MuJoCo:

Modeling and simulation of **Mu**lti-**Jo**int dynamics with **Co**ntact

Version 0.5.0

Emo Todorov, Yuval Tassa, Tom Erez

June 10, 2013

Preface

MuJoCo is a physics engine aiming to facilitate research and development in robotics, biomechanics, and other areas where fast and accurate simulation of complex dynamical systems is needed. It offers a unique combination of speed, accuracy and modeling power, yet it is not merely a better simulator. Instead it is the first full-featured simulator designed from the ground up for the purpose of model-based control.

Existing engines can of course be used for control applications, however they are not ideal, which is why many research groups have developed their own in-house simulation tools. This piecemeal approach replicates effort and impedes progress. Having spent considerable time ourselves on extending existing engines as well as developing project-specific simulators with limited scope, we realized that as difficult as it is to develop a full-features new engine from scratch, in the long run this will actually save time. It is of course easier to find the courage for such an endeavour if one already has tenure.

The primary development and maintenance has been done by Emo Todorov in his spare time away from faculty responsibilities. Tom Erez and Yuval Tassa have used the software extensively and have identified problems and their solutions, opportunities for refinement and speedup, and features that needed to be added. They have also developed state-of-the-art optimal control software utilizing MuJoCo, which will be documented separately. Thanks are also due to the NSF, NIH and DARPA. Although we did not think it would be fruitful to seek funding for developing a new physics engine, the regular research funding provided by these agencies has supported multiple projects that benefited from MuJoCo, in turn providing continued motivation for its development.

After using MuJoCo for over two years at the Movement Control Laboratory, University of Washington, we have found it to be a powerful tool enabling behavior synthesis and model-predictive control for complex robotic systems. It has become a cornerstone in our efforts to build more intelligent controllers for both simulated and physical systems. We hope that others will find MuJoCo equally useful.

Version 0.5.0 As the current version number suggests, we do not consider MuJoCo to be complete. Nevertheless the parts that are complete are usable and well tuned. We look forward to feedback from the research community to help us make it better. In terms of documentation, more will be written soon. The present document provides a brief overview of the software, followed by a description of the mathematical and algorithmic foundations. In addition, the software release contains well-documented header files exposing the C API to the library, as well as built-in help in the MATLAB toolbox and standalone executable. A first attempt at producing Doxygen documentation is also made.

1 Modeling and software overview

MuJoCo models can exist on three levels of description:

XML file in an intuitive file format This is the recommended way of designing models. The parser can also load URDF models (although they lack many of the features that MuJoCo supports) and convert them. The parser can be evoked via a command-line tool (convert), or via the standalone executable (wx) discussed later. We are also working on converters for SDF and OpenSim, but they need more testing.

C++ API for high-level model construction This API is not presently documented and exposed, but it will be. This is what the XML parser uses internally to construct models. Once the API is available, users will be able to construct or modify models programmatically instead of writing XML files.

C structure (mjModel) used in all runtime computations Once a high-level model is constructed in runtime (as a tree of C++ objects), it can be compiled into a low-level model. The entire low-level model is contained in a single C structure called mjModel. It has the organization of structure-of-arrays for computational efficiency. mjModel can also be loaded and saved into binary files, as well as serialized in memory.

The software itself can currently be used in three ways:

Static link library for C/C++ programs The low-level engine (which we call the MuJoCo library) is written in ANSI C and is highly optimized for realtime computations. To gain full advantage of MuJoCo – both in terms of speed and ability to design custom computations – one should use it in the library form. The precompiled library is available for Windows 32 and 64 bit, MacOS, and Ubunu 64 bit (although it probably works on any Linux installation). The library only operates on mjModel, and not on any of the higher-level model descriptions. Apart from mjModel, it uses another large C structure called mjData – which contains the state information and intermediate results of the computations. mjData serves as a scratchpad where functions read their inputs and write their outputs.

MATLAB mex toolbox The top-level functionality of the MuJoCo library (i.e. the stepper function and the main algorithms) is also exposed in MATLAB, using a thin layer of code that translates function calls and data back and forth. This is the best way to learn MuJoCo and prototype algorithms built on top of it. Indeed most of our research until recently was done with the MATLAB toolbox. The toolbox also includes 3D rendering using native MATLAB graphics, and as such is platform-independent in principle. In practice MATLAB 3D graphics do not work equally well on all platforms. Thus for now we only have the Windows version (32 bit and 64 bit), as well as an experimental MacOS version.

