MuJoCo: A physics engine for model-based control

Conference Paper *in* Proceedings of the ... IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE/RSJ International Conference on Intelligent Robots and Systems · October 2012

DOI: 10.1109/IROS.2012.6386109

3 authors, including:

Yuval Tassa
Google Inc.
42 PUBLICATIONS **4,839** CITATIONS

MuJoCo: A physics engine for model-based control

Emanuel Todorov, Tom Erez and Yuval Tassa
University of Washington

***Abstract*— We describe a new physics engine tailored to model-based control. Multi-joint dynamics are represented in generalized coordinates and computed via recursive algorithms. Contact responses are computed via efficient new algorithms we have developed, based on the modern velocity-stepping approach which avoids the difficulties with spring-dampers. Models are specified using either a high-level C++ API or an intuitive XML file format. A built-in compiler transforms the user model into an optimized data structure used for runtime computation. The engine can compute both forward and inverse dynamics. The latter are well-defined even in the presence of contacts and equality constraints. The model can include tendon wrapping as well as actuator activation states (e.g. pneumatic cylinders or muscles). To facilitate optimal control applications and in particular sampling and finite differencing, the dynamics can be evaluated for different states and controls in parallel. Around 400,000 dynamics evaluations per second are possible on a 12-core machine, for a 3D homanoid with 18 dofs and 6 active contacts. We have already used the engine in a number of control applications. It will soon be made publicly available.**

I. Introduction

As robotic hardware becomes more complex and capable, the importance of simulation tools increases. Existing physics engines can be used to test controllers that are already designed. However they lack the speed, accuracy and overall feature sets needed to automate the controller design process itself. On the other hand, software packages for automatic controller design (to the extent that they exists) are not integrated with physics engines, effectively limiting them to scenarios where the plant dynamics can be written down explicitly. In the absence of adequate tools, the field continues to rely on manual controller designs – which may be a large part of the reason why present-day robots do not perform as well as one may have hoped given the impressive sensors, actuators and computing power that are available.

Before presenting our work, we briefly discuss the requirements for controller design software. First and foremost, such software should be based on a suitable mathematical and algorithmic foundation. While many approaches to automatic control exist, and a proper comparison of their merits is beyond the scope of this paper, we believe that numerical optimization is the most powerful and generally applicable tool for automating processes that would otherwise require human intelligence. Indeed most algorithms in Artificial Intelligence, Machine Learning, Computer Vision, System Identification, State Estimation come down to numerical optimization. In the context of robotic control, numerical optimization is the basis of Optimal Control and is equally applicable to Path Planning, Model-predictive Control, and in general tuning the parameters of any parametric controller.

The essence of control optimization is to automatically construct many candidate controllers, evaluate their performance in simulation, and use the data to construct better controllers. This process can rely on sampling (as in evolutionary algorithms or Reinforcement Learning) or on gradient information – which is usually obtained via finite differencing because the dynamics of a complex robot are too complex to differentiate analytically, especially in the presence of contact dynamics. Either way, optimizing a controller requires a vast number of dynamics evaluations for different states and controls. For example, in our recent work on *de novo* synthesis of humanoid running gaits [3] we needed around $200,000,000$ evaluations – which takes less than 10 minutes on a 12-core machine using the software described here. With a time-step of 15 msec (at which our simulations are stable) this is roughly $5,000$ times faster than real-time. If we used an engine that merely runs in real-time, as for example the Open Dynamics Engine (ODE) when used in a similar context [5], we would have to wait for a month. The three orders-of-magnitude speed advantage can be broken down as follows: one order of magnitude is due to faster computation, one order of magnitude is due to parallel processing which fully utilizes all available processors, and one order of magnitude is due to higher accuracy and stability allowing larger time-steps; for comparison, ODE requires time-steps below 1 msec to achieve stable locomotion [5].

Controller optimization also calls for high simulation accuracy. The approximations used in gaming engines are often justified with the argument that accuracy is not so important as long as the simulation is stable. This may be true in scenarios where one can tune the engine's parameters to make an existing controller look realistic. In the context of control optimization, however, the controller is being "tuned" to the engine and not the other way around. As Sims [7] pointed out, if the physics engine allows cheating the optimization algorithm will find a way to exploit it – and produce a controller that achieves its goal (in the sense of optimizing the specified cost function) in a physically unrealistic way. This has been our experience as well.

