

MuJoCo XLA (MJX)


Starting with version 3.0.0, MuJoCo includes MuJoCo XLA (MJX) under the [mjax](#) directory. MJX allows MuJoCo to run on compute hardware supported by the [XLA](#) compiler via the [JAX](#) framework. MJX runs on a [all platforms supported by JAX](#): Nvidia and AMD GPUs, Apple Silicon, and [Google Cloud TPUs](#).

The MJX API is consistent with the main simulation functions in the MuJoCo API, although it is currently missing some features. While the [API documentation](#) is applicable to both libraries, we indicate features unsupported by MJX in the [notes](#) below.

MJX is distributed as a separate package called `mujoco-mjx` on [PyPI](#). Although it depends on the main `mujoco` package for model compilation and visualization, it is a re-implementation of MuJoCo that uses the same algorithms as the MuJoCo implementation. However, in order to properly leverage JAX, MJX deliberately diverges from the MuJoCo API in a few places, see below.

MJX is a successor to the [generalized physics pipeline](#) in Google's [Brax](#) physics and reinforcement learning library. MJX was built by core contributors to both MuJoCo and Brax, who will together continue to support both Brax (for its reinforcement learning algorithms and included environments) and MJX (for its physics algorithms). A future version of Brax will depend on the `mujoco-mjx` package, and Brax's existing [generalized pipeline](#) will be deprecated. This change will be largely transparent to users of Brax.

Tutorial notebook

The following IPython notebook demonstrates the use of MJX along with reinforcement learning to train humanoid and quadruped robots to locomote:  [Open in Colab](#).

Installation

The recommended way to install this package is via [PyPI](#):

```
pip install mujoco-mjx
```

 [stable](#) ▼

A copy of the MuJoCo library is provided as part of this package's dependencies and does **not** need to be downloaded or installed separately.

Basic usage

Once installed, the package can be imported via `from mujoco import mjx`. Structs, functions, and enums are available directly from the top-level `mjx` module.

Structs

Before running MJX functions on an accelerator device, structs must be copied onto the device via the `mjx.put_model` and `mjx.put_data` functions. Placing an `mjModel` on device yields an `mjx.Model`. Placing an `mjData` on device yields an `mjx.Data`:

```
model = mujoco.MjModel.from_xml_string("...")
data = mujoco.MjData(model)
mjx_model = mjx.put_model(model)
mjx_data = mjx.put_data(model, data)
```

These MJX variants mirror their MuJoCo counterparts but have a few key differences:

1. `mjx.Model` and `mjx.Data` contain JAX arrays that are copied onto device.
2. Some fields are missing from `mjx.Model` and `mjx.Data` for features that are [unsupported](#) in MJX.
3. JAX arrays in `mjx.Model` and `mjx.Data` support adding batch dimensions. Batch dimensions are a natural way to express domain randomization (in the case of `mjx.Model`) or high-throughput simulation for reinforcement learning (in the case of `mjx.Data`).
4. Numpy arrays in `mjx.Model` and `mjx.Data` are structural fields that control the output of JIT compilation. Modifying these arrays will force JAX to recompile MJX functions. As an example, `jnt_limited` is a numpy array passed by reference from `mjModel`, which determines if joint limit constraints should be applied. If `jnt_limited` is modified, JAX will re-compile MJX functions. On the other hand, `jnt_range` is a JAX array that can be modified at runtime, and will only apply to joints with limits as specified by the `jnt_limited` field.

Neither `mjx.Model` nor `mjx.Data` are meant to be constructed manually. An `mjx.Data` may be created by calling `mjx.make_data`, which mirrors the `mj_makeData` function in MuJoCo:

```
model = mujoco.MjModel.from_xml_string("...")
mjx_model = mjx.put_model(model)
```

 [stable](#) ▼

```
mjx_data = mjx.make_data(model)
```

Using `mjx.make_data` may be preferable when constructing batched `mjx.Data` structures inside of a `vmap`.

Functions

MuJoCo functions are exposed as MJX functions of the same name, but following [PEP 8](#)-compliant names. Most of the [main simulation](#) and some of the [sub-components](#) for forward simulation are available from the top-level `mjx` module.

MJX functions are not [JIT compiled](#) by default – we leave it to the user to JIT MJX functions, or JIT their own functions that reference MJX functions. See the [minimal example](#) below.

Enums and constants

MJX enums are available as `mjx.EnumType.ENUM_VALUE`, for example `mjx.JointType.FREE`. Enums for unsupported MJX features are omitted from the MJX enum declaration. MJX declares no constants but references MuJoCo constants directly.

