Extensions

This section describes MuJoCo's mechanisms for user-authored extensions. At present, extensibility is provided by via engine plugins and resource providers.

Engine plugins

Engine plugins, introduced in MuJoCo 2.3.0, allow user-defined logic to be inserted into various parts of MuJoCo's computational pipeline. For example, custom sensor and actuator types can be implemented as plugins. Plugin features are referenced in the XML content of an MJCF model, allowing MJCF to remain an abstract physical description of a system even if the simulation requirements extend beyond MuJoCo's built-in capabilities.

The plugin mechanism was designed to overcome the disadvantages of MuJoCo's physics callbacks. These global callbacks (usage example) are still available and useful for fast prototyping or when the user wishes to implement functionality in Python, but are generally deprecated as a stable mechanism for extended functionality. The central features of the plugin mechanism are:

- **Thread safety:** Plugin instances (see below) are thread-local, avoiding collisions.
- **Statefulness:** Plugins can be stateful, and their state will be (de)serialized correctly.
- **Interoperability:** Different plugins can coexist without interference.

Both users and developers of plugins should familiarize themselves with two key concepts:

Plugin

A **plugin** is a collection of functions and static attributes that implement its capabilities, bundled into an mjpPlugin struct. Plugin functions are **stateless**: they depend only on the arguments passed to them. When a plugin requires an internal state, it declares this state and allows MuJoCo to manage it and pass it in. This enables (de)serialization of the full simulation state. A plugin can therefore be regarded as the "pure logic" part of the functionality and is often library. A plugin is neither a model element nor is it associated wi model elements.

Plugin instance

A plugin **instance** represents the self-contained runtime state that is operated on by the plugin: when the plugin logic is executed, the instance state is passed in by the engine. A plugin instance is itself a model element of type mjOBJ_PLUGIN. There are `mjModel.nplugin` instances with id's in `[0 nplugin-1]`. Like other elements, instances can have names, with mj_name2id and mj_id2name mapping between id's and names. Unlike the plugin code which is loaded once into a global table, multiple instances of the same plugin can be defined and have a one-to-many relationship with other model elements.

one-to-one:

In this simplest case, each instance is referenced once in the model. For example, two sensors may declare that their values are computed by two plugin instances of the same plugin. In this case, every time the sensor output is computed, the plugin logic will be executed separately.

one-to-many:

Alternatively, the behavior of multiple elements can be backed by a single plugin instance. There are two main scenarios where this is useful:

- The values of different element types are linked to the same physical entity and computation. For example consider a motor with an internal thermometer. This would manifest as an actuator and sensor, both associated with the same plugin instance which computes both torque outputs and temperature readings.
- It is advantageous to batch the computation of multiple related elements together, for example where the computed value is the output of a neural network. The canonical example here is a robot that is equipped with `N` motors, where motor dynamics are modeled as a neural network. In this case, it can be substantially faster to produce the torque output of all N actuators in a single forward pass than for each motor separately.

Below, we begin by describing plugins from a user perspective:

- Types of plugin capabilities.
- How plugins are declared and configured in an MJCF model.
- How plugin states are incorporated into mjData, and what users need to do to safely duplicate and serialize mjData structs when plugin instances are present.

Next, we describe the logistics of plugin registration that are relevant t
developers of plugins. This is followed by a section that targets plugin

Plugin capabilities

A plugin is described by the contents of its associated mjpPlugin struct. The `capabilityflags` member is an integer bitfield describing the plugin's capabilities, where bit semantics are defined in the enum mjtPluginCapabilityBit. Using a bitfield allows plugins to support multiple types of computation. The currently supported plugin capabilities are:

- Actuator plugin
- Sensor plugin
- Passive force plugin
- Signed distance field plugin

Additional capabilities will be added in the future as required.

Declaration in MJCF

First, a plugin dependency must be declared through `<extension><plugin>`. When the model is parsed, if any plugin is declared but not registered (see below), a model compilation error is raised. If only a single MJCF element is backed by a plugin, instances can be implicitly created in-place. If multiple elements are backed by the same plugin, instance declaration must be explicit:

```
<mujoco>
  <extension>
    <plugin plugin="mujoco.test.simple_sensor_plugin"/>
    <plugin plugin="mujoco.test.actuator_sensor_plugin">
      <instance name="explicit_instance"/>
    </plugin>
  </extension>
  ...
  <sensor>
    <plugin name="sensor0" plugin="mujoco.test.simple_sensor_plugin"/>
    <plugin name="sensor1" plugin="mujoco.test.simple_sensor_plugin"/>
    <plugin name="sensor2" instance="explicit_instance"/>
  </sensor>
  ...
  <actuator>
    <plugin name="actuator2" instance="explicit_instance"/>
  </actuator>
</mujoco>
```

In the example above, `sensor0` and `sensor1` are each backed by a simple plugin that does not share computation among elements, so an instance is implicitly created for each sensor by directly referencing the plugin identifier. In contrast, `se` `actuator2` are backed by a plugin that shares computation, so they mu shared instance that was explicitly declared.

