

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) 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 mjData 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 n  **stable** ▼
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)  **stable** ▼ 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 `timestep` diagnostics. The main simulation function is



stable ▼

```

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  **stable**  `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

 [stable](#) ▼

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 LDL 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 `se` are omitted. The functions being called are components of the simulator, and 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.

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 dy  [stable](#) 
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_actuator
```

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 “`fwdin`” 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 an 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.

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 thr

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

 [stable](#) ▼

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

// 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. 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.

 **stable** ▼

Model changes

Model editing framework

The discussion below regarding `mjModel` changes at runtime was written before the 3.2.0 introduction of the [Model Editing](#) framework. It is still valid, but the new framework is the safe and recommended way to modify models.

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_data` 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 `mjSpec`, 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 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

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. `qM` uses a custom indexing format designed for matrices that correspond to tree topology, while `qLD` uses the standard CSR format. `qM` will be migrated to CSR in an upcoming change. 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



Variable	Size	Meaning
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.

Internal stack

MuJoCo allocates and manages dynamic memory in an “arena” space in `mjData.arena`. The arena memory space contains two types of dynamically allocated memory:

- Memory related to constraints, since the number of contacts is unknown at the beginning of a step.
- Memory for temporary variables, managed by an internal stack mechanism.

See [Memory allocation](#) for details regarding the layout of the arena and internal stack.

Most top-level MuJoCo functions allocate space on the `mjData` 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. 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, 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.

```
// mark an mjData stack frame
mj_markStack(d);

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

// restore the mjData stack frame
mj_freeStack(d);
```



The function [mj_stackAllocNum](#) 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. Note that [mj_stackAllocNum](#) is only used for allocating `mjtNum` arrays, the most common type of array. [mj_stackAllocInt](#) is provided for integer array allocation, and [mj_stackAllocByte](#) is provided for allocation of arbitrary number of bytes and alignment.



Errors and warnings

When a terminal error occurs, MuJoCo calls the function [mju_error](#) internally. 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, print the error message and "Press Enter to exit..." to standard output. Then wait for any keyboard input, and then terminate the simulator with failure.

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  **stable**  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 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 `fwdiv` is enabled in `mjModel.opt.enableflags`, the field `mjData.fwdiv` is also populated. It contains the difference between the forward and inverse dynamics, in terms of generalized forces and constraint forces. Recall 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.fwdiv`, 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.

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  **stable**  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.

 **stable** ▼

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 torque. 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 transform

 stable ▼

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.

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_invPose` constructs the opposite pose, while `mju_trnVecPose` transforms a 3D vector 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.

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