Python Bindings

Starting with version 2.1.2, MuJoCo comes with native Python bindings that are developed in C++ using pybind11. Unlike previous Python bindings, these are officially supported by the MuJoCo development team and will be kept up-to-date with the latest developments in MuJoCo itself.

The Python bindings are distributed as the `mujoco` package on PyPI. These are low-level bindings that are meant to give as close to a direct access to the MuJoCo library as possible. However, in order to provide an API and semantics that developers would expect in a typical Python library, the bindings deliberately diverge from the raw MuJoCo API in a number of places, which are documented throughout this page.

DeepMind's dm_control reinforcement learning library (which prior to version 1.0.0 implemented its own MuJoCo bindings based on `ctypes`) has been updated to depend on the `mujoco` package and continues to be supported by DeepMind. Changes in dm_control should be largely transparent to users of previous versions, however code that depended directly on its low-level API may need to be updated. Consult the migration guide for detail.

For mujoco-py users, we include notes below to aid migration.

Tutorial notebook

A MuJoCo tutorial using the Python bindings is available here: Open in Colab

Installation

The recommended way to install this package is via PyPI:

```
pip install mujoco
```

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

Building from source

> **Note**
>
> Building from source is only necessary if you are modifying the Python bindings (or are trying to run on exceptionally old Linux systems). If that's not the case, then we recommend installing the prebuilt binaries from PyPI.

1. Make sure you have CMake and a C++17 compiler installed.
2. Download the latest binary release from GitHub. On macOS, the download corresponds to a DMG file from which you can drag `MuJoCo.app` into your `/Applications` folder.
3. Clone the entire `mujoco` repository from GitHub and `cd` into the python directory:
   ```
   git clone https://github.com/deepmind/mujoco.git
   cd mujoco/python
   ```
4. Create a virtual environment:
   ```
   python3 -m venv /tmp/mujoco
   source /tmp/mujoco/bin/activate
   ```
5. Generate a source distribution tarball with the `make_sdist.sh` script.
   ```
   cd python
   bash make_sdist.sh
   ```
   The `make_sdist.sh` script generates additional C++ header files that are needed to build the bindings, and also pulls in required files from elsewhere in the repository outside the `python` directory into the sdist. Upon completion, the script will create a `dist` directory with a `mujoco-x.y.z.tar.gz` file (where `x.y.z` is the version number).
6. Use the generated source distribution to build and install the bindings. You'll need to specify the path to the MuJoCo library you downloaded earlier in the `MUJOCO_PATH` environment variable.

   > **Note**
   >
   > For macOS, this can be the path to a directory that contains the `mujoco.framework`. In particular, you can set `MUJOCO_PATH=/Applications/MuJoCo.app` if you installed MuJoCo as suggested in step 1.

   ```
   cd dist
   MUJOCO_PATH=/PATH/TO/MUJOCO pip install mujoco-x.y.z.tar.gz
   ```

The Python bindings should now be installed! To check that they've been successfully installed, `cd` outside of the `mujoco` directory and run `python -c "import mujoco"`.

Interactive viewer

An interactive GUI viewer is available as part of the Python package. (This is the same viewer as the `simulate` application that ships with the MuJoCo binary releases.)

Three distinct use cases are supported:

1. Launching as a standalone application:
   - `python -m mujoco.viewer` launches an empty visualization session, where a model can be loaded by drag-and-drop.
   - `python -m mujoco.viewer --mjcf=/path/to/some/mjcf.xml` launches a visualization session for the specified model file.

2. Launching from a Python program/script – import the module via `from mujoco import viewer` and launch the GUI using one of the following invocations:
   - `viewer.launch()` launches an empty visualization session, where a model can be loaded by drag-and-drop.
   - `viewer.launch(model)` launches a visualzation session for the given `mjModel` where the visualizer internally creates its own instance of `mjData`
   - `viewer.launch(model, data)` is the same as above, except that the visualizer operates directly on the given `mjData` instance – upon exit the `data` object will have been modified.
3. Launching from an interactive Python session (aka REPL): when working interactively either in a `python` or `ipython` shell, the visualizer can be launched in a "passive" mode via `viewer.launch_repl(model, data)`, where the user remains in full control of modifying or stepping the physics. In this mode, the user can interact with the visualizer using the mouse and keyboard as usual, however the physics will be frozen unless the user explicitly calls `mj_step` (or perform any other modification of the `mjData` or `mjModel`) in the REPL terminal. Note that since the visualizer does not modify `mjData` in this mode, mouse-drag perturbations will not work unless the user explicitly handles incoming GUI perturbation events in the REPL session.

Basic usage

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

Structs

MuJoCo data structures are exposed as Python classes. In order to conform to PEP 8 naming guidelines, struct names begin with a capital letter, for example `mjData` becomes `mujoco.MjData` in Python.

