Programming

Introduction

This chapter is the MuJoCo programming guide. A separate chapter contains the API Reference documentation. MuJoCo is a dynamic library compatible with Windows, Linux and macOS, which requires a process with AVX instructions. The library exposes the full functionality of the simulator through a compiler-independent shared-memory C API. It can also be used in C++ programs.

The MuJoCo codebase is organized into subdirectories corresponding to different major areas of functionality:

Engine
: The simulator (or physics engine) is written in C. It is responsible for all runtime computations.

Parser
: The XML parser is written in C++. It can parse MJCF models and URDF models, converting them into an internal mjCModel C++ object which is not directly exposed to the user.

Compiler
: The compiler is written in C++. It takes an mjCModel C++ object constructed by the parser, and converts it into an mjModel C structure used at runtime.

Abstract visualizer
: The abstract visualizer is written in C. It generates a list of abstract geometric entities representing the simulation state, with all information needed for actual rendering. It also provides abstract mouse hooks for camera and perturbation control.

OpenGL renderer
: The renderer is written in C and is based on fixed-function OpenGL. It does not have all the features of state-of-the-art rendering engines (and can be replaced with such an engine if desired) but nevertheless it provides efficient and informative 3D rendering.

UI framework
: The UI framework (new in MuJoCo 2.0) is written in C. UI elements are rendered in OpenGL. It has its own event mechanism and abstract hooks for keyboard and mouse input. The code samples use it with GLFW, but it can also be used with other window libraries.

Getting started

MuJoCo is an open-source project. Pre-built dynamic libraries are available for x86_64 and arm64 machines running Windows, Linux, and macOS. These can be downloaded from the GitHub Releases page. Users who do not intend to develop or modify core MuJoCo code are encouraged to use our pre-built libraries, as these come bundled with the same versions of dependencies that we regularly test against, and benefit from build flags that have been tuned for performance. Our pre-built libraries are almost entirely self-contained and do not require any other library to be present, outside the standard C runtime. We also hide all symbols apart from those that form MuJoCo's public API, thus ensuring that it can coexist with any other libraries that may be loaded into the process (including other versions of libraries that MuJoCo depends on).

The pre-built distribution is a single .zip on Windows, .dmg on macOS, and .tar.gz on Linux. There is no installer. On Windows and Linux, simply extract the archive in a directory of your choice. From the `bin` subdirectory, you can now run the precompiled code samples, for example:

```
Windows:           simulate ..\model\humanoid.xml
Linux and macOS:   ./simulate ../model/humanoid.xml
```

The directory structure is shown below. Users can re-organize it if needed, as well as install the dynamic libraries in other directories and set the path accordingly. The only file created automatically is MUJOCO_LOG.TXT in the executable directory; it contains error and warning messages, and can be deleted at any time.

```
bin     - dynamic libraries, executables, MUJOCO_LOG.TXT
doc     - README.txt and REFERENCE.txt
include - header files needed to develop with MuJoCo
model   - model collection
sample  - code samples and makefile need to build them
```

After verifying that the simulator works, you may also want to re-compile the code samples to ensure that you have a working development environment. We provide Makefiles for Windows, macOS, and Linux, and also a cross-platform CMake setup that can be used to build sample applications independently of the MuJoCo library itself. If you are using the vanilla Makefile, we assume that you are using Visual Studio on Windows and LLVM/Clang on Linux. On Windows, you also need to either open a Visual Studio command prompt with native x64 tools or call the `vcvarsall.bat` script that comes with your MSVC installation to set up the appropriate environment variables.

On macOS, the DMG disk image contains `MuJoCo.app`, which you can double-click to launch the `simulate` GUI. You can also drag `MuJoCo.app` into the `/Application` on your system, as you would to install any other app. While `MuJoCo.app` may look like a file, it is in fact an Application Bundle, which is a directory that contains executable binaries for all of MuJoCo's sample applications, along with an embedded framework, which is a subdirectory containing the MuJoCo dynamic library and all of its public headers. In other words, `MuJoCo.app` contains all the same files that are shipped in the archive on Windows and Linux. To see this, right click (or control-click) on `MuJoCo.app` and click "Show Package Contents".

As mentioned above, `mujoco.framework` contains the library and headers that are necessary to build any application that depends on MuJoCo. If you are using Xcode, you can import it as a framework dependency on your project. (This also works for Swift projects without any modification). If you are building manually, you can use `-F` and `-framework mujoco` to specify the header search path and the library search path respectively. The macOS Makefile provides an example for this.

Building MuJoCo from source

To build MuJoCo from source, you will need CMake and a working C++17 compiler installed. The steps are:

1. Clone the `mujoco` repository from GitHub.
2. Create a new build directory somewhere, and `cd` into it.
3. Run `cmake $PATH_TO_CLONED_REPO` to configure the build.
4. Run `cmake --build .` to build.

MuJoCo's build system automatically fetches dependencies from upstream repositories over the Internet using CMake's FetchContent module.

The main CMake setup will build the MuJoCo library itself along with all sample applications, but the Python bindings are not built. Those come with their own build instructions, which can be found in the Python Bindings section of the documentation.

Additionally, the CMake setup also implements an installation phase which will copy and organize the output files to a target directory. Specify the directory using `cmake $PATH_TO_CLONED_REPO -DCMAKE_INSTALL_PREFIX=<my_install_dir>`. After successfully building MuJoCo following the instructions above, you can install it using `cmake --install .`.

Header files

The distribution contains several header files which are identical on all platforms. They are also available from the links below, to make this documentation self-contained.

mujoco.h (source)
: This is the main header file and must be included in all programs using MuJoCo. It defines all API functions and global variables, and includes the next 5 files which provide the necessary type definitions.

mjmodel.h (source)
: This file defines the C structure mjModel which is the runtime representation of the model being simulated. It also defines a number of primitive types and other structures needed to define mjModel.

mjdata.h (source)
: This file defines the C structure mjData which is the workspace where all computations read their inputs and write their outputs. It also defines primitive types and other structures needed to define mjData.

mjvisualize.h (source)
: This file defines the primitive types and structures needed by the abstract visualizer.

mjrender.h (source)
: This file defines the primitive types and structures needed by the OpenGL renderer.

mjui.h (source)
: This file defines the primitive types and structures needed by the UI framework.

mjtnum.h (source)
: Defines MuJoCo's `mjtNum` floating-point type to be either `double` or `float`. See mjtNum.

mjxmacro.h (source)
: This file is optional and is not included by mujoco.h. It defines X Macros that can automate the mapping of mjModel and mjData into scripting languages, as well as other operations that require accessing all fields of mjModel and mjData. See code sample testxml.cc.

mjexport.h (source)
: Macros used for exporting public symbols from the MuJoCo library. This header should not be used directly by client code.

glfw3.h
: This file is optional and is not included by mujoco.h. It is the only header file needed for the GLFW library. See code sample simulate.cc.

Versions and compatibility

MuJoCo has been used extensively since 2010 and is quite mature (even though our version numbering scheme is quite conservative). Nevertheless it remains under active development, and we have many exciting ideas for new features and are also making changes based on user feedback. This leads to unavoidable changes in both the modeling language in the API. While we encourage users to upgrade to the latest version, we recognize that this is not always feasible, especially when other developers release software that relies on MuJoCo. Therefore we have introduced simple mechanisms to help avoid version conflicts, as follows.

The situation is more subtle if existing code was developed with a certain version of MuJoCo, and is now being compiled and linked with a different version. If the definitions of the API functions used in that code have changed, either the compiler or the linker will generate errors. But even if the function definitions have not changed, it may still be a good idea to assert that the software version is the same. To this end, the main header (mujoco.h) defines the symbol mjVERSION_HEADER and the library provides the function mj_version. Thus the header and library versions can be compared with:

```
// recommended version check
if( mjVERSION_HEADER!=mj_version() )
  complain();
```

Note that only the main header defines this symbol. We assume that the collection of headers released with each software version will stay together and will not be mixed between versions. To avoid complications with floating-point comparisons, the above symbol and function use integers that are 100x the version number, so for example in software version 2.1 the symbol mjVERSION_HEADER is defined as 210.

Naming convention

All symbols defined in the API start with the prefix “mj”. The character after “mj” in the prefix determines the family to which the symbol belongs. First we list the prefixes corresponding to type definitions.

mj
: Core simulation data structure (C struct), for example mjModel. If all characters after the prefix are capital, for example mjMIN, this is a macro or a symbol (#define).

mjt
: Primitive type, for example mjtGeom. Except for mjtByte and mjtNum, all other definitions in this family are enums.

mjf
: Callback function type, for example mjfGeneric.

mjv
: Data structure related to abstract visualization, for example mjvCamera.

mjr
: Data structure related to OpenGL rendering, for example mjrContext.

mjui
: Data structure related to UI framework, for example mjuiSection.

Next we list the prefixes corresponding to function definitions. Note that function prefixes always end with underscore.

mj_
: Core simulation function, for example mj_step. Almost all such functions have pointers to mjModel and mjData as their first two arguments, possibly followed by other arguments. They usually write their outputs to mjData.

mju_
: Utility function, for example mju_mulMatVec. These functions are self-contained in the sense that they do not have mjModel and mjData pointers as their arguments.

mjv_
: Function related to abstract visualization, for example mjv_updateScene.

mjr_
: Function related to OpenGL rendering, for example mjr_render.

mjui_
: Function related to UI framework, for example mjui_update.

mjcb_
: Global callback function pointer, for example mjcb_control. The user can install custom callbacks by setting these global pointers to user-defined functions.

mjd_
: Functions for computing derivatives, for example mjd_transitionFD.

Using OpenGL

The use of MuJoCo’s native OpenGL renderer will be explained in OpenGL Rendering. For rendering, MuJoCo uses OpenGL 1.5 in the compatibility profile with the `ARB_framebuffer_object` and `ARB_vertex_buffer_object` extensions. OpenGL symbols are loaded via GLAD the first time the mjr_makeContext function is called. This means that the MuJoCo library itself does not have an explicit dependency on OpenGL and can be used on systems without OpenGL support, as long as `mjr_` functions are not called.

Applications that use MuJoCo’s built-in rendering functionalities are responsible for linking against an appropriate OpenGL context creation library and for ensuring that there is an OpenGL context that is made current on the running thread. On Windows and macOS, there is a canonical OpenGL library provided by the operating system. On Linux, MuJoCo currently supports GLX for rendering to an X11 window, OSMesa for headless software rendering, and EGL for hardware accelerated headless rendering.

Before version 2.1.4, MuJoCo used GLEW rather than GLAD to manage OpenGL symbols, which required linking against different GLEW libraries at build time depending on the GL implementation used. In order to avoid having manage OpenGL dependency when no rendering was required, “nogl” builds of the library was made available. Since OpenGL symbols are now lazily resolved at runtime after the switch to GLAD, the “nogl” libraries are no longer provided.

Code samples

MuJoCo comes with several code samples providing useful functionality. Some of them are quite elaborate (simulate.cc in particular) but nevertheless we hope that they will help users learn how to program with the library.

testspeed.cc

This code sample tests the simulation speed for a given model. The command line arguments are the model file, the number of time steps to simulate, the number of parallel threads to use, and a flag to enable internal profiling (the last two are optional). When N threads are specified with N>1, the code allocates a single mjModel and per-thread mjData, and runs N identical simulations in parallel. The idea is to test performance with all cores active, similar to Reinforcement Learning scenarios where samples are collected in parallel. The optimal N usually equals the number of logical cores. By default the simulation starts from the model reference configuration qpos0 and qvel=0. However if a keyframe named “test” is present in the model, it is used as the initial state state.

The timing code is straightforward: the simulation of the passive dynamics is advanced for the specified number of steps, while collecting statistics about the number of contacts, scalar constraints, and CPU times from internal profiling. The results are then printed in the console. To simulate controlled dynamics instead of passive dynamics one can either install the control

callback mjcb_control, or set control signals explicitly as explained in the simulation loop section below.

testxml.cc

This code sample tests the parser, compiler and XML writer. The testing code does the following:

- Parse and compile a specified XML model in MJCF or URDF. This yields an mjModel structure ready for simulation;
- Save the model as a temporary MJCF file, using a "canonical" subset of MJCF where a number of conversions have already been performed by the compiler;
- Parse and compile the temporary MJCF file. This yields a second mjModel structure ready for simulation;
- Compare the two mjModel structures field by field, and print the field with the largest numerical difference. Since MJCF is a text format, the real-valued numbers saved in it have lower precision than the double precision used internally, thus we cannot expect the two models to be identical on the bit level. But we can expect the largest difference to be on the order of 1e-6. A substantially larger difference indicates a bug in the parser, compiler or XML writer - and should be reported.

