API Reference

Introduction

This chapter is the reference manual for MuJoCo. It is generated from the header files included with MuJoCo, but also contains additional text not available in the headers.

Type definitions

Primitive types

MuJoCo defines a large number of primitive types described here. Except for mjtNum and mjtByte, all other primitive types are C enums used to define various integer constants. Note that the rest of the API does not use these enum types directly. Instead it uses ints, and only the documentation/comments state that certain ints correspond to certain enum types. This is because we want the API to be compiler-independent, and the C standard does not dictate how many bytes must be used to represent an enum type. Nevertheless we recommend using these types when calling the API functions (and letting the compiler do the enum-to-int type cast).

mjtNum

```
#ifdef mjUSEDOUBLE
    typedef double mjtNum;
#else
    typedef float mjtNum;
#endif
```

Defined in mjtnum.h

This is the floating-point type used throughout the simulator. If the symbol `mjUSEDOUBLE` is defined in `mjmodel.h`, this type is defined as `double`, otherwise it is defined as `float`. Currently only the double-precision version of MuJoCo is distributed, although the entire code base works with single-precision as well. We may release the single-precision version in the future for efficiency reasons, but the double-precision version will always be available. Thus it is safe to write user code assuming double precision. However, our preference is to write code that works with either single or double precision. To this end we provide math utility functions that are always defined with the correct floating-point type.

Note that changing `mjUSEDOUBLE` in `mjtnum.h` will not change how the library was compiled, and instead will result in numerous link errors. In general, the header files distributed with precompiled MuJoCo should never be changed by the user.

mjtByte

```
typedef unsigned char mjtByte;
```

Defined in mjmodel.h

Byte type used to represent boolean variables.

mjtDisableBit

```
typedef enum mjtDisableBit_ {     // disable default feature bitflags
  mjDSBL_CONSTRAINT   = 1<<0,     // entire constraint solver
  mjDSBL_EQUALITY     = 1<<1,     // equality constraints
  mjDSBL_FRICTIONLOSS = 1<<2,     // joint and tendon frictionloss constraints
  mjDSBL_LIMIT        = 1<<3,     // joint and tendon limit constraints
  mjDSBL_CONTACT      = 1<<4,     // contact constraints
  mjDSBL_PASSIVE      = 1<<5,     // passive forces
  mjDSBL_GRAVITY      = 1<<6,     // gravitational forces
  mjDSBL_CLAMPCTRL    = 1<<7,     // clamp control to specified range
  mjDSBL_WARMSTART    = 1<<8,     // warmstart constraint solver
  mjDSBL_FILTERPARENT = 1<<9,     // remove collisions with parent body
  mjDSBL_ACTUATION    = 1<<10,    // apply actuation forces
  mjDSBL_REFSAFE      = 1<<11,    // integrator safety: make ref[0]>=2*timestep
  mjDSBL_SENSOR       = 1<<12,    // sensors

  mjNDISABLE          = 13        // number of disable flags
} mjtDisableBit;
```

Defined in mjmodel.h

Constants which are powers of 2. They are used as bitmasks for the field `disableflags` of mjOption. At runtime this field is `m->opt.disableflags`. The number of these constants is given by `mjNDISABLE` which is also the length of the global string array mjDISABLESTRING with text descriptions of these flags.

mjtEnableBit

```
typedef enum mjtEnableBit_ {      // enable optional feature bitflags
  mjENBL_OVERRIDE     = 1<<0,     // override contact parameters
  mjENBL_ENERGY       = 1<<1,     // energy computation
  mjENBL_FWDINV       = 1<<2,     // record solver statistics
  mjENBL_SENSORNOISE  = 1<<3,     // add noise to sensor data
                                  // experimental features:
  mjENBL_MULTICCD     = 1<<4,     // multi-point convex collision detection

  mjNENABLE           = 5         // number of enable flags
} mjtEnableBit;
```

Defined in mjmodel.h

Constants which are powers of 2. They are used as bitmasks for the field `enableflags` of mjOption. At runtime this field is `m->opt.enableflags`. The number of these constants is given by `mjNENABLE` which is also the length of the global string array mjENABLESTRING with text descriptions of these flags.

mjtJoint

```
typedef enum mjtJoint_ {          // type of degree of freedom
  mjJNT_FREE          = 0,        // global position and orientation (quat)       (7)
  mjJNT_BALL,                     // orientation (quat) relative to parent        (4)
  mjJNT_SLIDE,                    // sliding distance along body-fixed axis       (1)
  mjJNT_HINGE                     // rotation angle (rad) around body-fixed axis  (1)
} mjtJoint;
```

Defined in mjmodel.h

Primitive joint types. These values are used in `m->jnt_type`. The numbers in the comments indicate how many positional coordinates each joint type has. Note that ball joints and rotational components of free joints are represented as unit quaternions - which have 4 positional coordinates but 3 degrees of freedom each.

mjtGeom

```
typedef enum mjtGeom_ {           // type of geometric shape
  // regular geom types
  mjGEOM_PLANE        = 0,        // plane
  mjGEOM_HFIELD,                  // height field
  mjGEOM_SPHERE,                  // sphere
  mjGEOM_CAPSULE,                 // capsule
  mjGEOM_ELLIPSOID,               // ellipsoid
  mjGEOM_CYLINDER,                // cylinder
  mjGEOM_BOX,                     // box
  mjGEOM_MESH,                    // mesh

  mjNGEOMTYPES,                   // number of regular geom types

  // rendering-only geom types: not used in mjModel, not counted in mjNGEOMTYPES
  mjGEOM_ARROW        = 100,      // arrow
  mjGEOM_ARROW1,                  // arrow without wedges
  mjGEOM_ARROW2,                  // arrow in both directions
  mjGEOM_LINE,                    // line
  mjGEOM_SKIN,                    // skin
  mjGEOM_LABEL,                   // text label

  mjGEOM_NONE         = 1001      // missing geom type
} mjtGeom;
```

Defined in mjmodel.h

Geometric types supported by MuJoCo. The first group are "official" geom types that can be used in the model. The second group are geom types that cannot be used in the model but are used by the visualizer to add decorative elements. These values are used in `m->geom_type` and `m->site_type`.

mjtCamLight

```
typedef enum mjtCamLight_ {       // tracking mode for camera and light
  mjCAMLIGHT_FIXED    = 0,        // pos and rot fixed in body
  mjCAMLIGHT_TRACK,               // pos tracks body, rot fixed in global
  mjCAMLIGHT_TRACKCOM,            // pos tracks subtree com, rot fixed in body
  mjCAMLIGHT_TARGETBODY,          // pos fixed in body, rot tracks target body
  mjCAMLIGHT_TARGETBODYCOM        // pos fixed in body, rot tracks target subtree com
} mjtCamLight;
```

Defined in mjmodel.h

Dynamic modes for cameras and lights, specifying how the camera/light position and orientation are computed. These values are used in `m->cam_mode` and `m->light_mode`.

mjtTexture

```
typedef enum mjtTexture_ {        // type of texture
  mjTEXTURE_2D        = 0,        // 2d texture, suitable for planes and hfields
  mjTEXTURE_CUBE,                 // cube texture, suitable for all other geom types
  mjTEXTURE_SKYBOX                // cube texture used as skybox
} mjtTexture;
```

Defined in mjmodel.h

Texture types, specifying how the texture will be mapped. These values are used in `m->tex_type`.

mjtIntegrator

```
typedef enum mjtIntegrator_ {     // integrator mode
  mjINT_EULER         = 0,        // semi-implicit Euler
  mjINT_RK4,                      // 4th-order Runge Kutta
  mjINT_IMPLICIT                  // implicit in velocity
} mjtIntegrator;
```

Defined in mjmodel.h

Numerical integrator types. These values are used in `m->opt.integrator`.

mjtCollision

```
typedef enum mjtCollision_ {      // collision mode for selecting geom pairs
  mjCOL_ALL           = 0,        // test precomputed and dynamic pairs
  mjCOL_PAIR,                     // test predefined pairs only
  mjCOL_DYNAMIC                   // test dynamic pairs only
} mjtCollision;
```

Defined in mjmodel.h

Collision modes specifying how candidate geom pairs are generated for near-phase collision checking. These values are used in `m->opt.collision`.

mjtCone

```
typedef enum mjtCone_ {           // type of friction cone
  mjCONE_PYRAMIDAL    = 0,        // pyramidal
  mjCONE_ELLIPTIC                 // elliptic
} mjtCone;
```

Defined in mjmodel.h

Available friction cone types. These values are used in `m->opt.cone`.

mjtJacobian

```
typedef enum mjtJacobian_ {       // type of constraint Jacobian
```

```
  mjJAC_DENSE          = 0,       // dense
  mjJAC_SPARSE,                   // sparse
  mjJAC_AUTO                      // dense if nv<60, sparse otherwise
} mjtJacobian;
```

Defined in mjmodel.h

Available Jacobian types. These values are used in `m->opt.jacobian`.

mjtSolver

```
typedef enum mjtSolver_ {         // constraint solver algorithm
  mjSOL_PGS            = 0,       // PGS    (dual)
  mjSOL_CG,                       // CG     (primal)
  mjSOL_NEWTON                    // Newton (primal)
} mjtSolver;
```

Defined in mjmodel.h

Available constraint solver algorithms. These values are used in `m->opt.solver`.

mjtEq

```
typedef enum mjtEq_ {             // type of equality constraint
  mjEQ_CONNECT        = 0,        // connect two bodies at a point (ball joint)
  mjEQ_WELD,                      // fix relative position and orientation of two bodies
  mjEQ_JOINT,                     // couple the values of two scalar joints with cubic
  mjEQ_TENDON,                    // couple the lengths of two tendons with cubic
  mjEQ_DISTANCE                   // unsupported, will cause an error if used
} mjtEq;
```

Defined in mjmodel.h

Equality constraint types. These values are used in `m->eq_type`.

mjtWrap

```
typedef enum mjtWrap_ {           // type of tendon wrap object
  mjWRAP_NONE          = 0,       // null object
  mjWRAP_JOINT,                   // constant moment arm
  mjWRAP_PULLEY,                  // pulley used to split tendon
  mjWRAP_SITE,                    // pass through site
  mjWRAP_SPHERE,                  // wrap around sphere
  mjWRAP_CYLINDER                 // wrap around (infinite) cylinder
} mjtWrap;
```

Defined in mjmodel.h

Tendon wrapping object types. These values are used in `m->wrap_type`.

mjtTrn

```
typedef enum mjtTrn_ {            // type of actuator transmission
  mjTRN_JOINT          = 0,       // force on joint
  mjTRN_JOINTINPARENT,            // force on joint, expressed in parent frame
  mjTRN_SLIDERCRANK,              // force via slider-crank linkage
  mjTRN_TENDON,                   // force on tendon
  mjTRN_SITE,                     // force on site
  mjTRN_BODY,                     // adhesion force on a body's geoms

  mjTRN_UNDEFINED      = 1000     // undefined transmission type
} mjtTrn;
```

Defined in mjmodel.h

Actuator transmission types. These values are used in `m->actuator_trntype`.

mjtDyn

```
typedef enum mjtDyn_ {            // type of actuator dynamics
  mjDYN_NONE           = 0,       // no internal dynamics; ctrl specifies force
  mjDYN_INTEGRATOR,               // integrator: da/dt = u
  mjDYN_FILTER,                   // linear filter: da/dt = (u-a) / tau
  mjDYN_MUSCLE,                   // piece-wise linear filter with two time constants
  mjDYN_USER                      // user-defined dynamics type
} mjtDyn;
```

Defined in mjmodel.h

Actuator dynamics types. These values are used in `m->actuator_dyntype`.

mjtGain

```
typedef enum mjtGain_ {           // type of actuator gain
  mjGAIN_FIXED         = 0,       // fixed gain
  mjGAIN_AFFINE,                  // const + kp*length + kv*velocity
  mjGAIN_MUSCLE,                  // muscle FLV curve computed by mju_muscleGain()
  mjGAIN_USER                     // user-defined gain type
} mjtGain;
```

Defined in mjmodel.h

Actuator gain types. These values are used in `m->actuator_gaintype`.

mjtBias

```
typedef enum mjtBias_ {           // type of actuator bias
  mjBIAS_NONE          = 0,       // no bias
  mjBIAS_AFFINE,                  // const + kp*length + kv*velocity
  mjBIAS_MUSCLE,                  // muscle passive force computed by mju_muscleBias()
  mjBIAS_USER                     // user-defined bias type
} mjtBias;
```

Defined in mjmodel.h

Actuator bias types. These values are used in `m->actuator_biastype`.

mjtObj

```
typedef enum mjtObj_ {            // type of MujoCo object
  mjOBJ_UNKNOWN        = 0,       // unknown object type
  mjOBJ_BODY,                     // body
  mjOBJ_XBODY,                    // body, used to access regular frame instead of i-frame
  mjOBJ_JOINT,                    // joint
```

```
  mjOBJ_DOF,                    // dof
  mjOBJ_GEOM,                   // geom
  mjOBJ_SITE,                   // site
  mjOBJ_CAMERA,                 // camera
  mjOBJ_LIGHT,                  // light
  mjOBJ_MESH,                   // mesh
  mjOBJ_SKIN,                   // skin
  mjOBJ_HFIELD,                 // heightfield
  mjOBJ_TEXTURE,                // texture
  mjOBJ_MATERIAL,               // material for rendering
  mjOBJ_PAIR,                   // geom pair to include
  mjOBJ_EXCLUDE,                // body pair to exclude
  mjOBJ_EQUALITY,               // equality constraint
  mjOBJ_TENDON,                 // tendon
  mjOBJ_ACTUATOR,               // actuator
  mjOBJ_SENSOR,                 // sensor
  mjOBJ_NUMERIC,                // numeric
  mjOBJ_TEXT,                   // text
  mjOBJ_TUPLE,                  // tuple
  mjOBJ_KEY,                    // keyframe
  mjOBJ_PLUGIN                  // plugin instance
} mjtObj;
```

Defined in mjmodel.h

MuJoCo object types. These values are used in the support functions mj_name2id and mj_id2name to convert between object names and integer ids.

mjtConstraint

```
typedef enum mjtConstraint_ {     // type of constraint
  mjCNSTR_EQUALITY    = 0,        // equality constraint
  mjCNSTR_FRICTION_DOF,           // dof friction
  mjCNSTR_FRICTION_TENDON,        // tendon friction
  mjCNSTR_LIMIT_JOINT,            // joint limit
  mjCNSTR_LIMIT_TENDON,           // tendon limit
  mjCNSTR_CONTACT_FRICTIONLESS,   // frictionless contact
  mjCNSTR_CONTACT_PYRAMIDAL,      // frictional contact, pyramidal friction cone
  mjCNSTR_CONTACT_ELLIPTIC        // frictional contact, elliptic friction cone
} mjtConstraint;
```

Defined in mjmodel.h

Constraint types. These values are not used in mjModel, but are used in the mjData field `d->efc_type` when the list of active constraints is constructed at each simulation time step.

mjtConstraintState

```
typedef enum mjtConstraintState_ { // constraint state
  mjCNSTRSTATE_SATISFIED = 0,     // constraint satisfied, zero cost (limit, contact)
  mjCNSTRSTATE_QUADRATIC,         // quadratic cost (equality, friction, limit, contact)
  mjCNSTRSTATE_LINEARNEG,         // linear cost, negative side (friction)
  mjCNSTRSTATE_LINEARPOS,         // linear cost, positive side (friction)
  mjCNSTRSTATE_CONE               // squared distance to cone cost (elliptic contact)
} mjtConstraintState;
```

Defined in mjmodel.h

These values are used by the solver internally to keep track of the constraint states.

mjtSensor

```
typedef enum mjtSensor_ {         // type of sensor
  // common robotic sensors, attached to a site
  mjSENS_TOUCH        = 0,        // scalar contact normal forces summed over sensor zone
  mjSENS_ACCELEROMETER,           // 3D linear acceleration, in local frame
  mjSENS_VELOCIMETER,             // 3D linear velocity, in local frame
  mjSENS_GYRO,                    // 3D angular velocity, in local frame
  mjSENS_FORCE,                   // 3D force between site's body and its parent body
  mjSENS_TORQUE,                  // 3D torque between site's body and its parent body
  mjSENS_MAGNETOMETER,            // 3D magnetometer
  mjSENS_RANGEFINDER,             // scalar distance to nearest geom or site along z-axis

  // sensors related to scalar joints, tendons, actuators
  mjSENS_JOINTPOS,                // scalar joint position (hinge and slide only)
  mjSENS_JOINTVEL,                // scalar joint velocity (hinge and slide only)
  mjSENS_TENDONPOS,               // scalar tendon position
  mjSENS_TENDONVEL,               // scalar tendon velocity
  mjSENS_ACTUATORPOS,             // scalar actuator position
  mjSENS_ACTUATORVEL,             // scalar actuator velocity
  mjSENS_ACTUATORFRC,             // scalar actuator force

  // sensors related to ball joints
  mjSENS_BALLQUAT,                // 4D ball joint quaternion
  mjSENS_BALLANGVEL,              // 3D ball joint angular velocity

  // joint and tendon limit sensors, in constraint space
  mjSENS_JOINTLIMITPOS,           // joint limit distance-margin
  mjSENS_JOINTLIMITVEL,           // joint limit velocity
  mjSENS_JOINTLIMITFRC,           // joint limit force
  mjSENS_TENDONLIMITPOS,          // tendon limit distance-margin
  mjSENS_TENDONLIMITVEL,          // tendon limit velocity
  mjSENS_TENDONLIMITFRC,          // tendon limit force

  // sensors attached to an object with spatial frame: (x)body, geom, site, camera
  mjSENS_FRAMEPOS,                // 3D position
  mjSENS_FRAMEQUAT,               // 4D unit quaternion orientation
  mjSENS_FRAMEXAXIS,              // 3D unit vector: x-axis of object's frame
  mjSENS_FRAMEYAXIS,              // 3D unit vector: y-axis of object's frame
  mjSENS_FRAMEZAXIS,              // 3D unit vector: z-axis of object's frame
  mjSENS_FRAMELINVEL,             // 3D linear velocity
  mjSENS_FRAMEANGVEL,             // 3D angular velocity
  mjSENS_FRAMELINACC,             // 3D linear acceleration
  mjSENS_FRAMEANGACC,             // 3D angular acceleration

  // sensors related to kinematic subtrees; attached to a body (which is the subtree root)
  mjSENS_SUBTREECOM,              // 3D center of mass of subtree
  mjSENS_SUBTREELINVEL,           // 3D linear velocity of subtree
  mjSENS_SUBTREEANGMOM,           // 3D angular momentum of subtree

  // global sensors
  mjSENS_CLOCK,                   // simulation time
```

```
  // plugin-controlled sensors
  mjSENS_PLUGIN,                  // plugin-controlled

  // user-defined sensor
  mjSENS_USER                     // sensor data provided by mjcb_sensor callback
} mjtSensor;
```

Defined in mjmodel.h

Sensor types. These values are used in `m->sensor_type`.

mjtStage

```
typedef enum mjtStage_ {          // computation stage
  mjSTAGE_NONE        = 0,        // no computations
  mjSTAGE_POS,                    // position-dependent computations
  mjSTAGE_VEL,                    // velocity-dependent computations
  mjSTAGE_ACC                     // acceleration/force-dependent computations
} mjtStage;
```

Defined in mjmodel.h

These are the compute stages for the skipstage parameters of mj_forwardSkip and mj_inverseSkip.

mjtDataType

```
typedef enum mjtDataType_ {       // data type for sensors
  mjDATATYPE_REAL     = 0,        // real values, no constraints
  mjDATATYPE_POSITIVE,            // positive values; 0 or negative: inactive
  mjDATATYPE_AXIS,                // 3D unit vector
  mjDATATYPE_QUATERNION           // unit quaternion
} mjtDataType;
```

Defined in mjmodel.h

These are the possible sensor data types, used in `mjData.sensor_datatype`.

mjtWarning

```
typedef enum mjtWarning_ {        // warning types
  mjWARN_INERTIA      = 0,        // (near) singular inertia matrix
  mjWARN_CONTACTFULL,             // too many contacts in contact list
  mjWARN_CNSTRFULL,               // too many constraints
  mjWARN_VGEOMFULL,               // too many visual geoms
  mjWARN_BADQPOS,                 // bad number in qpos
  mjWARN_BADQVEL,                 // bad number in qvel
  mjWARN_BADQACC,                 // bad number in qacc
  mjWARN_BADCTRL,                 // bad number in ctrl

  mjNWARNING                      // number of warnings
} mjtWarning;
```