The requirements for speed and accuracy are obvious in principle. What is less obvious however is that, in the context of control optimization, these requirements become so demanding that none of the existing physics engines can meet them. Indeed we were unaware of this a few years back when we attempted to build the control functionality we needed on top of ODE, and were quickly disappointed. ODE as well as other game-oriented engines (such as NVIDIA PhysX and Bullet Physics) represent the system state in over-

complete Cartesian coordinates and enforce joint constraints numerically. This is sensible when simulating a large number of mostly disconnected bodies with few joint constraints, however it becomes both inaccurate and inefficient when simulating elaborate multi-joint systems such as humanoids. Another issue with game engines lies in the contact dynamics, formulated as (approximations to) linear complementarity problems or LCPs [8]. Although this approach is a significant improvement over earlier spring-damper models of contact, it still requires manual tuning and small time steps. On the other end of the spectrum are engines such as SD/FAST and OpenSim, which represent the system state and perform all computations in joint coordinates, and thus do not need to enforce joint constraints numerically (except for occasional loop joints). However the latter engines either ignore contacts, or use spring-dampers resulting in large penetrations or in stiff dynamics. This limits their applications to robotics where contact dynamics are key.

These observations indicated that we need a new engine, representing the state in joint coordinates and simulating contacts in ways that are related to LCP but better. Thus the name MuJoCo – which stands for **Mu**lti-**Jo**int dynamics with **Co**ntact. We developed several new formulations of the physics of contact [11], [12], [10] and implemented the resulting algorithms in MuJoCo. Note that contact simulation is an area of active research, unlike simulation of smooth multi-joint dynamics where the book has basically been written [4]. This is why we have decided to provide multiple mechanisms for modeling contact dynamics, and allow the user to select the one most suitable for a given system. In addition, MuJoCo has several unique features which are rarely needed for simulation purposes but greatly facilitate control applications:

- The same dynamical system can be evaluated in parallel (in shared memory or distributed architectures) for different states and controls. This is useful for approximating derivatives via finite differencing, which in turn enables numerical optimization.
- Inverse dynamics can always be computed, even in the presence of contacts and equality constraints. Inverse dynamics are useful for analysis of recorded data as well as computed torque control applications.
- Actuator dynamics such as the pressures inside pneumatic or hydraulic cylinders as well as the activations of biological muscles can be modeled. Actuators can transmit forces via linkages or tendons. Tendons can wrap around 3D shapes, and can have hard length limits.
- There are multiple ways to access MuJoCo's functionality: a standalone executable with interactive interface allowing the user to 'reach into' the simulation with a 6D mouse; interfaces to MATLAB and soon ROS; and a static C library that can be linked to user programs.
- The user can invoke the entire pipeline or only pieces of it, facilitating the implementation of non-standard computations. MuJoCo's functionality can be further extended with a variety of callbacks specifying control laws, passive force fields, custom equality constraints, and custom contact solvers.
- Models are created in an intuitive XML format or using a C++ API, and then compiled automatically into low-level data structures optimized for runtime computation. XML model files in other formats (e.g. URDF) can be converted to the native MuJoCo format. The native format is designed to be human readable and editable.

The rest of the paper is organized as follows. Section II describes the main algorithms implemented in MuJoCo. Section III provides an overview of the modeling convention. Section IV presents timing tests and comparisons to SD/FAST – which does not handle contacts, but is the best prior engine for multi-joint dynamics in our opinion. Section V provides a summary and outlines future work.

II. Algorithmic foundations

Here we provide a summary of the numerical algorithms implemented in MuJoCo. We start with notation and smooth dynamics which are fairly standard, then explain the contact simulation algorithms in more detail, followed by computational complexity and inverse dynamics.

