base class and implement the required methods.

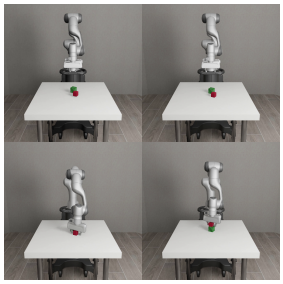
3 Benchmark Environments

3.1 Task Descriptions

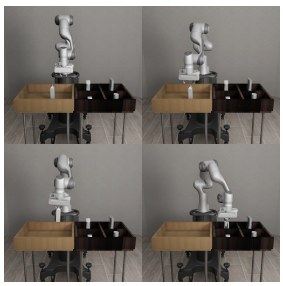
We provide a brief description of each environment below, along with a sequence of frames that depict a successful rollout.



Block Lifting: A cube is placed on the tabletop in front of a single robot arm. The robot arm must lift the cube above a certain height. The cube location is randomized at the beginning of each episode.



Block Stacking: Two cubes are placed on the tabletop in front of a single robot arm. The robot must place one cube on top of the other cube. The cube locations are randomized at the beginning of each episode.



Pick-and-Place: Four objects are placed in a bin in front of a single robot arm. There are four containers next to the bin. The robot must place each object into its corresponding container. This task also has easier single-object variants. The object locations are randomized at the beginning of each episode.

Nut Assembly: Two colored pegs (one square and one round) are mounted on the tabletop, and two colored nuts (one square and one round) are placed on the table in front of a single robot arm. The robot must fit the square nut onto the square peg and the round nut onto the round peg. This task also has easier single nut-and-peg variants. The nut locations are randomized at the beginning of each episode.

Door Opening: A door with a handle is mounted in free space in front of a single robot arm. The robot arm must learn to turn the handle and open the door. The door location is randomized at the beginning of each episode.

Table Wiping: A table with a whiteboard surface and some markings is placed in front of a single robot arm, which has a whiteboard eraser mounted on its hand. The robot arm must learn to wipe the whiteboard surface and clean all of the markings. The whiteboard markings are randomized at the beginning of each episode.

Two Arm Lifting: A large pot with two handles is placed on a table top. Two robot arms are placed on the same side of the table or on opposite ends of the table. The two robot arms must each grab a handle and lift the pot together, above a certain height, while keeping the pot level. The pot location is randomized at the beginning of each episode.

Two Arm Peg-In-Hole: Two robot arms are placed either next to each other or opposite each other. One robot arm holds a board with a square hole in the center, and the other robot arm holds a long peg. The two robot arms must coordinate to insert the peg into the hole. The initial arm configurations are randomized at the beginning of each episode.

Two Arm Handover: A hammer is placed on a narrow table. Two robot arms are placed on the same side of the table or on opposite ends of the table. The two robot arms must coordinate so that the arm closer to the hammer picks it up and hands it to the other arm. The hammer location and size is randomized at the beginning of each episode.

3.2 Benchmarking Results

We provide a standardized set of benchmarking experiments as baselines for future experiments. Specifically, we test Soft Actor-Critic (SAC) [4], the state-of-the-art model-free RL algorithm, on a select combination of tasks (all) using a combination of proprioceptive and object-specific observations, robots (**Panda**, **Sawyer**), and controllers (`OSC_POSE`, `JOINT_VELOCITY`). Our experiments were implemented and executed in an extended version of **rlkit**, a popular PyTorch-based RL framework and algorithm library. For ease of replicability, we have released our official experimental results on a **benchmark repository**.

All agents were trained for 500 epochs with 500 steps per episode, and utilize the same standardized algorithm hyperparameters (see our benchmarking repo above for exact parameter values). The agents receive the low-dimensional physical states as input to the policy. These experiments ran on 2 CPUs and 12G VRAM and no GPU, each taking about two days to complete. We normalize the per-step rewards to 1.0 such that the maximum possible per-episode return is 500. In Figure 4, we show the per-task experiments conducted, with each experiment's training curve showing the evaluation return mean's average and standard deviation over five random seeds.

We select two of the easiest environments, **Block Lifting** and **Door Opening**, for an ablation study between the operational space controllers (`OSC_POSE`) and the joint velocity controllers (`JOINT_VELOCITY`). We observe that the choice of controllers alone has an evident impact on the efficiency of learning. Both robots learn to solve the tasks faster with the operational space controllers, which we hypothesize is credited to the accelerated exploration in task space;