The code uses the X Macros described in the Reference chapter. This is a convenient way to apply the same operation to all fields in mjModel, without explicitly typing their names. The code sample simulate.cc also uses X Macros to implement a watch, where the user can type the name of any mjData field which is resolved at runtime.

compile.cc

This code sample evokes the built-in parser and compiler. It implements all possible model conversions from (MJCF, URDF, MJB) format to (MJCF, MJB, TXT) format. Models saved as MJCF use a canonical subset of our format as described in the Modeling chapter, and therefore MJCF-to-MJCF conversion will generally result in a different file. The TXT format is a human-readable road-map to the model. It cannot be loaded by MuJoCo, but can be a very useful aid during model development. It is in one-to-one correspondence with the compiled mjModel. Note also that one can use the function mj_printData to create a text file which is in one-to-one correspondence with mjData, although this is not done by the code sample.

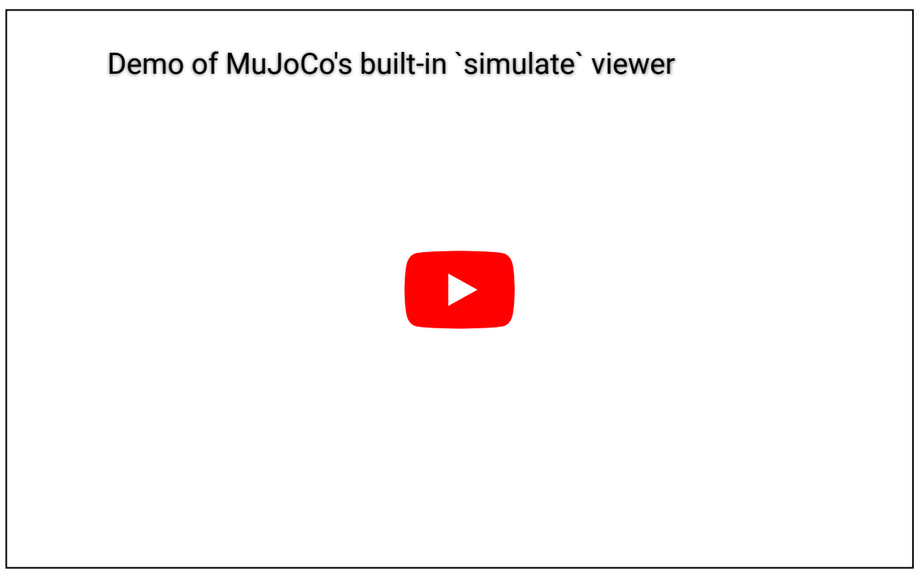
basic.cc

This code sample is a minimal interactive simulator. The model file must be provided as command-line argument. It opens an OpenGL window using the platform-independent GLFW library, and renders the simulation state at 60 fps while advancing the simulation in real-time. Press Backspace to reset the simulation. The mouse can be used to control the camera: left drag to rotate, right drag to translate in the vertical plane, shift right drag to translate in the horizontal plane, scroll or middle drag to zoom.

The Visualization programming guide below explains how visualization works. This code sample is a minimal illustration of the concepts in that guide.

simulate.cc

This code sample is a fully-featured interactive simulator. It opens an OpenGL window using the platform-independent GLFW library, and renders the simulation state in it. There is built-in help, simulation statistics, profiler, sensor data plots. The model file can be specified as a command-line argument, or loaded at runtime using drag-and-drop functionality. As of MuJoCo 2.0, this code sample uses the native UI to render various controls, and provides an illustration of how the new UI framework is intended to be used. Below is a screen-capture of `simulate` in action:



Interaction is done with the mouse; built-in help with a summary of available commands is available by pressing the `F1` key. Briefly, an object is selected by left-double-click. The user can then apply forces and torques on the selected object by holding Ctrl and dragging the mouse. Dragging the mouse alone (without Ctrl) moves the camera. There are keyboard shortcuts for pausing the simulation, resetting, and re-loading the model file. The latter functionality is very useful while editing the model in an XML editor.

The code is quite long yet reasonably commented, so it is best to just read it. Here we provide a high-level overview. The `main()` function initializes both MuJoCo and GLFW, opens a window, and install GLFW callbacks for mouse and keyboard handling. Note that there is no render callback; GLFW puts the user in charge, instead of running a rendering loop behind the scenes. The main loop handles UI events and rendering. The simulation is handled in a background thread, which is synchronized with the main thread.

The mouse and keyboard callbacks perform whatever action is necessary. Many of these actions invoke functionality provided by MuJoCo's abstract visualization mechanism. Indeed this mechanism is designed to be hooked to mouse and keyboard events more or less directly, and provides camera as well as perturbation control.

The profiler and sensor data plots illustrate the use of the mjr_figure function that can plot elaborate 2D figures with grids, annotation, axis scaling etc. The information presented in the profiler is extracted from the diagnostic fields of mjData. It is a very useful tool for tuning the

parameters of the constraint solver algorithms. The outputs of the sensors defined in the model are visualized as a bar graph.

Note that the profiler shows timing information collected with high-resolution timers. On Windows, depending on the power settings, the OS may reduce the CPU frequency; this is because simulate.cc sleeps most of the time in order to slow down to realtime. This results in inaccurate timings. To avoid this problem, change the Windows power plan so that the minimum processor state is 100%.

record.cc

This code sample simulates the passive dynamics of a given model, renders it offscreen, reads the color and depth pixel values, and saves them into a raw data file that can then be converted into a movie file with tools such as ffmpeg. The rendering is simplified compared to simulate.cc because there is no user interaction, visualization options or timing; instead we simply render with the default settings as fast as possible. The dimensions and number of multi-samples for the offscreen buffer are specified in the MuJoCo model, while the simulation duration, frames-per- second to be rendered (usually much less than the physics simulation rate), and output file name are specified as command-line arguments. For example, a 5 second animation at 60 frames per second is created with:

```
render humanoid.xml 5 60 rgb.out
```

The default humanoid.xml model specifies offscreen rendering with 800x800 resolution. With this information in hand, we can compress the (large) raw date file into a playable movie file:

```
ffmpeg -f rawvideo -pixel_format rgb24 -video_size 800x800
  -framerate 60 -i rgb.out -vf "vflip" video.mp4
```

This sample can be compiled in three ways which differ in how the OpenGL context is created: using GLFW with an invisible window, using OSMesa, or using EGL. The latter two options are only available on Linux and are envoked by defining the symbols MJ_OSMESA or MJ_EGL when compiling record.cc. The functions `initOpenGL` and `closeOpenGL` create and close the OpenGL context in three different ways depending on which of the above symbols is defined.

Note that the MuJoCo rendering code does not depend on how the OpenGL context was created. This is the beauty of OpenGL: it leaves context creation to the platform, and the actual rendering is then standard and works in the same way on all platforms. In retrospect, the decision to leave context creation out of the standard has led to unnecessary proliferation of overlapping technologies, which differ not only between platforms but also within a platform in the case of Linux. The addition of a couple of extra functions (such as those provided by OSMesa for example) could have avoided a lot of confusion. EGL is a newer standard from Khronos which aims to do this, and it is gaining popularity. But we cannot yet assume that all users have it installed.

derivative.cc

This code sample illustrates the numerical approximation of forward and inverse dynamics derivatives via finite differences. The process involves a number of epochs. In each epoch the simulation is advanced for a specified number of steps, derivatives are computed at the last state, and timing and accuracy statistics are collected. The averages over epochs are printed at the end.

The code can be incorporated in user projects where derivatives are needed, and can also be used as a stand-alone tool for estimating CPU time and numerical accuracy. Accuracy is estimated in the function `checkderiv()` using several mathematical identities about the derivatives of inverse functions; the residuals being computed would be zero if the derivatives were exact. Note that these identities involve matrix multiplications which may affect the accuracy estimates. Timing tests are applied only to the parallel section, where the function `worker()` is executed in multiple threads using OpenMP. There are fewer threads than forward/inverse dynamics evaluations, thus each thread executes multiple evaluations. For a more general discussion of parallel processing in MuJoCo see multi-threading below.

Recall than for a differentiable function `f(x)` the derivative can be approximated as

```
df/dx = (f(x+eps)-f(x))/eps
```

where `eps` is a small number. One can also use the centered finite difference method, which is two times slower but more accurate. Here `f` is one of the functions

```
forward dynamics:  qacc(qfrc_applied, qvel, qpos)
inverse dynamics:  qfrc_inverse(qacc, qvel, qpos)
```

The code sample computes six Jacobian matrices, containing the derivative of each function with respect to its three arguments. The results are stored in the array `deriv`. All six Jacobian matrices are square, with dimensionality equal to the number of degrees of freedom `mjModel.nv`. When the model configuration includes quaternion joints, mjData.qpos has larger dimensionality than the other vectors, however the derivative is only defined in the tangent space to the configuration manifold. This is why, when differentiating with respect to the elements of `mjData.qpos`, we do not directly add `eps` but instead use the function mju_quatIntegrate to perturb the quaternion in the tangent space, keeping it normalized. This technique should also be used in any other situation where quaternions need to be perturbed.

There are some important subtleties in this code that improve speed as well as accuracy. To speed up the computation, we re-use intermediate results whenever possible. This relies on the skip mechanism described under forward dynamics and inverse dynamics below. We first perturb force dimensions, keeping position and velocity fixed. In this way we avoid recomputing results that depend on position and velocity but not on force. Then we perturb velocity dimensions, and avoid recomputing results that depend on position but not on velocity or force. Finally we perturb position dimensions - which requires full computation because everything depends on position.

Accuracy depends on the value of `eps` which is user-adjustable, as well as the shape of the function. In the case of forward dynamics however, the function evaluation involves an iterative constraint solver, and this must be handled with care. In general, the difference between `f(x+eps)` and `f(x)` is very small, thus any noise affecting the two function evaluations differently can make the resulting derivatives meaningless. Different warm-starts or different number of solver iterations can act as such noise here. Therefore we fix the warm-start `mjData.qacc` to a value pre-computed at the center point, using `nwarmup` extra major iterations to obtain a more accurate warm-start. We

also fix the number of solver iterations to `niter` and set `mjModel.opt.tolerance = 0;` this disables the early termination mechanism. Note that the original simulation options are restored in the serial code which advances the state.

We emphasize that the above subtleties are not high-order corrections that can be incorporated later. In the presence of unilateral constraints, numerical derivatives are hard to compute and there is no shortcut around it; indeed they would not even be defined if it wasn't for our soft-constraint model. Making the constraints softer results in more accurate results. This effect is so strong that in some situations it makes sense to intentionally work with the wrong model, i.e., a model that is softer than desired, so as to obtain more accurate derivatives.

uitools

(uitools.h) (uitools.cc) This is not a stand-alone code sample, but rather a small utility used to hook up the new UI to GLFW. It is used in simulate.cc and can also be used in user projects that involve the new UI. If GLFW is replaced with a different window library, this is the only file that would have to be changed in order to access the UI functionality.

Simulation

Initialization

After the version check, the next step is to allocate and initialize the main data structures needed for simulation, namely mjModel and mjData. Additional initialization steps related to visualization and callbacks will be discussed later.

mjModel and mjData should never be allocated directly by the user. Instead they are allocated and initialized by the corresponding API functions. These are very elaborate data structures, containing (arrays of) other structures, preallocated data arrays for all intermediate results, as well as an internal stack. Our strategy is to allocate all necessary heap memory at the beginning of the simulation, and free it after the simulation is done, so that we never have to call the C memory allocation and deallocation functions during the simulation. This is done for speed, avoidance of memory fragmentation, future GPU portability, and ease of managing the state of the entire simulator during a reset. It also means however that the maximal variable-memory allocation given by the memory attribute in the size MJCF element, which affects the allocation of `mjData`, must be set to a sufficiently large value. If this maximal size is exceeded during simulation, it is not increased dynamically, but instead an error is generated. See also diagnostics below.