Minimal example

```
# Throw a ball at 100 different velocities.

import jax
import mujoco
from mujoco import mjx

XML=r"""
<mujoco>
  <worldbody>
    <body>
      <freejoint/>
      <geom size=".15" mass="1" type="sphere"/>
    </body>
  </worldbody>
</mujoco>
"""

model = mujoco.MjModel.from_xml_string(XML)
mjx_model = mjx.put_model(model)

@jax.vmap
def batched_step(vel):
    mjx_data = mjx.make_data(mjx_model)
    qvel = mjx_data.qvel.at[0].set(vel)
```

 [stable](#) ▼

```
mjx_data = mjx_data.replace(qvel=qvel)
pos = mjx.step(mjx_model, mjx_data).qpos[0]
return pos

vel = jax.numpy.arange(0.0, 1.0, 0.01)
pos = jax.jit(batched_step)(vel)
print(pos)
```

Helpful Command Line Scripts

We provide two command line scripts with the `mujoco-mjx` package:

```
mjx-testspeed --mjcf=/PATH/TO/MJCF/ --base_path=.
```

This command takes in a path to an MJCF file along with optional arguments (use `--help` for more information) and computes helpful metrics for performance tuning. The command will output, among other things, the total simulation time, the total steps per second and the total realtime factor (here total is across all available devices).

```
mjx-viewer --help
```

This command launches the MJX model in the simulate viewer, allowing you to visualize and interact with the model. Note this steps the simulation using MJX physics (not C MuJoCo) so it can be helpful for example for debugging solver parameters.

Feature Parity


MJX supports most of the main simulation features of MuJoCo, with a few exceptions. MJX will raise an exception if asked to copy to device an `mjModel` with field values referencing unsupported features.

The following features are **fully supported** in MJX:

| Category | Feature |
|-----------------------------------|--|
| Dynamics | Forward |
| Joint | <code>FREE</code> , <code>BALL</code> , <code>SLIDE</code> , <code>HINGE</code> |
| Transmission | <code>JOINT</code> , <code>JOINTINPARENT</code> , <code>SITE</code> , <code>TENDON</code> |
| Actuator Dynamics | <code>NONE</code> , <code>INTEGRATOR</code> , <code>FILTER</code> , <code>FILTEREXACT</code> , <code>MUSCLE</code> |
| Actuator Gain | <code>FIXED</code> , <code>AFFINE</code> , <code>MUSCLE</code> |
| Actuator Bias | <code>NONE</code> , <code>AFFINE</code> , <code>MUSCLE</code> |

| Category | Feature |
|---------------------------------|--|
| Tendon Wrapping | JOINT, SITE, PULLEY, SPHERE, CYLINDER |
| Geom | PLANE, HFIELD, SPHERE, CAPSULE, BOX, MESH are fully implemented. ELLIPSOID and CYLINDER are implemented but only collide with other primitives, note that BOX is implemented as a mesh. |
| Constraint | EQUALITY, LIMIT_JOINT, CONTACT_FRICTIONLESS, CONTACT_PYRAMIDAL, CONTACT_ELLIPTIC, FRICTION_DOF, FRICTION_TENDON |
| Equality | CONNECT, WELD, JOINT, TENDON |
| Integrator | EULER, RK4, IMPLICITFAST (IMPLICITFAST not supported with fluid drag) |
| Cone | PYRAMIDAL, ELLIPTIC |
| Condim | 1, 3, 4, 6 (1 is not supported with ELLIPTIC) |
| Solver | CG, NEWTON |
| Dynamics | Inverse |
| Fluid Model | Inertia model |
| Tendons | Fixed , Spatial |
| Sensors | MAGNETOMETER, CAMPROJECTION, RANGEFINDER, JOINTPOS, TENDONPOS, ACTUATORPOS, BALLQUAT, FRAMEPOS, FRAMEXAXIS, FRAMEYAXIS, FRAMEZAXIS, FRAMEQUAT, SUBTREECOM, CLOCK, VELOCIMETER, GYRO, JOINTVEL, TENDONVEL, ACTUATORVEL, BALLANGVEL, FRAMELINVEL, FRAMEANGVEL, SUBTREELINVEL, SUBTREEANGMOM, TOUCH, ACCELEROMETER, FORCE, TORQUE, ACTUATORFRC, JOINTACTFRC, TENDONACTFRC, FRAMELINACC, FRAMEANGACC (ACCELEROMETER, FORCE, TORQUE not supported with connect or weld equality constraints) |

