Globals

Global variable and constant definitions can be classified as:

- Callbacks:
  - Error callbacks.
  - Memory callbacks.
  - Physics callbacks.
- The collision table containing narrow-phase collision functions.
- String constants.
- Numeric constants.
- Macros.
- X Macros.

Error callbacks

All user callbacks (i.e., global function pointers whose name starts with 'mjcb') are initially set to NULL, which disables them and allows the default processing to take place. To install a callback, simply set the corresponding global pointer to a user function of the correct type. Keep in mind that these are global and not model-specific. So if you are simulating multiple models in parallel, they use the same set of callbacks.

mju_user_error

This is called from within the main error function mju_error. When installed, this function overrides the default error processing. Once it prints error messages (or whatever else the user wants to do), it must **exit** the program. MuJoCo is written with the assumption that mju_error will not return. If it does, the behavior of the software is undefined.

```
extern void (*mju_user_error)(const char*);
```

mju_user_warning

stable

This is called from within the main warning function mju_warning. It is similar to the error handler, but instead it must return without exiting the program.

```
extern void (*mju_user_warning)(const char*);
```

Memory callbacks

The purpose of the memory callbacks is to allow the user to install custom memory allocation and deallocation mechanisms. One example where we have found this to be useful is a MATLAB wrapper for MuJoCo, where mex files are expected to use MATLAB's memory mechanism for permanent memory allocation.

mju_user_malloc

If this is installed, the MuJoCo runtime will use it to allocate all heap memory it needs (instead of using aligned malloc). The user allocator must allocate memory aligned on 8-byte boundaries. Note that the parser and compiler are written in C++ and sometimes allocate memory with the "new" operator which bypasses this mechanism.

```
extern void* (*mju_user_malloc)(size_t);
```

mju_user_free

If this is installed, MuJoCo will free any heap memory it allocated by calling this function (instead of using aligned free).

```
extern void (*mju_user_free)(void*);
```

Physics callbacks

The physics callbacks are the main mechanism for modifying the behavior of the simulator, beyond setting various options. The options control the operation of the default pipeline, while callbacks extend the pipeline at well-defined places. This enables advanced users to implement many interesting functions which we have not thought of, while still taking advantage of the default pipeline. As with all other callbacks, there is no automated error checking - instead we assume that the authors of callback functions know what they are doing.

stable

Custom physics callbacks will often need parameters that are not standard in MJCF. This is largely why we have provided custom fields as well as user data arrays in MJCF.

The idea is to "instrument" the MJCF model by entering the necessary user parameters, and then write callbacks that look for those parameters and perform the corresponding computations. We strongly encourage users to write callbacks that check the model for the presence of user parameters before accessing them - so that when a regular model is loaded, the callback disables itself automatically instead of causing the software to crash.

mjcb_passive

This is used to implement a custom passive force in joint space; if the force is more naturally defined in Cartesian space, use the end-effector Jacobian to map it to joint space. By "passive" we do not mean a force that does no positive work (as in physics), but simply a force that depends only on position and velocity but not on control. There are standard passive forces in MuJoCo arising from springs, dampers, viscosity and density of the medium. They are computed in `mjData.qfrc_passive` before mjcb_passive is called. The user callback should add to this vector instead of overwriting it (otherwise the standard passive forces will be lost).

```
extern mjfGeneric mjcb_passive;
```

mjcb_control

This is the most commonly used callback. It implements a control law, by writing in the vector of controls `mjData.ctrl`. It can also write in `mjData.qfrc_applied` and `mjData.xfrc_applied`. The values written in these vectors can depend on position, velocity and all other quantities derived from them, but cannot depend on contact forces and other quantities that are computed after the control is specified. If the callback accesses the latter fields, their values do not correspond to the current time step.