Defined in mjdata.h

Warning types. The number of warning types is given by `mjNWARNING` which is also the length of the array `mjData.warning`.

mjtTimer

```
typedef enum mjtTimer_ {
  // main api
  mjTIMER_STEP        = 0,        // step
  mjTIMER_FORWARD,                // forward
  mjTIMER_INVERSE,                // inverse

  // breakdown of step/forward
  mjTIMER_POSITION,               // fwdPosition
  mjTIMER_VELOCITY,               // fwdVelocity
  mjTIMER_ACTUATION,              // fwdActuation
  mjTIMER_ACCELERATION,           // fwdAcceleration
  mjTIMER_CONSTRAINT,             // fwdConstraint

  // breakdown of fwdPosition
  mjTIMER_POS_KINEMATICS,         // kinematics, com, tendon, transmission
  mjTIMER_POS_INERTIA,            // inertia computations
  mjTIMER_POS_COLLISION,          // collision detection
  mjTIMER_POS_MAKE,               // make constraints
  mjTIMER_POS_PROJECT,            // project constraints

  mjNTIMER                        // number of timers
} mjtTimer;
```

Defined in mjdata.h

Timer types. The number of timer types is given by `mjNTIMER` which is also the length of the array `mjData.timer`, as well as the length of the string array mjTIMERSTRING with timer names.

mjtCatBit

```
typedef enum mjtCatBit_ {         // bitflags for mjvGeom category
  mjCAT_STATIC        = 1,        // model elements in body 0
  mjCAT_DYNAMIC       = 2,        // model elements in all other bodies
  mjCAT_DECOR         = 4,        // decorative geoms
  mjCAT_ALL           = 7         // select all categories
} mjtCatBit;
```

Defined in mjvisualize.h

These are the available categories of geoms in the abstract visualizer. The bitmask can be used in the function mjr_render to specify which categories should be rendered.

mjtMouse

```
typedef enum mjtMouse_ {          // mouse interaction mode
  mjMOUSE_NONE        = 0,        // no action
  mjMOUSE_ROTATE_V,               // rotate, vertical plane
  mjMOUSE_ROTATE_H,               // rotate, horizontal plane
  mjMOUSE_MOVE_V,                 // move, vertical plane
  mjMOUSE_MOVE_H,                 // move, horizontal plane
  mjMOUSE_ZOOM,                   // zoom
  mjMOUSE_SELECT                  // selection
} mjtMouse;
```

Defined in mjvisualize.h

These are the mouse actions that the abstract visualizer recognizes. It is up to the user to intercept mouse events and translate them into these actions, as illustrated in simulate.cc.

mjtPertBit

```
typedef enum mjtPertBit_ {          // mouse perturbations
  mjPERT_TRANSLATE    = 1,          // translation
  mjPERT_ROTATE       = 2           // rotation
} mjtPertBit;
```

Defined in mjvisualize.h

These bitmasks enable the translational and rotational components of the mouse perturbation. For the regular mouse, only one can be enabled at a time. For the 3D mouse (SpaceNavigator) both can be enabled simultaneously. They are used in `mjvPerturb.active`.

mjtCamera

```
typedef enum mjtCamera_ {           // abstract camera type
  mjCAMERA_FREE       = 0,          // free camera
  mjCAMERA_TRACKING,                // tracking camera; uses trackbodyid
  mjCAMERA_FIXED,                   // fixed camera; uses fixedcamid
  mjCAMERA_USER                     // user is responsible for setting OpenGL camera
} mjtCamera;
```

Defined in mjvisualize.h

These are the possible camera types, used in `mjvCamera.type`.

mjtLabel

```
typedef enum mjtLabel_ {            // object labeling
  mjLABEL_NONE        = 0,          // nothing
  mjLABEL_BODY,                     // body labels
  mjLABEL_JOINT,                    // joint labels
  mjLABEL_GEOM,                     // geom labels
  mjLABEL_SITE,                     // site labels
  mjLABEL_CAMERA,                   // camera labels
  mjLABEL_LIGHT,                    // light labels
  mjLABEL_TENDON,                   // tendon labels
  mjLABEL_ACTUATOR,                 // actuator labels
  mjLABEL_CONSTRAINT,               // constraint labels
  mjLABEL_SKIN,                     // skin labels
  mjLABEL_SELECTION,                // selected object
  mjLABEL_SELPNT,                   // coordinates of selection point
  mjLABEL_CONTACTFORCE,             // magnitude of contact force

  mjNLABEL                          // number of label types
} mjtLabel;
```

Defined in mjvisualize.h

These are the abstract visualization elements that can have text labels. Used in `mjvOption.label`.

mjtFrame

```
typedef enum mjtFrame_ {            // frame visualization
  mjFRAME_NONE        = 0,          // no frames
  mjFRAME_BODY,                     // body frames
  mjFRAME_GEOM,                     // geom frames
  mjFRAME_SITE,                     // site frames
  mjFRAME_CAMERA,                   // camera frames
  mjFRAME_LIGHT,                    // light frames
  mjFRAME_CONTACT,                  // contact frames
  mjFRAME_WORLD,                    // world frame

  mjNFRAME                          // number of visualization frames
} mjtFrame;
```

Defined in mjvisualize.h

These are the MuJoCo objects whose spatial frames can be rendered. Used in `mjvOption.frame`.

mjtVisFlag

```
typedef enum mjtVisFlag_ {          // flags enabling model element visualization
  mjVIS_CONVEXHULL    = 0,          // mesh convex hull
  mjVIS_TEXTURE,                    // textures
  mjVIS_JOINT,                      // joints
  mjVIS_CAMERA,                     // cameras
  mjVIS_ACTUATOR,                   // actuators
  mjVIS_ACTIVATION,                 // activations
  mjVIS_LIGHT,                      // lights
  mjVIS_TENDON,                     // tendons
  mjVIS_RANGEFINDER,                // rangefinder sensors
  mjVIS_CONSTRAINT,                 // point constraints
  mjVIS_INERTIA,                    // equivalent inertia boxes
  mjVIS_SCLINERTIA,                 // scale equivalent inertia boxes with mass
  mjVIS_PERTFORCE,                  // perturbation force
  mjVIS_PERTOBJ,                    // perturbation object
  mjVIS_CONTACTPOINT,               // contact points
  mjVIS_CONTACTFORCE,               // contact force
  mjVIS_CONTACTSPLIT,               // split contact force into normal and tanget
  mjVIS_TRANSPARENT,                // make dynamic geoms more transparent
  mjVIS_AUTOCONNECT,                // auto connect joints and body coms
  mjVIS_COM,                        // center of mass
  mjVIS_SELECT,                     // selection point
  mjVIS_STATIC,                     // static bodies
  mjVIS_SKIN,                       // skin

  mjNVISFLAG                        // number of visualization flags
} mjtVisFlag;
```

Defined in mjvisualize.h

These are indices in the array `mjvOption.flags`, whose elements enable/disable the visualization of the corresponding model or decoration element.

mjtRndFlag

```
typedef enum mjtRndFlag_ {          // flags enabling rendering effects
```

```
  mjRND_SHADOW        = 0,        // shadows
  mjRND_WIREFRAME,                // wireframe
  mjRND_REFLECTION,               // reflections
  mjRND_ADDITIVE,                 // additive transparency
  mjRND_SKYBOX,                   // skybox
  mjRND_FOG,                      // fog
  mjRND_HAZE,                     // haze
  mjRND_SEGMENT,                  // segmentation with random color
  mjRND_IDCOLOR,                  // segmentation with segid+1 color
  mjRND_CULL_FACE,                // cull backward faces

  mjNRNDFLAG                      // number of rendering flags
} mjtRndFlag;
```

Defined in mjvisualize.h

These are indices in the array `mjvScene.flags`, whose elements enable/disable OpenGL rendering effects.

mjtStereo

```
typedef enum mjtStereo_ {         // type of stereo rendering
  mjSTEREO_NONE       = 0,        // no stereo; use left eye only
  mjSTEREO_QUADBUFFERED,          // quad buffered; revert to side-by-side if no hardware sup
  mjSTEREO_SIDEBYSIDE             // side-by-side
} mjtStereo;
```

Defined in mjvisualize.h

These are the possible stereo rendering types. They are used in `mjvScene.stereo`.

mjtGridPos

```
typedef enum mjtGridPos_ {        // grid position for overlay
  mjGRID_TOPLEFT      = 0,        // top left
  mjGRID_TOPRIGHT,                // top right
  mjGRID_BOTTOMLEFT,              // bottom left
  mjGRID_BOTTOMRIGHT              // bottom right
} mjtGridPos;
```

Defined in mjrender.h

These are the possible grid positions for text overlays. They are used as an argument to the function mjr_overlay.

mjtFramebuffer

```
typedef enum mjtFramebuffer_ {      // OpenGL framebuffer option
  mjFB_WINDOW         = 0,        // default/window buffer
  mjFB_OFFSCREEN                  // offscreen buffer
} mjtFramebuffer;
```

Defined in mjrender.h

These are the possible framebuffers. They are used as an argument to the function mjr_setBuffer.

mjtFontScale

```
typedef enum mjtFontScale_ {        // font scale, used at context creation
  mjFONTSCALE_50      = 50,       // 50% scale, suitable for low-res rendering
  mjFONTSCALE_100     = 100,      // normal scale, suitable in the absence of DPI scaling
  mjFONTSCALE_150     = 150,      // 150% scale
  mjFONTSCALE_200     = 200,      // 200% scale
  mjFONTSCALE_250     = 250,      // 250% scale
  mjFONTSCALE_300     = 300       // 300% scale
} mjtFontScale;
```

Defined in mjrender.h

These are the possible font sizes. The fonts are predefined bitmaps stored in the dynamic library at three different sizes.

mjtFont

```
typedef enum mjtFont_ {           // font type, used at each text operation
  mjFONT_NORMAL       = 0,        // normal font
  mjFONT_SHADOW,                  // normal font with shadow (for higher contrast)
  mjFONT_BIG                      // big font (for user alerts)
} mjtFont;
```

Defined in mjrender.h

These are the possible font types.

mjtButton

```
typedef enum mjtButton_ {         // mouse button
  mjBUTTON_NONE = 0,              // no button
  mjBUTTON_LEFT,                  // left button
  mjBUTTON_RIGHT,                 // right button
  mjBUTTON_MIDDLE                 // middle button
} mjtButton;
```

Defined in mjui.h

Mouse button IDs used in the UI framework.

mjtEvent

```
typedef enum mjtEvent_ {          // mouse and keyboard event type
  mjEVENT_NONE = 0,               // no event
  mjEVENT_MOVE,                   // mouse move
  mjEVENT_PRESS,                  // mouse button press
  mjEVENT_RELEASE,                // mouse button release
  mjEVENT_SCROLL,                 // scroll
  mjEVENT_KEY,                    // key press
  mjEVENT_RESIZE                  // resize
} mjtEvent;
```

Defined in mjui.h

Event types used in the UI framework.

mjtItem

```
typedef enum mjtItem_ {            // UI item type
  mjITEM_END = -2,                 // end of definition list (not an item)
  mjITEM_SECTION = -1,             // section (not an item)
  mjITEM_SEPARATOR = 0,            // separator
  mjITEM_STATIC,                   // static text
  mjITEM_BUTTON,                   // button

  // the rest have data pointer
  mjITEM_CHECKINT,                 // check box, int value
  mjITEM_CHECKBYTE,                // check box, mjtByte value
  mjITEM_RADIO,                    // radio group
  mjITEM_RADIOLINE,                // radio group, single line
  mjITEM_SELECT,                   // selection box
  mjITEM_SLIDERINT,                // slider, int value
  mjITEM_SLIDERNUM,                // slider, mjtNum value
  mjITEM_EDITINT,                  // editable array, int values
  mjITEM_EDITNUM,                  // editable array, mjtNum values
  mjITEM_EDITTXT,                  // editable text

  mjNITEM                          // number of item types
} mjtItem;
```

Defined in mjui.h

Item types used in the UI framework.

Function types

MuJoCo callbacks have corresponding function types. They are defined in mjdata.h and in mjui.h. The actual callback functions are documented later.

mjfGeneric

```
typedef void (*mjfGeneric)(const mjModel* m, mjData* d);
```

This is the function type of the callbacks mjcb_passive and mjcb_control.

mjfConFilt

```
typedef int (*mjfConFilt)(const mjModel* m, mjData* d, int geom1, int geom2);
```

This is the function type of the callback mjcb_contactfilter. The return value is 1: discard, 0: proceed with collision check.

mjfSensor

```
typedef void (*mjfSensor)(const mjModel* m, mjData* d, int stage);
```

This is the function type of the callback mjcb_sensor.

mjfTime

```
typedef mjtNum (*mjfTime)(void);
```

This is the function type of the callback mjcb_time.

mjfAct

```
typedef mjtNum (*mjfAct)(const mjModel* m, const mjData* d, int id);
```

This is the function type of the callbacks mjcb_act_dyn, mjcb_act_gain and mjcb_act_bias.

mjfCollision

```
typedef int (*mjfCollision)(const mjModel* m, const mjData* d,
                            mjContact* con, int g1, int g2, mjtNum margin);
```

This is the function type of the callbacks in the collision table mjCOLLISIONFUNC.

mjfItemEnable

```
typedef int (*mjfItemEnable)(int category, void* data);
```

This is the function type of the predicate function used by the UI framework to determine if each item is enabled or disabled.

Data structures

MuJoCo uses several data structures shown below. They are taken directly from the header files which contain comments for each field.

mjVFS

```
struct mjVFS_ {                    // virtual file system for loading from memory
  int   nfile;                     // number of files present
  char  filename[mjMAXVFS][mjMAXVFSNAME]; // file name without path
  int   filesize[mjMAXVFS];        // file size in bytes
  void* filedata[mjMAXVFS];        // buffer with file data
};
typedef struct mjVFS_ mjVFS;
```

Defined in mjmodel.h

This is the data structure with the virtual file system. It can only be constructed programmatically, and does not have an analog in MJCF.

mjOption

```
struct mjOption_ {                 // physics options
  // timing parameters
  mjtNum timestep;                 // timestep
  mjtNum apirate;                  // update rate for remote API (Hz)

  // solver parameters
  mjtNum impratio;                 // ratio of friction-to-normal contact impedance
  mjtNum tolerance;                // main solver tolerance
  mjtNum noslip_tolerance;         // noslip solver tolerance
  mjtNum mpr_tolerance;            // MPR solver tolerance
```

```
  // physical constants
  mjtNum gravity[3];              // gravitational acceleration
  mjtNum wind[3];                 // wind (for lift, drag and viscosity)
  mjtNum magnetic[3];             // global magnetic flux
  mjtNum density;                 // density of medium
  mjtNum viscosity;               // viscosity of medium

  // override contact solver parameters (if enabled)
  mjtNum o_margin;                // margin
  mjtNum o_solref[mjNREF];        // solref
  mjtNum o_solimp[mjNIMP];        // solimp

  // discrete settings
  int integrator;                 // integration mode (mjtIntegrator)
  int collision;                  // collision mode (mjtCollision)
  int cone;                       // type of friction cone (mjtCone)
  int jacobian;                   // type of Jacobian (mjtJacobian)
  int solver;                     // solver algorithm (mjtSolver)
  int iterations;                 // maximum number of main solver iterations
  int noslip_iterations;          // maximum number of noslip solver iterations
  int mpr_iterations;             // maximum number of MPR solver iterations
  int disableflags;               // bit flags for disabling standard features
  int enableflags;                // bit flags for enabling optional features
};
typedef struct mjOption_ mjOption;
```

Defined in mjmodel.h

This is the data structure with simulation options. It corresponds to the MJCF element option. One instance of it is embedded in mjModel.

mjVisual

```
struct mjVisual_ {              // visualization options
  struct {                      // global parameters
    float fovy;                 // y-field of view for free camera (degrees)
    float ipd;                  // inter-pupilary distance for free camera
    float azimuth;              // initial azimuth of free camera (degrees)
    float elevation;            // initial elevation of free camera (degrees)
    float linewidth;            // line width for wireframe and ray rendering
    float glow;                 // glow coefficient for selected body
    float realtime;             // initial real-time factor (1: real time)
    int offwidth;               // width of offscreen buffer
    int offheight;              // height of offscreen buffer
  } global;

  struct {                      // rendering quality
    int   shadowsize;           // size of shadowmap texture
    int   offsamples;           // number of multisamples for offscreen rendering
    int   numslices;            // number of slices for builtin geom drawing
    int   numstacks;            // number of stacks for builtin geom drawing
    int   numquads;             // number of quads for box rendering
  } quality;

  struct {                      // head light
    float ambient[3];           // ambient rgb (alpha=1)
    float diffuse[3];           // diffuse rgb (alpha=1)
    float specular[3];          // specular rgb (alpha=1)
    int   active;               // is headlight active
  } headlight;

  struct {                      // mapping
    float stiffness;            // mouse perturbation stiffness (space->force)
    float stiffnessrot;         // mouse perturbation stiffness (space->torque)
    float force;                // from force units to space units
    float torque;               // from torque units to space units
    float alpha;                // scale geom alphas when transparency is enabled
    float fogstart;             // OpenGL fog starts at fogstart * mjModel.stat.extent
    float fogend;               // OpenGL fog ends at fogend * mjModel.stat.extent
    float znear;                // near clipping plane = znear * mjModel.stat.extent
    float zfar;                 // far clipping plane = zfar * mjModel.stat.extent
    float haze;                 // haze ratio
    float shadowclip;           // directional light: shadowclip * mjModel.stat.extent
    float shadowscale;          // spot light: shadowscale * light.cutoff
    float actuatortendon;       // scale tendon width
  } map;

  struct {                      // scale of decor elements relative to mean body size
    float forcewidth;           // width of force arrow
    float contactwidth;         // contact width
    float contactheight;        // contact height
    float connect;              // autoconnect capsule width
    float com;                  // com radius
    float camera;               // camera object
    float light;                // light object
    float selectpoint;          // selection point
    float jointlength;          // joint length
    float jointwidth;           // joint width
    float actuatorlength;       // actuator length
    float actuatorwidth;        // actuator width
    float framelength;          // bodyframe axis length
    float framewidth;           // bodyframe axis width
    float constraint;           // constraint width
    float slidercrank;          // slidercrank width
  } scale;

  struct {                      // color of decor elements
    float fog[4];               // fog
    float haze[4];              // haze
    float force[4];             // external force
    float inertia[4];           // inertia box
    float joint[4];             // joint
    float actuator[4];          // actuator, neutral
    float actuatornegative[4];  // actuator, negative limit
    float actuatorpositive[4];  // actuator, positive limit
    float com[4];               // center of mass
    float camera[4];            // camera object
    float light[4];             // light object
    float selectpoint[4];       // selection point
    float connect[4];           // auto connect
    float contactpoint[4];      // contact point
    float contactforce[4];      // contact force
    float contactfriction[4];   // contact friction force
```

```
    float contacttorque[4];       // contact torque
    float contactgap[4];          // contact point in gap
    float rangefinder[4];         // rangefinder ray
    float constraint[4];          // constraint
    float slidercrank[4];         // slidercrank
    float crankbroken[4];         // used when crank must be stretched/broken
  } rgba;
};
typedef struct mjVisual_ mjVisual;
```

Defined in mjmodel.h

This is the data structure with abstract visualization options. It corresponds to the MJCF element visual. One instance of it is embedded in mjModel.

mjStatistic

```
struct mjStatistic_ {             // model statistics (in qpos0)
  mjtNum meaninertia;             // mean diagonal inertia
  mjtNum meanmass;                // mean body mass
  mjtNum meansize;                // mean body size
  mjtNum extent;                  // spatial extent
  mjtNum center[3];               // center of model
};
typedef struct mjStatistic_ mjStatistic;
```

Defined in mjmodel.h

This is the data structure with model statistics precomputed by the compiler or set by the user. It corresponds to the MJCF element statistic. One instance of it is embedded in mjModel.