First we must call one of the functions that allocates and initializes mjModel and returns a pointer to it. The available options are

```
// option 1: parse and compile XML from file
mjModel* m = mj_loadXML("mymodel.xml", NULL, errstr, errstr_sz);

// option 2: parse and compile XML from virtual file system
mjModel* m = mj_loadXML("mymodel.xml", vfs, errstr, errstr_sz);

// option 3: load precompiled model from MJB file
mjModel* m = mj_loadModel("mymodel.mjb", NULL);

// option 4: load precompiled model from virtual file system
mjModel* m = mj_loadModel("mymodel.mjb", vfs);

// option 5: deep copy from existing mjModel
mjModel* m = mj_copyModel(NULL, mexisting);
```

All these functions return a NULL pointer if there is an error or warning. In the case of XML parsing and model compilation, a description of the error is returned in the string provided as argument. For the remaining functions, the low-level mju_error or mju_warning is called with the error/warning message; see error handling. Once we have a pointer to the mjModel that was allocated by one of the above functions, we pass it as argument to all API functions that need model access. Note that most functions treat this pointer as `const`; more on this in model changes below.

The virtual file system (VFS) was introduced in MuJoCo 1.50. It allows disk resources to be loaded in memory or created programmatically by the user, and then MuJoCo's load functions search for files in the VFS before accessing the disk. See Virtual file system in the API Reference chapter.

In addition to mjModel which holds the model description, we also need mjData which is the workspace where all computations are performed. Note that mjData is specific to a given mjModel. The API functions generally assume that users know what they are doing, and perform minimal argument checking. If the mjModel and mjData passed to any API function are incompatible (or NULL) the resulting behavior is unpredictable. mjData is created with

```
// option 1: create mjDada corresponding to given mjModel
mjData* d = mj_makeData(m);

// option 2: deep copy from existing mjData
mjData* d = mj_copyData(NULL, m, dexisting);
```

Once both mjModel and mjData are allocated and initialized, we can call the various simulation functions. When we are done, we can delete them with

```
// deallocate existing mjModel
mj_deleteModel(m);

// deallocate existing mjData
mj_deleteData(d);
```

The code samples illustrate the complete initialization and termination sequence.

MuJoCo simulations are deterministic with one exception: sensor noise can be generated when this feature is enabled. This is done by calling the C function rand() internally. To generate the same random number sequence, call srand() with a desired seed after the model is loaded and before the simulation starts. The model compiler calls srand(123) internally, so as to generate random dots for procedural textures. Therefore the noise sequence in the sensor data will change if the specification of procedural textures changes, and the user does not call srand() after model compilation.

Simulation loop

There are multiple ways to run a simulation loop in MuJoCo. The simplest way is to call the top-level simulation function mj_step in a loop such as

```
// simulate until t = 10 seconds
while( d->time<10 )
  mj_step(m, d);
```

This by itself will simulate the passive dynamics, because we have not provided any control signals or applied forces. The default (and recommended) way to control the system is to implement a control callback, for example

```
// simple controller applying damping to each dof
void mycontroller(const mjModel* m, mjData* d)
{
  if( m->nu==m->nv )
    mju_scl(d->ctrl, d->qvel, -0.1, m->nv);
}
```

This illustrates two concepts. First, we are checking if the number of controls `mjModel.nu` equals the number of DoFs `mjModel.nv`. In general, the same callback may be used with multiple models depending on how the user code is structured, and so it is a good idea to check the model dimensions in the callback. Second, MuJoCo has a library of BLAS-like functions that are very useful; indeed a large part of the code base consists of calling such functions internally. The mju_scl function above scales the velocity vector `mjData.qvel` by a constant feedback gain and copies the result into the control vector `mjData.ctrl`. To install this callback, we simply assign it to the global control callback pointer mjcb_control:

```
// install control callback
mjcb_control = mycontroller;
```

Now if we call mj_step, our control callback will be executed whenever the control signal is needed by the simulation pipeline, and as a result we will end up simulating the controlled dynamics (except damping does not really do justice to the notion of control, and is better implemented as a passive joint property, but these are finer points).

Instead of relying on a control callback, we could set the control vector `mjData.ctrl` directly. Alternatively we could set applied forces as explained in state and control. If we could compute these control- related quantities before `mj_step` is called, then the simulation loop for the controlled dynamics (without using a control callback) would become

```
while( d->time<10 ) {
  // set d->ctrl or d->qfrc_applied or d->xfrc_applied
  mj_step(m, d);
}
```

Why would we not be able to compute the controls before `mj_step` is called? After all, isn't this what causality means? The answer is subtle but important, and has to do with the fact that we are simulating in discrete time. The top- level simulation function `mj_step` basically does two things: compute the forward dynamics in continuous time, and then integrate over a time period specified by `mjModel.opt.timestep`. Forward dynamics computes the acceleration `mjData.qacc` at time `mjData.time`, given the state and control at time `mjData.time`. The numerical integrator then advances the state and time to `mjData.time + mjModel.opt.timestep`. Now, the control is required to be a function of the state at time `mjData.time`. However a general feedback controller can be a very complex function, depending on various features of the state - in particular all the features computed by MuJoCo as intermediate results of the simulation. These may include contacts, Jacobians, passive forces. None of these quantities are available before `mj_step` is called (or rather, they are available but outdated by one time step). In contrast, when `mj_step` calls our control callback, it does so as late in the computation as possible - namely after all the intermediate results dependent on the state but not on the control have been computed.

The same effect can be achieved without using a control callback. This is done by breaking `mj_step` in two parts: before the control is needed, and after the control is needed. The simulation loop now becomes

```
while( d->time<10 ) {
  mj_step1(m, d);
  // set d->ctrl or d->qfrc_applied or d->xfrc_applied
  mj_step2(m, d);
}
```

There is one complication however: this only works with Euler integration. The Runge-Kutta integrator (as well as other advanced integrators we plan to implement) need to evaluate the entire dynamics including the feedback control law multiple times per step, which can only be done using a control callback. But with Euler integration, the above separation of `mj_step` into mj_step1 and mj_step2 is sufficient to provide the control law with the intermediate results of the computation.

To make the above discussion more clear, we provide the internal implementation of mj_step, mj_step1 and mj_step2, omitting some code that computes timing diagnostics. The main simulation function is

```
void mj_step(const mjModel* m, mjData* d) {
  // common to all integrators
  mj_checkPos(m, d);
  mj_checkVel(m, d);
  mj_forward(m, d);
  mj_checkAcc(m, d);

  // compare forward and inverse solutions if enabled
  if( mjENABLED(mjENBL_FWDINV) )
    mj_compareFwdInv(m, d);

  // use selected integrator
  if( m->opt.integrator==mjINT_RK4 )
    mj_RungeKutta(m, d, 4);
  else
    mj_Euler(m, d);
}
```

The checking functions reset the simulation automatically if any numerical values have become invalid or too large. The control callback (if any) is called from within the forward dynamics function.

Next we show the implementation of the two-part stepping approach, although the specifics will make sense only after we explain the forward dynamics later. Note that the control callback is now called directly, since we have essentially unpacked the forward dynamics function. Note also that we always call the Euler integrator in mj_step2 regardless of the setting of `mjModel.opt.integrator`.

```
void mj_step1(const mjModel* m, mjData* d)
{
  mj_checkPos(m, d);
  mj_checkVel(m, d);
  mj_fwdPosition(m, d);
  mj_sensorPos(m, d);
  mj_energyPos(m, d);
  mj_fwdVelocity(m, d);
  mj_sensorVel(m, d);
  mj_energyVel(m, d);

  // if we had a callback we would be using mj_step, but call it anyway
  if( mjcb_control )
    mjcb_control(m, d);
}

void mj_step2(const mjModel* m, mjData* d)
{
  mj_fwdActuation(m, d);
  mj_fwdAcceleration(m, d);
  mj_fwdConstraint(m, d);
  mj_sensorAcc(m, d);
  mj_checkAcc(m, d);

  // compare forward and inverse solutions if enabled
  if( mjENABLED(mjENBL_FWDINV) )
    mj_compareFwdInv(m, d);

  // integrate with Euler; ignore integrator option
  mj_Euler(m, d);
}
```

State and control

MuJoCo has a well-defined state that is easy to set, reset and advance through time. This is closely related to the notion of state of a dynamical system. Dynamical systems are usually described in the general form

```
dx/dt = f(t,x,u)
```

where `t` is the time, `x` is the state vector, `u` is the control vector, and `f` is the function that computes the time-derivative of the state. This is a continuous-time formulation, and indeed the physics model simulated by MuJoCo is defined in continuous time. Even though the numerical integrator operates in discrete time, the main part of the computation - namely the function mj_forward - corresponds to the continuous-time dynamics function `f(t,x,u)` above. Here we explain this correspondence.

The state vector in MuJoCo is:

```
x = (mjData.time, mjData.qpos, mjData.qvel, mjData.act)
```

For a second-order dynamical system the state contains only position and velocity, however MuJoCo can also model actuators (such as cylinders and biological muscles) that have their own activation states assembled in the vector `mjData.act`. While the physics model is time-invariant, user-defined control laws may be time-varying; in particular control laws obtained from trajectory optimizers would normally be indexed by `mjData.time`.

The reason for the “official” caveat above is because user callbacks may store additional state variables that change over time and affect the callback outputs; indeed the field `mjData.userdata` exists mostly for that purpose. Other state-like quantities that are part of mjData and are treated as inputs by forward dynamics are `mjData.mocap_pos` and mjData.mocap_quat. These quantities are unusual in that they are meant to change at each time step (normally driven by a motion capture device), however this change is implemented by the user, while the simulator treats them as constants. In that sense they are no different from all the constants in mjModel, or the function callback pointers set by the user: such constants affect the computation, but are not part of the state vector of a dynamical system.

The warm-start mechanism in the constraint solver effectively introduces another state variable. This mechanism uses the output of forward dynamics from the previous time step, namely the acceleration vector `mjData.qacc`, to estimate the current constraint forces via inverse dynamics. This estimate then initializes the optimization algorithm in the solver. If this algorithm runs until convergence the warm-start will affect the speed of convergence but not the final solution (since the underlying optimization problem is convex and does not have local minima), but in practice the algorithm is often terminated early, and so the warm-start has some (usually very small) effect on the solution.

Next we turn to the controls and applied forces. The control vector in MuJoCo is

```
u = (mjData.ctrl, mjData.qfrc_applied, mjData.xfrc_applied)
```

These quantities specify control signals (`mjData.ctrl`) for the actuators defined in the model, or directly apply forces and torques specified in joint space (`mjData.qfrc_applied`) or in Cartesian space (mjData.xfrc_applied).

Finally, calling mj_forward which corresponds to the abstract dynamics function `f(t,x,u)` computes the time-derivative of the state vector. The corresponding fields of mjData are

```
dx/dt = f(t,x,u) = (1, mjData.qvel, mjData.qacc, mjData.act_dot)
```

In the presence of quaternions (i.e., when free or ball joints are used), the position vector `mjData.qpos` has higher dimensionality than the velocity vector `mjData.qvel` and so this is not a simple time-derivative in the sense of scalars, but instead takes quaternion algebra into account.

To illustrate how the simulation state can be manipulated, suppose we have two mjData pointers src and dst corresponding to the same mjModel, and we want to copy the entire simulation state from one to the other (leaving out internal diagnostics which do not affect the simulation). This can be done as

```
// copy simulation state
dst->time = src->time;
mju_copy(dst->qpos, src->qpos, m->nq);
mju_copy(dst->qvel, src->qvel, m->nv);
mju_copy(dst->act,  src->act,  m->na);

// copy mocap body pose and userdata
mju_copy(dst->mocap_pos,  src->mocap_pos,  3*m->nmocap);
mju_copy(dst->mocap_quat, src->mocap_quat, 4*m->nmocap);
mju_copy(dst->userdata, src->userdata, m->nuserdata);

// copy warm-start acceleration
mju_copy(dst->qacc_warmstart, src->qacc_warmstart, m->nv);
```

Now, assuming the controls are also the same (see below) and that any installed callbacks are not relying on user-defined state variables that are different between src and dst, calling mj_forward(m, src) or mj_step(m, src) yields the same result as calling mj_forward(m, dst) or mj_step(m, dst) respectively. Similarly, calling mj_inverse(m, src) yields the same result as calling mj_inverse(m, dst). More on inverse dynamics later.