A. Equations of motion and smooth dynamics

We will use the following notation:

$\mathbf{q}$	position in generalized coordinates
$\mathbf{v}$	velocity in generalized coordinates
M	inertia matrix in generalized coordinates
$\mathbf{b}$	"bias" forces: Coriolis, centrifugal, gravity, springs
τ	external/applied forces
ϕ	equality constraints: $\phi(\mathbf{q}) = 0$
J_E	Jacobian of equality constraints
$\mathbf{v}_E^*$	desired velocity in equality constraint coordinates
$\mathbf{f}_E$	impulse caused by equality constraints
J_C	Jacobian of active contacts
$\mathbf{v}_C$	velocity in contact coordinates
$\mathbf{f}_C$	impulse caused by contacts
h	time step

3D rotations (ball joints and base orientations) are represented as unit quaternions, while their rotational velocities are represented as 3D vectors. Thus in general we have $\dim(\mathbf{v}) < \dim(\mathbf{q})$, so we write $\mathbf{v}$ instead of $\dot{\mathbf{q}}$.

The equations of motion in continuous time are

$$M(\mathbf{q})\, d\mathbf{v} = (\mathbf{b}(\mathbf{q}, \mathbf{v}) + \tau)\, dt + J_E(\mathbf{q})^{\mathsf{T}} \mathbf{f}_E(\mathbf{q}, \mathbf{v}, \tau) + J_C(\mathbf{q})^{\mathsf{T}} \mathbf{f}_C(\mathbf{q}, \mathbf{v}, \tau)$$

Note that we have not divided by dt. This is because constraint and contact forces are impulsive, and the latter are generally inconsistent with continuous-time formulations (e.g. Painleve's paradox). Indeed in order to model hard contacts (as opposed to spring-damper approximations), we will have to adopt a discrete-time velocity-based formulation [8] which is now standard in gaming engines. The procedure for solving the above equations of motion consists of the following steps:

1) Compute the Cartesian positions and orientations of all rigid bodies (i.e. the forward kinematics), detect potential collisions (with some safety margin), and construct the Jacobians J_E, J_C.
2) Compute the inertia matrix M using the Composite Rigid Body (CRB) algorithm, the bias forces $\mathbf{b}$ using the Recursive Newton Euler (RNE) algorithm, and the sparse $\mathrm{L}^\mathsf{T}\mathrm{DL}$ factorization of M.
3) Express the equality constraint impulse $\mathbf{f}_E$ as a function of the (still unknown) contact impulse $\mathbf{f}_C$, and project the dynamics in the subspace tangent to the equality constraint manifold. Apply constraint stabilization.
4) Further project the dynamics in contact coordinates, and solve for the contact impulse $\mathbf{f}_C$ and the resulting contact velocity $\mathbf{v}_C$.
5) Integrate numerically to obtain the next state.

Steps 1, 2 are standard [4]. We now clarify steps 3, 4, 5. The desired next-step velocity $\mathbf{v}_E^*(\mathbf{q})$ in equality constraint coordinates is computed from the constraint violation $\phi(\mathbf{q})$, in such a way that if the system follows $\mathbf{v}_E^*$ exactly, the violation will decay to 0 as a critically-damped spring. This resembles Baumgarte stabilization, except that here $\mathbf{v}_E^*$ is directly enforced at the next time step – which can be more accurate than traditional Baumgarte stabilization.

We now focus on the computations in a single time step. Replacing $d\mathbf{v}$ with $(\mathbf{v}_{t+h} - \mathbf{v}_t)$ and dt with h, and taking into account the velocity transformations specified by the Jacobians, the equations of motion in discrete time become

$$M(\mathbf{q}_t)\,\mathbf{v}_{t+h} = \mathbf{s}(\mathbf{q}_t, \mathbf{v}_t, \tau_t) + J_E(\mathbf{q}_t)^\mathsf{T}\mathbf{f}_E + J_C(\mathbf{q}_t)^\mathsf{T}\mathbf{f}_C \tag{1a}$$

$$J_E(\mathbf{q}_t)\,\mathbf{v}_{t+h} = \mathbf{v}_E^*(\mathbf{q}_t) \tag{1b}$$

$$J_C(\mathbf{q}_t)\,\mathbf{v}_{t+h} = \mathbf{v}_C \tag{1c}$$

where the vector $\mathbf{s}$ is defined as

$$\mathbf{s}(\mathbf{q}_t, \mathbf{v}_t, \tau_t) = M(\mathbf{q}_t)\,\mathbf{v}_t + h\mathbf{b}(\mathbf{q}_t, \mathbf{v}_t) + h\tau_t$$

We will now drop the time indices and functional dependencies for clarity. At this point the quantities $M, \mathbf{s}, J_E, J_C, \mathbf{v}_E^*$ are known, while $\mathbf{f}_E, \mathbf{f}_C, \mathbf{v}_C, \mathbf{v}$ remain to be computed.