Standalone executable (MuJoCo Studio) The executable (wx) is meant to evolve into a "studio", incorporating MuJoCo as well as tools that automate various computations. For now it is mostly a glorified viewer/explorer, nevertheless it is extremely useful in constructing models and exploring their dynamics (before starting to do serious work with them). MuJoCo Studio has fast custom OpenGL graphics that enable interactive visualization with many visual aids. It also allows the user to "reach in" the simulation (via the mouse or even better a 3D Connexion Space Navigator), apply perturbations to selected bodies and see how they affect the simulation. The GUI also exposes all the dynamics options from mjModel, making it easy to adjust numerical parameters and instantly see their effect.

The executable is built using the wxWidgets platform-independent framework. Presently only the Windows version is available. We have compiled and run it on Ubuntu (and should be able to do so on MacOS as well), however it needs finetuning before it can be released for any OS other than Windows. When working with models that have complex meshes, a good GPU is recommended. Alternatively, one can automatically fit the meshes with simple geometric primitives. This is also how the MATLAB toolbox should be used – because MATLAB does provide the GPU access needed to render complex meshes interactively.

2 Mathematical and algorithmic foundations

Here we explain the mathematical and algorithmic approach behind MuJoCo. Keep in mind that the engine can be used without knowledge of the material in this chapter – by simply calling `mj_step(model,data)` repeatedly to advance the simulation.

2.1 Summary of computational methods

- Multi-joint dynamics are expressed in generalized or joint coordinates. This allows the use efficient recursive algorithms, while avoiding the need to impose joint constraints numerically. Equality constraints other than regular joints (e.g. loop joints) are imposed numerically.

- Impulsive dynamics resulting from inequality constraints (contacts, joint limits and dry friction) are handled with a velocity-stepping scheme. In particular, the impulse is computed by a dedicated solver as the solution to an optimization problem, which then determines the next-step velocity directly, without having to integrate (potentially infinite) interaction forces over time.

- Stabilization of both equality and inequality constraints is handled carefully, so as to avoid injecting energy to the extent possible. We model the temporal evolution of constraint violations via critical damping (with user-specified time constant) and set the desired error correction velocities accordingly. These desired velocities become recommendations to the impulse solver which computes the actual next-step velocity.

- Actuation is modeled in a very flexible way, allowing torque motors, pneumatic and hydraulic cylinders, biological muscles and user-defined actuators with custom dynamics. Actuators can have their own state variables – in which case the overall system dynamics become 3rd order.

- 3D geometry is modeled as rigid "geoms" which can be simple geometric primitives, convex meshes, or heightfields. A unique aspect of MuJoCo is the support for tendons, i.e. strings that follow a minimal path in space subject to routing and wrapping constraints. Tendons can be used to model muscles, cylinders mounted directly, strings in a marionette, tensegrity structures. Collision detection is based on methods very similar to gaming engines. Tendon routing is based on analytical formulas.

- MuJoCo is designed not only for forward simulation, but also for inverse dynamics computations which are useful for data analysis and control synthesis. Constraints generally make the dynamics non-invertible. To this end we have introduced approximations that are optional, but when enabled they make the entire computational pipeline analytically invertible. They also smooth out the forward dynamics, facilitating numerical differentiation. Inverse dynamics are not yet documented and exposed, yet this has been a major design goal and so it affects other design choices we have made throughout the engine.

2.2 Notation and relevant quantities

Bellow we summarize our notation for the key quantities, and the corresponding field names in the main data structure (mjData) holding the model state information. They are separated into inputs, intermediate results, and outputs. The actual computations are described later.

Inputs

mjData field	Symbol	Meaning
`qpos`	q	positions in generalized coordinates
`qvel`	v	velocities in generalized coordinates
`act`	a	activations of the actuators (for 3rd order dynamics)
`ctrl`	u	user-supplied control signals (force or activation-related)
`qfrc_applied`		generalized forces applied directly (bypassing actuators)
`pert_xfrc`		Cartesian forces applied to selected bodies
`dt`	h	discrete time step