mjModel

```
struct mjModel_ {
  // ------------------------------ sizes

  // sizes needed at mjModel construction
  int nq;                         // number of generalized coordinates = dim(qpos)
  int nv;                         // number of degrees of freedom = dim(qvel)
  int nu;                         // number of actuators/controls = dim(ctrl)
  int na;                         // number of activation states = dim(act)
  int nbody;                      // number of bodies
  int njnt;                       // number of joints
  int ngeom;                      // number of geoms
  int nsite;                      // number of sites
  int ncam;                       // number of cameras
  int nlight;                     // number of lights
  int nmesh;                      // number of meshes
  int nmeshvert;                  // number of vertices in all meshes
  int nmeshtexvert;               // number of vertices with texcoords in all meshes
  int nmeshface;                  // number of triangular faces in all meshes
  int nmeshgraph;                 // number of ints in mesh auxiliary data
  int nskin;                      // number of skins
  int nskinvert;                  // number of vertices in all skins
  int nskintexvert;               // number of vertiex with texcoords in all skins
  int nskinface;                  // number of triangular faces in all skins
  int nskinbone;                  // number of bones in all skins
  int nskinbonevert;              // number of vertices in all skin bones
  int nhfield;                    // number of heightfields
  int nhfielddata;                // number of data points in all heightfields
  int ntex;                       // number of textures
  int ntexdata;                   // number of bytes in texture rgb data
  int nmat;                       // number of materials
  int npair;                      // number of predefined geom pairs
  int nexclude;                   // number of excluded geom pairs
  int neq;                        // number of equality constraints
  int ntendon;                    // number of tendons
  int nwrap;                      // number of wrap objects in all tendon paths
  int nsensor;                    // number of sensors
  int nnumeric;                   // number of numeric custom fields
  int nnumericdata;               // number of mjtNums in all numeric fields
  int ntext;                      // number of text custom fields
  int ntextdata;                  // number of mjtBytes in all text fields
  int ntuple;                     // number of tuple custom fields
  int ntupledata;                 // number of objects in all tuple fields
  int nkey;                       // number of keyframes
  int nmocap;                     // number of mocap bodies
  int nplugin;                    // number of plugin instances
  int npluginattr;                // number of chars in all plugin config attributes
  int nuser_body;                 // number of mjtNums in body_user
  int nuser_jnt;                  // number of mjtNums in jnt_user
  int nuser_geom;                 // number of mjtNums in geom_user
  int nuser_site;                 // number of mjtNums in site_user
  int nuser_cam;                  // number of mjtNums in cam_user
  int nuser_tendon;               // number of mjtNums in tendon_user
  int nuser_actuator;             // number of mjtNums in actuator_user
  int nuser_sensor;               // number of mjtNums in sensor_user
  int nnames;                     // number of chars in all names

  // sizes set after mjModel construction (only affect mjData)
  int nM;                         // number of non-zeros in sparse inertia matrix
  int nD;                         // number of non-zeros in sparse derivative matrix
  int nemax;                      // number of potential equality-constraint rows
  int njmax;                      // number of available rows in constraint Jacobian
  int nconmax;                    // number of potential contacts in contact list
  int nstack;                     // number of fields in mjData stack
  int nuserdata;                  // number of extra fields in mjData
  int nsensordata;                // number of fields in sensor data vector
  int npluginstate;               // number of fields in the plugin state vector

  int nbuffer;                    // number of bytes in buffer

  // ------------------------------ options and statistics

  mjOption opt;                   // physics options
  mjVisual vis;                   // visualization options
  mjStatistic stat;               // model statistics

  // ------------------------------ buffers

  // main buffer
  void*     buffer;               // main buffer; all pointers point in it    (nbuffer)
```

```
// default generalized coordinates
mjtNum*   qpos0;                // qpos values at default pose              (nq x 1)
mjtNum*   qpos_spring;          // reference pose for springs               (nq x 1)

// bodies
int*      body_parentid;        // id of body's parent                      (nbody x 1)
int*      body_rootid;          // id of root above body                    (nbody x 1)
int*      body_weldid;          // id of body that this body is welded to   (nbody x 1)
int*      body_mocapid;         // id of mocap data; -1: none               (nbody x 1)
int*      body_jntnum;          // number of joints for this body           (nbody x 1)
int*      body_jntadr;          // start addr of joints; -1: no joints      (nbody x 1)
int*      body_dofnum;          // number of motion degrees of freedom      (nbody x 1)
int*      body_dofadr;          // start addr of dofs; -1: no dofs          (nbody x 1)
int*      body_geomnum;         // number of geoms                          (nbody x 1)
int*      body_geomadr;         // start addr of geoms; -1: no geoms        (nbody x 1)
mjtByte*  body_simple;          // body is simple (has diagonal M)          (nbody x 1)
mjtByte*  body_sameframe;       // inertial frame is same as body frame     (nbody x 1)
mjtNum*   body_pos;             // position offset rel. to parent body      (nbody x 3)
mjtNum*   body_quat;            // orientation offset rel. to parent body   (nbody x 4)
mjtNum*   body_ipos;            // local position of center of mass         (nbody x 3)
mjtNum*   body_iquat;           // local orientation of inertia ellipsoid   (nbody x 4)
mjtNum*   body_mass;            // mass                                     (nbody x 1)
mjtNum*   body_subtreemass;     // mass of subtree starting at this body    (nbody x 1)
mjtNum*   body_inertia;         // diagonal inertia in ipos/iquat frame     (nbody x 3)
mjtNum*   body_invweight0;      // mean inv inert in qpos0 (trn, rot)       (nbody x 2)
mjtNum*   body_gravcomp;        // antigravity force, units of body weight  (nbody x 1)
mjtNum*   body_user;            // user data                                (nbody x nuser_
int*      body_plugin;          // plugin instance id (-1 if not in use)    (nbody x 1)

// joints
int*      jnt_type;             // type of joint (mjtJoint)                 (njnt x 1)
int*      jnt_qposadr;          // start addr in 'qpos' for joint's data    (njnt x 1)
int*      jnt_dofadr;           // start addr in 'qvel' for joint's data    (njnt x 1)
int*      jnt_bodyid;           // id of joint's body                       (njnt x 1)
int*      jnt_group;            // group for visibility                     (njnt x 1)
mjtByte*  jnt_limited;          // does joint have limits                   (njnt x 1)
mjtNum*   jnt_solref;           // constraint solver reference: limit       (njnt x mjNREF)
mjtNum*   jnt_solimp;           // constraint solver impedance: limit       (njnt x mjNIMP)
mjtNum*   jnt_pos;              // local anchor position                    (njnt x 3)
mjtNum*   jnt_axis;             // local joint axis                         (njnt x 3)
mjtNum*   jnt_stiffness;        // stiffness coefficient                    (njnt x 1)
mjtNum*   jnt_range;            // joint limits                             (njnt x 2)
mjtNum*   jnt_margin;           // min distance for limit detection         (njnt x 1)
mjtNum*   jnt_user;             // user data                                (njnt x nuser_j

// dofs
int*      dof_bodyid;           // id of dof's body                         (nv x 1)
int*      dof_jntid;            // id of dof's joint                        (nv x 1)
int*      dof_parentid;         // id of dof's parent; -1: none             (nv x 1)
int*      dof_Madr;             // dof address in M-diagonal                (nv x 1)
int*      dof_simplenum;        // number of consecutive simple dofs        (nv x 1)
mjtNum*   dof_solref;           // constraint solver reference:frictionloss (nv x mjNREF)
mjtNum*   dof_solimp;           // constraint solver impedance:frictionloss (nv x mjNIMP)
mjtNum*   dof_frictionloss;     // dof friction loss                        (nv x 1)
mjtNum*   dof_armature;         // dof armature inertia/mass                (nv x 1)
mjtNum*   dof_damping;          // damping coefficient                      (nv x 1)
mjtNum*   dof_invweight0;       // diag. inverse inertia in qpos0           (nv x 1)
mjtNum*   dof_M0;               // diag. inertia in qpos0                   (nv x 1)

// geoms
int*      geom_type;            // geometric type (mjtGeom)                 (ngeom x 1)
int*      geom_contype;         // geom contact type                        (ngeom x 1)
int*      geom_conaffinity;     // geom contact affinity                    (ngeom x 1)
int*      geom_condim;          // contact dimensionality (1, 3, 4, 6)      (ngeom x 1)
int*      geom_bodyid;          // id of geom's body                        (ngeom x 1)
int*      geom_dataid;          // id of geom's mesh/hfield (-1: none)      (ngeom x 1)
int*      geom_matid;           // material id for rendering                (ngeom x 1)
int*      geom_group;           // group for visibility                     (ngeom x 1)
int*      geom_priority;        // geom contact priority                    (ngeom x 1)
mjtByte*  geom_sameframe;       // same as body frame (1) or iframe (2)     (ngeom x 1)
mjtNum*   geom_solmix;          // mixing coef for solref/imp in geom pair  (ngeom x 1)
mjtNum*   geom_solref;          // constraint solver reference: contact     (ngeom x mjNREF
mjtNum*   geom_solimp;          // constraint solver impedance: contact     (ngeom x mjNIMP
mjtNum*   geom_size;            // geom-specific size parameters            (ngeom x 3)
mjtNum*   geom_rbound;          // radius of bounding sphere                (ngeom x 1)
mjtNum*   geom_pos;             // local position offset rel. to body       (ngeom x 3)
mjtNum*   geom_quat;            // local orientation offset rel. to body    (ngeom x 4)
mjtNum*   geom_friction;        // friction for (slide, spin, roll)         (ngeom x 3)
mjtNum*   geom_margin;          // detect contact if dist<margin            (ngeom x 1)
mjtNum*   geom_gap;             // include in solver if dist<margin-gap     (ngeom x 1)
mjtNum*   geom_fluid;           // fluid interaction parameters             (ngeom x mjNFLU
mjtNum*   geom_user;            // user data                                (ngeom x nuser_
float*    geom_rgba;            // rgba when material is omitted            (ngeom x 4)

// sites
int*      site_type;            // geom type for rendering (mjtGeom)        (nsite x 1)
int*      site_bodyid;          // id of site's body                        (nsite x 1)
int*      site_matid;           // material id for rendering                (nsite x 1)
int*      site_group;           // group for visibility                     (nsite x 1)
mjtByte*  site_sameframe;       // same as body frame (1) or iframe (2)     (nsite x 1)
mjtNum*   site_size;            // geom size for rendering                  (nsite x 3)
mjtNum*   site_pos;             // local position offset rel. to body       (nsite x 3)
mjtNum*   site_quat;            // local orientation offset rel. to body    (nsite x 4)
mjtNum*   site_user;            // user data                                (nsite x nuser_
float*    site_rgba;            // rgba when material is omitted            (nsite x 4)

// cameras
int*      cam_mode;             // camera tracking mode (mjtCamLight)       (ncam x 1)
int*      cam_bodyid;           // id of camera's body                      (ncam x 1)
int*      cam_targetbodyid;     // id of targeted body; -1: none            (ncam x 1)
mjtNum*   cam_pos;              // position rel. to body frame              (ncam x 3)
mjtNum*   cam_quat;             // orientation rel. to body frame           (ncam x 4)
mjtNum*   cam_poscom0;          // global position rel. to sub-com in qpos0 (ncam x 3)
mjtNum*   cam_pos0;             // global position rel. to body in qpos0    (ncam x 3)
mjtNum*   cam_mat0;             // global orientation in qpos0              (ncam x 9)
mjtNum*   cam_fovy;             // y-field of view (deg)                    (ncam x 1)
mjtNum*   cam_ipd;              // inter-pupilary distance                  (ncam x 1)
mjtNum*   cam_user;             // user data                                (ncam x nuser_c

// lights
int*      light_mode;           // light tracking mode (mjtCamLight)        (nlight x 1)
int*      light_bodyid;         // id of light's body                       (nlight x 1)
int*      light_targetbodyid;   // id of targeted body; -1: none            (nlight x 1)
mjtByte*  light_directional;    // directional light                        (nlight x 1)
```

```
mjtByte*  light_castshadow;     // does light cast shadows                  (nlight x 1)
mjtByte*  light_active;         // is light on                              (nlight x 1)
mjtNum*   light_pos;            // position rel. to body frame              (nlight x 3)
mjtNum*   light_dir;            // direction rel. to body frame             (nlight x 3)
mjtNum*   light_poscom0;        // global position rel. to sub-com in qpos0 (nlight x 3)
mjtNum*   light_pos0;           // global position rel. to body in qpos0    (nlight x 3)
mjtNum*   light_dir0;           // global direction in qpos0                (nlight x 3)
float*    light_attenuation;    // OpenGL attenuation (quadratic model)     (nlight x 3)
float*    light_cutoff;         // OpenGL cutoff                            (nlight x 1)
float*    light_exponent;       // OpenGL exponent                          (nlight x 1)
float*    light_ambient;        // ambient rgb (alpha=1)                    (nlight x 3)
float*    light_diffuse;        // diffuse rgb (alpha=1)                    (nlight x 3)
float*    light_specular;       // specular rgb (alpha=1)                   (nlight x 3)

// meshes
int*      mesh_vertadr;         // first vertex address                     (nmesh x 1)
int*      mesh_vertnum;         // number of vertices                       (nmesh x 1)
int*      mesh_texcoordadr;     // texcoord data address; -1: no texcoord   (nmesh x 1)
int*      mesh_faceadr;         // first face address                       (nmesh x 1)
int*      mesh_facenum;         // number of faces                          (nmesh x 1)
int*      mesh_graphadr;        // graph data address; -1: no graph         (nmesh x 1)
float*    mesh_vert;            // vertex positions for all meshes          (nmeshvert x 3)
float*    mesh_normal;          // vertex normals for all meshes            (nmeshvert x 3)
float*    mesh_texcoord;        // vertex texcoords for all meshes          (nmeshtexvert x
int*      mesh_face;            // triangle face data                       (nmeshface x 3)
int*      mesh_graph;           // convex graph data                        (nmeshgraph x 1

// skins
int*      skin_matid;           // skin material id; -1: none               (nskin x 1)
int*      skin_group;           // group for visibility                     (nskin x 1)
float*    skin_rgba;            // skin rgba                                (nskin x 4)
float*    skin_inflate;         // inflate skin in normal direction         (nskin x 1)
int*      skin_vertadr;         // first vertex address                     (nskin x 1)
int*      skin_vertnum;         // number of vertices                       (nskin x 1)
int*      skin_texcoordadr;     // texcoord data address; -1: no texcoord   (nskin x 1)
int*      skin_faceadr;         // first face address                       (nskin x 1)
int*      skin_facenum;         // number of faces                          (nskin x 1)
int*      skin_boneadr;         // first bone in skin                       (nskin x 1)
int*      skin_bonenum;         // number of bones in skin                  (nskin x 1)
float*    skin_vert;            // vertex positions for all skin meshes     (nskinvert x 3)
float*    skin_texcoord;        // vertex texcoords for all skin meshes     (nskintexvert x
int*      skin_face;            // triangle faces for all skin meshes       (nskinface x 3)
int*      skin_bonevertadr;     // first vertex in each bone                (nskinbone x 1)
int*      skin_bonevertnum;     // number of vertices in each bone          (nskinbone x 1)
float*    skin_bonebindpos;     // bind pos of each bone                    (nskinbone x 3)
float*    skin_bonebindquat;    // bind quat of each bone                   (nskinbone x 4)
int*      skin_bonebodyid;      // body id of each bone                     (nskinbone x 1)
int*      skin_bonevertid;      // mesh ids of vertices in each bone        (nskinbonevert
float*    skin_bonevertweight;  // weights of vertices in each bone         (nskinbonevert

// height fields
mjtNum*   hfield_size;          // (x, y, z_top, z_bottom)                  (nhfield x 4)
int*      hfield_nrow;          // number of rows in grid                   (nhfield x 1)
int*      hfield_ncol;          // number of columns in grid                (nhfield x 1)
int*      hfield_adr;           // address in hfield_data                   (nhfield x 1)
float*    hfield_data;          // elevation data                           (nhfielddata x

// textures
int*      tex_type;             // texture type (mjtTexture)                (ntex x 1)
int*      tex_height;           // number of rows in texture image          (ntex x 1)
int*      tex_width;            // number of columns in texture image       (ntex x 1)
int*      tex_adr;              // address in rgb                           (ntex x 1)
mjtByte*  tex_rgb;              // rgb (alpha = 1)                          (ntexdata x 1)

// materials
int*      mat_texid;            // texture id; -1: none                     (nmat x 1)
mjtByte*  mat_texuniform;       // make texture cube uniform                (nmat x 1)
float*    mat_texrepeat;        // texture repetition for 2d mapping        (nmat x 2)
float*    mat_emission;         // emission (x rgb)                         (nmat x 1)
float*    mat_specular;         // specular (x white)                       (nmat x 1)
float*    mat_shininess;        // shininess coef                           (nmat x 1)
float*    mat_reflectance;      // reflectance (0: disable)                 (nmat x 1)
float*    mat_rgba;             // rgba                                     (nmat x 4)

// predefined geom pairs for collision detection; has precedence over exclude
int*      pair_dim;             // contact dimensionality                   (npair x 1)
int*      pair_geom1;           // id of geom1                              (npair x 1)
int*      pair_geom2;           // id of geom2                              (npair x 1)
int*      pair_signature;       // (body1+1)<<16 + body2+1                  (npair x 1)
mjtNum*   pair_solref;          // constraint solver reference: contact     (npair x mjNREF
mjtNum*   pair_solimp;          // constraint solver impedance: contact     (npair x mjNIMP
mjtNum*   pair_margin;          // detect contact if dist<margin            (npair x 1)
mjtNum*   pair_gap;             // include in solver if dist<margin-gap     (npair x 1)
mjtNum*   pair_friction;        // tangent1, 2, spin, roll1, 2              (npair x 5)

// excluded body pairs for collision detection
int*      exclude_signature;    // (body1+1)<<16 + body2+1                  (nexclude x 1)

// equality constraints
int*      eq_type;              // constraint type (mjtEq)                  (neq x 1)
int*      eq_obj1id;            // id of object 1                           (neq x 1)
int*      eq_obj2id;            // id of object 2                           (neq x 1)
mjtByte*  eq_active;            // enable/disable constraint                (neq x 1)
mjtNum*   eq_solref;            // constraint solver reference              (neq x mjNREF)
mjtNum*   eq_solimp;            // constraint solver impedance              (neq x mjNIMP)
mjtNum*   eq_data;              // numeric data for constraint              (neq x mjNEQDAT

// tendons
int*      tendon_adr;           // address of first object in tendon's path (ntendon x 1)
int*      tendon_num;           // number of objects in tendon's path       (ntendon x 1)
int*      tendon_matid;         // material id for rendering                (ntendon x 1)
int*      tendon_group;         // group for visibility                     (ntendon x 1)
mjtByte*  tendon_limited;       // does tendon have length limits           (ntendon x 1)
mjtNum*   tendon_width;         // width for rendering                      (ntendon x 1)
mjtNum*   tendon_solref_lim;    // constraint solver reference: limit       (ntendon x mjNR
mjtNum*   tendon_solimp_lim;    // constraint solver impedance: limit       (ntendon x mjNI
mjtNum*   tendon_solref_fri;    // constraint solver reference: friction    (ntendon x mjNR
mjtNum*   tendon_solimp_fri;    // constraint solver impedance: friction    (ntendon x mjNI
mjtNum*   tendon_range;         // tendon length limits                     (ntendon x 2)
mjtNum*   tendon_margin;        // min distance for limit detection         (ntendon x 1)
mjtNum*   tendon_stiffness;     // stiffness coefficient                    (ntendon x 1)
mjtNum*   tendon_damping;       // damping coefficient                      (ntendon x 1)
mjtNum*   tendon_frictionloss;  // loss due to friction                     (ntendon x 1)
mjtNum*   tendon_lengthspring;  // spring resting length range              (ntendon x 2)
```

```
mjtNum*   tendon_length0;       // tendon length in qpos0                 (ntendon x 1)
mjtNum*   tendon_invweight0;    // inv. weight in qpos0                   (ntendon x 1)
mjtNum*   tendon_user;          // user data                              (ntendon x nuse
float*    tendon_rgba;          // rgba when material is omitted          (ntendon x 4)

// list of all wrap objects in tendon paths
int*      wrap_type;            // wrap object type (mjtWrap)             (nwrap x 1)
int*      wrap_objid;           // object id: geom, site, joint           (nwrap x 1)
mjtNum*   wrap_prm;             // divisor, joint coef, or site id        (nwrap x 1)

// actuators
int*      actuator_trntype;     // transmission type (mjtTrn)             (nu x 1)
int*      actuator_dyntype;     // dynamics type (mjtDyn)                 (nu x 1)
int*      actuator_gaintype;    // gain type (mjtGain)                    (nu x 1)
int*      actuator_biastype;    // bias type (mjtBias)                    (nu x 1)
int*      actuator_trnid;       // transmission id: joint, tendon, site   (nu x 2)
int*      actuator_actadr;      // first activation address; -1: stateless (nu x 1)
int*      actuator_actnum;      // number of activation variables         (nu x 1)
int*      actuator_group;       // group for visibility                   (nu x 1)
mjtByte*  actuator_ctrllimited; // is control limited                     (nu x 1)
mjtByte*  actuator_forcelimited;// is force limited                       (nu x 1)
mjtByte*  actuator_actlimited;  // is activation limited                  (nu x 1)
mjtNum*   actuator_dynprm;      // dynamics parameters                    (nu x mjNDYN)
mjtNum*   actuator_gainprm;     // gain parameters                        (nu x mjNGAIN)
mjtNum*   actuator_biasprm;     // bias parameters                        (nu x mjNBIAS)
mjtNum*   actuator_ctrlrange;   // range of controls                      (nu x 2)
mjtNum*   actuator_forcerange;  // range of forces                        (nu x 2)
mjtNum*   actuator_actrange;    // range of activations                   (nu x 2)
mjtNum*   actuator_gear;        // scale length and transmitted force     (nu x 6)
mjtNum*   actuator_cranklength; // crank length for slider-crank          (nu x 1)
mjtNum*   actuator_acc0;        // acceleration from unit force in qpos0  (nu x 1)
mjtNum*   actuator_length0;     // actuator length in qpos0               (nu x 1)
mjtNum*   actuator_lengthrange; // feasible actuator length range         (nu x 2)
mjtNum*   actuator_user;        // user data                              (nu x nuser_act
int*      actuator_plugin;      // plugin instance id; -1: not a plugin   (nu x 1)

// sensors
int*      sensor_type;          // sensor type (mjtSensor)                (nsensor x 1)
int*      sensor_datatype;      // numeric data type (mjtDataType)        (nsensor x 1)
int*      sensor_needstage;     // required compute stage (mjtStage)      (nsensor x 1)
int*      sensor_objtype;       // type of sensorized object (mjtObj)     (nsensor x 1)
int*      sensor_objid;         // id of sensorized object                (nsensor x 1)
int*      sensor_reftype;       // type of reference frame (mjtObj)       (nsensor x 1)
int*      sensor_refid;         // id of reference frame; -1: global frame (nsensor x 1)
int*      sensor_dim;           // number of scalar outputs               (nsensor x 1)
int*      sensor_adr;           // address in sensor array                (nsensor x 1)
mjtNum*   sensor_cutoff;        // cutoff for real and positive; 0: ignore (nsensor x 1)
mjtNum*   sensor_noise;         // noise standard deviation               (nsensor x 1)
mjtNum*   sensor_user;          // user data                              (nsensor x nuse
int*      sensor_plugin;        // plugin instance id; -1: not a plugin   (nsensor x 1)

// plugin instances
int*      plugin;               // globally registered plugin slot number  (nplugin x 1)
int*      plugin_stateadr;      // address in the plugin state array       (nplugin x 1)
int*      plugin_statenum;      // number of states in the plugin instance (nplugin x 1)
char*     plugin_attr;          // config attributes of plugin instances   (npluginattr x
int*      plugin_attradr;       // address to each instance's config attrib (nplugin x 1)

// custom numeric fields
int*      numeric_adr;          // address of field in numeric_data        (nnumeric x 1)
int*      numeric_size;         // size of numeric field                   (nnumeric x 1)
mjtNum*   numeric_data;         // array of all numeric fields             (nnumericdata x

// custom text fields
int*      text_adr;             // address of text in text_data            (ntext x 1)
int*      text_size;            // size of text field (strlen+1)           (ntext x 1)
char*     text_data;            // array of all text fields (0-terminated) (ntextdata x 1)

// custom tuple fields
int*      tuple_adr;            // address of text in text_data            (ntuple x 1)
int*      tuple_size;           // number of objects in tuple              (ntuple x 1)
int*      tuple_objtype;        // array of object types in all tuples     (ntupledata x 1
int*      tuple_objid;          // array of object ids in all tuples       (ntupledata x 1
mjtNum*   tuple_objprm;         // array of object params in all tuples    (ntupledata x 1

// keyframes
mjtNum*   key_time;             // key time                                (nkey x 1)
mjtNum*   key_qpos;             // key position                            (nkey x nq)
mjtNum*   key_qvel;             // key velocity                            (nkey x nv)
mjtNum*   key_act;              // key activation                          (nkey x na)
mjtNum*   key_mpos;             // key mocap position                      (nkey x 3*nmoca
mjtNum*   key_mquat;            // key mocap quaternion                    (nkey x 4*nmoca
mjtNum*   key_ctrl;             // key control                             (nkey x nu)

// names
int*      name_bodyadr;         // body name pointers                      (nbody x 1)
int*      name_jntadr;          // joint name pointers                     (njnt x 1)
int*      name_geomadr;         // geom name pointers                      (ngeom x 1)
int*      name_siteadr;         // site name pointers                      (nsite x 1)
int*      name_camadr;          // camera name pointers                    (ncam x 1)
int*      name_lightadr;        // light name pointers                     (nlight x 1)
int*      name_meshadr;         // mesh name pointers                      (nmesh x 1)
int*      name_skinadr;         // skin name pointers                      (nskin x 1)
int*      name_hfieldadr;       // hfield name pointers                    (nhfield x 1)
int*      name_texadr;          // texture name pointers                   (ntex x 1)
int*      name_matadr;          // material name pointers                  (nmat x 1)
int*      name_pairadr;         // geom pair name pointers                 (npair x 1)
int*      name_excludeadr;      // exclude name pointers                   (nexclude x 1)
int*      name_eqadr;           // equality constraint name pointers       (neq x 1)
int*      name_tendonadr;       // tendon name pointers                    (ntendon x 1)
int*      name_actuatoradr;     // actuator name pointers                  (nu x 1)
int*      name_sensoradr;       // sensor name pointers                    (nsensor x 1)
int*      name_numericadr;      // numeric name pointers                   (nnumeric x 1)
int*      name_textadr;         // text name pointers                      (ntext x 1)
int*      name_tupleadr;        // tuple name pointers                     (ntuple x 1)
int*      name_keyadr;          // keyframe name pointers                  (nkey x 1)
int*      name_pluginadr;       // plugin instance name pointers           (nplugin x 1)
char*     names;                // names of all objects, 0-terminated      (nnames x 1)
};
typedef struct mjModel_ mjModel;
```