The control callback is called from within mj_forward and mj_step, just before the controls and applied forces are needed. When using the RK integrator, it will be called 4 times per step. The alternative way of specifying controls and applied forces is to set them before `mj_step`, or use `mj_step1` and `mj_step2`. The latter approach allows setting the controls after the position and velocity computations have been performed by `mj_step1`, allowing these results to be utilized in computing the control (similar to using mjcb_control). However, the only way to change the controls between sub-steps of the RK integrator is to define the control callback.

```
extern mjfGeneric mjcb_control;
```

stable

mjcb_contactfilter

This callback can be used to replace MuJoCo's default collision filtering. When installed, this function is called for each pair of geoms that have passed the broad-phase test (or are predefined geom pairs in the MJCF) and are candidates for near-phase collision. The default processing uses the contype and conaffinity masks, the parent-child filter and some other considerations related to welded bodies to decide if collision should be allowed. This callback replaces the default processing, but keep in mind that the entire mechanism is being replaced. So for example if you still want to take advantage of contype/conaffinity, you have to re-implement it in the callback.

```
extern mjfConFilt mjcb_contactfilter;
```

mjcb_sensor

This callback populates fields of `mjData.sensordata` corresponding to user-defined sensors. It is called if it is installed and the model contains user-defined sensors. It is called once per compute stage (mjSTAGE_POS, mjSTAGE_VEL, mjSTAGE_ACC) and must fill in all user sensor values for that stage. The user-defined sensors have dimensionality and data types defined in the MJCF model which must be respected by the callback.

```
extern mjfSensor mjcb_sensor;
```

mjcb_time

Installing this callback enables the built-in profiler, and keeps timing statistics in `mjData.timer`. The return type is mjtNum, while the time units are up to the user. simulate.cc assumes the unit is 1 millisecond. In order to be useful, the callback should use high-resolution timers with at least microsecond precision. This is because the computations being timed are very fast.

```
extern mjfTime mjcb_time;
```

mjcb_act_dyn

This callback implements custom activation dynamics: it must return the value of `mjData.act_dot` for the specified actuator. This is the time-derivative of the activation state vector `mjData.act`. It is called for model actuators with user dynamics (mjDYN_USER). If such actuators exist in the model but the callback is not installed, their time-derivative is set to 0.

```
extern mjfAct mjcb_act_dyn;
```

stable

mjcb_act_gain

This callback implements custom actuator gains: it must return the gain for the specified actuator with `mjModel.actuator_gaintype` set to mjGAIN_USER. If such actuators exist in the model and this callback is not installed, their gains are set to 1.

```
extern mjfAct mjcb_act_gain;
```

mjcb_act_bias

This callback implements custom actuator biases: it must return the bias for the specified actuator with `mjModel.actuator_biastype` set to mjBIAS_USER. If such actuators exist in the model and this callback is not installed, their biases are set to 0.

```
extern mjfAct mjcb_act_bias;
```

Collision table

mjCOLLISIONFUNC

Table of pairwise collision functions indexed by geom types. Only the upper-right triangle is used. The user can replace these function pointers with custom routines, replacing MuJoCo's collision mechanism. If a given entry is NULL, the corresponding pair of geom types cannot be collided. Note that these functions apply only to near-phase collisions. The broadphase mechanism is built-in and cannot be modified.

```
extern mjfCollision mjCOLLISIONFUNC[mjNGEOMTYPES][mjNGEOMTYPES];
```

String constants

The string constants described here are provided for user convenience. They correspond to the English names of lists of options, and can be displayed in menus or dialogs in a GUI. The code sample simulate.cc illustrates how they can be used.

mjDISABLESTRING

Names of the disable bits defined by mjtDisableBit.

stable

```
extern const char* mjDISABLESTRING[mjNDISABLE];
```

mjENABLESTRING

Names of the enable bits defined by mjtEnableBit.

```
extern const char* mjENABLESTRING[mjNENABLE];
```

mjTIMERSTRING

Names of the mjData timers defined by mjtTimer.

```
extern const char* mjTIMERSTRING[mjNTIMER];
```

mjLABELSTRING

Names of the visual labeling modes defined by mjtLabel.

```
extern const char* mjLABELSTRING[mjNLABEL];
```

mjFRAMESTRING

Names of the frame visualization modes defined by mjtFrame.

```
extern const char* mjFRAMESTRING[mjNFRAME];
```

mjVISSTRING

Descriptions of the abstract visualization flags defined by mjtVisFlag. For each flag there are three strings,

with the following meaning:

[0]: flag name;