Intermediate results

mjData field	Symbol	Meaning
`qM,qLD`	$M(q)$	inertia matrix and its LDL factorization
`qfrc_bias`	$c(q,v)$	Coriolis, centripetal and gravitational forces
	$\tau(q,v,a,u)$	net smooth forces: actuator, passive and applied forces
`eq_J`	$J_E(q)$	Jacobian of equality constraints
	$v_E = J_E v$	velocity in equality-constraint coordinates
`eq_f`	$f_E(q,v,\tau)$	impulse caused by equality constraints
`eq_Achol`	$A_E(q)$	inverse inertia in equality-constraint coordinates; saved in Cholesky factorized form
`flc_J...`	$J_I(q)$	Jacobian of inequality constraints; multiple mjData fields used to encode sparse matrix
	$v_I = J_I v$	velocity in inequality-constraint coordinates
`flc_f`	$f_I(q,v,\tau)$	impulse caused by inequality constraints
`flc_A`	$A_I(q)$	inverse inertia in inequality-constraint coordinates

"flc" stands for "friction, limit, contact". Some of the functional dependencies indicated above are standard. Others are not. In particular, the quantities f_E, f_I are computed by specialized solvers which take into account all smooth forces acting on the system. In this way the impulse solvers have the last word – which helps minimize constraint violations.

Outputs The main outputs of the forward dynamics are the next-step velocity v' saved in `qvel_next`, and the activation time-derivative $\dot{a}$ saved in `actdot`. Numerical integration for q, a is then applied, so that the state variables $q = q(t), v = v(t), a = a(t)$ assume their new values, denoted $q' = q(t+h), v' = v(t+h), a' = a(t+h)$. Since the entire computation is applied at a single point in time, we omit the time index t in the rest of the text.

2.3 Equations of motion and integration

The equations of motion in continuous time are

$$M\,dv = (c + \tau)\,dt + J_E^{\mathsf{T}} f_E + J_I^{\mathsf{T}} f_I$$

Note that we have not divided by dt because we are dealing with impulses. The above equation simply means that the integrals of the left and the right side are equal. Focusing on a single point in time, the equations of motion in discrete time can be written as

$$\begin{aligned} M\,(v' - v) &= (c + \tau)\,h + J_E^{\mathsf{T}} f_E + J_I^{\mathsf{T}} f_I \\ v' &= v + M^{-1}\left((c + \tau)\,h + J_E^{\mathsf{T}} f_E + J_I^{\mathsf{T}} f_I\right) \end{aligned}$$

In a nutshell, the engine computes all the quantities on the right side (as explained next), evaluates v' from the above formula, and advances the simulation via numerical integration.

Numerical integration Assuming the next-step velocities v' can be computed for any h, the goal of integration is to compute the next-step positions q'. MuJoCo provides five integrators. The first three are variations on the Euler method, given below along with the numeric constants used to select them (see `mjModel.option.integrator`):

`mjINT_EXPLICIT`	$q' = q + hv$
`mjINT_MIDPOINT`	$q' = q + 0.5h\,(v + v')$
`mjINT_SEMIIMPLICIT`	$q' = q + hv'$

"+" involves quaternion operations when necessary, while "implicit" refers to using quantities at the next time step to perform an update. The last integrator is semi-implicit because it is implicit in position, but the velocity update (performed by the computational pipeline described next) is still explicit. The semi-implicit integrator is the recommended choice of the Euler methods.

MuJoCo also implements two flavors of the 4th order Runge Kutta method (RK4). Note that RK4 is normally presented using explicit time derivatives, while here we use a velocity-stepping scheme. Nevertheless, we can define acceleration numerically as $\dot{v} = h^{-1}\,(v' - v)$ and apply the standard RK4 method. Of course this requires running the entire computational pipeline 4 times per time step. When integrating smooth systems this more than pays off because it enables much larger time steps without loosing accuracy, but here we are dealing with impulses so the advantages of RK4 are less clear and remain to be investigated. In the meantime, we provide one more method based on the following reasoning. RK4 should obviously be applied to the smooth dynamics, but there may be little justification in running the collision detector and impulse solver 4 times per time step. Thus we introduce the "interleaved" integrator, which applies RK4 to the smooth dynamics, and only applies impulses at the end of each time step. Summarizing, we have two more integrators:

`mjINT_RK4`	4th order Runge Kutta on the entire system
`mjINT_INTERLEAVE`	RK4 smooth dynamics, semi-implicit impulse dynamics

2.4 Computational pipeline

Computing the next-step velocity v' involves a long sequence of operations. We provide informal summaries here. The more elaborate operations are presented in more detail later.