Defined in mjmodel.h

This is the main data structure holding the MuJoCo model. It is treated as constant by the simulator.

mjContact

```
struct mjContact_ {              // result of collision detection functions
  // contact parameters set by geom-specific collision detector
  mjtNum dist;                   // distance between nearest points; neg: penetration
  mjtNum pos[3];                 // position of contact point: midpoint between geoms
  mjtNum frame[9];               // normal is in [0-2]

  // contact parameters set by mj_collideGeoms
  mjtNum includemargin;          // include if dist<includemargin=margin-gap
  mjtNum friction[5];            // tangent1, 2, spin, roll1, 2
  mjtNum solref[mjNREF];         // constraint solver reference
  mjtNum solimp[mjNIMP];         // constraint solver impedance

  // internal storage used by solver
  mjtNum mu;                     // friction of regularized cone, set by mj_makeConstraint
  mjtNum H[36];                  // cone Hessian, set by mj_updateConstraint

  // contact descriptors set by mj_collideGeoms
  int dim;                       // contact space dimensionality: 1, 3, 4 or 6
  int geom1;                     // id of geom 1
  int geom2;                     // id of geom 2

  // flag set by mj_fuseContact or mj_instantianteEquality
  int exclude;                   // 0: include, 1: in gap, 2: fused, 3: equality, 4: no dofs

  // address computed by mj_instantiateContact
  int efc_address;               // address in efc; -1: not included, -2-i: distance constra
};
typedef struct mjContact_ mjContact;
```

Defined in mjdata.h

This is the data structure holding information about one contact. `mjData.contact` is a preallocated array of mjContact data structures, populated at runtime with the contacts found by the collision detector. Additional contact information is then filled-in by the simulator.

mjWarningStat

```
struct mjWarningStat_ {          // warning statistics
  int lastinfo;                  // info from last warning
  int number;                    // how many times was warning raised
};
typedef struct mjWarningStat_ mjWarningStat;
```

Defined in mjdata.h

This is the data structure holding information about one warning type. `mjData.warning` is a preallocated array of mjWarningStat data structures, one for each warning type.

mjTimerStat

```
struct mjTimerStat_ {            // timer statistics
  mjtNum duration;               // cumulative duration
  int number;                    // how many times was timer called
};
typedef struct mjTimerStat_ mjTimerStat;
```

Defined in mjdata.h

This is the data structure holding information about one timer. `mjData.timer` is a preallocated array of mjTimerStat data structures, one for each timer type.

mjSolverStat

```
struct mjSolverStat_ {           // per-iteration solver statistics
  mjtNum improvement;            // cost reduction, scaled by 1/trace(M(qpos0))
  mjtNum gradient;               // gradient norm (primal only, scaled)
  mjtNum lineslope;              // slope in linesearch
  int nactive;                   // number of active constraints
  int nchange;                   // number of constraint state changes
  int neval;                     // number of cost evaluations in line search
  int nupdate;                   // number of Cholesky updates in line search
};
typedef struct mjSolverStat_ mjSolverStat;
```

Defined in mjdata.h

This is the data structure holding information about one solver iteration. `mjData.solver` is a preallocated array of mjSolverStat data structures, one for each iteration of the solver, up to a maximum of mjNSOLVER. The actual number of solver iterations is given by `mjData.solver_iter`.

mjData

```
struct mjData_ {
  // constant sizes
  int nstack;                    // number of mjtNums that can fit in the arena+stack space
  int nbuffer;                   // size of main buffer in bytes
  int nplugin;                   // number of plugin instances

  // stack pointer
  size_t pstack;                 // first available mjtNum address in stack

  // arena pointer
  size_t parena;                 // first available byte in arena

  // memory utilization stats
  int maxuse_stack;              // maximum stack allocation
  size_t maxuse_arena;           // maximum arena allocation
  int maxuse_con;                // maximum number of contacts
  int maxuse_efc;                // maximum number of scalar constraints

  // diagnostics
  mjWarningStat warning[mjNWARNING]; // warning statistics
  mjTimerStat timer[mjNTIMER];   // timer statistics
  mjSolverStat solver[mjNSOLVER]; // solver statistics per iteration
  int solver_iter;               // number of solver iterations
  int solver_nnz;                // number of non-zeros in Hessian or efc_AR
  mjtNum solver_fwdinv[2];       // forward-inverse comparison: qfrc, efc
```

```
// variable sizes
int ne;                         // number of equality constraints
int nf;                         // number of friction constraints
int nefc;                       // number of constraints
int ncon;                       // number of detected contacts

// global properties
mjtNum time;                    // simulation time
mjtNum energy[2];               // potential, kinetic energy

//------------------------------ end of info header

// buffers
void*     buffer;               // main buffer; all pointers point in it           (nbuffer
void*     arena;                // arena+stack buffer                (nstack*sizeof(mjtNum)

//------------------------------ main inputs and outputs of the computation

// state
mjtNum*   qpos;                 // position                                  (nq x 1)
mjtNum*   qvel;                 // velocity                                  (nv x 1)
mjtNum*   act;                  // actuator activation                       (na x 1)
mjtNum*   qacc_warmstart;       // acceleration used for warmstart           (nv x 1)
mjtNum*   plugin_state;         // plugin state                              (npluginstate x

// control
mjtNum*   ctrl;                 // control                                   (nu x 1)
mjtNum*   qfrc_applied;         // applied generalized force                 (nv x 1)
mjtNum*   xfrc_applied;         // applied Cartesian force/torque            (nbody x 6)

// mocap data
mjtNum*   mocap_pos;            // positions of mocap bodies                 (nmocap x 3)
mjtNum*   mocap_quat;           // orientations of mocap bodies              (nmocap x 4)

// dynamics
mjtNum*   qacc;                 // acceleration                              (nv x 1)
mjtNum*   act_dot;              // time-derivative of actuator activation    (na x 1)

// user data
mjtNum*   userdata;             // user data, not touched by engine          (nuserdata x 1)

// sensors
mjtNum*   sensordata;           // sensor data array                         (nsensordata x

// plugins
int*          plugin;           // copy of m->plugin, required for deletion (nplugin x 1)
uintptr_t*    plugin_data;      // pointer to plugin-managed data structure (nplugin x 1)

//------------------------------ POSITION dependent

// computed by mj_fwdPosition/mj_kinematics
mjtNum*   xpos;                 // Cartesian position of body frame          (nbody x 3)
mjtNum*   xquat;                // Cartesian orientation of body frame       (nbody x 4)
mjtNum*   xmat;                 // Cartesian orientation of body frame       (nbody x 9)
mjtNum*   xipos;                // Cartesian position of body com            (nbody x 3)
mjtNum*   ximat;                // Cartesian orientation of body inertia     (nbody x 9)
mjtNum*   xanchor;              // Cartesian position of joint anchor        (njnt x 3)
mjtNum*   xaxis;                // Cartesian joint axis                      (njnt x 3)
mjtNum*   geom_xpos;            // Cartesian geom position                   (ngeom x 3)
mjtNum*   geom_xmat;            // Cartesian geom orientation                (ngeom x 9)
mjtNum*   site_xpos;            // Cartesian site position                   (nsite x 3)
mjtNum*   site_xmat;            // Cartesian site orientation                (nsite x 9)
mjtNum*   cam_xpos;             // Cartesian camera position                 (ncam x 3)
mjtNum*   cam_xmat;             // Cartesian camera orientation              (ncam x 9)
mjtNum*   light_xpos;           // Cartesian light position                  (nlight x 3)
mjtNum*   light_xdir;           // Cartesian light direction                 (nlight x 3)

// computed by mj_fwdPosition/mj_comPos
mjtNum*   subtree_com;          // center of mass of each subtree            (nbody x 3)
mjtNum*   cdof;                 // com-based motion axis of each dof         (nv x 6)
mjtNum*   cinert;               // com-based body inertia and mass           (nbody x 10)

// computed by mj_fwdPosition/mj_tendon
int*      ten_wrapadr;          // start address of tendon's path            (ntendon x 1)
int*      ten_wrapnum;          // number of wrap points in path             (ntendon x 1)
int*      ten_J_rownnz;         // number of non-zeros in Jacobian row       (ntendon x 1)
int*      ten_J_rowadr;         // row start address in colind array         (ntendon x 1)
int*      ten_J_colind;         // column indices in sparse Jacobian         (ntendon x nv)
mjtNum*   ten_length;           // tendon lengths                            (ntendon x 1)
mjtNum*   ten_J;                // tendon Jacobian                           (ntendon x nv)
int*      wrap_obj;             // geom id; -1: site; -2: pulley             (nwrap*2 x 1)
mjtNum*   wrap_xpos;            // Cartesian 3D points in all path           (nwrap*2 x 3)

// computed by mj_fwdPosition/mj_transmission
mjtNum*   actuator_length;      // actuator lengths                          (nu x 1)
mjtNum*   actuator_moment;      // actuator moments                          (nu x nv)

// computed by mj_fwdPosition/mj_crb
mjtNum*   crb;                  // com-based composite inertia and mass      (nbody x 10)
mjtNum*   qM;                   // total inertia (sparse)                    (nM x 1)

// computed by mj_fwdPosition/mj_factorM
mjtNum*   qLD;                  // L'*D*L factorization of M (sparse)        (nM x 1)
mjtNum*   qLDiagInv;            // 1/diag(D)                                 (nv x 1)
mjtNum*   qLDiagSqrtInv;        // 1/sqrt(diag(D))                           (nv x 1)

//------------------------------ POSITION, VELOCITY dependent

// computed by mj_fwdVelocity
mjtNum*   ten_velocity;         // tendon velocities                         (ntendon x 1)
mjtNum*   actuator_velocity;    // actuator velocities                       (nu x 1)

// computed by mj_fwdVelocity/mj_comVel
mjtNum*   cvel;                 // com-based velocity [3D rot; 3D tran]      (nbody x 6)
mjtNum*   cdof_dot;             // time-derivative of cdof                   (nv x 6)

// computed by mj_fwdVelocity/mj_rne (without acceleration)
mjtNum*   qfrc_bias;            // C(qpos,qvel)                              (nv x 1)

// computed by mj_fwdVelocity/mj_passive
mjtNum*   qfrc_passive;         // passive force                             (nv x 1)

// computed by mj_fwdVelocity/mj_referenceConstraint
```

```
  mjtNum*   efc_vel;              // velocity in constraint space: J*qvel     (nefc x 1)
  mjtNum*   efc_aref;             // reference pseudo-acceleration            (nefc x 1)

  // computed by mj_sensorVel/mj_subtreeVel if needed
  mjtNum*   subtree_linvel;       // linear velocity of subtree com           (nbody x 3)
  mjtNum*   subtree_angmom;       // angular momentum about subtree com       (nbody x 3)

  // computed by mj_Euler
  mjtNum*   qH;                   // L'*D*L factorization of modified M       (nM x 1)
  mjtNum*   qHDiagInv;            // 1/diag(D) of modified M                  (nv x 1)

  // computed by mj_implicit
  int*      D_rownnz;             // non-zeros in each row                    (nv x 1)
  int*      D_rowadr;             // address of each row in D_colind          (nv x 1)
  int*      D_colind;             // column indices of non-zeros              (nD x 1)

  // computed by mj_implicit/mj_derivative
  mjtNum*   qDeriv;               // d (passive + actuator - bias) / d qvel   (nD x 1)

  // computed by mj_implicit/mju_factorLUSparse
  mjtNum*   qLU;                  // sparse LU of (qM - dt*qDeriv)            (nD x 1)

  //------------------------------- POSITION, VELOCITY, CONTROL/ACCELERATION dependent

  // computed by mj_fwdActuation
  mjtNum*   actuator_force;       // actuator force in actuation space        (nu x 1)
  mjtNum*   qfrc_actuator;        // actuator force                           (nv x 1)

  // computed by mj_fwdAcceleration
  mjtNum*   qfrc_smooth;          // net unconstrained force                  (nv x 1)
  mjtNum*   qacc_smooth;          // unconstrained acceleration               (nv x 1)

  // computed by mj_fwdConstraint/mj_inverse
  mjtNum*   qfrc_constraint;      // constraint force                         (nv x 1)

  // computed by mj_inverse
  mjtNum*   qfrc_inverse;         // net external force; should equal:        (nv x 1)
                                  // qfrc_applied + J'*xfrc_applied + qfrc_actuator

  // computed by mj_sensorAcc/mj_rnePostConstraint if needed; rotation:translation format
  mjtNum*   cacc;                 // com-based acceleration                   (nbody x 6)
  mjtNum*   cfrc_int;             // com-based interaction force with parent  (nbody x 6)
  mjtNum*   cfrc_ext;             // com-based external force on body         (nbody x 6)

  //------------------------------- ARENA-ALLOCATED ARRAYS

  // computed by mj_collision
  mjContact* contact;             // list of all detected contacts            (ncon x 1)

  // computed by mj_makeConstraint
  int*      efc_type;             // constraint type (mjtConstraint)          (nefc x 1)
  int*      efc_id;               // id of object of specified type           (nefc x 1)
  int*      efc_J_rownnz;         // number of non-zeros in Jacobian row      (nefc x 1)
  int*      efc_J_rowadr;         // row start address in colind array        (nefc x 1)
  int*      efc_J_rowsuper;       // number of subsequent rows in supernode   (nefc x 1)
  int*      efc_J_colind;         // column indices in Jacobian               (nefc x nv)
  int*      efc_JT_rownnz;        // number of non-zeros in Jacobian row    T (nv x 1)
  int*      efc_JT_rowadr;        // row start address in colind array      T (nv x 1)
  int*      efc_JT_rowsuper;      // number of subsequent rows in supernode T (nv x 1)
  int*      efc_JT_colind;        // column indices in Jacobian             T (nv x nefc)
  mjtNum*   efc_J;                // constraint Jacobian                      (nefc x nv)
  mjtNum*   efc_JT;               // constraint Jacobian transposed           (nv x nefc)
  mjtNum*   efc_pos;              // constraint position (equality, contact)  (nefc x 1)
  mjtNum*   efc_margin;           // inclusion margin (contact)               (nefc x 1)
  mjtNum*   efc_frictionloss;     // frictionloss (friction)                  (nefc x 1)
  mjtNum*   efc_diagApprox;       // approximation to diagonal of A           (nefc x 1)
  mjtNum*   efc_KBIP;             // stiffness, damping, impedance, imp'      (nefc x 4)
  mjtNum*   efc_D;                // constraint mass                          (nefc x 1)
  mjtNum*   efc_R;                // inverse constraint mass                  (nefc x 1)

  // computed by mj_fwdConstraint/mj_inverse
  mjtNum*   efc_b;                // linear cost term: J*qacc_smooth - aref   (nefc x 1)
  mjtNum*   efc_force;            // constraint force in constraint space     (nefc x 1)
  int*      efc_state;            // constraint state (mjtConstraintState)    (nefc x 1)

  // computed by mj_projectConstraint
  int*      efc_AR_rownnz;        // number of non-zeros in AR                (nefc x 1)
  int*      efc_AR_rowadr;        // row start address in colind array        (nefc x 1)
  int*      efc_AR_colind;        // column indices in sparse AR              (nefc x nefc)
  mjtNum*   efc_AR;               // J*inv(M)*J' + R                          (nefc x nefc)
};
typedef struct mjData_ mjData;
```