[1]: the string "0" or "1" indicating if the flag is on or off by default, as set by mjv_defaultOption;

[2]: one-character string with a suggested keyboard shortcut, used in simulate.cc.

```
extern const char* mjVISSTRING[mjNVISFLAG][3];
```

mjRNDSTRING

Descriptions of the OpenGL rendering flags defined by mjtRndFlag. The
for each flag have the same format as above, except the defaults here
mjv_makeScene.

stable

```
extern const char* mjRNDSTRING[mjNRNDFLAG][3];
```

Numeric constants

Many integer constants were already documented in the primitive types above. In addition, the header files define several other constants documented here. Unless indicated otherwise, each entry in the table below is defined in mjmodel.h. Note that some extended key codes are defined in mjui.h which are not shown in the table below. Their names are in the format `mjKEY_XXX`. They correspond to GLFW key codes.

symbol	value	description
`mjMINVAL`	1E-15	The minimal value allowed in any denominator, and in general any mathematical operation where 0 is not allowed. In almost all cases, MuJoCo silently clamps smaller values to mjMINVAL.
`mjPI`	π	The value of π. This is used in various trigonometric functions, and also for conversion from degrees to radians in the compiler.
`mjMAXVAL`	1E+10	The maximal absolute value allowed in mjData.qpos, mjData.qvel, mjData.qacc. The API functions mj_checkPos, mj_checkVel, mj_checkAcc use this constant to detect instability.
`mjMINMU`	1E-5	The minimal value allowed in any friction coefficient. Recall that MuJoCo's contact model allows different number of friction dimensions to be included, as specified by the condim attribute. If however a given friction dimension is included, its friction is not allowed to be smaller than this constant. Smaller values are automatically clamped to this constant.
`mjMINIMP`	0.0001	The minimal value allowed in any constraint impedance. Smaller values are automatically clamped to this constant.
`mjMAXIMP`	0.9999	The maximal value allowed in any constraint impedance. Larger values are automatic this constant.

stable

symbol	value	description
`mjMAXCONPAIR`	50	The maximal number of contacts points that can be generated per geom pair. MuJoCo's built-in collision functions respect this limit, and user-defined functions should also respect it. Such functions are called with a return buffer of size `mjMAXCONPAIR`; attempting to write more contacts in the buffer can cause unpredictable behavior.
`mjMAXTREEDEPTH`	50	The maximum depth of each body and mesh bounding volume hierarchy. If this large limit is exceeded, a warning is raised and ray casting may not be possible. For a balanced hierarchy, this implies 1E15 bounding volumes.
`mjNEQDATA`	11	The maximal number of real-valued parameters used to define each equality constraint. Determines the size of `mjModel.eq_data`. This and the next five constants correspond to array sizes which we have not fully settled. There may be reasons to increase them in the future, so as to accommodate extra parameters needed for more elaborate computations. This is why we maintain them as symbolic constants that can be easily changed, as opposed to the array size for representing quaternions for example – which has no reason to change.
`mjNDYN`	10	The maximal number of real-valued parameters used to define the activation dynamics of each actuator. Determines the size of `mjModel.actuator_dynprm`.
`mjNGAIN`	10	The maximal number of real-valued parameters used to define the gain of each actuator. Determines the size of `mjModel.actuator_gainprm`.
`mjNBIAS`	10	The maximal number of real-valued parameters used to define the bias of each actuator. Determines the size of `mjModel.actuator_biasprm`.
`mjNFLUID`	12	The number of per-geom fluid interaction parameters required by the ellipsoidal model.
`mjNREF`	2	The maximal number of real-valued para[illegible] to define the reference acceleration of each scalar