Kinematics Given the joint configuration q, the Cartesian positions and orientations of all bodies (as well as other model elements rigidly attached to the bodies) are computed in a forward pass through the kinematic tree. Orientations are accumulated using quaternion multiplication. Once the body configurations are known, tendon and actuator lengths and moment arms are also computed with specialized formulas.

Inertia The inertia matrix $M(q)$ is computed using the Composite Rigid Body algorithm (CRB), and is factorized using LDL factorization. Since M has a sparsity pattern corresponding to a tree, sparse LDL factorization can be performed without any fill-in. We use a custom sparse matrix representation of M adapted to the underlying tree structure, as described later. Note that we are not using the Featherstone $O(n)$ forward dynamics approach. This is because the inertia matrix is needed for any modern impulse solver. On the other hand, as Featherstone has pointed out, the advantages of his $O(n)$ are approach are unclear when applied to a kinematic tree such a humanoid – because branching introduces so much sparsity that the two approaches become similar.

Collision detection All geom pairs that are enabled for collision are checked. If the distance is below a user-defined safety margin (called `mindist`), one or more contacts are registered depending on the geom types. For each contact we compute the point in the middle between the two contacting (or penetrating) surfaces, the penetration distance, and the contact frame with the X axis oriented along the normal. Geom pairs can be enabled for collision using geom-specific collision masks (same mechanism as in the Open Dynamics Engine), or by pre-compiling the list of all allowed geom pairs (which is the most efficient mechanism). There is a placeholder for a broadphase based on hierarchical space decomposition, but this functionality is not yet implemented. Ideal geometric shapes such as spheres, capsules, planes, boxes, cylinders and ellipsoids are collided by specialized routines when possible. There is also a general routine based on the Minkowski Portal Refinement (MPR) algorithm as implemented in the LIBCCD library, which is used for convex meshes as well as all geom type pairs that do not have a custom routine. MPR is related to the older GJK algorithm. Non-convex meshes (or "triangle soups") can be included in the model and rendered properly, however for collision purposes they are automatically replaced with their convex hull. While the engine allows extensions to the collision detection mechanism via user callbacks, we recommend decomposing non-convex meshes into unions of convex meshes (with the HACD library for example). The user can define frictionless contacts, frictional contacts, as well as contacts that have torsional and rolling friction in addition to sliding friction. These friction types can be mixed in the same model. The friction parameters are not used in the collision detection phase but rather in the impulse phase described later. Another important contact parameter is softness, see below.

Constraint construction The Jacobians $J_E(q)$ and $J_I(q)$ of the active equality and inequality constraints are computed. Inequality constraints include contacts, joint and tendon limits, and (dry) joint friction. Since the number of inequality constraints can be large, J_I is optionally represented as a sparse matrix (using a convention similar to MATLAB). The inverse inertia matrices $A_E = J_E M^{-1} J_E^T$ and $A_I = J_I M_E^{-1} J_I^T$ are computed. In the absence of equality constraints we have $M_E = M$. When equality constraints are present, M_E is modified so as to increase the (apparent) inertia in the subspace corresponding to the equality constraints, as explained later. These matrix computations have been highly optimized because in some cases the computation of A_I can become the bottleneck of the entire pipeline. Overall, in medium-sized problems typical for robotics applications, we have found our math utility functions to be faster than Intel MKL – most likely because we avoid the overhead that is only needed for larger problems. These functions are exposed in the library API and can be used without the rest of MuJoCo.

The three steps above take most of the CPU time in the forward dynamics. Note however that they only depend on the joint positions q. This can be exploited when computing derivatives via finite differencing – which requires computing the forward dynamics for many values of (q, v, a, u) arranged on a grid. In particular, the grid can be traversed in such a way that the position q remains the same for several forward dynamics computations; in this case the results from the position-dependent steps can be reused. We rely on this approach routinely in our optimal control work, and obtain substantial speedups.

Control, actuation, perturbations, passive forces At this point MuJoCo needs the controls u, which the user code can compute by using the state information (q, v, a) as well as all of the above quantities that are already computed (end-effector positions and Jacobians in particular can be helpful in constructing control laws). The actuator forces are then computed according to the actuator type. The user can also specify external forces that are not generated by the modeled actuators, but for example by an external push on the system. MuJoCo can also model passive force fields such as joint and tendon springs and dampers, as well as body-level viscosity. All these applied forces are summed into the vector τ.