Defined in mjdata.h

This is the main data structure holding the simulation state. It is the workspace where all functions read their modifiable inputs and write their outputs.

mjvPerturb

```
struct mjvPerturb_ {              // object selection and perturbation
  int      select;                // selected body id; non-positive: none
  int      skinselect;            // selected skin id; negative: none
  int      active;                // perturbation bitmask (mjtPertBit)
  int      active2;               // secondary perturbation bitmask (mjtPertBit)
  mjtNum   refpos[3];             // desired position for selected object
  mjtNum   refquat[4];            // desired orientation for selected object
  mjtNum   localpos[3];           // selection point in object coordinates
  mjtNum   scale;                 // relative mouse motion-to-space scaling (set by initPertu
};
typedef struct mjvPerturb_ mjvPerturb;
```

Defined in mjvisualize.h

This is the data structure holding information about mouse perturbations.

mjvCamera

```
struct mjvCamera_ {               // abstract camera
  // type and ids
  int      type;                  // camera type (mjtCamera)
  int      fixedcamid;            // fixed camera id
  int      trackbodyid;           // body id to track
```

```
  // abstract camera pose specification
  mjtNum   lookat[3];             // lookat point
  mjtNum   distance;              // distance to lookat point or tracked body
  mjtNum   azimuth;               // camera azimuth (deg)
  mjtNum   elevation;             // camera elevation (deg)
};
typedef struct mjvCamera_ mjvCamera;
```

Defined in mjvisualize.h

This is the data structure describing one abstract camera.

mjvGLCamera

```
struct mjvGLCamera_ {             // OpenGL camera
  // camera frame
  float    pos[3];                // position
  float    forward[3];            // forward direction
  float    up[3];                 // up direction

  // camera projection
  float    frustum_center;        // hor. center (left,right set to match aspect)
  float    frustum_bottom;        // bottom
  float    frustum_top;           // top
  float    frustum_near;          // near
  float    frustum_far;           // far
};
typedef struct mjvGLCamera_ mjvGLCamera;
```

Defined in mjvisualize.h

This is the data structure describing one OpenGL camera.

mjvGeom

```
struct mjvGeom_ {                 // abstract geom
  // type info
  int      type;                  // geom type (mjtGeom)
  int      dataid;                // mesh, hfield or plane id; -1: none
  int      objtype;               // mujoco object type; mjOBJ_UNKNOWN for decor
  int      objid;                 // mujoco object id; -1 for decor
  int      category;              // visual category
  int      texid;                 // texture id; -1: no texture
  int      texuniform;            // uniform cube mapping
  int      texcoord;              // mesh geom has texture coordinates
  int      segid;                 // segmentation id; -1: not shown

  // OpenGL info
  float    texrepeat[2];          // texture repetition for 2D mapping
  float    size[3];               // size parameters
  float    pos[3];                // Cartesian position
  float    mat[9];                // Cartesian orientation
  float    rgba[4];               // color and transparency
  float    emission;              // emission coef
  float    specular;              // specular coef
  float    shininess;             // shininess coef
  float    reflectance;           // reflectance coef
  char     label[100];            // text label

  // transparency rendering (set internally)
  float    camdist;               // distance to camera (used by sorter)
  float    modelrbound;           // geom rbound from model, 0 if not model geom
  mjtByte  transparent;           // treat geom as transparent
};
typedef struct mjvGeom_ mjvGeom;
```

Defined in mjvisualize.h

This is the data structure describing one abstract visualization geom - which could correspond to a model geom or to a decoration element constructed by the visualizer.

mjvLight

```
struct mjvLight_ {                // OpenGL light
  float    pos[3];                // position rel. to body frame
  float    dir[3];                // direction rel. to body frame
  float    attenuation[3];        // OpenGL attenuation (quadratic model)
  float    cutoff;                // OpenGL cutoff
  float    exponent;              // OpenGL exponent
  float    ambient[3];            // ambient rgb (alpha=1)
  float    diffuse[3];            // diffuse rgb (alpha=1)
  float    specular[3];           // specular rgb (alpha=1)
  mjtByte  headlight;             // headlight
  mjtByte  directional;           // directional light
  mjtByte  castshadow;            // does light cast shadows
};
typedef struct mjvLight_ mjvLight;
```

Defined in mjvisualize.h

This is the data structure describing one OpenGL light.

mjvOption

```
struct mjvOption_ {               // abstract visualization options
  int      label;                 // what objects to label (mjtLabel)
  int      frame;                 // which frame to show (mjtFrame)
  mjtByte  geomgroup[mjNGROUP];   // geom visualization by group
  mjtByte  sitegroup[mjNGROUP];   // site visualization by group
  mjtByte  jointgroup[mjNGROUP];  // joint visualization by group
  mjtByte  tendongroup[mjNGROUP]; // tendon visualization by group
  mjtByte  actuatorgroup[mjNGROUP]; // actuator visualization by group
  mjtByte  skingroup[mjNGROUP];   // skin visualization by group
  mjtByte  flags[mjNVISFLAG];     // visualization flags (indexed by mjtVisFlag)
};
typedef struct mjvOption_ mjvOption;
```

Defined in mjvisualize.h

This structure contains options that enable and disable the visualization of various elements.

mjvScene

```
struct mjvScene_ {                  // abstract scene passed to OpenGL renderer
  // abstract geoms
  int      maxgeom;                 // size of allocated geom buffer
  int      ngeom;                   // number of geoms currently in buffer
  mjvGeom* geoms;                   // buffer for geoms (ngeom)
  int*     geomorder;               // buffer for ordering geoms by distance to camera (ngeom)

  // skin data
  int      nskin;                   // number of skins
  int*     skinfacenum;             // number of faces in skin (nskin)
  int*     skinvertadr;             // address of skin vertices (nskin)
  int*     skinvertnum;             // number of vertices in skin (nskin)
  float*   skinvert;                // skin vertex data (nskin)
  float*   skinnormal;              // skin normal data (nskin)

  // OpenGL lights
  int      nlight;                  // number of lights currently in buffer
  mjvLight lights[mjMAXLIGHT];      // buffer for lights (nlight)

  // OpenGL cameras
  mjvGLCamera camera[2];            // left and right camera

  // OpenGL model transformation
  mjtByte  enabletransform;         // enable model transformation
  float    translate[3];            // model translation
  float    rotate[4];               // model quaternion rotation
  float    scale;                   // model scaling

  // OpenGL rendering effects
  int      stereo;                  // stereoscopic rendering (mjtStereo)
  mjtByte  flags[mjNRNDFLAG];       // rendering flags (indexed by mjtRndFlag)

  // framing
  int      framewidth;              // frame pixel width; 0: disable framing
  float    framergb[3];             // frame color
};
typedef struct mjvScene_ mjvScene;
```

Defined in mjvisualize.h

This structure contains everything needed to render the 3D scene in OpenGL.

mjvFigure

```
struct mjvFigure_ {                 // abstract 2D figure passed to OpenGL renderer
  // enable flags
  int     flg_legend;               // show legend
  int     flg_ticklabel[2];         // show grid tick labels (x,y)
  int     flg_extend;               // automatically extend axis ranges to fit data
  int     flg_barplot;              // isolated line segments (i.e. GL_LINES)
  int     flg_selection;            // vertical selection line
  int     flg_symmetric;            // symmetric y-axis

  // style settings
  float   linewidth;                // line width
  float   gridwidth;                // grid line width
  int     gridsize[2];              // number of grid points in (x,y)
  float   gridrgb[3];               // grid line rgb
  float   figurergba[4];            // figure color and alpha
  float   panergba[4];              // pane color and alpha
  float   legendrgba[4];            // legend color and alpha
  float   textrgb[3];               // text color
  float   linergb[mjMAXLINE][3];    // line colors
  float   range[2][2];              // axis ranges; (min>=max) automatic
  char    xformat[20];              // x-tick label format for sprintf
  char    yformat[20];              // y-tick label format for sprintf
  char    minwidth[20];             // string used to determine min y-tick width

  // text labels
  char    title[1000];              // figure title; subplots separated with 2+ spaces
  char    xlabel[100];              // x-axis label
  char    linename[mjMAXLINE][100]; // line names for legend

  // dynamic settings
  int     legendoffset;             // number of lines to offset legend
  int     subplot;                  // selected subplot (for title rendering)
  int     highlight[2];             // if point is in legend rect, highlight line
  int     highlightid;              // if id>=0 and no point, highlight id
  float   selection;                // selection line x-value

  // line data
  int     linepnt[mjMAXLINE];       // number of points in line; (0) disable
  float   linedata[mjMAXLINE][2*mjMAXLINEPNT]; // line data (x,y)

  // output from renderer
  int     xaxispixel[2];            // range of x-axis in pixels
  int     yaxispixel[2];            // range of y-axis in pixels
  float   xaxisdata[2];             // range of x-axis in data units
  float   yaxisdata[2];             // range of y-axis in data units
};
typedef struct mjvFigure_ mjvFigure;
```

Defined in mjvisualize.h

This structure contains everything needed to render a 2D plot in OpenGL. The buffers for line points etc. are preallocated, and the user has to populate them before calling the function mjr_figure with this data structure as an argument.

mjrRect

```
struct mjrRect_ {                   // OpenGL rectangle
  int left;                         // left (usually 0)
  int bottom;                       // bottom (usually 0)
  int width;                        // width (usually buffer width)
  int height;                       // height (usually buffer height)
};
typedef struct mjrRect_ mjrRect;
```

Defined in mjrender.h (57)

This structure specifies a rectangle.

mjrContext

```
struct mjrContext_ {                // custom OpenGL context
  // parameters copied from mjVisual
  float lineWidth;                  // line width for wireframe rendering
  float shadowClip;                 // clipping radius for directional lights
  float shadowScale;                // fraction of light cutoff for spot lights
  float fogStart;                   // fog start = stat.extent * vis.map.fogstart
  float fogEnd;                     // fog end = stat.extent * vis.map.fogend
  float fogRGBA[4];                 // fog rgba
  int shadowSize;                   // size of shadow map texture
  int offWidth;                     // width of offscreen buffer
  int offHeight;                    // height of offscreen buffer
  int offSamples;                   // number of offscreen buffer multisamples

  // parameters specified at creation
  int fontScale;                    // font scale
  int auxWidth[mjNAUX];             // auxiliary buffer width
  int auxHeight[mjNAUX];            // auxiliary buffer height
  int auxSamples[mjNAUX];           // auxiliary buffer multisamples

  // offscreen rendering objects
  unsigned int offFBO;              // offscreen framebuffer object
  unsigned int offFBO_r;            // offscreen framebuffer for resolving multisamples
  unsigned int offColor;            // offscreen color buffer
  unsigned int offColor_r;          // offscreen color buffer for resolving multisamples
  unsigned int offDepthStencil;     // offscreen depth and stencil buffer
  unsigned int offDepthStencil_r; // offscreen depth and stencil buffer for resolving multisa

  // shadow rendering objects
  unsigned int shadowFBO;           // shadow map framebuffer object
  unsigned int shadowTex;           // shadow map texture

  // auxiliary buffers
  unsigned int auxFBO[mjNAUX];      // auxiliary framebuffer object
  unsigned int auxFBO_r[mjNAUX];    // auxiliary framebuffer object for resolving
  unsigned int auxColor[mjNAUX];    // auxiliary color buffer
  unsigned int auxColor_r[mjNAUX];// auxiliary color buffer for resolving

  // texture objects and info
  int ntexture;                     // number of allocated textures
  int textureType[100];             // type of texture (mjtTexture) (ntexture)
  unsigned int texture[100];        // texture names

  // displaylist starting positions
  unsigned int basePlane;           // all planes from model
  unsigned int baseMesh;            // all meshes from model
  unsigned int baseHField;          // all hfields from model
  unsigned int baseBuiltin;         // all buildin geoms, with quality from model
  unsigned int baseFontNormal;      // normal font
  unsigned int baseFontShadow;      // shadow font
  unsigned int baseFontBig;         // big font

  // displaylist ranges
  int rangePlane;                   // all planes from model
  int rangeMesh;                    // all meshes from model
  int rangeHField;                  // all hfields from model
  int rangeBuiltin;                 // all builtin geoms, with quality from model
  int rangeFont;                    // all characters in font

  // skin VBOs
  int nskin;                        // number of skins
  unsigned int* skinvertVBO;        // skin vertex position VBOs (nskin)
  unsigned int* skinnormalVBO;      // skin vertex normal VBOs (nskin)
  unsigned int* skintexcoordVBO;  // skin vertex texture coordinate VBOs (nskin)
  unsigned int* skinfaceVBO;        // skin face index VBOs (nskin)

  // character info
  int charWidth[127];               // character widths: normal and shadow
  int charWidthBig[127];            // chacarter widths: big
  int charHeight;                   // character heights: normal and shadow
  int charHeightBig;                // character heights: big

  // capabilities
  int glInitialized;                // is OpenGL initialized
  int windowAvailable;              // is default/window framebuffer available
  int windowSamples;                // number of samples for default/window framebuffer
  int windowStereo;                 // is stereo available for default/window framebuffer
  int windowDoublebuffer;           // is default/window framebuffer double buffered

  // framebuffer
  int     currentBuffer;            // currently active framebuffer: mjFB_WINDOW or mjFB_OFFSCR
};
typedef struct mjrContext_ mjrContext;
```

Defined in mjrender.h (67)

This structure contains the custom OpenGL rendering context, with the ids of all OpenGL resources uploaded to the GPU.

mjuiState

```
struct mjuiState_ {                 // mouse and keyboard state
  // constants set by user
  int nrect;                        // number of rectangles used
  mjrRect rect[mjMAXUIRECT];        // rectangles (index 0: entire window)
  void* userdata;                   // pointer to user data (for callbacks)

  // event type
  int type;                         // (type mjtEvent)

  // mouse buttons
  int left;                         // is left button down
  int right;                        // is right button down
  int middle;                       // is middle button down
  int doubleclick;                  // is last press a double click
  int button;                       // which button was pressed (mjtButton)
  double buttontime;                // time of last button press

  // mouse position
  double x;                         // x position
  double y;                         // y position
  double dx;                        // x displacement
  double dy;                        // y displacement
  double sx;                        // x scroll
```

```
  double sy;                      // y scroll

  // keyboard
  int control;                    // is control down
  int shift;                      // is shift down
  int alt;                        // is alt down
  int key;                        // which key was pressed
  double keytime;                 // time of last key press

  // rectangle ownership and dragging
  int mouserect;                  // which rectangle contains mouse
  int dragrect;                   // which rectangle is dragged with mouse
  int dragbutton;                 // which button started drag (mjtButton)
};
typedef struct mjuiState_ mjuiState;
```

Defined in mjui.h

This structure contains the keyboard and mouse state used by the UI framework.

mjuiThemeSpacing

```
struct mjuiThemeSpacing_ {        // UI visualization theme spacing
  int total;                      // total width
  int scroll;                     // scrollbar width
  int label;                      // label width
  int section;                    // section gap
  int itemside;                   // item side gap
  int itemmid;                    // item middle gap
  int itemver;                    // item vertical gap
  int texthor;                    // text horizontal gap
  int textver;                    // text vertical gap
  int linescroll;                 // number of pixels to scroll
  int samples;                    // number of multisamples
};
typedef struct mjuiThemeSpacing_ mjuiThemeSpacing;
```

Defined in mjui.h

This structure defines the spacing of UI items in the theme.

mjuiThemeColor

```
struct mjuiThemeColor_ {          // UI visualization theme color
  float master[3];                // master background
  float thumb[3];                 // scrollbar thumb
  float secttitle[3];             // section title
  float sectfont[3];              // section font
  float sectsymbol[3];            // section symbol
  float sectpane[3];              // section pane
  float shortcut[3];              // shortcut background
  float fontactive[3];            // font active
  float fontinactive[3];          // font inactive
  float decorinactive[3];         // decor inactive
  float decorinactive2[3];        // inactive slider color 2
  float button[3];                // button
  float check[3];                 // check
  float radio[3];                 // radio
  float select[3];                // select
  float select2[3];               // select pane
  float slider[3];                // slider
  float slider2[3];               // slider color 2
  float edit[3];                  // edit
  float edit2[3];                 // edit invalid
  float cursor[3];                // edit cursor
};
typedef struct mjuiThemeColor_ mjuiThemeColor;
```

Defined in mjui.h

This structure defines the colors of UI items in the theme.

mjuiItem

```
struct mjuiItemSingle_ {          // check and button-related
  int modifier;                   // 0: none, 1: control, 2: shift; 4: alt
  int shortcut;                   // shortcut key; 0: undefined
};


struct mjuiItemMulti_ {           // static, radio and select-related
  int nelem;                      // number of elements in group
  char name[mjMAXUIMULTI][mjMAXUINAME]; // element names
};


struct mjuiItemSlider_ {          // slider-related
  double range[2];                // slider range
  double divisions;               // number of range divisions
};


struct mjuiItemEdit_ {            // edit-related
  int nelem;                      // number of elements in list
  double range[mjMAXUIEDIT][2];   // element range (min>=max: ignore)
};


struct mjuiItem_ {                // UI item
  // common properties
  int type;                       // type (mjtItem)
  char name[mjMAXUINAME];         // name
  int state;                      // 0: disable, 1: enable, 2+: use predicate
  void *pdata;                    // data pointer (type-specific)
  int sectionid;                  // id of section containing item
  int itemid;                     // id of item within section

  // type-specific properties
  union {
    struct mjuiItemSingle_ single; // check and button
    struct mjuiItemMulti_ multi;   // static, radio and select
    struct mjuiItemSlider_ slider; // slider
    struct mjuiItemEdit_ edit;     // edit
  };
```

```
  // internal
  mjrRect rect;                   // rectangle occupied by item
};
typedef struct mjuiItem_ mjuiItem;
```

Defined in mjui.h

This structure defines one UI item.

mjuiSection

```
struct mjuiSection_ {             // UI section
  // properties
  char name[mjMAXUINAME];         // name
  int state;                      // 0: closed, 1: open
  int modifier;                   // 0: none, 1: control, 2: shift; 4: alt
  int shortcut;                   // shortcut key; 0: undefined
  int nitem;                      // number of items in use
  mjuiItem item[mjMAXUIITEM];     // preallocated array of items

  // internal
  mjrRect rtitle;                 // rectangle occupied by title
  mjrRect rcontent;               // rectangle occupied by content
};
typedef struct mjuiSection_ mjuiSection;
```

Defined in mjui.h

This structure defines one section of the UI.

mjUI

```
struct mjUI_ {                    // entire UI
  // constants set by user
  mjuiThemeSpacing spacing;       // UI theme spacing
  mjuiThemeColor color;           // UI theme color
  mjfItemEnable predicate;        // callback to set item state programmatically
  void* userdata;                 // pointer to user data (passed to predicate)
  int rectid;                     // index of this ui rectangle in mjuiState
  int auxid;                      // aux buffer index of this ui
  int radiocol;                   // number of radio columns (0 defaults to 2)

  // UI sizes (framebuffer units)
  int width;                      // width
  int height;                     // current heigth
  int maxheight;                  // height when all sections open
  int scroll;                     // scroll from top of UI

  // mouse focus
  int mousesect;                  // 0: none, -1: scroll, otherwise 1+section
  int mouseitem;                  // item within section
  int mousehelp;                  // help button down: print shortcuts

  // keyboard focus and edit
  int editsect;                   // 0: none, otherwise 1+section
  int edititem;                   // item within section
  int editcursor;                 // cursor position
  int editscroll;                 // horizontal scroll
  char edittext[mjMAXUITEXT];     // current text
  mjuiItem* editchanged;          // pointer to changed edit in last mjui_event

  // sections
  int nsect;                      // number of sections in use
  mjuiSection sect[mjMAXUISECT];  // preallocated array of sections
};
typedef struct mjUI_ mjUI;
```

Defined in mjui.h

This structure defines the entire UI.

mjuiDef

```
struct mjuiDef_ {                 // table passed to mjui_add()
  int type;                       // type (mjtItem); -1: section
  char name[mjMAXUINAME];         // name
  int state;                      // state
  void* pdata;                    // pointer to data
  char other[mjMAXUITEXT];        // string with type-specific properties
};
typedef struct mjuiDef_ mjuiDef;
```

Defined in mjui.h

This structure defines one entry in the definition table used for simplified UI construction.