All structs other than `mjModel` have constructors in Python. For structs that have an `mj_defaultFoo`-style initialization function, the Python constructor calls the default initializer automatically, so for example `mujoco.MjOption()` creates a new `mjOption` instance that is pre-initialized with mj_defaultOption. Otherwise, the Python constructor zero-initializes the underlying C struct.

Structs with a `mj_makeFoo`-style initialization function have corresponding constructor overloads in Python, for example `mujoco.MjvScene(model, maxgeom=10)` in Python creates a new `mjvScene` instance that is initialized with `mjv_makeScene(model, [the new mjvScene instance], 10)` in C. When this form of initialization is used, the corresponding deallocation function `mj_freeFoo/mj_deleteFoo` is automatically called when the Python object is deleted. The user does not need to manually free resources.

The `mujoco.MjModel` class does not a have Python constructor. Instead, we provide three static factory functions that create a new `mjModel` instance: `mujoco.MjModel.from_xml_string`, `mujoco.MjModel.from_xml_path`, and `mujoco.MjModel.from_binary_path`. The first function accepts a model XML as a string, while the latter two functions accept the path to either an XML or MJB model file. All three functions optionally accept a Python dictionary which is converted into a MuJoCo Virtual file system for use during model compilation.

Functions

MuJoCo functions are exposed as Python functions of the same name. Unlike with structs, we do not attempt to make the function names PEP 8-compliant, as MuJoCo uses both underscores and CamelCases. In most cases, function arguments appear exactly as they do in C, and keyword arguments are supported with the same names as declared in mujoco.h. Python bindings to C functions that accept array input arguments expect NumPy arrays or iterable objects that are convertible to NumPy arrays (e.g. lists). Output arguments (i.e. array arguments that MuJoCo expect to write values back to the caller) must always be writeable NumPy arrays.

In the C API, functions that take dynamically-sized arrays as inputs expect a pointer argument to the array along with an integer argument that specifies the array's size. In Python, the size arguments are omitted since we can automatically (and indeed, more safely) deduce it from the NumPy array. When calling these functions, pass all arguments other than array sizes in the same order as they appear in mujoco.h, or use keyword arguments. For example, mj_jac should be called as `mujoco.mj_jac(m, d, jacp, jacr, point, body)` in Python.

The bindings **releases the Python Global Interpreter Lock (GIL)** before calling the underlying MuJoCo function. This allows for some thread-based parallelism, however users should bear in mind that the GIL is only released for the duration of the MuJoCo C function itself, and not during the execution of any other Python code.

> **Note**
>
> One place where the bindings do offer added functionality is the top-level mj_step function. Since it is often called in a loop, we have added an additional `nstep` argument, indicating how many times the underlying mj_step should be called. If not specified, `nstep` takes the default value of 1. The following two code snippets perform the same computation, but the first one does so without acquiring the GIL in between subsequent physics steps:
>
> ```
> mj_step(model, data, nstep=20)
> ```
>
> ```
> for _ in range(20):
>   mj_step(model, data)
> ```

Enums and constants

MuJoCo enums are available as `mujoco.mjtEnumType.ENUM_VALUE`, for example `mujoco.mjtObj.mjOBJ_SITE`. MuJoCo constants are available with the same name directly under the `mujoco` module, for example `mujoco.mjVISSTRING`.

Minimal example

```
import mujoco

XML=r"""
```

```
<mujoco>
  <asset>
    <mesh file="gizmo.stl"/>
  </asset>
  <worldbody>
    <body>
      <freejoint/>
      <geom type="mesh" name="gizmo" mesh="gizmo"/>
    </body>
  </worldbody>
</mujoco>
"""

ASSETS=dict()
with open('/path/to/gizmo.stl', 'rb') as f:
  ASSETS['gizmo.stl'] = f.read()

model = mujoco.MjModel.from_xml_string(XML, ASSETS)
data = mujoco.MjData(model)
while data.time < 1:
  mujoco.mj_step(model, data)
  print(data.geom_xpos)
```

Named access

Most well-designed MuJoCo models assign names to objects (joints, geoms, bodies, etc.) of interest. When the model is compiled down to an `mjModel` instance, these names become associated with numeric IDs that are used to index into the various array members. For convenience and code readability, the Python bindings provide "named access" API on `MjModel` and `MjData`. Each `name_fooadr` field in the `mjModel` struct defines a name category `foo`.