Coriolis, centripetal, gravitational forces The forces $c(q, v)$ are the sum of Coriolis, centripetal and gravitational forces. They are computed simultaneously using the Recursive Newton Euler algorithm (RNE). Recall that RNE is a method for inverse dynamics, computing the applied force $\widetilde{\tau}$ needed to achieve prescribed acceleration $\dot{v}$ in state (q, v):

$$M(q)\,\dot{v} - c(q, v) = \widetilde{\tau}$$

Our notation is a bit unusual: we flip the sign of c and include gravity in it. RNE is used in forward dynamics by setting $\dot{v} = 0$, executing the algorithm to compute $\widetilde{\tau}$ in the above equation (not to be confused with the actual τ we already computed), and setting $c = -\widetilde{\tau}$. There is also a second RNE implementation (`mj_rnePost`) that can be called after the next-step velocity is computed. It takes into account the results of the impulse solver, and computes useful features such as interaction forces between bodies.

Impulses The final step is to compute the impulses f_E and f_I resulting from the constraints. This is a complex and non-standard topic addressed in a separate section below. Here we only provide a summary. We first eliminate f_E from the equations of motion, by expressing f_E as a function of the quantities we already computed and the still unknown f_I. This yields a modified equation in which the only remaining unknowns are v' and f_I. The reason for doing this is because the computation of f_I is the most complicated step (and the only one that requires an iterative method) thus we want to isolate it by first applying all analytical transformations that are possible. Then we project the equations of motion in inequality-constraint coordinates, and solve them subject frictional contact constraints.

2.4.1 Modifying the standard pipeline

The entire sequence of computations summarized above can be evoked with a single function: `mj_forward(model,data)`. However advanced users may prefer to modify the sequence and create custom computations. MuJoCo was developed as a research tool and is designed to make such modifications easy, in three ways. First, most of the steps can be enabled or disabled using the corresponding bits in `mjModel.option.disableflags`. Second, user callbacks can be plugged in at various places. Third, instead of using the high-level function, one can call multiple lower-level functions directly. In particular the following sequence of function calls achieves the same effect as `mj_forward(model,data)`:

```
mj_solvePosition(model,data,0);
mj_solveControl(model,data,0);
mj_solveActuation(model,data);
mj_solveVelocity(model,data);
mj_solveImpulse(model,data);
```

The position-dependent computations performed by `mj_solvePosition(model,data)` can be further broken down as:

```
mj_kinematics(model,data);
mj_com(model,data);
mj_tendon(model,data);
mj_transmission(model,data);
mj_crb(model,data);
mj_factorM(model,data);
mj_makeConstraint(model,data);
mj_projectConstraint(model,data);
mj_makeImpulse(model,data);
mj_projectImpulse(model,data);
```

All these functions operate on mjModel which is not modified, and on mjData which serves as a scratchpad: the outputs of one function become the inputs to the next.

2.5 Impulse solver

Impulses arising from equality and inequality constraints are handled very differently. In both cases however we use a similar mechanism for constraint stabilization, which we describe first.

2.5.1 Constraint stabilization

Physics engines often violate constraints and this is the case here as well. MuJoCo has a significant advantage over gaming engines in this regard, because it represents the system configuration q in joint coordinates, so it cannot even express a violation of joint constraints. Similarly, MuJoCo never violates the friction cone constraints on the contact forces. This is because the impulse solver in MuJoCo sets those forces directly, and it does so by solving an optimization problem which is always trivially feasible.

Other constraints however can be violated, in particular equality constraints other than joints, and inequality constraints corresponding to joint limits, tendon limits, and non-penetration between geoms. Such constraints depend on the body configuration which in turn depends nonlinearly on q, yet they are considered in linearized form (via the Jacobians), thus numerical integration in q-space with non-zero timestep can take the system outside the constraint manifold. We need a mechanism to correct such violations. Note that any such mechanism is non-physical (because the laws of physics are by definition undefined for constraint violations), and in particular it has the potential to inject energy and cause instabilities. Baumgarte's method is the most common approach: it basically adds a PD controller acting normal to the constraint manifold. This is also the starting point of our method (with parameters set to create critical damping), but we make an important change. Instead of letting this PD controller apply forces to our simulation, we only let it "recommend" the next-step velocity in constraint coordinates. This recommendation is then given to the impulse solver which ultimately decides what to do.