Using (1a, 1b) and the fact that M is always invertible, we can express $\mathbf{f}_E$ as a function of $\mathbf{f}_C$:

$$\mathbf{f}_E = \left(J_E M^{-1} J_E^\mathsf{T}\right)^{-1}\left(\mathbf{v}_E^* - J_E M^{-1}\left(\mathbf{s} + J_C^\mathsf{T}\mathbf{f}_C\right)\right)$$

Substituting this in (1a) and solving for $\mathbf{v}$ yields

$$\mathbf{v} = P J_C^\mathsf{T}\mathbf{f}_C + \mathbf{r} \tag{2}$$

where $P, \mathbf{r}$ are defined as

$$P = M^{-1} - M^{-1}J_E^\mathsf{T}\left(J_E M^{-1} J_E^\mathsf{T}\right)^{-1} J_E M^{-1}$$

$$\mathbf{r} = P\mathbf{s} + M^{-1}J_E^\mathsf{T}\left(J_E M^{-1} J_E^\mathsf{T}\right)^{-1}\mathbf{v}_E^*$$

This is the projection in the subspace tangent to the equality constraint manifold (step 3). When there are no equality constraints we have $P = M^{-1}$ and $\mathbf{r} = M^{-1}\mathbf{s}$. In terms of implementation, multiplication by M^{-1} is done using sparse back-substitution (following sparse factorization), and the recurring matrix $M^{-1}J_E^\mathsf{T} = \left(J_E M^{-1}\right)^\mathsf{T}$ is computed only once and reused when necessary.

At this point we know $P, \mathbf{r}, J_C$ in (2) and need to compute $\mathbf{f}_C$, which corresponds to step 4 and will be described later. Once this is done, we use (2) to obtain $\mathbf{v}_{t+h}$, and then integrate the position. MuJoCo provides two integrators. One is the semi-implicit Euler method:

$$\mathbf{q}_{t+h} = \mathbf{q}_t \text{ " + " } h\mathbf{v}_{t+h}$$

Semi-implicit refers to the fact that we are using the next-step velocity. This approach tends to be more accurate than explicit integration, and is key to velocity-stepping schemes. The plus sign is in quotes because 3D rotations are expressed as unit quaternions while their velocities are 3D vectors, and so we use formulas for quaternion integration rather than simple addition. The other integrator is 4^{th}-order Runge-Kutta, modified to work with next-step velocities rather than accelerations.

B. Solving for the contact impulse

We now return to step 4. Substituting (2) in (1c) yields the following equation in contact coordinates:

$$A\mathbf{f}_C + \mathbf{v}_0 = \mathbf{v}_C \tag{3}$$

where $A, \mathbf{v}_0$ are defined as

$$A = J_C P J_C^\mathsf{T}$$

$$\mathbf{v}_0 = J_C\mathbf{r}$$

The matrix A is the inverse inertia projected in the subspace tangent to the equality constraint manifold, and then expressed in the contact coordinates. The vector $\mathbf{v}_0$ is the contact velocity which results in the absence of an impulse. The quantities $A, \mathbf{v}_0$ are known, while $\mathbf{f}_C, \mathbf{v}_C$ need to be computed.

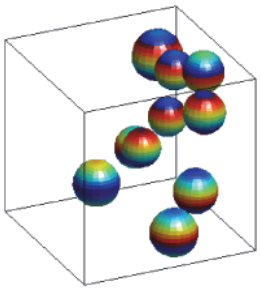
Let $\dim(\mathbf{f}_C) = k$. We have to compute $2k$ scalar quantities while (3) provides only k equality constraints. Thus we need an additional k effective constraints in order to obtain a unique solution. These additional constraints come from the Coulomb friction model with complementarity conditions [8], or some approximation to it as discussed later. Focusing for the moment on a single contact, let the contact impulse $\mathbf{f}_C$ be partitioned as $\left[f^\mathcal{N}; \mathbf{f}^\mathcal{F}\right]$ where $f^\mathcal{N}$ is the normal component and $\mathbf{f}^\mathcal{F} \in \mathbb{R}^2$ is the tangential/friction component, and similarly for $\mathbf{v}_C$. Along the normal we have

$$f^\mathcal{N} \geq 0, \quad v^\mathcal{N} \geq 0, \quad f^\mathcal{N} v^\mathcal{N} = 0 \tag{4}$$