The entire mjData can also be copied with the function mj_copyData. This involves less code but is much slower. Indeed using the above code to copy the state and then calling mj_forward to recompute everything can sometimes be faster than copying mjData. This is because the preallocated buffers in mjData are large enough to hold the intermediate results in the worst case where all possible constraints are active, but in practice only a small fraction of constraints tend to be active simultaneously.

To illustrate how the control vector can be manipulated, suppose we want to clear all controls and applied forces before calling mj_step, so as to make sure we are simulating the passive dynamics (assuming no control callback of course). This can be done as

```
// clear controls and applied forces
mju_zero(dst->ctrl, m->nu);
mju_zero(dst->qfrc_applied, m->nv);
mju_zero(dst->xfrc_applied, 6*m->nbody);
```

If the user has installed a control callback mjcb_control different from the default callback (which is a NULL pointer), the user callback would be expected to set some of the above fields to non-zero. Note that MuJoCo will not clear these controls/forces at the end of the time step. This is the responsibility of the user.

Also relevant in this context is the function mj_resetData. It sets `mjData.qpos` equal to the model reference configuration `mjModel.qpos0`, `mjData.mocap_pos` and `mjData.mocap_quat` equal to the corresponding fixed body poses from mjModel; and all other state and control variables to 0.

Forward dynamics

The goal of forward dynamics is to compute the time-derivative of the state, namely the acceleration vector mjData.qacc and the activation time-derivative `mjData.act_dot`. Along the way it computes everything else needed to simulate the dynamics, including active contacts and other constraints, joint-space inertia and its LTDL decomposition, constraint forces, sensor data and so on. All these intermediate results are available in mjData and can be used in custom computations. As illustrated in the simulation loop section above, the main stepper function mj_step calls mj_forward to do most of the work, and then calls the numerical integrator to advance the simulation state to the next discrete point in time.

The forward dynamics function mj_forward internally calls mj_forwardSkip with skip arguments (mjSTAGE_NONE, 0), where the latter function is implemented as

```
void mj_forwardSkip(const mjModel* m, mjData* d, int skipstage, int skipsensor) {
  // position-dependent
  if( skipstage<mjSTAGE_POS )
  {
    mj_fwdPosition(m, d);
    if( !skipsensor )
      mj_sensorPos(m, d);
    if( mjENABLED(mjENBL_ENERGY) )
      mj_energyPos(m, d);
  }

  // velocity-dependent
  if( skipstage<mjSTAGE_VEL )
  {
    mj_fwdVelocity(m, d);
    if( !skipsensor )
      mj_sensorVel(m, d);
    if( mjENABLED(mjENBL_ENERGY) )
      mj_energyVel(m, d);
  }

  // acceleration-dependent
  if( mjcb_control )
    mjcb_control(m, d);
  mj_fwdActuation(m, d);
  mj_fwdAcceleration(m, d);
  mj_fwdConstraint(m, d);
  if( !skipsensor )
    mj_sensorAcc(m, d);
}
```

Note that this is the same sequence of calls as in mj_step1 and mj_step2 above, except that checking of real values and computing features such as sensor and energy are omitted. The functions being called are components of the simulation pipeline. In turn they call sub-components.

The integer argument skipstage determines which parts of the computation will be skipped. The possible skip levels are

mjSTAGE_NONE

Skip nothing. Run all computations.

mjSTAGE_POS
: Skip computations that depend on position but not on velocity or control or applied force. Examples of such computations include forward kinematics, collision detection, inertia matrix computation and decomposition. These computations typically take the most CPU time and should be skipped when possible (see below).

mjSTAGE_VEL
: Skip computations that depend on position and velocity but not on control or applied force. Examples include the computation of Coriolis and centrifugal forces, passive damping forces, reference accelerations for constraint stabilization.

The intermediate result fields of mjData are organized into sections according to which part of the state is needed in order to compute them. Calling mj_forwardSkip with mjSTAGE_POS assumes that the fields in the first section (position dependent) have already been computed and does not recompute them. Similarly, mjSTAGE_VEL assumes that the fields in the first and second sections (position and velocity dependent) have already been computed.

When can we use the above machinery and skip some of the computations? In a regular simulation this is not possible. However, MuJoCo is designed not only for simulation but also for more advanced applications such as model-based optimization, machine learning etc. In such settings one often needs to sample the dynamics at a cloud of nearby states, or approximate derivatives via finite differences - which is another form of sampling. If the samples are arranged on a grid, where only the position or only the velocity or only the control is different from the center point, then the above mechanism can improve performance by about a factor of 2. The code sample derivative.cc illustrates this approach, and also shows how multi-threading can be used for additional speedup.

Inverse dynamics

The computation of inverse dynamics is a unique feature of MuJoCo, and is not found in any other modern engine capable of simulating contacts. Inverse dynamics are well-defined and very efficient to compute, thanks to our soft-constraint model described in the Overview chapter. In fact once the position and velocity-dependent computations that are shared with forward dynamics have been performed, the recovery of constraint and applied forces given the acceleration comes down to an analytical formula. This is so fast that we actually use inverse dynamics (with the acceleration computed at the previous time step) to warm-start the iterative constraint solver in forward dynamics.

The inputs to inverse dynamics are the same as the state vector in forward dynamics as illustrated in state and control, but without `mjData.act` and `mjData.time`. Assuming no callbacks that depend on user- defined state variables, the inputs to inverse dynamics are the following fields of mjData:

```
(mjData.qpos, mjData.qvel, mjData.qacc, mjData.mocap_pos, mjData.mocap_quat)
```

The main output is `mjData.qfrc_inverse`. This is the force that must have acted on the system in order to achieve the observed acceleration `mjData.qacc`. If forward dynamics were to be computed exactly, by running the iterative solver to full convergence, we would have

```
mjData.qfrc_inverse = mjData.qfrc_applied + Jacobian'*mjData.xfrc_applied + mjData.qfrc_actua
```

where `mjData.qfrc_actuator` is the joint-space force produced by the actuators and the Jacobian is the mapping from joint to Cartesian space. When the “fwdinv” flag in `mjModel.opt.enableflags` is set, the above identity is used to monitor the quality of the forward dynamics solution. In particular, the two components of `mjData.solver_fwdinv` are set to the L2 norm of the difference between the forward and inverse solutions, in terms of joint forces and constraint forces respectively.

Similar to forward dynamics, `mj_inverse` internally calls mj_inverseSkip with skip arguments `(mjSTAGE_NONE, 0)`. The skip mechanism is the same as in forward dynamics, and can be used to speed up structured sampling. The result `mjData.qfrc_inverse` is obtained by using the Recursive Newton-Euler algorithm to compute the net force acting on the system, and then subtracting from it all internal forces.

Inverse dynamics can be used as an analytical tool when experimental data are available. This is common in robotics as well as biomechanics. It can also be used to compute the joint torques needed to drive the system along a given reference trajectory; this is known as computed torque control. In the context of state estimation, system identification and optimal control, it can be used within an optimization loop to find sequences of states that minimize physics violation along with other costs. Physics violation can be quantified as the norm of any unexplained external force computed by inverse dynamics.

Multi-threading

When MuJoCo is used for simulation as explained in the simulation loop section, it runs in a single thread. We have experimented with multi-threading parts of the simulation pipeline that are computationally expensive and amenable to parallel processing, and have concluded that the speedup is not worth using up the extra processor cores. This is because MuJoCo is already fast compared to the overhead of launching and synchronizing multiple threads within the same time step. If users start working with large simulations involving many floating bodies, we may eventually implement within-step multi-threading, but for now this use case is not common.

Rather than speed up a single simulation, we prefer to use multi-threading to speed up sampling operations that are common in more advanced applications. Simulation is inherently serial over time (the output of one mj_step is the input to the next), while in sampling many calls to either forward or inverse dynamics can be executed in parallel since there are no dependencies among them, except perhaps for a common initial state. The code sample derivative.cc illustrates one important example of sampling, namely the approximation of dynamics derivatives via finite differences. Here we will not repeat the material from that section, but will instead explain MuJoCo’s general approach to parallel processing.

MuJoCo was designed for multi-threading from its beginning. Unlike most existing simulators where the notion of dynamical system state is difficult to map to the software state and is often distributed among multiple objects, in MuJoCo we have the unified data structure mjData which contains everything that changes over time. Recall the discussion of state and control. The key idea is to create one mjData for each thread, and then use it for all per-thread computations. Below is the general template, using OpenMP to simplify thread management.

```
// prepare OpenMP
int nthread = omp_get_num_procs();      // get number of logical cores
omp_set_dynamic(0);                     // disable dynamic scheduling
omp_set_num_threads(nthread);           // number of threads = number of logical cores

// allocate per-thread mjData
mjData* d[64];
for( int n=0; n<nthread; n++ )
    d[n] = mj_makeData(m);

// ... serial code, perhaps using its own mjData* dmain

// parallel section
#pragma omp parallel
{
  int n = omp_get_thread_num();       // thread-private variable with thread id (0 to nthread

  // ... initialize d[n] from results in serial code

  // thread function
  worker(m, d[n]);                    // shared mjModel (read-only), per-thread mjData (read-
}

// delete per-thread mjData
for( int n=0; n<nthread; n++ )
  mj_deleteData(d[n]);
```

Since all top-level API functions treat mjModel as `const`, this multi-threading scheme is safe. Each thread only writes to its own mjData. Therefore no further synchronization among threads is needed.

The above template reflects a particular style of parallel processing. Instead of creating a large number of threads, one for each work item, and letting OpenMP distribute them among processors, we rely on manual scheduling. More precisely, we create as many threads as there are processors, and then within the `worker` function we distribute the work explicitly among threads (not shown here, but see derivative.cc for an example). This approach is more efficient because the thread-specific mjData is large compared to the processor cache.

We also use a shared mjModel for cache-efficiency. In some situations it may not be possible to use the same mjModel for all threads. One obvious reason is that mjModel may need to be modified within the thread function. Another reason is that the mjOption structure which is contained within mjModel may need to be adjusted (so as to control the number of solver iterations for example), although this is likely to be the same for all parallel threads and so the adjustment can be made in the shared model before the parallel section.

How the thread-specific mjData is initialized and what the thread function does is of course application-dependent. Nevertheless, the general efficiency guidelines from the earlier sections apply here. Copying the state into the thread-specific mjData and running MuJoCo to fill in the rest may be faster than using mj_copyData. Furthermore, the skip mechanism available in both forward and inverse dynamics is particularly useful in parallel sampling applications, because the samples usually have structure allowing some computations to be re-used. Finally, keep in mind that the forward solver is iterative and good warm-start can substantially reduce the number of necessary iterations. When samples are close to each other in state and control space, the solution for one sample (ideally in the center) can be used to warm-start all the other samples. In this setting it is important to make sure that the different results between nearby samples reflect genuine differences between the samples, and not different warm-start or termination of the iterative solver.

Model changes

The MuJoCo model contained in mjModel is supposed to represent constant physical properties of the system, and in theory should not change after compilation. Of course in practice things are not that simple. It is often desirable to change the physics options in `mjModel.opt`, so as to experiment with different aspects of the physics or to create custom computations. Indeed these options are designed in such a way that the user can make arbitrary changes to them between time steps.

The general rule is that real-valued parameters are safe to change, while structural integer parameters are not because that may result in incorrect sizes or indexing. This rule does not hold universally though. Some real-valued parameters such as inertias are expected to obey certain properties. On the other hand, some structural parameters such as object types may be possible to change, but that depends on whether any sizes or indexes depend on them. Arrays of type mjtByte can be changed safely, since they are binary indicators that enable and disable certain features. The only exception here is `mjModel.tex_rgb` which is texture data represented as mjtByte.

When changing mjModel fields that corresponds to resources uploaded to the GPU, the user must also call the corresponding upload function: `mjr_uploadTexture`, `mjr_uploadMesh`, `mjr_uploadHField`. Otherwise the data used for simulation and for rendering will no longer be consistent.

A related consideration has to do with changing real-valued fields of mjModel that have been used by the compiler to compute other real-valued fields: if we make a change, we want it to propagate. That is what the function mj_setConst does: it updates all derived fields of mjModel. These are fields whose names end with "0", corresponding to precomputed quantities when the model is in the reference configuration `mjModel.qpos0`.