For each name category `foo`, `mujoco.MjModel` and `mujoco.MjData` objects provide a method `foo` that takes a single string argument, and returns an accessor object for all arrays corresponding to the entity `foo` of the given name. The accessor object contains attributes whose names correspond to the fields of either `mujoco.MjModel` or `mujoco.MjData` but with the part before the underscore removed. In addition, accessor objects also provide `id` and `name` properties, which can be used as replacements for `mj_name2id` and `mj_id2name` respectively. For example:

- `m.geom('gizmo')` returns an accessor for arrays in the `MjModel` object `m` associated with the geom named "gizmo".
- `m.geom('gizmo').rgba` is a NumPy array view of length 4 that specifies the RGBA color for the geom. Specifically, it corresponds to the portion of `m.geom_rgba[4*i:4*i+4]` where `i = mujoco.mj_name2id(m, mujoco.mjtObj.mjOBJ_GEOM, 'gizmo')`.
- `m.geom('gizmo').id` is the same number as returned by `mujoco.mj_name2id(m, mujoco.mjtObj.mjOBJ_GEOM, 'gizmo')`.
- `m.geom(i).name` is `'gizmo'`, where `i = mujoco.mj_name2id(m, mujoco.mjtObj.mjOBJ_GEOM, 'gizmo')`.

Additionally, the Python API define a number of aliases for some name categories corresponding to the XML element name in the MJCF schema that defines an entity of that category. For example, `m.joint('foo')` is the same as `m.jnt('foo')`. A complete list of these aliases are provided below.

The accessor for joints is somewhat different that of the other categories. Some `mjModel` and `mjData` fields (those of size size `nq` or `nv`) are associated with degrees of freedom (DoFs) rather than joints. This is because different types of joints have different numbers of DoFs. We nevertheless associate these fields to their corresponding joints, for example through `d.joint('foo').qpos` and `d.joint('foo').qvel`, however the size of these arrays would differ between accessors depending on the joint's type.

Named access is guaranteed to be O(1) in the number of entities in the model. In other words, the time it takes to access an entity by name does not grow with the number of names or entities in the model. (This is currently **not** the case for the mj_name2id function, which performs a linear scan.)

For completeness, we provide here a complete list of all name categories in MuJoCo, along with their corresponding aliases defined in the Python API.

- `body`
- `jnt` or `joint`
- `geom`
- `site`
- `cam` or `camera`
- `light`
- `mesh`
- `skin`
- `hfield`
- `tex` or `texture`
- `mat` or `material`
- `pair`
- `exclude`
- `eq` or `equality`
- `tendon` or `ten`
- `actuator`
- `sensor`
- `numeric`
- `text`
- `tuple`
- `key` or `keyframe`

Rendering

MuJoCo itself expects users to set up a working OpenGL context before calling any of its `mjr_` rendering routine. The Python bindings provide a basic class `mujoco.GLContext` that helps users set up such a context for offscreen rendering. To create a context, call `ctx = mujoco.GLContext(max_width, max_height)`. Once the context is created, it must be made current before MuJoCo rendering functions can be called, which you can do so via `ctx.make_current()`. Note that a context can only be made current on one thread at any given time, and all subsequent rendering calls must be made on the same thread.

The context is freed automatically when the `ctx` object is deleted, but in some multi-threaded scenario it may be necessary to explicitly free the underlying OpenGL context. To do so, call `ctx.free()`, after which point it is the user's responsibility to ensure that no further rendering calls are made on the context.

Once the context is created, users can follow MuJoCo's standard rendering, for example as documented in the Visualization section.

Error handling

MuJoCo reports irrecoverable errors via the mju_error mechanism, which immediately terminates the entire process. Users are permitted to install a custom error handler via the mju_user_error callback, but it too is expected to terminate the process, otherwise the behavior of MuJoCo after the callback returns is undefined. In actuality, it is sufficient to ensure that error callbacks do not return *to MuJoCo*, but it is permitted to use longjmp to skip MuJoCo's call stack back to the external callsite.

The Python bindings utilizes longjmp to allow it to convert irrecoverable MuJoCo errors into Python exceptions of type `mujoco.FatalError` that can be caught and processed in the usual Pythonic way. Furthermore, it installs its error callback in a thread-local manner using a currently private API, thus allowing for concurrent calls into MuJoCo from multiple threads.

Callbacks

MuJoCo allows users to install custom callback functions to modify certain parts of its computation pipeline. For example, mjcb_sensor can be used to implement custom sensors, and mjcb_control can be used to implement custom actuators. Callbacks are exposed through the function pointers prefixed `mjcb_` in mujoco.h.

For each callback `mjcb_foo`, users can set it to a Python callable via `mujoco.set_mjcb_foo(some_callable)`. To reset it, call `mujoco.set_mjcb_foo(None)`. To retrieve the currently installed callback, call `mujoco.get_mjcb_foo()`. (The getter **should not** be used if the callback is not installed via the Python bindings.) The bindings automatically acquire the GIL each time the callback is entered, and release it before reentering MuJoCo. This is likely to incur a severe performance impact as callbacks are triggered several times throughout MuJoCo's computation pipeline and is unlikely to be suitable for "production" use case. However, it is expected that this feature will be useful for prototyping complex models.