Configuration in MJCF

Plugins can declare custom attributes that represent specialized configurable parameters. For example, a DC motor model may expose the resistance, inductance, and capacitance as configuration attributes. In MJCF, the values of these attributes can be specified via `<config>` elements, where each `<config>` has a key and a value. Valid keys and values are specified by the plugin developers, but are declared to MuJoCo during plugin registration time so that the MuJoCo model compiler can raise errors for invalid values.

```
<mujoco>
  <extension>
    <plugin plugin="mujoco.test.simple_actuator_plugin">
      <instance name="explicit_instance">
        <config key="resistance" value="1.0"/>
        <config key="inductance" value="2.0"/>
      </instance>
    </plugin>
  </extension>
  ...
  <actuator>
    <plugin name="actuator0" instance="explicit_instance"/>
    <plugin name="actuator1" plugin="mujoco.test.simple_actuator_plugin">
        <config key="resistance" value="3.0"/>
        <config key="inductance" value="4.0"/>
    </plugin>
  </actuator>
</mujoco>
```

In the example above, `actuator0` refers to a pre-existing plugin instance that was created and configured via the `<instance>` element, while `actuator1` is implicitly creating and configuring a new plugin instance in-place. Note that it would be an error to add `<config>` child elements directly to `actuator0` because a new plugin instance is not being created there.

Plugin state

While plugin code should be stateless, individual plugin instances are permitted to hold time-dependent state that is intended to evolve alongside MuJoCo physics, for example temperature variables in thermodynamically coupled actuator models. Separately, it may also be desirable for plugin instances to memoize potentially expensive parts of their operation. For example, sensor or actuator plugins that are backed by pretrained neural networks will want to preload their weight compilation time. It is important for us to distinguish between these two types of per-instance plugin payload. The term **plugin state** refers to the time-dependent state of

the plugin instance that consists of *floating point* values, while the term **plugin data** refers to *arbitrary data structures* consisting of memoized payload that should be considered implementation detail for the plugin's computation.

Crucially, plugin data must be reconstructible only from plugin configuration attributes, the plugin state, and MuJoCo state variables. This means that the plugin data is not expected to be serializable, and will not be serialized by MuJoCo when it copies or stores data. On the other hand, plugin state is considered an integral part of the physics and must be serialized alongside MuJoCo's other state variables in order for the physics to be faithfully restored.

Plugins must declare the number of floating point values required for each instance via the `nstate` callback of its mjpPlugin struct. Note that this number can depend on the exact configuration of the instance. During mj_makeData, MuJoCo allocates the requisite number of slots in the `plugin_state` field of mjData for each plugin instance. The `plugin_stateadr` field in mjModel indicates the position within the overall `plugin_state` array at which each plugin instance can find its state values.

Plugin data, however, is entirely opaque from MuJoCo's point of view. During mj_makeData, MuJoCo calls the `init` callback from the relevant mjpPlugin. In this callback, the plugin is permitted to allocate or otherwise create an arbitrary data structure that it requires to function and stores its pointer in the `plugin_data` field of mjData that is being created. During mj_deleteData, MuJoCo calls the `destroy` callback from the same mjpPlugin, and the plugin is responsible for deallocating its internal resources associated with the instance.

When mjData is being copied via mj_copyData, MuJoCo will copy over the plugin state. However, the plugin code is responsible for setting up the plugin data for the newly copied mjData. To facilitate this, MuJoCo calls the `copy` callback from mjpPlugin for each plugin instance present.

Actuator states

When writing stateful actuator plugins, there are two choices for where to save the actuator state. One option is using `plugin_state` as described above, and the other is to use `mjData.act` by implementing the callback on mjpPlugin.

When using the latter option, the actuator plugin's state will be added to `mjData.act`, and MuJoCo will automatically integrate `mjData.act_dot` values between timesteps. One advantage of this approach is that finite-differencing functions like mjd_transitionFD will work as they do for native actuators. The `mjpPlugin.advance` callback will be called after `act_dot` is integrated, and actuator plugins the `act` values at that point, if the built-in integrator is not appropriate.