Finally, if changes are made to mjModel at runtime, it may be desirable to save them back to the XML. The function mj_saveLastXML does that in a limited sense: it copies all real-valued parameters from mjModel back to the internal mjCModel, and then saves it as XML. This does not cover all possible changes that the user could have made. The only way to guarantee that all changes are saved is to save the model as a binary MJB file with the function mj_saveModel, or even better, make the changes directly in the XML. Unfortunately there are situations where changes need to be made programmatically, as in system identification for example, and this can only be done with the compiled model. So in summary, we have reasonable but not perfect mechanisms for saving model changes. The reason for this lack of perfection is that we are working with a compiled model, so this is like changing a binary executable and asking a "decompiler" to make corresponding changes to the C code - it is just not possible in general.

Data layout and buffer allocation

All matrices in MuJoCo are in **row-major** format. For example, the linear memory array (a0, a1, … a5) represents the 2-by-3 matrix

```
a0 a1 a2
a3 a4 a5
```

This convention has traditionally been associated with C, while the opposite column-major convention has been associated with Fortran. There is no particular reason to choose one over the other, but whatever the choice is, it is essential to keep it in mind at all times. All MuJoCo utility functions that operate on matrices, such as mju_mulMatMat, mju_mulMatVec etc. assume this matrix layout. For vectors there is of course no difference between row-major and column-major formats.

When possible, MuJoCo exploits sparsity. This can make all the difference between O(N) and O(N^3) scaling. The inertia matrix `mjData.qM` and its LTDL factorization `mjData.qLD` are always represented as sparse, using a custom indexing format designed for matrices that correspond to tree topology. The functions mj_factorM, mj_solveM, mj_solveM2 and mj_mulM are used for sparse factorization, substitution and matrix-vector multiplication. The user can also convert these matrices to dense format with the function mj_fullM although MuJoCo never does that internally.

The constraint Jacobian matrix `mjData.efc_J` is represented as sparse whenever the sparse Jacobian option is enabled. The function mj_isSparse can be used to determine if sparse format is currently in use. In that case the transposed Jacobian `mjData.efc_JT` is also computed, and the inverse constraint inertia `mjData.efc_AR` becomes sparse. Sparse matrices are stored in the compressed sparse row (CSR) format. For a generic matrix A with dimensionality m-by-n, this format is:

Variable	Size	Meaning
A	m * n	Real-valued data
A_rownnz	m	Number of non-zeros per row
A_rowadr	m	Starting index of row data in A and A_colind
A_colind	m * n	Column indices

Thus A[A_rowadr[r]+k] is the element of the underlying dense matrix at row r and column A_colind[A_rowadr[r]+k], where k < A_rownnz[r]. Normally m*n storage is not necessary (assuming the matrix is indeed sparse) but we allocate space for the worst-case scenario. Furthermore, in operations that can change the sparsity pattern, it is more efficient to spread out the data so that we do not have to perform many memory moves when inserting new data. We call this sparse layout "uncompressed". It is still a valid layout, but instead of A_rowadr[r] = A_rowadr[r-1] + A_rownnz[r] which is the standard convention, we set A_rowadr[r] = r*n. MuJoCo uses sparse matrices internally

To represent 3D orientations and rotations, MuJoCo uses unit quaternions - namely 4D unit vectors arranged as q = (w, x, y, z). Here (x, y, z) is the rotation axis unit vector scaled by sin(a/2), where a is the rotation angle in radians, and w = cos(a/2). Thus the quaternion corresponding to a null rotation is (1, 0, 0, 0). This is the default setting of all quaternions in MJCF.

MuJoCo also uses 6D spatial vectors internally. These are quantities in mjData prefixed with 'c', namely cvel, cacc, cdot, etc. They are spatial motion and force vectors that combine a 3D rotational component followed by a 3D translational component. We do not provide utility functions for working with them, and documenting them is beyond our scope here. See Roy Featherstone's webpage on Spatial Algebra. The unusual order (rotation before translation) is based on this material, and was apparently standard convention in the past.

The data structures mjModel and mjData contain many pointers to preallocated buffers. The constructors of these data structures (mj_makeModel and mj_makeData) allocate one large buffer, namely `mjModel.buffer` and `mjData.buffer`, and then partition it and set all the other pointers in it. mjData also contains a stack outside this main buffer, as discussed below. Even if two pointers appear one after the other, say `mjData.qpos` and `mjData.qvel`, do not assume that the data arrays are contiguous and there is no gap between them. The constructors implement byte-alignment for each data array, and skip bytes when necessary. So if you want to copy `mjData.qpos` and `mjData.qvel`, the correct way to do it is the hard way:

```
// do this
mju_copy(myqpos, d->qpos, m->nq);
mju_copy(myqvel, d->qvel, m->nv);

// DO NOT do this, there may be padding at the end of d->qpos
mju_copy(myqposqvel, d->qpos, m->nq + m->nv);
```

The X Macros defined in the optional header file `mjxmacro.h` can be used to automate allocation of data structure that match mjModel and mjData, for example when writing a MuJoCo wrapper for a scripting language. In the code sample testxml.cc we use these unusual macros to compare all data arrays from two instances of mjModel and find the one with the largest difference. Apparently X Macros were invented in the 1960's for assembly language, and remain a great idea.

Internal stack

MuJoCo allocates and manages its own stack of mjtNums. `mjData.stack` is the pointer to the preallocated memory buffer. `mjData.nstack` is the maximum number of mjtNums that the stack can hold, as determined by the nstack attribute of the size element in MJCF. `mjData.pstack` is the first available address in the stack; this is our custom stack pointer.

Most top-level MuJoCo functions allocate space on the stack, use it for internal computations, and then deallocate it. They cannot do this with the regular C stack because the allocation size is determined dynamically at runtime. And calling the heap memory management functions would be inefficient and result in fragmentation - thus a custom stack. When any MuJoCo function is called, upon return the value of `mjData.pstack` is the same. The only exception is the function mj_resetData and its variants: they set `mjData.pstack = 0`. Note that this function is called internally when an instability is detected in `mj_step`, `mj_step1` and `mj_step2`. So if user functions take advantage of the custom stack (as they should), this needs to be done in-between MuJoCo calls that have the potential to reset the simulation.

Below is the general template for using the custom stack in user code. This assumes that `mjData\* d` is defined in the scope. If not, saving and restoring the stack pointer should be done manually

instead of using the mjMARKSTACK and mjFREESTACK macros.

```
// save stack pointer in the "hidden" variable _mark
mjMARKSTACK

// allocate space
mjtNum* myqpos = mj_stackAlloc(d, m->nq);
mjtNum* myqvel = mj_stackAlloc(d, m->nv);

// restore stack from _mark
mjFREESTACK
```

The function mj_stackAlloc checks if there is enough space, and if so it advances the stack pointer, otherwise it triggers an error. It also keeps track of the maximum stack allocation; see diagnostics below.

Errors, warnings, memory allocation

When a terminal error occurs, MuJoCo calls the function mju_error internally. This function has a single argument which is the error message. The helper functions mju_error_i and mju_error_s are also used, but they simply construct the error message using a printf format string and an additional integer or string argument, and then call mju_error. Here is what mju_error does:

1. Append the error message at the end of the file MUJOCO_LOG.TXT in the program directory (create the file if it does not exist). Also write the date and time along with the error message.
2. If the user error callback mju_user_error is installed, call that function with the error message as argument. Otherwise printf the error message, printf “Press Enter to exit...”, getchar() and exit(1).

If a user error callback is installed, it must **not** return, otherwise the behavior of the simulator is undefined. The idea here is that if mju_error is called, the simulation cannot continue and the user is expected to make some change such that the error condition is avoided. The error messages are self-explanatory.

One situation where it is desirable to continue even after an error is an interactive simulator that fails to load a model file. This could be because the user provided the wrong file name, or because model compilation failed. This is handled by a special mechanism which avoids calling mju_error. The model loading functions mj_loadXML and mj_loadModel return NULL if the operation fails, and there is no need to exit the program. In the case of mj_loadXML there is an output argument containing the parser or compiler error that caused the failure, while mj_loadModel generates corresponding warnings (see below).

Internally mj_loadXML actually uses the mju_error mechanism, by temporarily installing a “user” handler that triggers a C++ exception, which is then intercepted. This is possible because the parser, compiler and runtime are compiled and linked together, and use the same copy of the C/C++ memory manager and standard library. If the user implements an error callback that triggers a C++ exception, this will be in their workspace which is not necessarily the same as the MuJoCo library workspace, and so it is not clear what will happen; the outcome probably depends on the compiler and platform. It is better to avoid this approach and simply exit when mju_error is called (which is the default behavior in the absence of a user handler).

MuJoCo can also generate warnings. They indicate conditions that are likely to cause numerical inaccuracies, but can also indicate problems with loading a model and other problematic situations where the simulator is nevertheless able to continue normal operation. The warning mechanism has two levels. The high-level is implemented with the function mj_warning. It registers a warning in mjData as explained in more detail in the diagnostics section below, and also calls the low-level function mju_warning. Alternatively, the low-level function may be called directly (from within mj_loadModel for example) without registering a warning in mjData. This is done in places where mjData is not available.

mju_warning does the following: if the user callback mju_user_warning is installed, it calls that callback. Otherwise it appends the warning message to MUJOCO_LOG.TXT and also does a printf, similar to mju_error but without exiting. When MuJoCo wrappers are developed for environments such as MATLAB, it makes sense to install a user callback which prints warnings in the command window (with mexPrintf).

When MuJoCo allocates and frees memory on the heap, it always uses the functions mju_malloc and mju_free. These functions call the user callbacks mju_user_malloc and mju_user_free when installed, otherwise they call the standard C functions malloc and free. The reason for this indirection is because users may want MuJoCo to use a heap under their control. In MATLAB for example, a user callback for memory allocation would use mxmalloc and mexMakeArrayPersistent.

Diagnostics

MuJoCo has several built-in diagnostics mechanisms that can be used to fine-tune the model. Their outputs are grouped in the diagnostics section at the beginning of mjData.

When the simulator encounters a situation that is not a terminal error but is nevertheless suspicious and likely to result in inaccurate numerical results, it triggers a warning. There are several possible warning types, indexed by the enum type mjtWarning. The array `mjData.warning` contains one mjWarningStat data structure per warning type, indicating how many times each warning type has been triggered since the last reset and any information about the warning (usually the index of the problematic model element). The counters are cleared upon reset. When a warning of a given type is first triggered, the warning text is also printed by mju_warning as documented in error and memory above. All this is done by the function mj_warning which the simulator calls internally when it encounters a warning. The user can also call this function directly to emulate a warning.

When a model needs to be optimized for high-speed simulation, it is important to know where in the pipeline the CPU time is spent. This can in turn suggest which parts of the model to simplify or how to design the user application. MuJoCo provides an extensive profiling mechanism. It involves multiple timers indexed by the enum type mjtTimer. Each timer corresponds to a top-level API function, or to a component of such a function. Similar to warnings, timer information accumulates and is only cleared on reset. The array `mjData.timer` contains one mjTimerStat data structure per timer. The average duration per call for a given timer (corresponding to `mj_step` in the example below) can be computed as:

```
mjtNum avtm = d->timer[mjTIMER_STEP].duration / mjMAX(1, d->timer[mjTIMER_STEP].number);
```

This mechanism is built into MuJoCo, but it only works when the timer callback mjcb_time is installed by the user. Otherwise all timer durations are 0. The reason for this design is because there is no platform-independent way to implement high-resolution timers in C without bringing in additional dependencies. Also, most of the time the user does not need timing, and in that case there is no reason to call timing functions.

One part of the simulation pipeline that needs to be monitored closely is the iterative constraint solver. The simplest diagnostic here is `mjData.solver_iter` which shows how many iterations the solver took on the last call to mj_step or `mj_forward`. Note that the solver has tolerance parameters for early termination, so this number is usually smaller than the maximum number of iterations allowed. The array `mjData.solver` contains one mjSolverStat data structure per iteration of the constraint solver, with information about the constraint state and line search.

When the option fwdinv is enabled in `mjModel.opt.enableflags`, the field `mjData.fwdinv` is also populated. It contains the difference between the forward and inverse dynamics, in terms of generalized forces and constraint forces. Recall that that the inverse dynamics use analytical formulas and are always exact, thus any discrepancy is due to poor convergence of the iterative solver in the forward dynamics. The numbers in `mjData.solver` near termination have similar order-of-magnitude as the numbers in `mjData.fwdinv`, but nevertheless these are two different diagnostics.