Alternatively, if a callback is implemented in a native dynamic library, users can use ctypes to obtain a Python handle to the C function pointer and pass it to `mujoco.set_mjcb_foo`. The bindings will then retrieve the underlying function pointer and assign it directly to the raw callback pointer, and the GIL will **not** be acquired each time the callback is entered.

Code Sample: open-loop rollout

We include a code sample showing how to add additional C/C++ functionality, exposed as a Python module via pybind11. The sample, implemented in `rollout.cc` and wrapped in `rollout.py`, implements a common use case where tight loops implemented outside of Python are beneficial: rolling out a trajectory (i.e., calling `mj_step()` in a loop), given an intial state and sequence of controls, and returning subsequent states and sensor values. The canonical usage form is

```
state, sensordata = rollout.rollout(model, data, initial_state, ctrl)
```

`initial_state` is a `nstate x nqva` array, with `nstate` initial states of length `nqva`, where `nqva = model.nq + model.nv + model.na` is the size of the full MuJoCo mechanical state: positions (`data.qpos`), velocities (`data.qvel`) and actuator activations (`data.act`). `ctrl` is a `nstate x nstep x nu` array of control sequences.

The `rollout` function is designed to be completely stateless, so all inputs of the stepping pipeline are set and any values already present in the given `MjData` instance will have no effect on the output. In order to facilitate this, all inputs including `time` and `qacc_warmstart` are set to default values, as are auxillary controls (`qfrc_applied`, `xfrc_applied` and `mocap_{pos,quat}`). These can also be optionally set by the user.

Since the Global Interpreter Lock can be released, this function can be efficiently threaded using Python threads. See the `test_threading` function in `rollout_test.py` for an example of threaded operation.

Migration Notes for mujoco-py

In mujoco-py, the main entry point is the MjSim class. Users construct a stateful `MjSim` instance from an MJCF model (similar to `dm_control.Physics`), and this instance holds references to an `mjModel` instance and its associated `mjData`. In contrast, the MuJoCo Python bindings (`mujoco`) take a more low-level approach, as explained above: following the design principle of the C library, the `mujoco` module itself is stateless, and merely wraps the underlying native structs and functions.

While a complete survey of mujoco-py is beyond the scope of this document, we offer below implementation notes for a non-exhaustive list of specific mujoco-py features:

`mujoco_py.load_model_from_xml(bstring)`

This factory function constructs a stateful `MjSim` instance. When using `mujoco`, the user should call the factory function `mujoco.MjModel.from_xml_*` as described above. The user is then responsible for holding the resulting `MjModel` struct instance and explicitly generating the corresponding `MjData` by calling `mujoco.MjData(model)`.

`sim.reset()`, `sim.forward()`, `sim.step()`

Here as above, `mujoco` users needs to call the underlying library functions, passing instances of `MjModel` and `MjData`: mujoco.mj_resetData(model, data), mujoco.mj_forward(model, data), and mujoco.mj_step(model, data).

`sim.get_state()`, `sim.set_state(state)`, `sim.get_flattened_state()`, `sim.set_state_from_flattened(state)`

The MuJoCo library's computation is deterministic given a specific input, as explained in the Programming section. mujoco-py implements methods for getting and setting some of the relevant fields (and similarly `dm_control.Physics` offers methods that correspond to the flattened case). `mujoco` do not offer such abstraction, and the user is expected to get/set the values of the relevant fields explicitly.

`sim.model.get_joint_qvel_addr(joint_name)`

This is a convenience method in mujoco-py that returns a list of contiguous indices corresponding to this joint. The list starts from `model.jnt_qposadr[joint_index]`, and its length depends on the joint type. `mujoco` doesn't offer this functionality, but this list can be easily constructed using `model.jnt_qposadr[joint_index]` and `xrange`.

`sim.model.*_name2id(name)`

mujoco-py creates dicts in `MjSim` that allow for efficient lookup of indices for objects of different types: `site_name2id`, `body_name2id` etc. These functions replace the function mujoco.mj_name2id(model, type_enum, name) whose current implementation is inefficient. `mujoco` offers a different approach for using entity names – named access, as well as access to the native mj_name2id.

`sim.save(fstream, format_name)`

This is the one context in which the MuJoCo library (and therefore also `mujoco`) is stateful: it holds a copy in memory of the last XML that was compiled, which is used in mujoco.mj_saveLastXML(fname). Note that mujoco-py's implementation has a convenient extra feature, whereby the pose (as determined by `sim.data`'s state) is transformed to a keyframe that's added to the model before saving. This extra feature is not currently available in `mujoco`.