stable

symbol	value	description
		constraint. Determines the size of all `mjModel.XXX_solref` fields.
`mjNIMP`	5	The maximal number of real-valued parameters used to define the impedance of each scalar constraint. Determines the size of all `mjModel.XXX_solimp` fields.
`mjNSOLVER`	200	The number of iterations where solver statistics can be stored in `mjData.solver`. This array is used to store diagnostic information about each iteration of the constraint solver. The actual number of iterations is given by `mjData.solver_iter`.
`mjNISLAND`	20	The number of islands for which solver statistics can be stored in `mjData.solver`. This array is used to store diagnostic information about each iteration of the constraint solver. The actual number of islands for which the solver was run is given by `mjData.nsolver_island`.
`mjNGROUP`	6	The number of geom, site, joint, tendon and actuator groups whose rendering can be enabled and disabled via mjvOption. Defined in mjvisualize.h.
`mjMAXOVERLAY`	500	The maximal number of characters in overlay text for rendering. Defined in mjvisualize.h.
`mjMAXLINE`	100	The maximal number of lines per 2D figure (mjvFigure). Defined in mjvisualize.h.
`mjMAXLINEPNT`	1000	The maximal number of points in each line in a 2D figure. Note that the buffer `mjvFigure.linepnt` has length `2*mjMAXLINEPNT` because each point has X and Y coordinates. Defined in mjvisualize.h.
`mjMAXPLANEGRID`	200	The maximal number of grid lines in each dimension for rendering planes. Defined in mjvisualize.h.
`mjNAUX`	10	Number of auxiliary buffers that can be allocated in mjrContext. Defined in mjrender.h.
`mjMAXTEXTURE`	1000	Maximum number of textures allowed. Defined in mjrender.h.
`mjMAXTHREAD`	128	Maximum number OS threads that can be used in a thread pool. Defined in mjthread.h.

stable

symbol	value	description
`mjMAXUISECT`	10	Maximum number of UI sections. Defined in mjui.h.
`mjMAXUIITEM`	200	Maximum number of items per UI section. Defined in mjui.h.
`mjMAXUITEXT`	500	Maximum number of characters in UI fields ‘edittext’ and ‘other’. Defined in mjui.h.
`mjMAXUINAME`	40	Maximum number of characters in any UI name. Defined in mjui.h.
`mjMAXUIMULTI`	20	Maximum number of radio and select items in UI group. Defined in mjui.h.
`mjMAXUIEDIT`	5	Maximum number of elements in UI edit list. Defined in mjui.h.
`mjMAXUIRECT`	15	Maximum number of UI rectangles. Defined in mjui.h.
`mjVERSION_HEADER`	332	The version of the MuJoCo headers; changes with every release. This is an integer equal to 100x the software version, so 210 corresponds to version 2.1. Defined in mujoco.h. The API function mj_version returns a number with the same meaning but for the compiled library.

Macros

mjUSESINGLE

Compile-time flag, see mjtNum.

mjDISABLED

```
#define mjDISABLED(x) (m->opt.disableflags & (x))
```

Check if a given standard feature has been disabled via the physics options, assuming mjModel* m is defined. x is of type mjtDisableBit.

mjENABLED

stable

```
#define mjENABLED(x) (m->opt.enableflags & (x))
```

Check if a given optional feature has been enabled via the physics options, assuming mjModel* m is defined. x is of type mjtEnableBit.

mjMAX

```
#define mjMAX(a,b) (((a) > (b)) ? (a) : (b))
```

Return maximum value. To avoid repeated evaluation with mjtNum types, use the function mju_max.

mjMIN

```
#define mjMIN(a,b) (((a) < (b)) ? (a) : (b))
```

Return minimum value. To avoid repeated evaluation with mjtNum types, use the function mju_min.

mjPLUGIN_LIB_INIT

```
#define mjPLUGIN_LIB_INIT                                                          \
  static void _mjplugin_dllmain(void);                                             \
  mjEXTERNC int __stdcall mjDLLMAIN(void* hinst, unsigned long reason, void* reserved) {  \
    if (reason == 1) {                                                             \
      _mjplugin_dllmain();                                                         \
    }                                                                              \
    return 1;                                                                      \
  }                                                                                \
  static void _mjplugin_dllmain(void)
```

Register a plugin as a dynamic library. See plugin registration for more details.

X Macros

The X Macros are not needed in most user projects. They are used internally to allocate the model, and are also available for users who know how to use this programming technique. See the header file mjxmacro.h for the actual definitions. They are particularly useful in writing MuJoCo wrappers for scripting languages, where dynamic structures matching the MuJoCo data structures need to be constructed programmatically.

stable

stable