Since MuJoCo's runtime works with compiled models, memory is preallocated when a model is compiled or loaded. Recall the memory attribute of the size element in MJCF. It determines the preallocated space for dynamic arrays. How is the user supposed to know what the appropriate value is? If there were a reliable recipe we would have implemented it in the compiler, but there isn't one. The theoretical worst-case, namely all geoms contacting all other geoms, calls for huge allocation which is almost never needed in practice. Our approach is to provide default settings in MJCF which are sufficient for most models, and allow the user to adjust them manually with the above attribute. If the simulator runs out of dynamic memory at runtime it will trigger an error. When such errors are triggered, the user should increase memory. The field `mjData.maxuse_arena` is designed to help with this adjustment. It keeps track of the maximum arena use since the last reset. So one strategy is to make very large allocation, then monitor `mjData.maxuse_memory` statistics during typical simulations, and use it to reduce the allocation.

The kinetic and potential energy are computed and stored in `mjData.energy` when the corresponding flag in `mjModel.opt.enableflags` is set. This can be used as another diagnostic. In general, simulation instability is associated with increasing energy. In some special cases (when all unilateral constraints, actuators and dissipative forces are disabled) the underlying physical system is energy-conserving. In that case any temporal fluctuations in the total energy indicate inaccuracies in numerical integration. For such systems the Runge-Kutta integrator has much better performance than the default semi-implicit Euler integrator.

Finally, the user can implement additional diagnostics as needed. Two examples were provided in the code samples `testxml.cc` and `derivative.cc`, where we computed model mismatches after save and load, and assessed the accuracy of the numerical derivatives respectively. Key to such diagnostics is to implement two different algorithms or simulation paths that compute the same quantity, and compare the results numerically. This type of sanity check is essential when dealing with complex dynamical systems where we do not really know what the numerical output should be; if we knew that, we would not be using a simulator in the first place.

Jacobians

The derivative of any vector function with respect to its vector argument is called Jacobian. When this term is used in multi-joint kinematics and dynamics, it refers to the derivative of some spatial quantity as a function of the system configuration. In that case the Jacobian is also a linear map that operates on vectors in the (co)tangent space to the configuration manifold - such as velocities, momenta, accelerations, forces. One caveat here is that the system configuration encoded in `mjData.qpos` has dimensionality `mjModel.nq`, while the tangent space has dimensionality `mjModel.nv`, and the latter is smaller when quaternion joints are present. So the size of the Jacobian matrix is N-by-`mjModel.nv` where N is the dimensionality of the spatial quantity being differentiated.

MuJoCo can differentiate analytically many spatial quantities. These include tendon lengths, actuator transmission lengths, end-effector poses, contact and other constraint violations. In the case of tendons and actuator transmissions the corresponding quantities are `mjData.ten_moment` and `mjData.actuator_moment`; we call them moment arms but mathematically they are Jacobians. The Jacobian matrix of all scalar constraint violations is stored in `mjData.efc_J`. Note that we are talking about constraint violations rather than the constraints themselves. This is because constraint violations have units of length, i.e., they are spatial quantities that we can differentiate. Constraints are more abstract entities and it is not clear what it means to differentiate them.

Beyond these automatically-computed Jacobians, we provide support functions allowing the user to compute additional Jacobians on demand. The main function for doing this is mj_jac. It is given a 3D point and a MuJoCo body to which this point is considered to be attached. `mj_jac` then computes both the translational and rotational Jacobians, which tell us how a spatial frame anchored at the given point will translate and rotate if we make a small change to the kinematic configuration. More precisely, the Jacobian maps joint velocities to end-effector velocities, while the transpose of the Jacobian maps end-effector forces to joint forces. There are also several other `mj_jacXXX` functions; these are convenience functions that call the main `mj_jac` function with different points of interest - such as a body center of mass, geom center etc.

The ability to compute end-effector Jacobians exactly and efficiently is a key advantage of working in joint coordinates. Such Jacobians are the foundation of many control schemes that map end-effector errors to actuator commands suitable for suppressing those errors. The computation of end-effector Jacobians in MuJoCo via the `mj_jac` function is essentially free in terms of CPU cost; so do not hesitate to use this function.

Contacts

Collision detection and solving for contact forces were explained in detail in the Computation chapter. Here we further clarify contact processing from a programming perspective.

The collision detection stage finds contacts between geoms, and records them in the array `mjData.contact` of mjContact data structures. They are sorted such that multiple contacts between the same pair of bodies are contiguous (note that one body can have multiple geoms attached to

it), and the body pairs themselves are sorted such that the first body acts as the major index and the second body as the minor index. Not all detected contacts are included in the contact force computation. When a contact is included, its mjContact.exclude field is 0, and its mjContact.efc_address is the address in the list of active scalar constraints. Reasons for exclusion can be the gap attribute of geom, as well as certain kinds of internal processing that use virtual contacts for intermediate computations.

The list `mjData.contact` is generated by the position stage of both forward and inverse dynamics. This is done automatically. However the user can override the internal collision detection functions, for example to implement non-convex mesh collisions, or to replace some of the convex collision functions we use with geom-specific primitives beyond the ones provided by MuJoCo. The global 2D array mjCOLLISIONFUNC contains the collision function pointer for each pair of geom types (in the upper-left triangle). To replace them, simply set these pointers to your functions. The collision function type is mjfCollision. When user collision functions detect contacts, they should construct an mjvContact structure for each contact and then call the function mj_addContact to add that contact to `mjData.contact`. The reference documentation of mj_addContact explains which fields of mjContact must be filled in by custom collision functions. Note that the functions we are talking about here correspond to near-phase collisions, and are called only after the list of candidate geom pairs has been constructed by the internal broad-phase collision mechanism.

After the constraint forces have been computed, the vector of forces for contact `i` starts at:

```
mjtNum* contactforce = d->efc_force + d->contact[i].efc_address;
```

and similarly for all other `efc_XXX` vectors. Keep in mind that the contact friction cone can be pyramidal or elliptic, depending on which solver is selected in `mjModel.opt`. The function mj_isPyramidal can be used to determine which friction cone type is used. For pyramidal cones, the interpretation of the contact force (whose address we computed above) is non-trivial, because the components are forces along redundant non-orthogonal axes corresponding to the edges of the pyramid. The function mj_contactForce can be used to convert the force generated by a given contact into a more intuitive format: a 3D force followed by a 3D toque. The torque component will be zero when condim is 1 or 3, and non-zero otherwise. This force and torque are expressed in the contact frame given by mjContact.frame. Unlike all other matrices in mjData, this matrix is stored in transposed form. Normally a 3-by-3 matrix corresponding to a coordinate frame would have the frame axes along the columns. Here the axes are along the rows of the matrix. Thus, given that MuJoCo uses row-major format, the contact normal axis (which is the X axis of the contact frame by our convention) is in position mjContact.frame[0-2], the Y axis is in [3-5] and the Z axis is in [6-8]. The reason for this arrangement is because we can have frictionless contacts where only the normal axis is used, so it makes sense to have its coordinates in the first 3 positions of `mjContact.frame`.

Coordinate frames and transformations

There are multiple coordinate frames used in MuJoCo. The top-level distinction is between joint coordinates and Cartesian coordinates. The mapping from the vector of joints coordinates to the Cartesian positions and orientations of all bodies is called forward kinematics and is the first step in the physics pipeline. The opposite mapping is called inverse kinematics but it is not uniquely defined and is not implemented in MuJoCo. Recall that mappings between the tangent spaces (i.e., joint velocities and forces to Cartesian velocities and forces) are given by the body Jacobians.

Here we explain further subtleties and subdivisions of the coordinate frames, and summarize the available transformation functions. In joint coordinates, the only complication is that the position vector `mjData.qpos` has different dimensionality than the velocity and acceleration vectors `mjData.qvel` and `mjData.qacc` due to quaternion joints. The function mj_differentiatePos "subtracts" two joint position vectors and returns a velocity vector. Conversely, the function mj_integratePos takes a position vector and a velocity vector, and returns a new position vector which has been displaced by the given velocity.

Cartesian coordinates are more complicated because there are three different coordinate frames that we use: local, global, and com-based. Local coordinates are used in mjModel to represent the static offsets between a parent and a child body, as well as the static offsets between a body and any geoms, sites, cameras and lights attached to it. These static offsets are applied in addition to any joint transformations. So `mjModel.body_pos`, `mjModel.body_quat` and all other spatial quantities in mjModel are expressed in local coordinates. The job of forward kinematics is to accumulate the joint transformations and static offsets along the kinematic tree and compute all positions and orientations in global coordinates. The quantities in mjData that start with "x" are expressed in global coordinates. These are `mjData.xpos`, `mjData.geom_xpos` etc. Frame orientations are usually stored as 3-by-3 matrices (xmat), except for bodies whose orientation is also stored as a unit quaternion `mjData.xquat`. Given this body quaternion, the quaternions of all other objects attached to the body can be reconstructed by a quaternion multiplication. The function mj_local2Global converts from local body coordinates to global Cartesian coordinates.

mju_negPose and mju_trnVecPose. A pose is a grouping of a 3D position and a unit quaternion orientation. There is no separate data structure; the grouping is in terms of logic. This represents a position and orientation in space, or in other words a spatial frame. Note that OpenGL uses 4-by-4 matrices to represent the same information, except here we use a quaternion for orientation. The function mju_mulPose multiplies two poses, meaning that it transforms the first pose by the second pose (the order is important). `mju_negPose` constructs the opposite pose, while `mju_trnVecPose` transforms a 3D vector by a pose, mapping it from local coordinates to global coordinates if we think of the pose as a coordinate frame. If we want to manipulate only the orientation part, we can do that with the analogous quaternion utility functions mju_mulQuat, mju_negQuat and mju_rotVecQuat.

Finally, there is the com-based frame. This is used to represent 6D spatial vectors containing a 3D angular velocity or acceleration or torque, followed by a 3D linear velocity or acceleration or force. Note the backwards order: rotation followed by translation. `mjData.cdof` and `mjData.cacc` are example of such vectors; the names start with "c". These vectors play a key role in the multi-joint dynamics computation. Explaining this is beyond our scope here; see Featherstone's excellent slides on the subject. In general, the user should avoid working with such quantities directly. Instead use the functions mj_objectVelocity, mj_objectAcceleration and the low-level mju_transformSpatial to obtain linear and angular velocities, accelerations and forces for a given body. Still, for the interested reader, we summarize the most unusual aspect of the "c" quantities. Suppose we want to represent a body spinning in place. One might expect a spatial velocity that has non-zero angular velocity and zero linear velocity. However this is not the case. The rotation is

interpreted as taking place around an axis through the center of the coordinate frame, which is outside the body (we use the center of mass of the kinematic tree). Such a rotation will not only rotate the body but also translate it. Therefore the spatial vector must have non-zero linear velocity to compensate for the side-effect of rotation around an off-body axis. If you call mj_objectVelocity, the resulting 6D quantity will be represented in a frame that is centered at the body and aligned with the world. Thus the linear component will now be zero as expected. This function will also put translation in front of rotation, which is our convention for local and global coordinates.

Visualization

MuJoCo has a native 3D visualizer. Its use is illustrated in the simulate.cc code sample and in the simpler basic.cc code sample. While it is not a full-featured rendering engine, it is a convenient, efficient and reasonably good-looking visualizer that facilitates research and development. It renders not only the simulation state but also decorative elements such as contact points and forces, equivalent inertia boxes, convex hulls, kinematic trees, constraint violations, spatial frames and text labels; these can provide insight into the physics simulation and help fine-tune the model.

The visualizer is tightly integrated with the simulator and supports both onscreen and offscreen rendering, as illustrated in the record.cc code sample. This makes it suitable for synthetic computer vision and machine learning applications, especially in could environments. VR integration is also available as of MuJoCo version 1.40, facilitating applications that utilize new head-mounted displays such as Oculus Rift and HTC Vive.

Visualization in MuJoCo is a two-stage process:

Abstract visualization and interaction
: This stage populates the mjvScene data structure with a list of geometric objects, lights, cameras and everything else needed to produce a 3D rendering. It also provides abstract keyboard and mouse hooks for user interaction. The relevant data structure and function names have the prefix `mjv`.

OpenGL rendering
: This stage takes the mjvScene data structure populated in the abstract visualization stage, and renders it. It also provides basic 2d drawing and framebuffer access, so that most applications would not need to call OpenGL directly. The relevant data structure and function names have the prefix `mjr`.