The following features are **in development** and coming soon:

| Category | Feature |  stable ▼ |
|----------------------|---|--|
| Geom | SDF. Collisions between (SPHERE, BOX, MESH, HFIELD) and CYLINDER. Collisions between (BOX, MESH, | |

| Category | Feature |
|----------------------------|--|
| | <code>HFIELD</code>) and <code>ELLIPSOID</code> . |
| Integrator | <code>IMPLICIT</code> |
| Fluid Model | Ellipsoid model |
| Sensors | All except <code>PLUGIN</code> , <code>USER</code> |
| Lights | Positions and directions of lights |

The following features are **unsupported**:

| Category | Feature |
|--|--|
| margin and gap | Unimplemented for collisions with <code>Mesh</code> Geom . |
| Transmission | <code>SLIDERCRANK</code> , <code>BODY</code> |
| Actuator Dynamics | <code>USER</code> |
| Actuator Gain | <code>USER</code> |
| Actuator Bias | <code>USER</code> |
| Solver | <code>PGS</code> |
| Sensors | <code>PLUGIN</code> , <code>USER</code> |

MJX – The Sharp Bits

GPUs and TPUs have unique performance tradeoffs that MJX is subject to. MJX specializes in simulating big batches of parallel identical physics scenes using algorithms that can be efficiently vectorized on [SIMD hardware](#). This specialization is useful for machine learning workloads such as [reinforcement learning](#) that require massive data throughput.

There are certain workflows that MJX is ill-suited for:

Single scene simulation

Simulating a single scene (1 instance of [mjData](#)), MJX can be **10x** slower than MuJoCo, which has been carefully optimized for CPU. MJX works best when simulating thousands or tens of thousands of scenes in parallel.

 [stable](#) ▼

Collisions between large meshes

MJX supports collisions between convex mesh geometries. However the convex collision algorithms in MJX are implemented differently than in MuJoCo. MJX uses a branchless version of the [Separating Axis Test](#) (SAT) to determine if geometries are colliding with convex meshes, while MuJoCo uses either MPR or GJK/EPA, see [Collision Detection](#) for more details. SAT works well for smaller meshes but suffers in both runtime and memory for larger meshes.

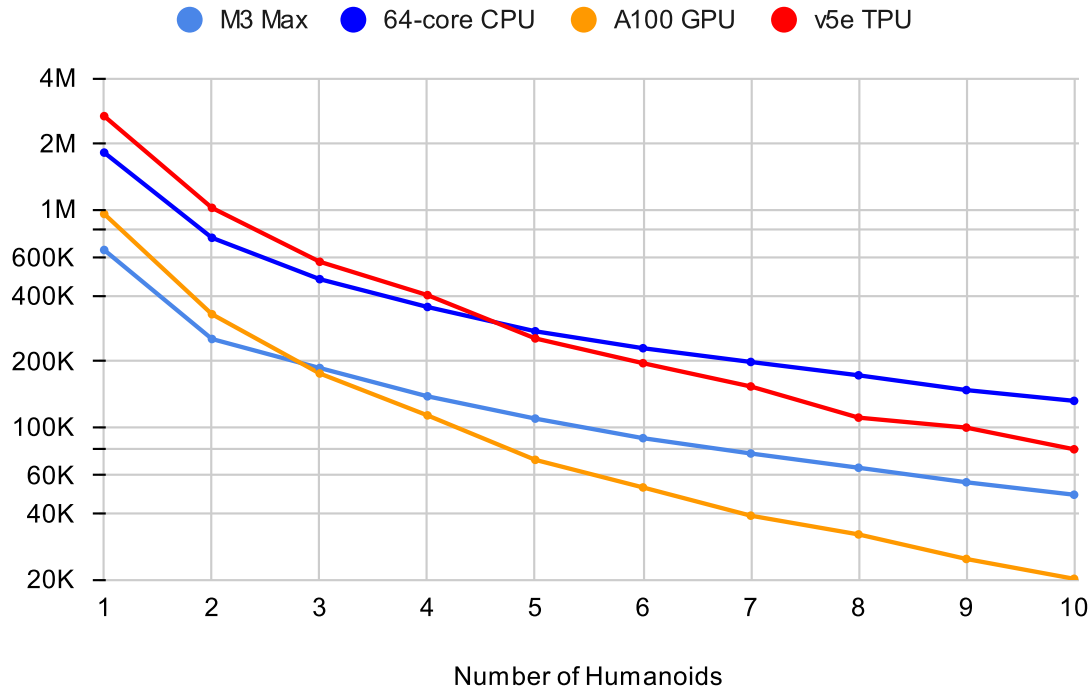
For collisions with convex meshes and primitives, the convex decomposition of the mesh should have roughly **200 vertices or less** for reasonable performance. For convex-convex collisions, the convex mesh should have roughly **fewer than 32 vertices**. We recommend using [maxhullvert](#) in the MuJoCo compiler to achieve desired convex mesh properties. With careful tuning, MJX can simulate scenes with mesh collisions – see the MJX [shadow hand](#) config for an example. Speeding up mesh collision detection is an active area of development for MJX.