These conditions correspond to the fact that the contact impulse cannot pull the bodies towards each other, the bodies cannot penetrate, and if the contact is breaking then there can be no contact impulse. In the tangent plane we have

$$\mathbf{v}^\mathcal{F} \text{ parallel to } \mathbf{f}^\mathcal{F}, \quad \left\langle \mathbf{v}^\mathcal{F}, \mathbf{f}^\mathcal{F} \right\rangle \leq 0$$
$$\left\|\mathbf{f}^\mathcal{F}\right\| \leq \mu f^\mathcal{N} \tag{5}$$

The first line means that if there is slip then the friction force must act in the direction opposite to the slip velocity. The



mu \ nc	7	16	27
0.1	3.8	4.8	6.5
0.5	2.6	5.3	7.5
1.0	2.8	4.9	10.2
2.0	2.9	4.6	16.2

Fig. 1. Left: the algorithm is tested on a system consisting of several balls moving inside a cube. Right: average number of Newton-like iterations as a function of the number of contacts **nc** and the friction coefficient **mu**.

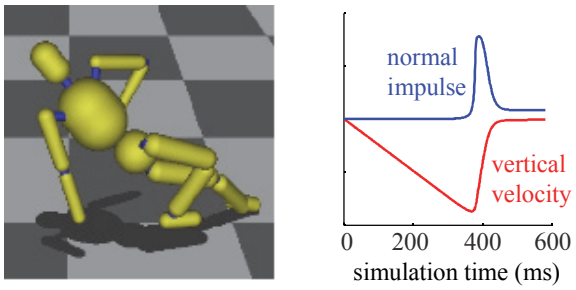


Fig. 2. Left: Rendering of a humanoid (used for testing) in the MuJoCo interactive 3D GUI. Right: a ball-drop test of the convex contact solver. Note how the contact impulse is smoothed, and yet there is no penetration.

second line means that the contact force must lie inside the friction cone, where μ is the coefficient of friction.

Condition (4) is called a complementarity condition. Together with a linear equation such as (3) it can form a linear complementarity problem (LCP). Condition (5) on the other hand does not fit in the LCP framework because it involves nonlinearities. One approach is to replace the friction cone with a pyramid and convert the problem into an LCP [8], which can then be solved with Lemke's algorithm or other methods such as the PATH solver. These methods however do not scale well for larger problems [1], which is not surprising given that the problem is generally NP-hard [6]. Furthermore the pyramid approximation introduces errors.

Instead of following the LCP approach, MuJoCo implements three new algorithms for contact simulation based on our recent work, as summarized next.

C. Implicit complementarity solver

The most accurate of the solvers provided in MuJoCo is based on [11]. It aims to find an exact solution to (3, 4, 5). The key idea is as follows. Instead of treating $\mathbf{f}_C$ and $\mathbf{v}_C$ as independent quantities constrained by complementarity, we express both as functions of a new hybrid variable $\mathbf{x}$. This is possible thanks to the complementarity conditions. In the normal direction for example, if the corresponding component of $\mathbf{x}$ is positive it encodes force (in which case the velocity is 0), otherwise it encodes velocity and the force is 0. We then solve the nonlinear equation

$$A\mathbf{f}_C(\mathbf{x}) + \mathbf{v}_0 = \mathbf{v}_C(\mathbf{x})$$

by converting it to an unconstrained optimization problem:

$$\min_{\mathbf{x}} \|A\mathbf{f}_C(\mathbf{x}) + \mathbf{v}_0 - \mathbf{v}_C(\mathbf{x})\|^2$$

The optimization uses a customized non-smooth Newton method; in the line-search phase it exploits the fact that the functions $\mathbf{f}_C(\mathbf{x})$ and $\mathbf{v}_C(\mathbf{x})$ are non-smooth only at planes and friction-cone boundaries. See [11] for details.