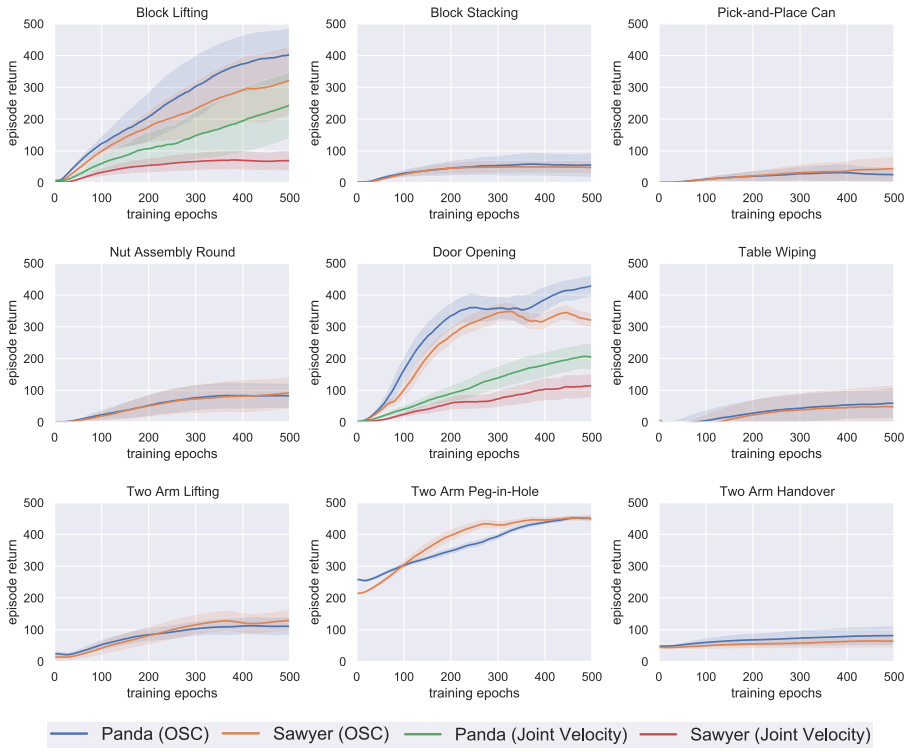


Figure 4: Benchmarking results on the nine standardized environments in `robosuite`. For the Two Arm tasks, we use two Panda arms for Panda (OSC) and two Sawyer arms for Sawyer (OSC)

this highlights the potential of this impedance-based controller to improve task performance on robotic tasks that were previously limited by their action space parameterization. The SAC algorithm is able to solve three of the nine environments, including **Block Lifting**, **Door Opening**, and **Two Arm Peg-in-Hole**, while making slow progress in the other environments, which requires intelligent exploration in longer task horizons. For future experiments, we recommend using the nine environments with the Panda robot and the operational space controller, i.e., the blue curves of Panda (OSC) in Figure 4, for standardized and fair comparisons.

4 Conclusion

`robosuite` provides a simulation framework and benchmark of environments for research and development of robot learning solutions. It includes a suite of standardized tasks for rigorous evaluation and reproducible research. This framework is built on top of the **MuJoCo** physics engine. Since its debut in 2017, robosuite has been used by the AI and robotics research community in a variety of topics, including reinforcement learning [3, 11], imitation learning [10], sim-to-real transfer [11], etc. With the presented v1.5 version, we hope to facilitate more diverse research and reproducible and benchmarkable advances. We invite the community to benchmark their solutions in the provided tasks, and to contribute to robosuite for future releases.

Acknowledgement

We would like to thank members of the Stanford People, AI, & Robots (PAIR) Group for their support and feedback to `robosuite`. In particular, the following people have made their contributions in different stages of this project: Jiren Zhu, Joan Creus-Costa, Jim Fan, Zihua Liu, Orien Zeng, Anchit Gupta, Michelle Lee, Rachel Gardner, Danfei Xu, Andrew Kondrich, Jonathan Booher, Albert Tung, Zhenyu Jiang, Yuqi Xie, You Liang Tan. We wholeheartedly welcome the community to contribute to `robosuite` through issues and pull requests. New contributors will be listed on our project website: **robosuite.ai**

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[3] Linxi Fan*, Yuke Zhu*, Jiren Zhu, Zihua Liu, Orien Zeng, Anchit Gupta, Joan Creus-Costa, Silvio Savarese, and Li Fei-Fei. Surreal: Open-source

reinforcement learning framework and robot manipulation benchmark. In *Conference on Robot Learning*, 2018.

[4] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[5] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *AAAI*, 2018.

[6] Oussama Khatib. Inertial properties in robotic manipulation: An object-level framework. *The international journal of robotics research*, 14(1):19–36, 1995.

[7] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[8] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An interactive 3d environment for visual AI. *arXiv preprint arXiv:1712.05474*, 2017.

[9] Oliver Kroemer, Scott Niekum, and George Konidaris. A review of robot learning for manipulation: Challenges, representations, and algorithms. *arXiv preprint arXiv:1907.03146*, 2019.

[10] Ajay Mandlekar, Yuke Zhu, Animesh Garg, Jonathan Booher, Max Spero, Albert Tung, Julian Gao, John Emmons, Anchit Gupta, Emre Orbay, et al. Roboturk: A crowdsourcing platform for robotic skill learning through imitation. In *Conference on Robot Learning*, pages 879–893, 2018.

[11] Roberto Martín-Martín, Michelle A Lee, Rachel Gardner, Silvio Savarese, Jeannette Bohg, and Animesh Garg. Variable impedance control in end-effector space: An action space for reinforcement learning in contact-rich tasks. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1010–1017. IEEE, 2019.

[12] Harish Ravichandar, Athanasios S Polydoros, Sonia Chernova, and Aude Billard. Recent advances in robot learning from demonstration. *Annual Review of Control, Robotics, and Autonomous Systems*, 3, 2020.

[13] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[14] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. dm_control: Software and tasks for continuous control. *arXiv preprint arXiv:2006.12983*, 2020.

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

[16] Fei Xia, William B Shen, Chengshu Li, Priya Kasimbeg, Micael Edmond Tchapmi, Alexander Toshev, Roberto Martín-Martín, and Silvio Savarese. Interactive gibson benchmark: A benchmark for interactive navigation in cluttered environments. *IEEE Robotics and Automation Letters*, 5(2):713–720, 2020.

[17] Kevin Zakka. Mink: Python inverse kinematics based on MuJoCo, July 2024.

[18] Yuke Zhu, Ziyu Wang, Josh Merel, Andrei Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, et al. Reinforcement and imitation learning for diverse visuomotor skills. *Robotics: Science and Systems*, 2018.