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.

MJOPTION_SCALARS

Scalar fields of mjOption.

MJOPTION_VECTORS

Vector fields of mjOption.

MJMODEL_INTS

Int fields of mjModel.

MJMODEL_POINTERS

Pointer fields of mjModel.

MJDATA_SCALAR

Scalar fields of mjData.

MJDATA_VECTOR

Vector fields of mjData.

MJDATA_POINTERS

Pointer fields of mjData.

Global variables

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

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

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.

mju_user_warning

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

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.

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

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

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.

mju_user_free

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

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

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.

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

```
extern mjfGeneric 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).

mjcb_control

```
extern mjfGeneric 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.

mjcb_contactfilter

```
extern mjfConFilt 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.

mjcb_sensor

```
extern mjfSensor 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.

mjcb_time

```
extern mjfTime 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.

mjcb_act_dyn

```
extern mjfAct 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.

mjcb_act_gain

```
extern mjfAct 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.

mjcb_act_bias

```
extern mjfAct 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.

Collision table

mjCOLLISIONFUNC

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

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.

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

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

Names of the disable bits defined by mjtDisableBit.

mjENABLESTRING

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

Names of the enable bits defined by mjtEnableBit.

mjTIMERSTRING

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

Names of the mjData timers defined by mjtTimer.

mjLABELSTRING

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

Names of the visual labeling modes defined by mjtLabel.

mjFRAMESTRING

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

Names of the frame visualization modes defined by mjtFrame.

mjVISSTRING

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

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.

mjRNDSTRING

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

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

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	pi	The value of pi. 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 automatically clamped to this constant.
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.
mjMAXVFS	200	The maximal number of files in the virtual file system.
mjMAXVFSNAME	100	The maximal number of characters in the name of each file in the virtual file system.
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 parameters used to define the reference acceleration of each scalar 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	1000	The size of the preallocated array `mjData.solver`. This is used to store diagnostic information about each iteration of the

		constraint solver. The actual number of iterations is given by `mjData.solver_iter`.
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.
mjMAXUISECT	10	Maximum number of UI sections. Defined in mjui.h.
mjMAXUIITEM	80	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	211	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.

API functions

The main header mujoco.h exposes a very large number of functions. However the functions that most users are likely to need are a small fraction. For example, simulate.cc which is as elaborate as a MuJoCo application is likely to get, calls around 40 of these functions, while `basic.cc` calls around 20. The rest are explosed just in case someone has a use for them. This includes us as users of MuJoCo – we do our own work with the public library instead of relying on internal builds.

Activation

The functions in this section are maintained for backward compatibility with the now-removed activation mechanism.

mj_activate

```
int mj_activate(const char* filename);
```

Does nothing, returns 1.

mj_deactivate

```
void mj_deactivate(void);
```

Does nothing.

Virtual file system

Virtual file system (VFS) functionality was introduced in MuJoCo 1.50. It enables the user to load all necessary files in memory, including MJB binary model files, XML files (MJCF, URDF and included files), STL meshes, PNGs for textures and height fields, and HF files in our custom height field format. Model and resource files in the VFS can also be constructed programmatically (say using a Python library that writes to memory). Once all desired files are in the VFS, the user can call mj_loadModel or mj_loadXML with a pointer to the VFS. When this pointer is not NULL, the loaders will first check the VFS for any file they are about to load, and only access the disk if the file is not found in the VFS. The file names stored in the VFS have their name and extension but the path information is stripped; this can be bypassed however by using a custom path symbol in the file names, say “mydir_myfile.xml”.

The entire VFS is contained in the data structure mjVFS. All utility functions for maintaining the VFS operate on this data structure. The common usage pattern is to first clear it with mj_defaultVFS, then add disk files to it with mj_addFileVFS (which allocates memory buffers and loads the file content in memory), then call mj_loadXML or mj_loadModel, and then clear everything with mj_deleteVFS.

mj_defaultVFS

```
void mj_defaultVFS(mjVFS* vfs);
```

Initialize VFS to empty (no deallocation).

mj_addFileVFS

```
int mj_addFileVFS(mjVFS* vfs, const char* directory, const char* filename);
```

Add file to VFS, return 0: success, 1: full, 2: repeated name, -1: not found on disk.

mj_makeEmptyFileVFS

```
int mj_makeEmptyFileVFS(mjVFS* vfs, const char* filename, int filesize);
```

Make empty file in VFS, return 0: success, 1: full, 2: repeated name.

mj_findFileVFS

```
int mj_findFileVFS(const mjVFS* vfs, const char* filename);
```

Return file index in VFS, or -1 if not found in VFS.

mj_deleteFileVFS

```
int mj_deleteFileVFS(mjVFS* vfs, const char* filename);
```

Delete file from VFS, return 0: success, -1: not found in VFS.

mj_deleteVFS

```
void mj_deleteVFS(mjVFS* vfs);
```

Delete all files from VFS.

Parse and compile

The key function here is mj_loadXML. It invokes the built-in parser and compiler, and either returns a pointer to a valid mjModel, or NULL - in which case the user should check the error information in the user-provided string. The model and all files referenced in it can be loaded from disk or from a VFS when provided.

mj_loadXML

```
mjModel* mj_loadXML(const char* filename, const mjVFS* vfs,
                    char* error, int error_sz);
```

Parse XML file in MJCF or URDF format, compile it, return low-level model. If vfs is not NULL, look up files in vfs before reading from disk. If error is not NULL, it must have size error_sz.

mj_saveLastXML

```
int mj_saveLastXML(const char* filename, const mjModel* m,
                   char* error, int error_sz);
```

Update XML data structures with info from low-level model, save as MJCF. If error is not NULL, it must have size error_sz.

mj_freeLastXML

```
void mj_freeLastXML(void);
```

Free last XML model if loaded. Called internally at each load.

mj_printSchema

```
int mj_printSchema(const char* filename, char* buffer, int buffer_sz,
                   int flg_html, int flg_pad);
```

Print internal XML schema as plain text or HTML, with style-padding or ` `.

Main simulation

These are the main entry points to the simulator. Most users will only need to call `mj_step`, which computes everything and advanced the simulation state by one time step. Controls and applied forces must either be set in advance (in mjData.ctrl, qfrc_applied and xfrc_applied), or a control callback mjcb_control must be installed which will be called just before the controls and applied forces are needed. Alternatively, one can use `mj_step1` and `mj_step2` which break down the simulation pipeline into computations that are executed before and after the controls are needed; in this way one can set controls that depend on the results from `mj_step1`. Keep in mind though that the RK4 solver does not work with mj_step1/2.

mj_forward performs the same computations as `mj_step` but without the integration. It is useful after loading or resetting a model (to put the entire mjData in a valid state), and also for out-of-order computations that involve sampling or finite-difference approximations.

mj_inverse runs the inverse dynamics, and writes its output in mjData.qfrc_inverse. Note that mjData.qacc must be set before calling this function. Given the state (qpos, qvel, act), mj_forward maps from force to acceleration, while mj_inverse maps from acceleration to force. Mathematically these functions are inverse of each other, but numerically this may not always be the case because the forward dynamics rely on a constraint optimization algorithm which is usually terminated early. The difference between the results of forward and inverse dynamics can be computed with the function mj_compareFwdInv, which can be though of as another solver accuracy check (as well as a general sanity check).

The skip version of mj_forward and mj_inverse are useful for example when qpos was unchanged but qvel was changed (usually in the context of finite differencing). Then there is no point repeating the computations that only depend on qpos. Calling the dynamics with skipstage = mjSTAGE_POS will achieve these savings.

mj_step

```
void mj_step(const mjModel* m, mjData* d);
```

Advance simulation, use control callback to obtain external force and control.

mj_step1

```
void mj_step1(const mjModel* m, mjData* d);
```

Advance simulation in two steps: before external force and control is set by user.

mj_step2

```
void mj_step2(const mjModel* m, mjData* d);
```

Advance simulation in two steps: after external force and control is set by user.

mj_forward

```
void mj_forward(const mjModel* m, mjData* d);
```

Forward dynamics: same as mj_step but do not integrate in time.

mj_inverse

```
void mj_inverse(const mjModel* m, mjData* d);
```

Inverse dynamics: qacc must be set before calling.

mj_forwardSkip

```
void mj_forwardSkip(const mjModel* m, mjData* d, int skipstage, int skipsensor);
```

Forward dynamics with skip; skipstage is mjtStage.

mj_inverseSkip

```
void mj_inverseSkip(const mjModel* m, mjData* d, int skipstage, int skipsensor);
```

Inverse dynamics with skip; skipstage is mjtStage.

Initialization

This section contains functions that load/initialize the model or other data structures. Their use is well illustrated in the code samples.

mj_defaultLROpt

```
void mj_defaultLROpt(mjLROpt* opt);
```

Set default options for length range computation.

mj_defaultSolRefImp

```
void mj_defaultSolRefImp(mjtNum* solref, mjtNum* solimp);
```

Set solver parameters to default values.

mj_defaultOption

```
void mj_defaultOption(mjOption* opt);
```

Set physics options to default values.

mj_defaultVisual

```
void mj_defaultVisual(mjVisual* vis);
```

Set visual options to default values.

mj_copyModel

```
mjModel* mj_copyModel(mjModel* dest, const mjModel* src);
```

Copy mjModel, allocate new if dest is NULL.

mj_saveModel

```
void mj_saveModel(const mjModel* m, const char* filename, void* buffer, int buffer_sz);
```

Save model to binary MJB file or memory buffer; buffer has precedence when given.

mj_loadModel

```
mjModel* mj_loadModel(const char* filename, const mjVFS* vfs);
```

Load model from binary MJB file. If vfs is not NULL, look up file in vfs before reading from disk.

mj_deleteModel

```
void mj_deleteModel(mjModel* m);
```

Free memory allocation in model.

mj_sizeModel

```
int mj_sizeModel(const mjModel* m);
```

Return size of buffer needed to hold model.

mj_makeData

```
mjData* mj_makeData(const mjModel* m);
```

Allocate mjData corresponding to given model.

mj_copyData

```
mjData* mj_copyData(mjData* dest, const mjModel* m, const mjData* src);
```

Copy mjData.

mj_resetData

```
void mj_resetData(const mjModel* m, mjData* d);
```

Reset data to defaults.

mj_resetDataDebug

```
void mj_resetDataDebug(const mjModel* m, mjData* d, unsigned char debug_value);
```

Reset data to defaults, fill everything else with debug_value.

mj_resetDataKeyframe

```
void mj_resetDataKeyframe(const mjModel* m, mjData* d, int key);
```

Reset data, set fields from specified keyframe.

mj_stackAlloc

```
mjtNum* mj_stackAlloc(mjData* d, int size);
```

Allocate array of specified size on mjData stack. Call mju_error on stack overflow.

mj_deleteData

```
void mj_deleteData(mjData* d);
```

Free memory allocation in mjData.

mj_resetCallbacks

```
void mj_resetCallbacks(void);
```

Reset all callbacks to NULL pointers (NULL is the default).

mj_setConst

```
void mj_setConst(mjModel* m, mjData* d);
```

Set constant fields of mjModel, corresponding to qpos0 configuration.

mj_setLengthRange

```
int mj_setLengthRange(mjModel* m, mjData* d, int index,
                      const mjLROpt* opt, char* error, int error_sz);
```

Set actuator_lengthrange for specified actuator; return 1 if ok, 0 if error.

Printing

These functions can be used to print various quantities to the screen for debugging purposes.

mj_printFormattedModel

```
void mj_printFormattedModel(const mjModel* m, const char* filename, const char* float_format)
```

Print `mjModel` to text file, specifying format. `float_format` must be a valid printf-style format string for a single float value.

mj_printModel

```
void mj_printModel(const mjModel* m, const char* filename);
```

Print model to text file.

mj_printFormattedData

```
void mj_printFormattedData(const mjModel* m, mjData* d, const char* filename,
                           const char* float_format);
```

Print `mjData` to text file, specifying format. `float_format` must be a valid printf-style format string for a single float value.

mj_printData

```
void mj_printData(const mjModel* m, mjData* d, const char* filename);
```

Print data to text file.

mju_printMat

```
void mju_printMat(const mjtNum* mat, int nr, int nc);
```

Print matrix to screen.

mju_printMatSparse

```
void mju_printMatSparse(const mjtNum* mat, int nr,
                        const int* rownnz, const int* rowadr, const int* colind);
```

Print sparse matrix to screen.

Components

These are components of the simulation pipeline, called internally from mj_step, mj_forward and mj_inverse. It is unlikely that the user will need to call them.

mj_fwdPosition

```
void mj_fwdPosition(const mjModel* m, mjData* d);
```

Run position-dependent computations.

mj_fwdVelocity

```
void mj_fwdVelocity(const mjModel* m, mjData* d);
```

Run velocity-dependent computations.

mj_fwdActuation

```
void mj_fwdActuation(const mjModel* m, mjData* d);
```

Compute actuator force qfrc_actuator.

mj_fwdAcceleration

```
void mj_fwdAcceleration(const mjModel* m, mjData* d);
```

Add up all non-constraint forces, compute qacc_smooth.

mj_fwdConstraint

```
void mj_fwdConstraint(const mjModel* m, mjData* d);
```

Run selected constraint solver.

mj_Euler

```
void mj_Euler(const mjModel* m, mjData* d);
```

Euler integrator, semi-implicit in velocity.

mj_RungeKutta

```
void mj_RungeKutta(const mjModel* m, mjData* d, int N);
```

Runge-Kutta explicit order-N integrator.

mj_invPosition

```
void mj_invPosition(const mjModel* m, mjData* d);
```

Run position-dependent computations in inverse dynamics.

mj_invVelocity

```
void mj_invVelocity(const mjModel* m, mjData* d);
```

Run velocity-dependent computations in inverse dynamics.

mj_invConstraint

```
void mj_invConstraint(const mjModel* m, mjData* d);
```

Apply the analytical formula for inverse constraint dynamics.

mj_compareFwdInv

```
void mj_compareFwdInv(const mjModel* m, mjData* d);
```

Compare forward and inverse dynamics, save results in fwdinv.

Sub components

These are sub-components of the simulation pipeline, called internally from the components above. It is very unlikely that the user will need to call them.

mj_sensorPos

```
void mj_sensorPos(const mjModel* m, mjData* d);
```

Evaluate position-dependent sensors.

mj_sensorVel

```
void mj_sensorVel(const mjModel* m, mjData* d);
```

Evaluate velocity-dependent sensors.

mj_sensorAcc

```
void mj_sensorAcc(const mjModel* m, mjData* d);
```

Evaluate acceleration and force-dependent sensors.

mj_energyPos

```
void mj_energyPos(const mjModel* m, mjData* d);
```

Evaluate position-dependent energy (potential).

mj_energyVel

```
void mj_energyVel(const mjModel* m, mjData* d);
```

Evaluate velocity-dependent energy (kinetic).

mj_checkPos

```
void mj_checkPos(const mjModel* m, mjData* d);
```

Check qpos, reset if any element is too big or nan.

mj_checkVel

```
void mj_checkVel(const mjModel* m, mjData* d);
```

Check qvel, reset if any element is too big or nan.

mj_checkAcc

```
void mj_checkAcc(const mjModel* m, mjData* d);
```

Check qacc, reset if any element is too big or nan.

mj_kinematics

```
void mj_kinematics(const mjModel* m, mjData* d);
```

Run forward kinematics.

mj_comPos

```
void mj_comPos(const mjModel* m, mjData* d);
```

Map inertias and motion dofs to global frame centered at CoM.

mj_camlight

```
void mj_camlight(const mjModel* m, mjData* d);
```

Compute camera and light positions and orientations.

mj_tendon

```
void mj_tendon(const mjModel* m, mjData* d);
```

Compute tendon lengths, velocities and moment arms.

mj_transmission

```
void mj_transmission(const mjModel* m, mjData* d);
```

Compute actuator transmission lengths and moments.

mj_crb

```
void mj_crb(const mjModel* m, mjData* d);
```

Run composite rigid body inertia algorithm (CRB).

mj_factorM

```
void mj_factorM(const mjModel* m, mjData* d);
```

Compute sparse $L^T D L$ factorizaton of inertia matrix.

mj_solveM

```
void mj_solveM(const mjModel* m, mjData* d, mjtNum* x, const mjtNum* y, int n);
```

Solve linear system $Mx = y$ using factorization: $x = (L^T D L)^{-1} y$

mj_solveM2

```
void mj_solveM2(const mjModel* m, mjData* d, mjtNum* x, const mjtNum* y, int n);
```

Half of linear solve: $x = \sqrt{D^{-1}}(L^T)^{-1} y$

mj_comVel

```
void mj_comVel(const mjModel* m, mjData* d);
```

Compute cvel, cdof_dot.

mj_passive

```
void mj_passive(const mjModel* m, mjData* d);
```

Compute qfrc_passive from spring-dampers, viscosity and density.

mj_subtreeVel

```
void mj_subtreeVel(const mjModel* m, mjData* d);
```

subtree linear velocity and angular momentum

mj_rne

```
void mj_rne(const mjModel* m, mjData* d, int flg_acc, mjtNum* result);
```

RNE: compute M(qpos)*qacc + C(qpos,qvel); flg_acc=0 removes inertial term.

mj_rnePostConstraint

```
void mj_rnePostConstraint(const mjModel* m, mjData* d);
```

RNE with complete data: compute cacc, cfrc_ext, cfrc_int.

mj_collision

```
void mj_collision(const mjModel* m, mjData* d);
```

Run collision detection.

mj_makeConstraint

```
void mj_makeConstraint(const mjModel* m, mjData* d);
```

Construct constraints.

mj_projectConstraint

```
void mj_projectConstraint(const mjModel* m, mjData* d);
```

Compute inverse constraint inertia efc_AR.

mj_referenceConstraint

```
void mj_referenceConstraint(const mjModel* m, mjData* d);
```

Compute efc_vel, efc_aref.

. _mj_constraintUpdate:

mj_constraintUpdate

```
void mj_constraintUpdate(const mjModel* m, mjData* d, const mjtNum* jar,
                         mjtNum cost[1], int flg_coneHessian);
```

Compute efc_state, efc_force, qfrc_constraint, and (optionally) cone Hessians. If cost is not NULL, set *cost = s(jar) where jar = Jac*qacc-aref.