There are several reasons for this separation. First, the two stages are conceptually different and separating them is good software design. Second, they have different dependencies, both internally and in terms of additional libraries; in particular, abstract visualization does not require any graphics libraries. Third, users who wish to integrate another rendering engine with MuJoCo can bypass the native OpenGL renderer but still take advantage of the abstract visualizer.

Below is a mixture of C code and pseudo-code in comments, illustrating the structure of a MuJoCo application which does both simulation and rendering. This is a short version of the basic.cc code sample. For concreteness we assume that GLFW is used, although it can be replaced with a different window library such as GLUT or one of its derivatives.

```
// MuJoCo data structures
mjModel* m = NULL;                  // MuJoCo model
mjData* d = NULL;                   // MuJoCo data
mjvCamera cam;                      // abstract camera
mjvOption opt;                      // visualization options
mjvScene scn;                       // abstract scene
mjrContext con;                     // custom GPU context

// ... load model and data

// init GLFW, create window, make OpenGL context current, request v-sync
glfwInit();
GLFWwindow* window = glfwCreateWindow(1200, 900, "Demo", NULL, NULL);
glfwMakeContextCurrent(window);
glfwSwapInterval(1);

// initialize visualization data structures
mjv_defaultCamera(&cam);
mjv_defaultPerturb(&pert);
mjv_defaultOption(&opt);
mjr_defaultContext(&con);

// create scene and context
mjv_makeScene(m, &scn, 1000);
mjr_makeContext(m, &con, mjFONTSCALE_100);

// ... install GLFW keyboard and mouse callbacks

// run main loop, target real-time simulation and 60 fps rendering
while( !glfwWindowShouldClose(window) ) {
  // advance interactive simulation for 1/60 sec
  //  Assuming MuJoCo can simulate faster than real-time, which it usually can,
  //  this loop will finish on time for the next frame to be rendered at 60 fps.
  //  Otherwise add a cpu timer and exit this loop when it is time to render.
  mjtNum simstart = d->time;
  while( d->time - simstart < 1.0/60.0 )
      mj_step(m, d);

  // get framebuffer viewport
  mjrRect viewport = {0, 0, 0, 0};
  glfwGetFramebufferSize(window, &viewport.width, &viewport.height);

  // update scene and render
  mjv_updateScene(m, d, &opt, NULL, &cam, mjCAT_ALL, &scn);
  mjr_render(viewport, &scn, &con);

  // swap OpenGL buffers (blocking call due to v-sync)
  glfwSwapBuffers(window);

  // process pending GUI events, call GLFW callbacks
  glfwPollEvents();
}

// close GLFW, free visualization storage
glfwTerminate();
```

```
mjv_freeScene(&scn);
mjr_freeContext(&con);

// ... free MuJoCo model and data
```

Abstract visualization and interaction

This stage populates the mjvScene data structure with a list of geometric objects, lights, cameras and everything else needed to produce a 3D rendering. It also provides abstract keyboard and mouse hooks for user interaction.

Cameras

There are two types of camera objects: an abstract camera represented with the stand-alone data structure mjvCamera, and a low-level OpenGL camera represented with the data structure mjvGLCamera which is embedded in mjvScene. When present, the abstract camera is used during scene update to automatically compute the OpenGL camera parameters, which are then used by the OpenGL renderer. Alternatively, the user can bypass the abstract camera mechanism and set the OpenGL camera parameters directly, as discussed in the Virtual Reality section below.

The abstract camera can represent three different camera types as determined by mjvCamera.type. The possible settings are defined by the enum mjtCamera:

mjCAMERA_FREE
: This is the most commonly used abstract camera. It can be freely moved with the mouse. It has a lookat point, distance to the lookat point, azimuth and elevation; twist around the line of sight is not allowed. The function mjv_moveCamera is a mouse hook for controlling all these camera properties interactively with the mouse. When simulate.cc first starts, it uses the free camera.

mjCAMERA_TRACKING
: This is similar to the free camera, except the lookat point is no longer a free parameter but instead is coupled to the MuJoCo body whose id is given by mjvCamera.trackbodyid. At each update, the lookat point is set to the center of mass of the kinematic subtree rooted at the specified body. There is also some filtering which produces smooth camera motion. The distance, azimuth and elevation are controlled by the user and are not modified automatically. This is useful for tracking a body as it moves around, without turning the camera. To switch from the free to the tracking camera in simulate.cc, hold Ctrl and right-double-click on the body of interest. Press Esc to go back to the free camera.

mjCAMERA_FIXED
: This refers to a camera explicitly defined in the model, unlike the free and tracking cameras which only exist in the visualizer and are not defined in the model. The id of the model camera is given by mjvCamera.fixedcamid. This camera is fixed in the sense that the visualizer cannot change its pose or any other parameters. However the simulator computes the camera pose at each time step, and if the camera is attached to a moving body or is in tracking or targeting mode, it will move.

mjCAMERA_USER
: This means that the abstract camera is ignored during an update and the low-level OpenGL cameras are not changed. It is equivalent to not specifying an abstract camera at all, i.e., passing a NULL pointer to mjvCamera in the update functions explained below.

The low-level mjvGLCamera is what determines the actual rendering. There are two such cameras embedded in mjvScene, one for each eye. Each has position, forward and up directions. Forward corresponds to the negative Z axis of the camera frame, while up corresponds to the positive Y axis. There is also a frustum in the sense of OpenGL, except we store the average of the left and right frustum edges and then during rendering compute the actual edges from the viewport aspect ratio assuming 1:1 pixel aspect ratio. The distance between the two camera positions corresponds to the inter-pupilary distance (ipd). When the low-level camera parameters are computed automatically from an abstract camera, the ipd as well as vertical field of view (fovy) are taken from `mjModel.vis.global.ipd`/`fovy` for free and tracking cameras, and from the camera-specific `mjModel.cam_ipd/fovy` for cameras defined in the model. When stereoscopic mode is not enabled, as determined by mjvScene.stereo, the camera data for the two eyes are internally averaged during rendering.

Selection

In many applications we need to click on a point and determine the 3D object to which this point/pixel belongs. This is done with the function mjv_select. Prior to MuJoCo 1.50 this function (called mjr_select) used OpenGL rendering in a special mode to recover the object identity and 3D position of the clicked point. Now it uses a new collision detection module that intersects a ray with all geoms in the model. This is actually engine-level functionality and does not depend on the visualizer (indeed it is also used to simulate rangefinder sensors independent of visualization), but the select function is implemented in the visualizer because it needs information about the camera and viewport.

The function mjv_select returns the index of the geom at the specified window coordinates, or -1 if there is no geom at those coordinates. The 3D position is also returned. See the code sample simulate.cc for an example of how to use this function. Internally, mjv_select calls the engine-level function mj_ray which in turn calls the per-geom functions mj_rayMesh, mj_rayHfield and mju_rayGeom. The user can implement custom selection mechanisms by calling these functions directly. In a VR application for example, it would make sense to use the hand-held controller as a "laser pointer" that can select objects.

Perturbations

Interactive perturbations have proven very useful in exploring the model dynamics as well as probing closed-loop control systems. The user is free to implement any perturbation mechanism of their choice by setting `mjData.qfrc_applied` or `mjData.xfrc_applied` to suitable forces (in generalized and Cartesian coordinates respectively).

Prior to MuJoCo version 1.40, user code had to maintain a collection of objects in order to implement perturbations. All these objects are now grouped into the data structure mjvPerturb. Its use is illustrated in simulate.cc. The idea is to select a MuJoCo body of interest, and provide a reference pose (i.e., a 3D position and quaternion orientation) for that body. These are stored in mjPerturb.refpos/quat. The function mjv_movePerturb is a mouse hook for controlling the reference

pose with the mouse. The function mjv_initPerturb is used to set the reference pose equal to the selected body pose at the onset of perturbation, so as to avoid jumps.

This perturbation object can then be used to move the selected body directly (when the simulation is paused or when the selected body is a mocap body), or to apply forces and torques to the body. This is done with the functions mjv_applyPerturbPose and mjv_applyPerturbForce respectively. The latter function writes the external perturbation force to `mjData.xfrc_applied` for the selected body. However it does not clear `mjData.xfrc_applied` for the remaining bodies, thus it is recommended to clear it in user code, in case the selected body changed and some perturbation force was left over from a previous time step. If there is more than one device that can apply perturbations or user code needs to add perturbations from other sources, the user must implement the necessary logic so that only the desired perturbations are present in `mjData.xfrc_applied` and any old perturbations are cleared.

In addition to affecting the simulation, the perturbation object is recognized by the abstract visualizer and can be rendered. This is done by adding a visual string to denote the positional difference, and a rotating cube to denote the reference orientation of the selected body. The perturbation forces themselves can also be rendered when the corresponding visualization flag in mjvOption is enabled.

Scene update

Finally, we bring all of the above elements together and explain how mjvScene is updated before being passed to the OpenGL rendering stage. This can be done with a single call to the function mjv_updateScene at each frame. mjvCamera and mjvPerturb are arguments to this function, or they can be NULL pointers in which case the corresponding functionality is disabled. In VR applications the parameters of mjvScene.camera[n], n=0,1 must also be set at each frame; this is done by user code outside mjv_updateScene. The function mjv_updateScene examines mjModel and mjData, constructs all geoms that need to be rendered (according to the specified visualization options), and populates the array mjvScene.geom with mjvGeom objects. Note that mjvGeom is an abstract geom, and is not in one-to-one correspondence with the simulation geoms in mjModel and mjData. In particular, mjvGeom contains the geom pose, scale, shape (primitive or mesh index in mjModel), material properties, textures (index in mjModel), labeling, and everything else needed for specify how rendering should be done. mjvScene also contains up to eight OpenGL lights which are copied from the model, as well as a headlight which is in light position 0 when present.

The above procedure is the most common approach, and it updates the entire scene at each frame. In addition, we provide two functions for finer control. mjv_updateCamera updates only the camera (i.e., maps the abstract mjvCamera to the low-level mjvGLCamera) but does not touch the geoms or lights. This is useful when the user is moving the camera rapidly but the simulation state has not changed - in that case there is no point in re-creating the lists of geoms and lights.

More advanced rendering effects can be achieved by manipulating the list of abstract geoms. For example, the user can add custom geoms at the end of the list. Sometimes it is desirable to render a sequence of simulation states (i.e., a trajectory) and not just the current state. For this purpose, we have provided the function mjv_addGeoms which adds the geoms corresponding to the current simulation state to the list already in mjvScene. It does not change the list of lights, because lighting is additive and having too many lights will make the scene too bright. Importantly, the user can select which geom categories will be added, via a bitmask of enum type mjtCatBit:

mjCAT_STATIC
: This selects MuJoCo geoms and sites belonging to the world body (which has body id 0).

mjCAT_DYNAMIC
: This selects MuJoCo geoms and sites belonging to bodies other than the world body.

mjCAT_DECOR
: This selects decorative elements such as force arrows, automatically-generated skeletons, equivalent inertia boxes, and any other elements that were added by the abstract visualizer and do not correspond to MuJoCo geoms and sites defined in the model.

mjCAT_ALL
: This selects all of the above categories.

The main update function mjv_updateScene would normally be called with mjCAT_ALL. It clears the geom list and calls mjv_addGeom to add only the geoms for the current model state. If we want to render a trajectory, we have to be careful to avoid visual clutter. So it makes sense to render one of the frames with mjCAT_ALL (usually the first or the last depending on the use case), and all other frames with mjCAT_DYNAMIC. Since the static/world objects are not moving, rendering them in each frame will only slow down the GPU and create visual aliasing. As for the decor elements, there could be situations where we want to render all of them - for example to visualize the evolution of contact forces over time. In summary, there is plenty of flexibility in how mjvScene is constructed. We have provided automation for the main use cases, but the user can also make programmatic changes as needed.

Virtual reality

In desktop applications it is convenient to use an abstract mjvCamera allowing intuitive mouse control, and then automatically map it to mjvGLCamera used for rendering. In VR applications the situation is very different. In that case the head/eyes of the user as well as the projection surface are being tracked, and therefore have physical presence in the room. If anything can be moved by the user (with a mouse or other input device) it is the position, orientation and scale of the model relative to the room. This is called model transformation, and is represented in mjvScene. The function mjv_moveModel is a mouse hook for controlling this transformation. When using an abstract mjvCamera during update, the model transformation is automatically disabled, by setting the flag mjvScene.enabletransform = 0 rather than clearing the actual parameters. In this way the user can switch between VR and desktop camera mode without losing the model transformation parameters.