More precisely, let $e(t), \dot{e}(t)$ denote one scalar component of the current constraint violation and its velocity. We want e to be critically-damped with user-specified natural frequency ω:

$$\ddot{e} + 2\omega\dot{e} + \omega^2 e = 0$$

Integrating this ODE analytically for time h with initial condition $e(t), \dot{e}(t)$, evaluating the solution $e(t+h)$, and then differentiating to obtain the next-step velocity yields

$$\dot{e}(t+h) = \big(\dot{e}(t) - h\omega\dot{e}(t) - h\omega^2 e(t)\big) \exp(-h\omega)$$

This is how we set the desired next-step velocity v_E^* in equality-constraint coordinates. When inequalities are violated, we use the above formula to specify not the desired next-step velocity but rather the minimum of the next-step velocity. This is because, if some other force acting on the system ends up fixing the penetration, we do not want the constraint stabilization mechanism to fight back and result in a sticky contact. We denote the vector of such velocities $v_I^{\min}$. For inequalities that are not violated (i.e. the contact distance is

positive), we set $v_I^{\min}$ to the velocity that will bring the distance to 0 in one timestep. For frictional contacts only the normal components of this vector is set in this way, while the frictional components are always 0. The user actually specifies ω^{-1} instead of ω, separately for equality and inequality constraints: `eqerrreduce` and `imperrreduce` in mjModel.option. A value of 0 means that violations are (attempted to be) fixed completely in one timestep – usually resulting in unstable behavior.

2.5.2 Eliminating the equality constraints

As outlined above, we now seek to express f_E as a function of the remaining quantities (including the still unknown f_I) and eliminate it. We will not actually compute f_E, but rather its effect on the next-state velocity. This will be done by exploiting the Gauss principle of least constraint – which says that if we have unconstrained dynamics $f = Ma$ and acceleration constraint $J_E a = k$, the constrained acceleration is the solution to the following optimization problem:

$$\min_a \left(a - M^{-1}f\right)^T M \left(a - M^{-1}f\right) \quad \text{s.t.} \quad J_E a = k$$

The constraint in our case is on the next-step velocity rather than the acceleration:

$$J_E v' = v_E^*$$

To apply the Gauss principle we (temporarily) switch to acceleration form, with

$$a = \frac{v' - v}{h}, \quad k = \frac{v_E^* - J_E v}{h}, \quad f = c + \tau + \frac{1}{h} J_I^{\mathsf{T}} f_I$$

Note that f denotes all forces acting on the system except for the constraint forces whose effect we are trying to compute.

At this point we will proceed in two ways and make both available in MuJoCo. One is to impose a hard constraint, the other is a soft constraint. The latter is needed to make the dynamics invertible (again this functionality is not yet exposed, however we are setting the stage for it). Even with a hard constraint we will merely follow the recommendation v_E^* of the constraint stabilizer – which still allows some softness because it does not attempt to fix violations in one timestep. The interplay between these two sources of softness needs investigation.

Hard constraint The solution to the above constrained optimization problem is

$$\begin{aligned} a &= M_E^{-1} f + M^{-1} J_E^T A_E^{-1} k \\ \text{where} \quad M_E^{-1} &= M^{-1} - M^{-1} J_E^T A_E^{-1} J_E M^{-1} \\ A_E &= J_E M^{-1} J_E^T \end{aligned}$$

Note that M_E^{-1} is singular, thus M_E – which is the effective inertia after the equality constraints have been taken into account – is infinite in the subspace corresponding to the

constraints. This is fine because we only need M_E^{-1}. Substituting a, k, f with their values in our setting and rearranging, we have

$$\begin{aligned} v' &= \widehat{v} + M_E^{-1} J_I^{\mathsf{T}} f_I \\ \text{where} \quad \widehat{v} &= v + hM_E^{-1}(c+\tau) + M^{-1} J_E^T A_E^{-1}(v_E^* - J_E v) \end{aligned}$$

$\widehat{v}$ only depends on terms that we already know. What remains to be computed is f_I.