Large, complex scenes with many contacts

Accelerators exhibit poor performance for [branching code](#). Branching is used in broad-phase collision detection, when identifying potential collisions between large numbers of bodies in a scene. MJX ships with a simple branchless broad-phase algorithm (see performance tuning) but it is not as powerful as the one in MuJoCo.

To see how this affects simulation, let us consider a physics scene with increasing numbers of humanoid bodies, varied from 1 to 10. We simulate this scene using CPU MuJoCo on an Apple M3 Max and a 64-core AMD 3995WX and time it using [testspeed](#), using `2 x numcore` threads. We time the MJX simulation on an Nvidia A100 GPU using a batch size of 8192 and an 8-chip [v5 TPU](#) machine using a batch size of 16384. Note the vertical scale is logarithmic.

Steps per second: compare CPU, GPU, TPU



The values for a single humanoid (leftmost datapoints) for the four timed architectures are **650K**, **1.8M**, **950K** and **2.7M** steps per second, respectively. Note that as we increase the number of humanoids (which increases the number of potential contacts in a scene), MJX throughput decreases more rapidly than MuJoCo.

Performance tuning

For MJX to perform well, some configuration parameters should be adjusted from their default MuJoCo values:

[option/iterations](#) and [option/ls_iterations](#)

The [iterations](#) and [ls_iterations](#) attributes—which control solver and linesearch iterations, respectively—should be brought down to just low enough that the simulation remains stable. Accurate solver forces are not so important in reinforcement learning in which domain randomization is often used to add noise to physics for sim-to-real. The `NEWTON` [Solver](#) delivers excellent convergence with very few (often just one) solver iterations, and performs well on GPU. `CG` is currently a better choice for TPU.

☐ [stable](#) ▼

[contact/pair](#)

Consider explicitly marking geoms for collision detection to reduce the number of contacts that MJX must consider during each step. Enabling only an explicit list of valid contacts can have a dramatic effect on simulation performance in MJX. Doing this well often requires an understanding of the task – for example, the [OpenAI Gym Humanoid](#) task resets when the humanoid starts to fall, so full contact with the floor is not needed.

maxhullvert

Set `maxhullvert` to 64 or less for better convex mesh collision performance.

option/flag/eulerdamp

Disabling `eulerdamp` can help performance and is often not needed for stability. Read the [Numerical Integration](#) section for details regarding the semantics of this flag.

option/jacobian

Explicitly setting “dense” or “sparse” may speed up simulation depending on your device. Modern TPUs have specialized hardware for rapidly operating over sparse matrices, whereas GPUs tend to be faster with dense matrices as long as they fit onto the device. As such, the behavior in MJX for the default “auto” setting is sparse if `nv >= 60` (60 or more degrees of freedom), or if MJX detects a TPU as the default backend, otherwise “dense”. For TPU, using “sparse” with the Newton solver can speed up simulation by 2x to 3x. For GPU, choosing “dense” may impart a more modest speedup of 10% to 20%, as long as the dense matrices can fit on the device.

Broadphase



While MuJoCo handles broadphase culling out of the box, MJX requires additional parameters. For an approximate version of broadphase, use the experimental custom numeric parameters `max_contact_points` and `max_geom_pairs`.

`max_contact_points` caps the number of contact points sent to the solver for each condim type. `max_geom_pairs` caps the total number of geom-pairs sent to respective collision functions for each geom-type pair. As an example, the [shadow hand](#) environment makes use of these parameters.

GPU performance

The following environment variables should be set:

```
XLA_FLAGS=--xla_gpu_triton_gemm_any=true
```

This enables the Triton-based GEMM (matmul) emitter for any GEMM  **stable**  supports. This can yield a 30% speedup on NVIDIA GPUs. If you have multiple

GPUs, you may also benefit from enabling flags related to [communication between GPUs](#).

Copyright © DeepMind Technologies Limited
Made with [Sphinx](#) and @pradyunsg's [Furo](#)