Since the underlying problem is NP-hard, the algorithm cannot always find the exact solution (which has 0 residual). Nevertheless it finds the exact solution almost all the time, and when it does not it still converges to a sensible solution. Furthermore the number of Newton-like iterations is surprisingly small, as illustrated in Fig. 1 taken from [11]. Each iteration involves factorization of a k-by-k matrix; this could potentially be improved using Hessian-free methods.

D. Convex solver

A favorable trade-off between speed and accuracy is obtained by replacing the nonlinear complementarity constraints (4, 5) with a convex optimization problem, whose solution is very similar in practice [1] yet it can be computed more efficiently. The algorithm is based on [12], which turns out to be related to [2] even though it was developed independently. The advantage of [12] is that the resulting contact dynamics are invertible (see below).

The idea here is that contact impulses act to reduce the relative velocity between contacting surfaces. It then makes sense to define the kinetic energy in contact space, which is $\mathbf{v}_C^T A^{-1} \mathbf{v}_C / 2$, and minimize it subject to friction-cone and non-penetration constraints. The remaining constraints from (4, 5) are omitted because they make the optimization problem non-convex. The friction cone is imposed as a hard constraint, while non-penetration $\left(v^{\mathcal{N}} \geq 0\right)$ is imposed with a cost $q(\mathbf{v}_C)$ because otherwise the inverse dynamics could not be defined for trajectories that happen to have penetration. We have

$$\mathbf{f}_C = \arg \min_{\mathbf{f} \in \text{Cone}} \frac{1}{2} \mathbf{f}^T (A + R) \mathbf{f} + \mathbf{f}^T \mathbf{v}_0 + q(A\mathbf{f} + \mathbf{v}_0) \quad (6)$$

R is a small (diagonal) regularization term. It is needed for three reasons: A is often singular; without R the inverse cannot be defined (see below); one can enable contact interactions from a distance – which can be very useful in continuation methods for numerical optimization – and control the contact force magnitude by increasing R with distance. In addition to enabling continuation, the convex solver is well suited for numerical optimization because it smooths the dynamics while still producing phenomenologically hard contacts. This is illustrated with a ball-drop test in Fig. 2, taken from [12].

The convex solver is actually a family of solvers, because the convex optimization problem (6) can be handled with a variety of numerical methods. Interior-point methods were advocated in [2], [12] and they are certainly a viable option, however projected methods (Newton, conjugate-gradient, or

Gauss-Seidel) may be faster and have comparable accuracy. By "projected" we mean that the cone constraints are enforced after each iteration, or a non-smooth line-search respecting the constraints is used. Projected methods are usually limited to box constraints, however we have been able to generalize them to cone constraints. Presently a cone-projected Gauss-Seidel method is our favorite [9].

E. Diagonal solver

The least accurate but fastest contact solver is a diagonal solver, which can be thought of as a mass-aware spring-damper. The difficulty with traditional spring-damper models is that, if the contact-space inertia is too large the contact will be springy and result in large penetration and subsequent oscillation, while if the inertia is too small the dynamics will be stiff and very difficult to integrate numerically. Since the contact-space inertia is configuration-dependent while the spring-damper coefficients are fixed, this problem may seem unavoidable. However, if we have access to the diagonal of the A matrix we can tune the spring-dampers online, and for example make sure that we always have critically-damped springs at all contacts. This is essentially what our diagonal solver does, except it does not explicitly simulate spring-dampers. Instead it computes the desired next-state velocity $\mathbf{v}_C^*$ which would result if penetrations decayed like critically-damped springs (similar to equality-constraint violations) and there was no slip, and then solves

$$\operatorname{diag}(A)\,\mathbf{f}_C = \mathbf{v}_C^* - \mathbf{v}_0 \quad \text{s.t.} \quad \mathbf{f}_C \in \text{Cone}$$

approximately. This is done by computing the components of $\mathbf{f}_C$ independently for each contact (the diagonal solver ignores contact interactions by definition) and enforcing the friction-cone constraints, with the same projection method as above. A regularization term R can again be included.