Users may specify the dyntype attribute on actuator plugins, to introduce a filter or an integrator between user inputs and actuator states. When they do, the state variable introduced by `dyntype` will be placed *after* the plugin's state variables in the `act` array.

Registration

Plugins must be registered with MuJoCo before they can be referenced in MJCF models.

One-off plugins that are intended to support a specific application (or throwaway plugins that are implemented to help troubleshoot issues with a model) can be statically linked into the application. This can be as simple as preparing an mjpPlugin struct in the `main` function, then passing it to mjp_registerPlugin to be registered with MuJoCo.

Generally, reusable plugins are expected to be packaged as dynamic libraries. A dynamic library containing one or more MuJoCo plugins should make sure that all plugins are registered when the library is loaded. In GCC-compatible compilers, this can be achieved by calling mjp_registerPlugin in a function that is declared with `__attribute__((constructor))`, while in MSVC this can be done in a DLL entry point (canonically known as `DllMain`). MuJoCo provides a convenience macro mjPLUGIN_LIB_INIT that expands to either of these constructs depending on the compiler used.

Users of plugins that are delivered as dynamic libraries as described above can load the library using the function mj_loadPluginLibrary. This is the preferred way to load dynamic libraries containing MuJoCo plugins (rather than, say, calling `dlopen` or `LoadLibraryA` directly) since the exact way in which MuJoCo expects dynamic libraries to auto-register plugins may change over time, but mj_loadPluginLibrary is expected to also evolve to reflect the best practices.

For applications that need to be able to load arbitrary user-provided MJCF models, it may be desirable to automatically scan and load all dynamic libraries found without a specific directory. Users who bring along an MJCF that requires a plugin can then be instructed to place the requisite plugin libraries in the relevant directory. For example, this is what is done in the simulate interactive viewer application. The mj_loadAllPluginLibraries function is provided for this scan-and-load use case.

Writing plugins

This section, targeted at developers, is incomplete. We encourage peo
write their own plugins to contact the MuJoCo development team for l

starting point for experienced developers is the associated tests and the first-party plugins in the first-party plugin directory.

A future version of this section will include:

- The content of the mjpPlugin struct.
- Which functions and properties need to be provided in order to define a plugin.
- How to declare custom MJCF attributes for a plugin.
- Things that developers need to keep in mind in order to ensure that plugins function correctly when mjData is copied, stepped, or reset.

There are several first-party plugin directories:

actuator

The plugins in the actuator/ directory implement custom actuators, so far only a PID controller. See the README for details.

elasticity

The plugins in the elasticity/ directory are passive forces based on continuum mechanics for 1-dimensional and 2-dimensional bodies. The 1D model is invariant under rotations and captures the large deformation of elastic cables, decoupling twisting and bending strains. The 2D model is a suitable for computing the bending stiffness of thin elastic plates (i.e. shells having a flat stress-free configuration). In this case, the elastic energy is quadratic and therefore the stiffness matrix is constant. For more information, please see the README.

sensor

The plugins in the sensor/ directory implement custom sensors. Currently the sole sensor plugin is the touch grid sensor, see the README for details.

sdf

The plugins in the sdf/ directory specify custom shapes in a mesh-free manner, by defining methods computing a signed distance field and its gradient at query points. This shape then acts as a new geom type in the collision table at the top of engine_collision_driver.c. For more information concerning the available SDFs and how to write your own implicit geometry, please see the README. The rest of this section will give more detail concerning the collision algorithm and the plugin engine interface.

Collision points are found by minimizing the function A + B + abs(max(A, B)), where A and B are the two colliding SDFs, via gradient descent. Because SDFs are non-convex, multiple starting points are required in order to converge to multiple lo stable number of starting points is set using sdf_initpoints, and are initialized using the Halton

sequence inside the intersection of the axis-aligned bounding boxes. The number of gradient descent iterations is set using sdf_iterations.

While *exact* SDFs—encoding the precise signed distance to the surface—are preferred, collisions are possible with any function whose value vanishes at the surface and grows monotonically away from it, with a negative sign in the interior. For such functions, it is still possible to find collisions, albeit with a possibly increased number of starting points.

The `sdf_distance` method is called by the compiler to produce a visual mesh for rendering using the marching cubes algorithm implemented by MarchingCubeCpp.

Future improvement to the gradient descent algorithm, such as a line search which takes advantage of the properties of SDFs, might reduce the number of iterations and/or starting points.

For the sdf plugin, the following methods need to be specified

`sdf_distance`:
: Returns the signed distance of the query point given in local coordinates.