Soft constraint The alternative is to replace the hard constraint with a soft cost, weighted by a scalar λ specified by the user. The acceleration is now defined as the solution to an unconstrained optimization problem:

$$\min_a \left(a - M^{-1} f\right)^T M \left(a - M^{-1} f\right) + \lambda \left(J_E a - k\right)^T \left(J_E a - k\right)$$

The solution to this problem is

$$\begin{aligned} a &= M_E^{-1}\left(f + \lambda J_E^T k\right) \\ \text{where} \quad M_E^{-1} &= \left(M + \lambda J_E^T J_E\right)^{-1} \end{aligned}$$

Here M_E^{-1} is defined differently, and is no longer singular. Substituting a, k, f now yields

$$\begin{aligned} v' &= \widehat{v} + M_E^{-1} J_I^{\mathsf{T}} f_I \\ \text{where} \quad \widehat{v} &= v + hM_E^{-1}(c+\tau) + \lambda M_E^{-1} J_E^T (v_E^* - J_E v) \end{aligned}$$

One can verify that in the limit $\lambda \to \infty$ the solution to the soft constraint converges to the solution of the hard constraint, at a rate λ^{-1}.

The soft constraint mechanism is not yet implemented.

2.5.3 Solving the inequality constraints

Both of the above approaches lead to the same equation:

$$v' = \widehat{v} + M_E^{-1} J_I^{\mathsf{T}} f_I$$

The only difference is in how M_E^{-1} and $\widehat{v}$ are computed. For the rest of the computation however, it does not matter how these quantities were computed. Note that in the absence of equality constraints, we simply have $M_E = M$ and $\widehat{v} = v + hM^{-1}(c+\tau)$.

We now project the above equation into inequality-constrained coordinates by multiplying by J_I:

$$J_I v' = J_I \widehat{v} + J_I M_E^{-1} J_I^{\mathsf{T}} f_I$$

To simplify notation in the rest of this section, we will write this as

$$Af + v_0 = v_1$$

where A is the inverse inertia, v_0 is the next-step velocity that would result if there was no impulse, and v_1 is the next-step velocity after the impulse.

The elements of the vectors we are now working with are as follows:

- if dry joint friction is defined in the model, the frictional dofs are included first;
- if any limits are detected – which could be joint limits, tendon limits, or frictionless contacts which are computationally equivalent to limits – then the limits are included next;
- finally any frictional contacts are included, each contact starting with the normal direction followed by tangential and optionally torsional and rolling friction.

The corresponding quantities in mjData have the prefix `flc` which stands for "friction, limit, contact". Several indexing variables are also available to navigate these vectors.

In the above equation, we know A, v_0 and wish to compute f, v_1. Note that in inverse dynamics computations (not yet exposed) the situation is partially reversed: we know A, v_1 and wish to compute f, v_0. Here we focus on the forward computation.

Since we have twice as many uknowns as equations, we need additional constraints. These come from a model of the laws of frictional contact – which is still a very active research topic. The most common approach is to use a linear complementarity formulation (LCP) however this problem is likely to be NP-hard and the available algorithms (such as Lemke's method) are too inefficient. Instead, we and others have developed complementarity-free methods based on convex approximations. We define the impulse f as the solution to the following constrained optimization problem:

$$\min_f \frac{1}{2} (v_1 - v_{\min})^T A^{-1} (v_1 - v_{\min}) + \frac{1}{2} f^T R f \quad \text{s.t.} \quad f \in \text{FrictionCone}$$

This corresponds to minimizing kinetic energy, except we use the previously defined quantity $v_{\min}$ as "reference" instead of 0, so as to correct for possible penetration. The matrix R is a diagonal regularizer which plays an important role in practice (and needs better documentation). Its magnitude is controlled by the `softness` parameters in the model. This minimization problem can be written as

$$\min_f \frac{1}{2} f^T (A + R) f + f^T (v_0 - v_{\min}) \quad \text{s.t.} \quad f \in \text{FrictionCone}$$

FrictionCone is an elliptical cone when torsional and rolling friction are enabled (because the friction coefficients are generally different). For joint friction it is an interval bounded by the user-defined friction loss. For limits it is $f \geq 0$.

One can consider many iterative algorithms for solving this convex optimization problem. Presently MuJoCo offers several algorithms, but until further notice we recommend using only `mjALG_GS` which is a cone-projected Gauss-Seidel solver. Implementing better convex solvers, as well as the solver described in our earlier work on implicit nonlinear complementarity, is an important direction for future work. It is notable however that in the present context Gauss-Seidel needs surprisingly few iterations – we usually use 5. This is in sharp contrast with gaming engines where the same type of solver is used to compute contact impulses and impose joint constraints simultaneously – which needs many more iterations.