F. Computational complexity

We now address the computational complexity of the entire pipeline, including the smooth and impulsive dynamics. Let $n = \dim(\mathbf{v})$, $m = \dim(\mathbf{v}_E)$, and $k = \dim(\mathbf{v}_C)$. Thus M, P are n-by-n, J_E is m-by-n, J_C is k-by-n, and A is k-by-k. The computational complexity of the different steps can be summarized as follows:

- computing $\mathbf{b}$ via RNE is $O(n)$
- computing M via CRB is $O(n^2)$
- factorizing M is $O(n^3)$
- computing J_E is $O(n^2 m)$
- computing J_C is $O(n^2 k)$
- computing P is $O(n^2 m)$
- computing A is $O(n^2 k)$
- factorizing A is $O(k^3)$.

While these theoretical results are important for understanding how the CPU time will scale with the number of DOFs, in practice the performance of the engine is dominated by other factors, for the following reasons. First, the values of n are small – rarely exceeding 50 and usually less. Second, the branch-induced sparsity of M makes sparse factorization a lot faster than $O(n^3)$ as shown in [4]. Thus, since the number of equality constraints m is usually small (if present at all), the bottleneck is the $O(n^2 k)$ computation of A and the repeated $O(k^3)$ factorizations inside the contact solvers. Third and perhaps most important, counting the number of floating point operations used to be essential when floating point arithmetic was slow, but this is no longer the case. Instead, modern processors use comparable numbers of cycles for floating point, integer and indexing operations – all of which are very fast. The bottleneck now is in memory access. Thus the performance of physics engines such as MuJoCo tends to be dominated by cache misses more than traditional computational complexity considerations, and the only way to assess performance reliably is to run extensive timing tests.

Forward dynamics in the absence of contacts can alternatively be computed using $O(n)$ recursive algorithms. However in the presence of branch-induced sparsity typical in robotics these algorithms are not much faster than the present approach [4]. More importantly, $O(n)$ forward dynamics does not compute the inertia matrix M which is needed to compute A, which in turn is needed for all contact simulation methods that avoid the shortcomings of spring-dampers.

G. Inverse dynamics

We now describe the computation of inverse dynamics, which is a unique feature of MuJoCo. It can be used to analyze data or to compute the torques that will cause a robot to follow a reference trajectory. Another important application is in direct trajectory optimization (also known as space-time optimization) where the variables being optimized are the sequence of positions $\mathbf{q}$. Velocities $\mathbf{v}$ are then defined via numeric differentiation, control forces τ are defined via inverse dynamics, and the cost of the trajectory is computed as a function of all these quantities. These ideas are developed at length in [3]. Here we limit ourselves to the inverse dynamics computation.

Ignoring equality constraints and contacts for the moment, the RNE algorithm can be used to compute

$$M(\mathbf{q})\,\dot{\mathbf{v}} - \mathbf{b}(\mathbf{q}, \mathbf{v}) = \mathbf{f}_{\text{tot}}$$

where $\mathbf{f}_{\text{tot}}$ is the sum of control forces τ which we aim to compute, plus constraint and contact forces which are also unknown. Indeed RNE was originally designed for inverse dynamics, although it is also applicable in forward dynamics for computing $\mathbf{b}$, by setting $\dot{\mathbf{v}} = 0$. The unusual negative sign in front of $\mathbf{b}$ is our notational convention.

Our goal now is to distribute $\mathbf{f}_{\text{tot}}$ as

$$\mathbf{f}_{\text{tot}} = \tau + h^{-1} J_E(\mathbf{q})^{\mathsf{T}}\,\mathbf{f}_E + h^{-1} J_C(\mathbf{q})^{\mathsf{T}}\,\mathbf{f}_C$$

where $\mathbf{f}_{\text{tot}}, h, J_E, J_C$ are known and $\tau, \mathbf{f}_E, \mathbf{f}_C$ are unknown. Importantly, we want the inverse dynamics to be well-defined for any $(\mathbf{q}, \mathbf{v}, \dot{\mathbf{v}})$ regardless of constraint and penetration violations. This is because such violations are likely to be present both in recorded data and in early stages of trajectory optimization. Of course violations should be difficult to achieve, i.e. the inferred control force τ should be large in the corresponding subspace. Thus, in essence, we want the

dynamics to become more springy. The forward dynamics as defined above do not have this property because $\mathbf{f}_E$ and $\mathbf{f}_C$ are computed with knowledge of τ, and are scaled automatically so as to cancel any components of τ that would cause violations.