Support

These are support functions that need access to mjModel and mjData, unlike the utility functions which do not need such access. Support functions are called within the simulator but some of them can also be useful for custom computations, and are documented in more detail below.

mj_addContact

```
int mj_addContact(const mjModel* m, mjData* d, const mjContact* con);
```

Add contact to d->contact list; return 0 if success; 1 if buffer full.

mj_isPyramidal

```
int mj_isPyramidal(const mjModel* m);
```

Determine type of friction cone.

mj_isSparse

```
int mj_isSparse(const mjModel* m);
```

Determine type of constraint Jacobian.

mj_isDual

```
int mj_isDual(const mjModel* m);
```

Determine type of solver (PGS is dual, CG and Newton are primal).

mj_mulJacVec

```
void mj_mulJacVec(const mjModel* m, mjData* d, mjtNum* res, const mjtNum* vec);
```

This function multiplies the constraint Jacobian mjData.efc_J by a vector. Note that the Jacobian can be either dense or sparse; the function is aware of this setting. Multiplication by J maps velocities from joint space to constraint space.

mj_mulJacTVec

```
void mj_mulJacTVec(const mjModel* m, mjData* d, mjtNum* res, const mjtNum* vec);
```

Same as mj_mulJacVec but multiplies by the transpose of the Jacobian. This maps forces from constraint space to joint space.

mj_jac

```
void mj_jac(const mjModel* m, const mjData* d, mjtNum* jacp, mjtNum* jacr,
            const mjtNum point[3], int body);
```

This function computes an "end-effector" Jacobian, which is unrelated to the constraint Jacobian above. Any MuJoCo body can be treated as end-effector, and the point for which the Jacobian is computed can be anywhere in space (it is treated as attached to the body). The Jacobian has translational (jacp) and rotational (jacr) components. Passing NULL for either pointer will skip part of the computation. Each component is a 3-by-nv matrix. Each row of this matrix is the gradient of the corresponding 3D coordinate of the specified point with respect to the degrees of freedom. The ability to compute end-effector Jacobians analytically is one of the advantages of working in minimal coordinates - so use it!

mj_jacBody

```
void mj_jacBody(const mjModel* m, const mjData* d, mjtNum* jacp, mjtNum* jacr, int body);
```

This and the remaining variants of the Jacobian function call mj_jac internally, with the center of the body, geom or site. They are just shortcuts; the same can be achieved by calling mj_jac directly.

mj_jacBodyCom

```
void mj_jacBodyCom(const mjModel* m, const mjData* d, mjtNum* jacp, mjtNum* jacr, int body);
```

Compute body center-of-mass end-effector Jacobian.

mj_jacSubtreeCom

```
void mj_jacSubtreeCom(const mjModel* m, mjData* d, mjtNum* jacp, int body);
```

Compute subtree center-of-mass end-effector Jacobian. `jacp` is 3 x nv.

mj_jacGeom

```
void mj_jacGeom(const mjModel* m, const mjData* d, mjtNum* jacp, mjtNum* jacr, int geom);
```

Compute geom end-effector Jacobian.

mj_jacSite

```
void mj_jacSite(const mjModel* m, const mjData* d, mjtNum* jacp, mjtNum* jacr, int site);
```

Compute site end-effector Jacobian.

mj_jacPointAxis

```
void mj_jacPointAxis(const mjModel* m, mjData* d, mjtNum* jacPoint, mjtNum* jacAxis,
                     const mjtNum point[3], const mjtNum axis[3], int body);
```

Compute translation end-effector Jacobian of point, and rotation Jacobian of axis.

mj_name2id

```
int mj_name2id(const mjModel* m, int type, const char* name);
```

Get id of object with specified name, return -1 if not found; type is mjtObj.

mj_id2name

```
const char* mj_id2name(const mjModel* m, int type, int id);
```

Get name of object with specified id, return 0 if invalid type or id; type is mjtObj.

mj_fullM

```
void mj_fullM(const mjModel* m, mjtNum* dst, const mjtNum* M);
```

Convert sparse inertia matrix M into full (i.e. dense) matrix.

mj_mulM

```
void mj_mulM(const mjModel* m, const mjData* d, mjtNum* res, const mjtNum* vec);
```

This function multiplies the joint-space inertia matrix stored in mjData.qM by a vector. qM has a custom sparse format that the user should not attempt to manipulate directly. Alternatively one can convert qM to a dense matrix with mj_fullM and then user regular matrix-vector multiplication, but this is slower because it no longer benefits from sparsity.

mj_mulM2

```
void mj_mulM2(const mjModel* m, const mjData* d, mjtNum* res, const mjtNum* vec);
```

Multiply vector by (inertia matrix)^(1/2).

mj_addM

```
void mj_addM(const mjModel* m, mjData* d, mjtNum* dst, int* rownnz, int* rowadr, int* colind)
```

Add inertia matrix to destination matrix. Destination can be sparse uncompressed, or dense when all int* are NULL

mj_applyFT

```
void mj_applyFT(const mjModel* m, mjData* d, const mjtNum force[3], const mjtNum torque[3],
                const mjtNum point[3], int body, mjtNum* qfrc_target);
```

This function can be used to apply a Cartesian force and torque to a point on a body, and add the result to the vector mjData.qfrc_applied of all applied forces. Note that the function requires a pointer to this vector, because sometimes we want to add the result to a different vector.

mj_objectVelocity

```
void mj_objectVelocity(const mjModel* m, const mjData* d,
                       int objtype, int objid, mjtNum res[6], int flg_local);
```

Compute object 6D velocity in object-centered frame, world/local orientation.

mj_objectAcceleration

```
void mj_objectAcceleration(const mjModel* m, const mjData* d,
                           int objtype, int objid, mjtNum res[6], int flg_local);
```

Compute object 6D acceleration in object-centered frame, world/local orientation.

mj_contactForce

```
void mj_contactForce(const mjModel* m, const mjData* d, int id, mjtNum result[6]);
```

Extract 6D force:torque given contact id, in the contact frame.

mj_differentiatePos

```
void mj_differentiatePos(const mjModel* m, mjtNum* qvel, mjtNum dt,
                         const mjtNum* qpos1, const mjtNum* qpos2);
```

This function subtracts two vectors in the format of qpos (and divides the result by dt), while respecting the properties of quaternions. Recall that unit quaternions represent spatial orientations. They are points on the unit sphere in 4D. The tangent to that sphere is a 3D plane of rotational velocities. Thus when we subtract two quaternions in the right way, the result is a 3D vector and not a 4D vector. This the output qvel has dimensionality nv while the inputs have dimensionality nq.

mj_integratePos

```
void mj_integratePos(const mjModel* m, mjtNum* qpos, const mjtNum* qvel, mjtNum dt);
```

This is the opposite of mj_differentiatePos. It adds a vector in the format of qvel (scaled by dt) to a vector in the format of qpos.

mj_normalizeQuat

```
void mj_normalizeQuat(const mjModel* m, mjtNum* qpos);
```

Normalize all quaternions in qpos-type vector.

mj_local2Global

```
void mj_local2Global(mjData* d, mjtNum xpos[3], mjtNum xmat[9], const mjtNum pos[3],
                     const mjtNum quat[4], int body, mjtByte sameframe);
```

Map from body local to global Cartesian coordinates.

mj_getTotalmass

```
mjtNum mj_getTotalmass(const mjModel* m);
```

Sum all body masses.

mj_setTotalmass

```
void mj_setTotalmass(mjModel* m, mjtNum newmass);
```

Scale body masses and inertias to achieve specified total mass.

mj_version

```
int mj_version(void);
```

Return version number: 1.0.2 is encoded as 102.

mj_versionString

```
const char* mj_versionString();
```

Return the current version of MuJoCo as a null-terminated string.

Ray collisions

Ray collision functionality was added in MuJoCo 1.50. This is a new collision detection module that uses analytical formulas to intersect a ray (p + x*v, x>=0) with a geom, where p is the origin of the ray and v is the vector specifying the direction. All functions in this family return the distance to the nearest geom surface, or -1 if there is no intersection. Note that if p is inside a geom, the ray will intersect the surface from the inside which still counts as an intersection.

All ray collision functions rely on quantities computed by mj_kinematics (see mjData), so must be called after mj_kinematics, or functions that call it (e.g. mj_fwdPosition).

mj_ray

```
mjtNum mj_ray(const mjModel* m, const mjData* d, const mjtNum pnt[3], const mjtNum vec[3],
              const mjtByte* geomgroup, mjtByte flg_static, int bodyexclude,
```

```
              int geomid[1]);
```

Intersect ray (pnt+x*vec, x>=0) with visible geoms, except geoms in bodyexclude. Return geomid and distance (x) to nearest surface, or -1 if no intersection.

geomgroup is an array of length mjNGROUP, where 1 means the group should be included. Pass geomgroup=NULL to skip group exclusion. If flg_static is 0, static geoms will be excluded. bodyexclude=-1 can be used to indicate that all bodies are included.

mj_rayHfield

```
mjtNum mj_rayHfield(const mjModel* m, const mjData* d, int geomid,
                    const mjtNum pnt[3], const mjtNum vec[3]);
```

Interect ray with hfield, return nearest distance or -1 if no intersection.

mj_rayMesh

```
mjtNum mj_rayMesh(const mjModel* m, const mjData* d, int geomid,
                  const mjtNum pnt[3], const mjtNum vec[3]);
```

Interect ray with mesh, return nearest distance or -1 if no intersection.

mju_rayGeom

```
mjtNum mju_rayGeom(const mjtNum pos[3], const mjtNum mat[9], const mjtNum size[3],
                   const mjtNum pnt[3], const mjtNum vec[3], int geomtype);
```

Interect ray with pure geom, return nearest distance or -1 if no intersection.

mju_raySkin

```
mjtNum mju_raySkin(int nface, int nvert, const int* face, const float* vert,
                   const mjtNum pnt[3], const mjtNum vec[3], int vertid[1]);
```

Interect ray with skin, return nearest vertex id.

Interaction

These function implement abstract mouse interactions, allowing control over cameras and perturbations. Their use is well illustrated in simulate.cc.

mjv_defaultCamera

```
void mjv_defaultCamera(mjvCamera* cam);
```

Set default camera.

mjv_defaultFreeCamera

```
void mjv_defaultFreeCamera(const mjModel* m, mjvCamera* cam);
```

Set default free camera.

mjv_defaultPerturb

```
void mjv_defaultPerturb(mjvPerturb* pert);
```

Set default perturbation.

mjv_room2model

```
void mjv_room2model(mjtNum modelpos[3], mjtNum modelquat[4], const mjtNum roompos[3],
                    const mjtNum roomquat[4], const mjvScene* scn);
```

Transform pose from room to model space.

mjv_model2room

```
void mjv_model2room(mjtNum roompos[3], mjtNum roomquat[4], const mjtNum modelpos[3],
                    const mjtNum modelquat[4], const mjvScene* scn);
```

Transform pose from model to room space.

mjv_cameraInModel

```
void mjv_cameraInModel(mjtNum headpos[3], mjtNum forward[3], mjtNum up[3],
                       const mjvScene* scn);
```

Get camera info in model space; average left and right OpenGL cameras.

mjv_cameraInRoom

```
void mjv_cameraInRoom(mjtNum headpos[3], mjtNum forward[3], mjtNum up[3],
                      const mjvScene* scn);
```

Get camera info in room space; average left and right OpenGL cameras.

mjv_frustumHeight

```
mjtNum mjv_frustumHeight(const mjvScene* scn);
```

Get frustum height at unit distance from camera; average left and right OpenGL cameras.

mjv_alignToCamera

```
void mjv_alignToCamera(mjtNum res[3], const mjtNum vec[3], const mjtNum forward[3]);
```

Rotate 3D vec in horizontal plane by angle between (0,1) and (forward_x,forward_y).

mjv_moveCamera

```
void mjv_moveCamera(const mjModel* m, int action, mjtNum reldx, mjtNum reldy,
                    const mjvScene* scn, mjvCamera* cam);
```

Move camera with mouse; action is mjtMouse.

mjv_movePerturb

```
void mjv_movePerturb(const mjModel* m, const mjData* d, int action, mjtNum reldx,
                     mjtNum reldy, const mjvScene* scn, mjvPerturb* pert);
```

Move perturb object with mouse; action is mjtMouse.

mjv_moveModel

```
void mjv_moveModel(const mjModel* m, int action, mjtNum reldx, mjtNum reldy,
                   const mjtNum roomup[3], mjvScene* scn);
```

Move model with mouse; action is mjtMouse.

mjv_initPerturb

```
void mjv_initPerturb(const mjModel* m, const mjData* d,
                     const mjvScene* scn, mjvPerturb* pert);
```

Copy perturb pos,quat from selected body; set scale for perturbation.

mjv_applyPerturbPose

```
void mjv_applyPerturbPose(const mjModel* m, mjData* d, const mjvPerturb* pert,
                          int flg_paused);
```

Set perturb pos,quat in d->mocap when selected body is mocap, and in d->qpos otherwise. Write d->qpos only if flg_paused and subtree root for selected body has free joint.

mjv_applyPerturbForce

```
void mjv_applyPerturbForce(const mjModel* m, mjData* d, const mjvPerturb* pert);
```

Set perturb force,torque in d->xfrc_applied, if selected body is dynamic.

mjv_averageCamera

```
mjvGLCamera mjv_averageCamera(const mjvGLCamera* cam1, const mjvGLCamera* cam2);
```

Return the average of two OpenGL cameras.

mjv_select

```
int mjv_select(const mjModel* m, const mjData* d, const mjvOption* vopt,
               mjtNum aspectratio, mjtNum relx, mjtNum rely,
               const mjvScene* scn, mjtNum selpnt[3], int geomid[1], int skinid[1]);
```

This function is used for mouse selection. Previously selection was done via OpenGL, but as of MuJoCo 1.50 it relies on ray intersections which are much more efficient. aspectratio is the viewport width/height. relx and rely are the relative coordinates of the 2D point of interest in the viewport (usually mouse cursor). The function returns the id of the geom under the specified 2D point, or -1 if there is no geom (note that they skybox if present is not a model geom). The 3D coordinates of the clicked point are returned in selpnt. See simulate.cc for an illustration.

Visualization

The functions in this section implement abstract visualization. The results are used by the OpenGL rendered, and can also be used by users wishing to implement their own rendered, or hook up MuJoCo to advanced rendering tools such as Unity or Unreal Engine. See simulate.cc for illustration of how to use these functions.

mjv_defaultOption

```
void mjv_defaultOption(mjvOption* opt);
```

Set default visualization options.

mjv_defaultFigure

```
void mjv_defaultFigure(mjvFigure* fig);
```

Set default figure.

mjv_initGeom

```
void mjv_initGeom(mjvGeom* geom, int type, const mjtNum size[3],
                  const mjtNum pos[3], const mjtNum mat[9], const float rgba[4]);
```

Initialize given geom fields when not NULL, set the rest to their default values.

mjv_makeConnector

```
void mjv_makeConnector(mjvGeom* geom, int type, mjtNum width,
                       mjtNum a0, mjtNum a1, mjtNum a2,
                       mjtNum b0, mjtNum b1, mjtNum b2);
```

Set (type, size, pos, mat) for connector-type geom between given points. Assume that mjv_initGeom was already called to set all other properties.

mjv_defaultScene

```
void mjv_defaultScene(mjvScene* scn);
```

Set default abstract scene.

mjv_makeScene

```
void mjv_makeScene(const mjModel* m, mjvScene* scn, int maxgeom);
```

Allocate resources in abstract scene.

mjv_freeScene

```
void mjv_freeScene(mjvScene* scn);
```

Free abstract scene.

mjv_updateScene

```
void mjv_updateScene(const mjModel* m, mjData* d, const mjvOption* opt,
                     const mjvPerturb* pert, mjvCamera* cam, int catmask, mjvScene* scn);
```

Update entire scene given model state.

mjv_addGeoms

```
void mjv_addGeoms(const mjModel* m, mjData* d, const mjvOption* opt,
                  const mjvPerturb* pert, int catmask, mjvScene* scn);
```

Add geoms from selected categories.

mjv_makeLights

```
void mjv_makeLights(const mjModel* m, mjData* d, mjvScene* scn);
```

Make list of lights.

mjv_updateCamera

```
void mjv_updateCamera(const mjModel* m, mjData* d, mjvCamera* cam, mjvScene* scn);
```

Update camera.

mjv_updateSkin

```
void mjv_updateSkin(const mjModel* m, mjData* d, mjvScene* scn);
```

Update skins.

OpenGL rendering

These functions expose the OpenGL renderer. See simulate.cc for an illustration of how to use these functions.

mjr_defaultContext

```
void mjr_defaultContext(mjrContext* con);
```

Set default mjrContext.

mjr_makeContext

```
void mjr_makeContext(const mjModel* m, mjrContext* con, int fontscale);
```

Allocate resources in custom OpenGL context; fontscale is mjtFontScale.

mjr_changeFont

```
void mjr_changeFont(int fontscale, mjrContext* con);
```

Change font of existing context.

mjr_addAux

```
void mjr_addAux(int index, int width, int height, int samples, mjrContext* con);
```

Add Aux buffer with given index to context; free previous Aux buffer.

mjr_freeContext

```
void mjr_freeContext(mjrContext* con);
```

Free resources in custom OpenGL context, set to default.

mjr_uploadTexture

```
void mjr_uploadTexture(const mjModel* m, const mjrContext* con, int texid);
```

Upload texture to GPU, overwriting previous upload if any.

mjr_uploadMesh

```
void mjr_uploadMesh(const mjModel* m, const mjrContext* con, int meshid);
```

Upload mesh to GPU, overwriting previous upload if any.

mjr_uploadHField

```
void mjr_uploadHField(const mjModel* m, const mjrContext* con, int hfieldid);
```

Upload height field to GPU, overwriting previous upload if any.

mjr_restoreBuffer

```
void mjr_restoreBuffer(const mjrContext* con);
```

Make con->currentBuffer current again.

mjr_setBuffer

```
void mjr_setBuffer(int framebuffer, mjrContext* con);
```

Set OpenGL framebuffer for rendering: mjFB_WINDOW or mjFB_OFFSCREEN. If only one buffer is available, set that buffer and ignore framebuffer argument.

mjr_readPixels

```
void mjr_readPixels(unsigned char* rgb, float* depth,
                    mjrRect viewport, const mjrContext* con);
```

Read pixels from current OpenGL framebuffer to client buffer. Viewport is in OpenGL framebuffer; client buffer starts at (0,0).

mjr_drawPixels

```
void mjr_drawPixels(const unsigned char* rgb, const float* depth,
                    mjrRect viewport, const mjrContext* con);
```

Draw pixels from client buffer to current OpenGL framebuffer. Viewport is in OpenGL framebuffer; client buffer starts at (0,0).

mjr_blitBuffer

```
void mjr_blitBuffer(mjrRect src, mjrRect dst,
                    int flg_color, int flg_depth, const mjrContext* con);
```

Blit from src viewpoint in current framebuffer to dst viewport in other framebuffer. If src, dst have different size and flg_depth==0, color is interpolated with GL_LINEAR.

mjr_setAux

```
void mjr_setAux(int index, const mjrContext* con);
```

Set Aux buffer for custom OpenGL rendering (call restoreBuffer when done).

mjr_blitAux

```
void mjr_blitAux(int index, mjrRect src, int left, int bottom, const mjrContext* con);
```

Blit from Aux buffer to con->currentBuffer.

mjr_text

```
void mjr_text(int font, const char* txt, const mjrContext* con,
              float x, float y, float r, float g, float b);
```

Draw text at (x,y) in relative coordinates; font is mjtFont.

mjr_overlay

```
void mjr_overlay(int font, int gridpos, mjrRect viewport,
                 const char* overlay, const char* overlay2, const mjrContext* con);
```

Draw text overlay; font is mjtFont; gridpos is mjtGridPos.

mjr_maxViewport

```
mjrRect mjr_maxViewport(const mjrContext* con);
```

Get maximum viewport for active buffer.

mjr_rectangle

```
void mjr_rectangle(mjrRect viewport, float r, float g, float b, float a);
```

Draw rectangle.

mjr_label

```
void mjr_label(mjrRect viewport, int font, const char* txt,
               float r, float g, float b, float a, float rt, float gt, float bt,
               const mjrContext* con);
```

Draw rectangle with centered text.

mjr_figure

```
void mjr_figure(mjrRect viewport, mjvFigure* fig, const mjrContext* con);
```

Draw 2D figure.

mjr_render

```
void mjr_render(mjrRect viewport, mjvScene* scn, const mjrContext* con);
```

Render 3D scene.

mjr_finish

```
void mjr_finish(void);
```

Call glFinish.

mjr_getError

```
int mjr_getError(void);
```

Call glGetError and return result.

mjr_findRect

```
int mjr_findRect(int x, int y, int nrect, const mjrRect* rect);
```

Find first rectangle containing mouse, -1: not found.

UI framework

mjui_themeSpacing

```
mjuiThemeSpacing mjui_themeSpacing(int ind);
```

Get builtin UI theme spacing (ind: 0-1).

mjui_themeColor

```
mjuiThemeColor mjui_themeColor(int ind);
```

Get builtin UI theme color (ind: 0-3).

mjui_add

```
void mjui_add(mjUI* ui, const mjuiDef* def);
```

Add definitions to UI.

mjui_addToSection

```
void mjui_addToSection(mjUI* ui, int sect, const mjuiDef* def);
```

Add definitions to UI section.

mjui_resize

```
void mjui_resize(mjUI* ui, const mjrContext* con);
```

Compute UI sizes.

mjui_update

```
void mjui_update(int section, int item, const mjUI* ui,
                 const mjuiState* state, const mjrContext* con);
```

Update specific section/item; -1: update all.

mjui_event

```
mjuiItem* mjui_event(mjUI* ui, mjuiState* state, const mjrContext* con);
```

Handle UI event, return pointer to changed item, NULL if no change.

mjui_render

```
void mjui_render(mjUI* ui, const mjuiState* state, const mjrContext* con);
```

Copy UI image to current buffer.

