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 .`.

As a reference, a working build configuration can be found in MuJoCo's [continuous integration setup](https://github.com/deepmind/mujoco/blob/main/.github/workflows/build.yml) on GitHub.

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.