`sdf_staticdistance`:
: This is the static version of the previous function, taking config attributes as additional inputs. This function is required because mesh creation occurs during model compilation before the plugin object has been instantiated.

`sdf_gradient`:
: Computes the gradient in local coordinates of the SDF at the query point.

`sdf_aabb`:
: Computes the axis-aligned bounding box in local coordinates. This volume is voxelized uniformly before the call to the marching cubes algorithm.

Resource providers

Resource providers extend MuJoCo to load assets (XML files, meshes, textures, and etc.) that don't necessarily come from the OS filesystem or the Virtual File System (mjVFS). For example, downloading assets from the Internet could be implemented as a resource provider. These extensions are handled abstractly in MuJoCo via the mjResource struct.

Overview

Creating a new resource provider works by registering a mjpResourceProvider struct via mjp_registerResourceProvider in a global table. Once a resource provider is registered it can be used by all loading functions. The mjpResourceProvider struct stores three types of fields:

Resource prefix

Resources are identified by prefixes in their name. The chosen prefix should have a valid Uniform Resource Identifier (URI) scheme syntax. Resource names should also have a valid URI syntax, however this isn't enforced. A resource name with the syntax `{prefix}:{filename}` will match a provider using the scheme `prefix`. For instance, a resource provider accessing assets via the Internet might use `http` as its scheme. In this case a resource with the name `http://www.example.com/myasset.obj` would match against this resource provider. Schemes are case-insensitive so that `HTTP://www.example.com/myasset.obj` will also match. Note the importance of the colon. URI syntax requires that a colon follows the prefix in a resource name in order to match against a scheme. For example `https://www.example.com/myasset.obj` would NOT be a match since the scheme is designated as `https`.

Callbacks

There are three callbacks that a resource provider is required to implement: open, read, and close. The other two callback getdir and modified are optional. More details on these callbacks are given below.

Data Pointer

Lastly, there's an opaque data pointer for the provider to pass data into the callbacks. This data pointer is constant within a given model.

Resource providers work via callbacks:

- mjfOpenResource: The open callback takes a single parameter of type mjResource. The name field of the resource should be used to verify that the resource exists and populate the resource data field with any extra information needed for the resource. On failure this callback should return 0 (false) or else 1 (true).
- mjfReadResource: The read callback takes as arguments a mjResource and a pointer to a void pointer called the `buffer`. The read callback should point the `buffer` pointer to the location of where the bytes of the resource can be read and return the number of bytes pointed to in the `buffer`. On failure, this callback should return -1.
- mjfCloseResource: This callback takes a single parameter of type mjResource, and should be used to free any memory allocated in the data field in the supplied

resource.

- mjfGetResourceDir: This callback is optional and is used to extract the directory from a resource name. For example, the resource name `http://www.example.com/myasset.obj` would have `http://www.example.com/` as its directory.
- mjfResourceModified: This callback is optional and is used to check if an existing opened resource has been modified from its original source.

Usage

When a resource provider is registered, it can be used immediately to open assets. If the asset filename has a prefix that matches with the prefix of a registered provider, then that provider will be used to load the asset.

Example

This section provides a basic example of a resource provider that reads from a data URI scheme. First we implement the callbacks:

```
int str_open_callback(mjResource* resource) {
  // call some util function to validate
  if (!is_valid_data_uri(resource->name)) {
    return 0; // return failure
  }

  // some upper bound for the data
  resource->data = mju_malloc(get_data_uri_size(resource->name));
  if (resource->data == NULL) {
    return 0; // return failure
  }

  // fill data from string (some util function)
  get_data_uri(resource->name, &data);
}

int str_read_callback(mjResource* resource, const void** buffer) {
  *buffer = resource->data;
  return get_data_uri_size(resource->name);
}

void str_close_callback(mjResource* resource) {
  mju_free(resource->data);
}
```

Next we create the resource provider and register it with MuJoCo:

```
mjpResourceProvider resourceProvider = {
  .prefix = "data",
  .open = str_open_callback,
  .read = str_read_callback,
  .close = str_close_callback,
  .getdir = NULL
};

// return positive number on success
if (!mjp_registerResourceProvider(&resourceProvider)) {
  // ...
  // return failure
}
```

Now we can write assets as strings in our MJCF files:

```
<asset>
  <texture name="grid" file="grid.png" type="2d"/>
  <mesh content-type="model/obj" file="data:model/obj;base65,I215IG9iamVjdA0KdiAxIDAgMA0KdiA
  ...
</asset>
```