Error and memory

mju_error

```
void mju_error(const char* msg);
```

Main error function; does not return to caller.

mju_error_i

```
void mju_error_i(const char* msg, int i);
```

Error function with int argument; msg is a printf format string.

mju_error_s

```
void mju_error_s(const char* msg, const char* text);
```

Error function with string argument.

mju_warning

```
void mju_warning(const char* msg);
```

Main warning function; returns to caller.

mju_warning_i

```
void mju_warning_i(const char* msg, int i);
```

Warning function with int argument.

mju_warning_s

```
void mju_warning_s(const char* msg, const char* text);
```

Warning function with string argument.

mju_clearHandlers

```
void mju_clearHandlers(void);
```

Clear user error and memory handlers.

mju_malloc

```
void* mju_malloc(size_t size);
```

Allocate memory; byte-align on 64; pad size to multiple of 64.

mju_free

```
void mju_free(void* ptr);
```

Free memory, using free() by default.

mj_warning

```
void mj_warning(mjData* d, int warning, int info);
```

High-level warning function: count warnings in mjData, print only the first.

mju_writeLog

```
void mju_writeLog(const char* type, const char* msg);
```

Write [datetime, type: message] to MUJOCO_LOG.TXT.

Standard math

The “functions” in this section are preprocessor macros replaced with the corresponding C standard library math functions. When MuJoCo is compiled with single precision (which is not currently available to the public, but we sometimes use it internally) these macros are replaced with the corresponding single-precision functions (not shown here). So one can think of them as having inputs and outputs of type mjtNum, where mjtNum is defined as double or float depending on how MuJoCo is compiled. We will not document these functions here; see the C standard library specification.

mju_sqrt

```
#define mju_sqrt    sqrt
```

mju_exp

```
#define mju_exp     exp
```

mju_sin

```
#define mju_sin     sin
```

mju_cos

```
#define mju_cos     cos
```

mju_tan

```
#define mju_tan     tan
```

mju_asin

```
#define mju_asin    asin
```

mju_acos

```
#define mju_acos    acos
```

mju_atan2

```
#define mju_atan2   atan2
```

mju_tanh

```
#define mju_tanh    tanh
```

mju_pow

```
#define mju_pow     pow
```

mju_abs

```
#define mju_abs     fabs
```

mju_log

```
#define mju_log     log
```

mju_log10

```
#define mju_log10   log10
```

mju_floor

```
#define mju_floor   floor
```

mju_ceil

```
#define mju_ceil    ceil
```

Vector math

mju_zero3

```
void mju_zero3(mjtNum res[3]);
```

Set res = 0.

mju_copy3

```
void mju_copy3(mjtNum res[3], const mjtNum data[3]);
```

Set res = vec.

mju_scl3

```
void mju_scl3(mjtNum res[3], const mjtNum vec[3], mjtNum scl);
```

Set res = vec*scl.

mju_add3

```
void mju_add3(mjtNum res[3], const mjtNum vec1[3], const mjtNum vec2[3]);
```

Set res = vec1 + vec2.

mju_sub3

```
void mju_sub3(mjtNum res[3], const mjtNum vec1[3], const mjtNum vec2[3]);
```

Set res = vec1 - vec2.

mju_addTo3

```
void mju_addTo3(mjtNum res[3], const mjtNum vec[3]);
```

Set res = res + vec.

mju_subFrom3

```
void mju_subFrom3(mjtNum res[3], const mjtNum vec[3]);
```

Set res = res - vec.

mju_addToScl3

```
void mju_addToScl3(mjtNum res[3], const mjtNum vec[3], mjtNum scl);
```

Set res = res + vec*scl.

mju_addScl3

```
void mju_addScl3(mjtNum res[3], const mjtNum vec1[3], const mjtNum vec2[3], mjtNum scl);
```

Set res = vec1 + vec2*scl.

mju_normalize3

```
mjtNum mju_normalize3(mjtNum res[3]);
```

Normalize vector, return length before normalization.

mju_norm3

```
mjtNum mju_norm3(const mjtNum vec[3]);
```

Return vector length (without normalizing the vector).

mju_dot3

```
mjtNum mju_dot3(const mjtNum vec1[3], const mjtNum vec2[3]);
```

Return dot-product of vec1 and vec2.

mju_dist3

```
mjtNum mju_dist3(const mjtNum pos1[3], const mjtNum pos2[3]);
```

Return Cartesian distance between 3D vectors pos1 and pos2.

mju_rotVecMat

```
void mju_rotVecMat(mjtNum res[3], const mjtNum vec[3], const mjtNum mat[9]);
```

Multiply vector by 3D rotation matrix: res = mat * vec.

mju_rotVecMatT

```
void mju_rotVecMatT(mjtNum res[3], const mjtNum vec[3], const mjtNum mat[9]);
```

Multiply vector by transposed 3D rotation matrix: res = mat' * vec.

mju_cross

```
void mju_cross(mjtNum res[3], const mjtNum a[3], const mjtNum b[3]);
```

Compute cross-product: res = cross(a, b).

mju_zero4

```
void mju_zero4(mjtNum res[4]);
```

Set res = 0.

mju_unit4

```
void mju_unit4(mjtNum res[4]);
```

Set res = (1,0,0,0).

mju_copy4

```
void mju_copy4(mjtNum res[4], const mjtNum data[4]);
```

Set res = vec.

mju_normalize4

```
mjtNum mju_normalize4(mjtNum res[4]);
```

Normalize vector, return length before normalization.

mju_zero

```
void mju_zero(mjtNum* res, int n);
```

Set res = 0.

mju_fill

```
void mju_fill(mjtNum* res, mjtNum val, int n);
```

Set res = val.

mju_copy

```
void mju_copy(mjtNum* res, const mjtNum* data, int n);
```

Set res = vec.

mju_sum

```
mjtNum mju_sum(const mjtNum* vec, int n);
```

Return sum(vec).

mju_L1

```
mjtNum mju_L1(const mjtNum* vec, int n);
```

Return L1 norm: sum(abs(vec)).

mju_scl

```
void mju_scl(mjtNum* res, const mjtNum* vec, mjtNum scl, int n);
```

Set res = vec*scl.

mju_add

```
void mju_add(mjtNum* res, const mjtNum* vec1, const mjtNum* vec2, int n);
```

Set res = vec1 + vec2.

mju_sub

```
void mju_sub(mjtNum* res, const mjtNum* vec1, const mjtNum* vec2, int n);
```

Set res = vec1 - vec2.

mju_addTo

```
void mju_addTo(mjtNum* res, const mjtNum* vec, int n);
```

Set res = res + vec.

mju_subFrom

```
void mju_subFrom(mjtNum* res, const mjtNum* vec, int n);
```

Set res = res − vec.

mju_addToScl

```
void mju_addToScl(mjtNum* res, const mjtNum* vec, mjtNum scl, int n);
```

Set res = res + vec*scl.

mju_addScl

```
void mju_addScl(mjtNum* res, const mjtNum* vec1, const mjtNum* vec2, mjtNum scl, int n);
```

Set res = vec1 + vec2*scl.

mju_normalize

```
mjtNum mju_normalize(mjtNum* res, int n);
```

Normalize vector, return length before normalization.

mju_norm

```
mjtNum mju_norm(const mjtNum* res, int n);
```

Return vector length (without normalizing vector).

mju_dot

```
mjtNum mju_dot(const mjtNum* vec1, const mjtNum* vec2, const int n);
```

Return dot-product of vec1 and vec2.

mju_mulMatVec

```
void mju_mulMatVec(mjtNum* res, const mjtNum* mat, const mjtNum* vec, int nr, int nc);
```

Multiply matrix and vector: res = mat * vec.

mju_mulMatTVec

```
void mju_mulMatTVec(mjtNum* res, const mjtNum* mat, const mjtNum* vec, int nr, int nc);
```

Multiply transposed matrix and vector: res = mat' * vec.

mju_mulVecMatVec

```
mjtNum mju_mulVecMatVec(const mjtNum* vec1, const mjtNum* mat, const mjtNum* vec2, int n);
```

Multiply square matrix with vectors on both sides: return vec1' * mat * vec2.

mju_transpose

```
void mju_transpose(mjtNum* res, const mjtNum* mat, int nr, int nc);
```

Transpose matrix: res = mat'.

mju_symmetrize

```
void mju_symmetrize(mjtNum* res, const mjtNum* mat, int n);
```

Symmetrize square matrix $R = \frac{1}{2}(M + M^T)$.

mju_eye

```
void mju_eye(mjtNum* mat, int n);
```

Set mat to the identity matrix.

mju_mulMatMat

```
void mju_mulMatMat(mjtNum* res, const mjtNum* mat1, const mjtNum* mat2,
                   int r1, int c1, int c2);
```

Multiply matrices: res = mat1 * mat2.

mju_mulMatMatT

```
void mju_mulMatMatT(mjtNum* res, const mjtNum* mat1, const mjtNum* mat2,
                    int r1, int c1, int r2);
```

Multiply matrices, second argument transposed: res = mat1 * mat2'.

mju_mulMatTMat

```
void mju_mulMatTMat(mjtNum* res, const mjtNum* mat1, const mjtNum* mat2,
                    int r1, int c1, int c2);
```

Multiply matrices, first argument transposed: res = mat1' * mat2.

mju_sqrMatTD

```
void mju_sqrMatTD(mjtNum* res, const mjtNum* mat, const mjtNum* diag, int nr, int nc);
```

Set res = mat' * diag * mat if diag is not NULL, and res = mat' * mat otherwise.

mju_transformSpatial

```
void mju_transformSpatial(mjtNum res[6], const mjtNum vec[6], int flg_force,
                          const mjtNum newpos[3], const mjtNum oldpos[3],
                          const mjtNum rotnew2old[9]);
```

Coordinate transform of 6D motion or force vector in rotation:translation format. rotnew2old is 3-by-3, NULL means no rotation; flg_force specifies force or motion type.

Quaternions

mju_rotVecQuat

```
void mju_rotVecQuat(mjtNum res[3], const mjtNum vec[3], const mjtNum quat[4]);
```

Rotate vector by quaternion.

mju_negQuat

```
void mju_negQuat(mjtNum res[4], const mjtNum quat[4]);
```

Negate quaternion.

mju_mulQuat

```
void mju_mulQuat(mjtNum res[4], const mjtNum quat1[4], const mjtNum quat2[4]);
```

Multiply quaternions.

mju_mulQuatAxis

```
void mju_mulQuatAxis(mjtNum res[4], const mjtNum quat[4], const mjtNum axis[3]);
```

Multiply quaternion and axis.

mju_axisAngle2Quat

```
void mju_axisAngle2Quat(mjtNum res[4], const mjtNum axis[3], mjtNum angle);
```

Convert axisAngle to quaternion.

mju_quat2Vel

```
void mju_quat2Vel(mjtNum res[3], const mjtNum quat[4], mjtNum dt);
```

Convert quaternion (corresponding to orientation difference) to 3D velocity.

mju_subQuat

```
void mju_subQuat(mjtNum res[3], const mjtNum qa[4], const mjtNum qb[4]);
```

Subtract quaternions, express as 3D velocity: qb*quat(res) = qa.

mju_quat2Mat

```
void mju_quat2Mat(mjtNum res[9], const mjtNum quat[4]);
```

Convert quaternion to 3D rotation matrix.

mju_mat2Quat

```
void mju_mat2Quat(mjtNum quat[4], const mjtNum mat[9]);
```

Convert 3D rotation matrix to quaternion.

mju_derivQuat

```
void mju_derivQuat(mjtNum res[4], const mjtNum quat[4], const mjtNum vel[3]);
```

Compute time-derivative of quaternion, given 3D rotational velocity.

mju_quatIntegrate

```
void mju_quatIntegrate(mjtNum quat[4], const mjtNum vel[3], mjtNum scale);
```

Integrate quaternion given 3D angular velocity.

mju_quatZ2Vec

```
void mju_quatZ2Vec(mjtNum quat[4], const mjtNum vec[3]);
```

Construct quaternion performing rotation from z-axis to given vector.

Poses

mju_mulPose

```
void mju_mulPose(mjtNum posres[3], mjtNum quatres[4],
                 const mjtNum pos1[3], const mjtNum quat1[4],
                 const mjtNum pos2[3], const mjtNum quat2[4]);
```

Multiply two poses.

mju_negPose

```
void mju_negPose(mjtNum posres[3], mjtNum quatres[4],
                 const mjtNum pos[3], const mjtNum quat[4]);
```

Negate pose.

mju_trnVecPose

```
void mju_trnVecPose(mjtNum res[3], const mjtNum pos[3], const mjtNum quat[4],
                    const mjtNum vec[3]);
```

Transform vector by pose.

Decompositions

mju_cholFactor

```
int mju_cholFactor(mjtNum* mat, int n, mjtNum mindiag);
```

Cholesky decomposition: mat = L*L'; return rank, decomposition performed in-place into mat.

mju_cholSolve

```
void mju_cholSolve(mjtNum* res, const mjtNum* mat, const mjtNum* vec, int n);
```

Solve mat * res = vec, where mat is Cholesky-factorized

mju_cholUpdate

```
int mju_cholUpdate(mjtNum* mat, mjtNum* x, int n, int flg_plus);
```

Cholesky rank-one update: L*L' +/- x*x'; return rank.

mju_eig3

```
int mju_eig3(mjtNum eigval[3], mjtNum eigvec[9], mjtNum quat[4], const mjtNum mat[9]);
```

Eigenvalue decomposition of symmetric 3x3 matrix.

mju_boxQP

```
int mju_boxQP(mjtNum* res, mjtNum* R, int* index, const mjtNum* H, const mjtNum* g, int n,
              const mjtNum* lower, const mjtNum* upper);
```

Minimize $\frac{1}{2}x^T H x + x^T g \quad \text{s.t.} \quad l \leq x \leq u$, return rank or -1 if failed.

inputs:

`n` - problem dimension

`H` - SPD matrix `n*n`

`g` - bias vector `n`

`lower` - lower bounds `n`

`upper` - upper bounds `n`

`res` - solution warmstart `n`

return value:

`nfree <= n` - rank of unconstrained subspace, -1 if failure

outputs (required):

`res` - solution `n`

`R` - subspace Cholesky factor `nfree*nfree`, allocated: `n*(n+7)`

outputs (optional):

`index` - set of free dimensions `nfree`, allocated: `n`

notes:

The initial value of `res` is used to warmstart the solver. `R` must have allocatd size `n*(n+7)`, but only `nfree*nfree` values are used in output. `index` (if given) must have allocated size `n`, but only `nfree` values are used in output. The convenience function mju_boxQPmalloc allocates the required data structures. Only the lower triangles of H and R and are read from and written to, respectively.

mju_boxQPmalloc

```
void mju_boxQPmalloc(mjtNum** res, mjtNum** R, int** index, mjtNum** H, mjtNum** g, int n,
                     mjtNum** lower, mjtNum** upper);
```

Allocate heap memory for box-constrained Quadratic Program. As in mju_boxQP, `index`, `lower`, and `upper` are optional. Free all pointers with `mju_free()`.

Miscellaneous

mju_muscleGain

```
mjtNum mju_muscleGain(mjtNum len, mjtNum vel, const mjtNum lengthrange[2],
                      mjtNum acc0, const mjtNum prm[9]);
```

Muscle active force, prm = (range[2], force, scale, lmin, lmax, vmax, fpmax, fvmax).

mju_muscleBias

```
mjtNum mju_muscleBias(mjtNum len, const mjtNum lengthrange[2],
                      mjtNum acc0, const mjtNum prm[9]);
```

Muscle passive force, prm = (range[2], force, scale, lmin, lmax, vmax, fpmax, fvmax).

mju_muscleDynamics

```
mjtNum mju_muscleDynamics(mjtNum ctrl, mjtNum act, const mjtNum prm[2]);
```

Muscle activation dynamics, prm = (tau_act, tau_deact).

mju_encodePyramid

```
void mju_encodePyramid(mjtNum* pyramid, const mjtNum* force, const mjtNum* mu, int dim);
```

Convert contact force to pyramid representation.

mju_decodePyramid

```
void mju_decodePyramid(mjtNum* force, const mjtNum* pyramid, const mjtNum* mu, int dim);
```

Convert pyramid representation to contact force.

mju_springDamper

```
mjtNum mju_springDamper(mjtNum pos0, mjtNum vel0, mjtNum Kp, mjtNum Kv, mjtNum dt);
```

Integrate spring-damper analytically, return pos(dt).

mju_min

```
mjtNum mju_min(mjtNum a, mjtNum b);
```

Return min(a,b) with single evaluation of a and b.

mju_max

```
mjtNum mju_max(mjtNum a, mjtNum b);
```

Return max(a,b) with single evaluation of a and b.

mju_sign

```
mjtNum mju_sign(mjtNum x);
```

Return sign of x: +1, -1 or 0.

mju_round

```
int mju_round(mjtNum x);
```

Round x to nearest integer.

mju_type2Str

```
const char* mju_type2Str(int type);
```

Convert type id (mjtObj) to type name.

mju_str2Type

```
int mju_str2Type(const char* str);
```

Convert type name to type id (mjtObj).

mju_writeNumBytes

```
const char* mju_writeNumBytes(const size_t nbytes);
```

Construct a human readable number of bytes using standard letter suffix.

mju_warningText

```
const char* mju_warningText(int warning, size_t info);
```

Construct a warning message given the warning type and info.

mju_isBad

```
int mju_isBad(mjtNum x);
```

Return 1 if nan or abs(x)>mjMAXVAL, 0 otherwise. Used by check functions.

mju_isZero

```
int mju_isZero(mjtNum* vec, int n);
```

Return 1 if all elements are 0.

mju_standardNormal

```
mjtNum mju_standardNormal(mjtNum* num2);
```

Standard normal random number generator (optional second number).

mju_f2n

```
void mju_f2n(mjtNum* res, const float* vec, int n);
```

Convert from float to mjtNum.

mju_n2f

```
void mju_n2f(float* res, const mjtNum* vec, int n);
```

Convert from mjtNum to float.

mju_d2n

```
void mju_d2n(mjtNum* res, const double* vec, int n);
```

Convert from double to mjtNum.

mju_n2d

```
void mju_n2d(double* res, const mjtNum* vec, int n);
```

Convert from mjtNum to double.

mju_insertionSort

```
void mju_insertionSort(mjtNum* list, int n);
```

Insertion sort, resulting list is in increasing order.

mju_insertionSortInt

```
void mju_insertionSortInt(int* list, int n);
```

Integer insertion sort, resulting list is in increasing order.

mju_Halton

```
mjtNum mju_Halton(int index, int base);
```

Generate Halton sequence.

mju_strncpy

```
char* mju_strncpy(char *dst, const char *src, int n);
```

Call strncpy, then set dst[n-1] = 0.

mju_sigmoid

```
mjtNum mju_sigmoid(mjtNum x);
```

Sigmoid function over 0<=x<=1 constructed from half-quadratics.

mjd_transitionFD

```
void mjd_transitionFD(const mjModel* m, mjData* d, mjtNum eps, mjtByte centered,
                      mjtNum* A, mjtNum* B, mjtNum* C, mjtNum* D);
```

Finite differenced state-transition and control-transition matrices dx(t+h) = A*dx(t) + B*du(t).
Required output matrix dimensions: A: (2*nv+na x 2*nv+na), B: (2*nv+na x nu).

Macros

mjMARKSTACK

```
#define mjMARKSTACK int _mark = d->pstack;
```

This macro is helpful when using the MuJoCo stack in custom computations. It works together with the next macro and the mj_stackAlloc function, and assumes that mjData* d is defined. The use pattern is this:

```
mjMARKSTACK
mjtNum* temp = mj_stackAlloc(d, 100);
// ... use temp as needed
mjFREESTACK
```

mjFREESTACK

```
#define mjFREESTACK d->pstack = _mark;
```

Reset the MuJoCo stack pointer to the variable _mark, normally saved by mjMARKSTACK.

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

```
#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.