Since we have introduced two spaces, namely model space and room space, we need to map between them as well as clarify which spatial quantities are defined with respect to which spatial frame. Everything accessible by the simulator lives in the model space. The room space is only accessible by the visualizer. The only quantities defined in room space are the mjvGLCamera parameters. The functions mjv_room2model, mjv_model2room, mjv_cameraInModel, mjv_cameraInRoom perform the necessary transformations, and are needed for VR applications.

We now outline the procedure for hooking up head tracking to MuJoCo's visualizer in a VR application. A code sample illustrating this will soon be posted. We assume that a tracking device provides in real-time the positions of the two eyes (usually generated by tracking the position and orientation of the head and assuming a user-specific ipd), as well as the forward and up camera directions. We copy these data directly into the two mjvGLCameras, which are in mjvScene.camera[n] where n=0 is the left eye and n=1 is the right eye. Note that the forward direction is normal to the projection surface, and not necessarily aligned with the gaze direction; indeed the gaze direction is unknown (unless we also have an eye-tracking device) and does not affect the rendering.

We must also set the mjvGLCamera frustum. How this is done depends on the nature of the VR system. For head-mounted displays such as the Oculus Rift and HTC Vive, the projection surface moves with the head, and so the frustum is fixed and provided by the SDK. In this case we simply copy it into mjvGLCamera, averaging the left and right edges to compute the frustum_center parameter. Alternatively, the projection surface can be a monitor which is stationary in the room (which is the case in the zSpace system). For such systems we must compute the frustum at each frame, by taking into account the spatial relations between the monitor and the eyes/cameras. This assumes that the monitor is also tracked. The natural approach here is to define the monitor as the center of the room coordinate frame, and track the head relative to it. In the zSpace system this is done by embedding the motion capture cameras in the monitor itself.

Apart from tracking the head and using the correct perspective projection, VR applications typically involve hand-held spatial controllers that must be mapped to the motion of simulated objects or otherwise interact with the simulation. The pose of these controllers is recorded by the motion capture system in room space. The transformation functions we provide (mjv_room2model in particular) can be used to map to model space. Once we have the pose of the controller in model space, we can use a MuJoCo mocap body (defined in the model) to insert the controller in the simulation. This is precisely why mocap bodies were introduced in MuJoCo. Such bodies are treated as fixed from the viewpoint of physics, yet the user is expected to move them programmatically at each simulation step. They can interact with the simulation through contacts, or better yet, through soft equality constraints to regular bodies which in turn make contacts. The latter approach is illustrated in the MPL models available on the Forum. It provides effective dynamic filtering and avoids contacts involving bodies that behave as if they are infinitely heavy (which is what a fixed body is). Note that the time-varying positions and orientations of the mocap bodies are stored in `mjData.mocap_pos/quat`, as opposed to storing them in mjModel. This is because mjModel is supposed to remain constant. The fixed mocap body pose stored in mjModel is only used at initialization and reset, when user code has not yet had a chance to update mjData.mocap_pos/quat.

OpenGL Rendering

This stage takes the mjvScene data structure populated in the abstract visualization stage, and renders it. It also provides basic 2d drawing and framebuffer access, so that most applications would not need to call OpenGL directly.

Context and GPU resources

The first step in the rendering process is create the model-specific GPU context mjrContext. This is done by first clearing the data structure with the function mjr_defaultContext, and then calling the function mjr_makeContext. This was already illustrated earlier; the relevant code is:

```
mjModel* m;
mjrContext con;

// clear mjrContext only once before first use
mjr_defaultContext(&con);

// create window with OpenGL context, make it current
GLFWwindow* window = glfwCreateWindow(1200, 900, "Demo", NULL, NULL);
glfwMakeContextCurrent(window);

// ... load MuJoCo model

// make model-specific mjrContext
mjr_makeContext(m, &con, mjFONTSCALE_100);

// ... load another MuJoCo model

// make mjrContext for new model (old context freed automatically)
mjr_makeContext(m, &con, mjFONTSCALE_100);

// free context when done
mjr_freeContext(&con);
```

How is mjrContext related to an OpenGL context? An OpenGL context is what enables the application to talk to the video driver and send rendering commands. It must exist and must be current in the calling thread before mjr_makeContext is called. GLFW and related libraries provide the necessary functions as shown above.

mjrContext is specific to MuJoCo. After creation, it contains references (called "names" in OpenGL) to all the resources that were uploaded to the GPU by mjr_makeContext. These include model-specific resources such as meshes and textures, as well as generic resources such as font bitmaps for the specified font scale, framebuffer objects for shadow mapping and offscreen rendering, and associated renderbuffers. It also contains OpenGL-related options copied from `mjModel.vis`, capabilities of the default window framebuffer that are discovered automatically, and the currently active buffer for rendering; see buffers below. Note that even though MuJoCo uses fixed-function OpenGL, it avoids immediate mode rendering and instead uploads all resources to the GPU upfront. This makes it as efficient as a modern shader, and possibly more efficient, because fixed-function OpenGL is now implemented via internal shaders that have been written by the video driver developers and tuned extensively.

Most of the fields of mjrContext remain constant after the call to mjr_makeContext. The only exception is mjrContext.currentBuffer which changes whenever the active buffer changes. Some of the GPU resources may also change because the user can upload modified resources with the functions mjr_uploadTexture, mjr_uploadMesh, mjr_uploadHField. This can be used to achieve dynamic effects such as inserting a video feed into the rendering, or modulating a terrain map. Such modifications affect the resources residing on the GPU, but their OpenGL names are reused, thus the change is not actually visible in mjrContext.

The user should **never** make changes to mjrContext directly. MuJoCo's renderer assumes that only it can manage mjrContext. In fact this kind of object would normally be opaque and its internal structure would not be exposed to the user. We are exposing it because MuJoCo has an open design and also because users may want to interleave their own OpenGL code with MuJoCo's renderer, in which case they may need read access to some fields of mjrContext. For example in VR applications the user needs to blit from MuJoCo's offscreen buffer to a texture provided by a VR SDK.

When a different MuJoCo model is loaded, mjr_makeContext must be called again. There is also the function mjr_freeContext which frees the GPU resources while preserving the initialization and capabilities flags. This function should be called when the application is about to exit. It is called automatically from within mjr_makeContext, so you do not need to call it directly when a different model is loaded, although it is not an error to do so. The function mjr_defaultContext must be called once before rendering starts, to clear the memory allocated for the data structure mjrContext. If you call it after calling mjr_makeContext, it will wipe out any record that GPU resources were allocated without freeing those resources, so don't do that.

Buffers for rendering

In addition to the default window framebuffer, OpenGL can support unlimited framebuffer objects (FBOs) for custom rendering. In MuJoCo we provide systematic support for two framebuffers: the default window framebuffer, and one offscreen framebuffer. They are referred to by the constants in the enum type mjtFramebuffer, namely mjFB_WINDOW and mjFB_OFFSCREEN. At any time, one of these two buffers is active for the purposes of MuJoCo rendering, meaning that all subsequent commands are directed to it. There are two additional framebuffer objects referenced in mjrContext, needed for shadow mapping and resolving multi-sample buffers, but these are used internally and the user should not attempt to access them directly.

The active buffer is set with the function mjr_setBuffer. This sets the value of mjrContext.activeBuffer and configures the OpenGL state accordingly. When mjr_makeContext is called, internally it calls mjr_setBuffer with argument mjFB_WINDOW, so that rendering starts in the window buffer by default. If the specified buffer does not exist, mjr_setBuffer automatically defaults to the other buffer (note that when using headless rendering on Linux, there may be no window framebuffer).

From the perspective of OpenGL, there are important differences between the window framebuffer and offscreen framebuffer, and these differences affect how the MuJoCo user interacts with the renderer. The window framebuffer is created and managed by the operating system and not by OpenGL. As a result, properties such as resolution, double-buffering, quad-buffered stereo, mutli-samples, v-sync are set outside OpenGL; this is done by GLFW calls in our code samples. All OpenGL can do is detect these properties; we do this in mjr_makeContext and record the results in the various window capabilities fields of mjrContext. This is why such properties are not part of the MuJoCo model; they are session/software-specific and not model-specific. In contrast, the offscreen framebuffer is managed entirely by OpenGL, and so we can create that buffer with whatever properties we want, namely with the resolution and multi-sample properties specified in `mjModel.vis`.

The user can directly access the pixels in the two buffers. This is done with the functions mjr_readPixels, mjr_drawPixels and mjr_blitBuffer. Read/draw transfer pixels from/to the active buffer to/from the CPU. Blit transfers pixels between the two buffers on the GPU and is therefore much faster. The direction is from the active buffer to the buffer that is not active. Note that mjr_blitBuffer has source and destination viewports that can have different size, allowing the image to be scaled in the process.

Drawing pixels

The main rendering function is mjr_render. Its arguments are a rectangular viewport for rendering, the mjvScene which was populated by the abstract visualizer, and the mjrContext which was created by mjr_makeContext. The viewport can be the entire active buffer, or part of it for custom effects. A viewport corresponding to the entire buffer can be obtained with the function mjr_maxViewport. Note that while the offscreen buffer size does not change, the window buffer size changes whenever the user resizes or maximizes the window. Therefore user code should not assume fixed viewport size. In the code sample simulate.cc we use a callback which is triggered whenever the window size changes, while in basic.cc we simply check the window size every time we render. On certain scaled displays (only on OSX it seems) the window size and framebuffer size can be different. So if you are getting the size with GLFW functions, use glfwGetFramebuferSize rather than glfwGetWindowSize. On the other hand, mouse coordinates are returned by the operating system in window rather than framebuffer units; thus the mouse interaction functions discussed earlier should use glfwGetWindowSize to obtain the window height needed to normalize the mouse displacement data.

mjr_render renders all mjvGeoms from the list mjvScene.geom. The abstract visualization options mjvOption are no longer relevant here; they are used by mjv_updateScene to determine which geoms to add, and as far as mjr_render is concerned these options are already baked-in. There is however another set of rendering options that are embedded in mjvScene, and these affect the OpenGL rendering process. The array mjvScene.flags contains flags indexed by the enum type mjtRndFlag and include options for enabling and disabling wireframe mode, shadows, reflections, skyboxes and fog. Shadows and reflections involve additional rendering passes. MuJoCo's renderer is very efficient, but depending on the model complexity and available GPU, it may be necessary to disable one or both of these effects in some cases.

The parameter mjvScene.stereo determines the stereo mode. The possible values are given by the enum type mjtStereo and are as follows:

mjSTEREO_NONE
: Stereo rendering is disabled. The average of the two OpenGL cameras in mjvScene is used. Note that the renderer always expects both cameras to be properly defined, even if stereo is not used.

mjSTEREO_QUADBUFFERED
: This mode works only when the active buffer is the window, and the window supports quad-buffered OpenGL. This requires a professional video card. The code sample simulate.cc attempts to open such a window. In this mode MuJoCo's renderer uses the GL_BACK_LEFT and GL_BACK_RIGHT buffers to render the two views (as determined by the two OpenGL cameras in mjvScene) when the window is double-buffered, and GL_FRONT_LEFT and GL_FRONT_RIGHT otherwise. If the window does not support quad-buffered OpenGL or the

active buffer is the offscreen buffer, the renderer reverts to the side-by-side mode described next.

mjSTEREO_SIDEBYSIDE
: This stereo mode does not require special hardware and is always available. The viewport given to mjr_render is split in two equal rectangles side-by-side. The left view is rendered on the left side and the right view on the right side. In principle users can cross their eyes and see stereo on a regular monitor, but the goal here is to show it in a stereoscopic device. Most head-mounted displays support this stereo mode.

In addition to the main mjr_render function, we provide several functions for "decorating" the image. These are 2d rendering functions and include mjr_overlay, mjr_text, mjr_rectangle, mjr_figure. The user can draw additional decorations with their own OpenGL code. This should be done after mjr_render, because mjr_render clears the viewport.

We also provide the functions mjr_finish and mjr_getError for explicit synchronization with the GPU and for OpenGL error checking. They simply call glFinish and glGetError internally. This together with the basic 2d drawing functions above is meant to provide enough functionality so that most users will not need to write OpenGL code. Of course we cannot achieve this in all cases, short of providing wrappers for all of OpenGL.