This leads to the following general approach to inverse dynamics in the presence of impulse solvers. Define a *posthoc* mode where all impulses are computed as if $\tau = 0$, and then apply the actual τ at the end of the time step by adding $hM^{-1}\tau$ to the next-step velocity. We use a modified M in this procedure, so that the inertia seen by the control forces is increased in the subspace corresponding to violations:

$$M \leftarrow M + w_E J_E^T J_E + w_C J_C^T J_C$$

The coefficients w_E, w_C can be adjusted by the user.

When there are no equality constraints and the contact solver has an exact inverse, the inverse dynamics can be computed without resorting to posthoc mode. The convex contact solver in particular is invertible [12], meaning that given A and $v_C = A\mathbf{f}_C + \mathbf{v}_0$ we can recover $\mathbf{f}_C$ and $\mathbf{v}_0$. Indeed using the fact that $\mathbf{f}_C$ is the global minimizer of (6) in the forward dynamics, we can show that the same $\mathbf{f}_C$ can be recovered in the inverse dynamics by solving the following convex optimization problem:

$$\mathbf{f}_C = \arg\min_{\mathbf{f}\in\text{Cone}} \frac{1}{2}\mathbf{f}^T R\mathbf{f} + \mathbf{f}^T\left(\mathbf{v}_C + A^T \nabla q\left(v_C\right)\right)$$

This is easier to solve numerically than (6) because the objective function is now quadratic. If we omit the penetration cost $q\left(v_C\right)$ and instead rely on energy minimization to set the normal velocity to near 0, the matrix A drops altogether, meaning that we can also omit the computation of M.

III. Modeling

A. Different ways to construct a MuJoCo model

MuJoCo models can exist on three levels of description:

- XML model file in a new format called MJCF;
- Sequence of C++ API calls for model construction;
- Low-level C structure generated by the built-in compiler, and optionally saved and loaded in binary files.

All descriptions contain the same information but in different formats. They are related as follows. The MJCF file is an intuitive text description written by the user. The built-in parser loads the MJCF file and creates a runtime C++ object describing the entire model. When using MuJoCo as an executable, this is the only way to create models. When MuJoCo is linked as a library to a user program, the user can still call the parser programmatically, but now has the option of making the same sequence of API calls that the parser would have made and thereby constructing the same model. The parser route is usually the most convenient. The API route is better when working with large models that can be created programmatically, e.g. a chain with 100 identical links. Once a valid model object is created in the runtime environment by either method, the built-in compiler converts it into a low-level C structure used for all subsequent computations.

Here is a fully functional XML file in MJCF format, describing a floating box which can collide with the ground:

```
<?xml version="1.0" encoding="utf-8"?>
<mujoco version="1.0">
  <world>
    <geom type="plane" pos="0 0 0"/>
    <body>
      <joint type="free"/>
      <geom type="box" pos="0 0 5"
            size="1 2 3"/>
    </body>
  </world>
</mujoco>
```

A physical simulation needs a lot more information than what this file specifies directly. The missing information is filled-in automatically using suitable defaults, which can themselves be redefined by the user. A full explanation of the modeling convention is beyond the scope of this paper, but one important feature is the ability to specify body inertial properties automatically, by inferring them from the geom shapes and (default) material density. In fact one can define multiple geoms per body, and MuJoCo will combine their masses and inertias and assign them to the body. Other conveniences include an option to use either local or global coordinates, and multiple ways for specifying orientations.

B. Elements of a MuJoCo model

a) Body: Bodies are the elements used to build kinematic trees. A MuJoCo model consists of one or several kinematic trees, which can have floating bases including isolated objects. Bodies have mass and inertia matrix but do not have any geometric properties; they are just coordinate frames with inertial properties. Internally each body has a local coordinate frame which is centered at the center of mass and is aligned with the principal axes of inertia. A 'world' body is always defined.

b) Joint: Joints are defined inside bodies. They create motion degrees of freedom between the body and its parent; in the absence of joints the parent and child bodies are welded together. Note that this is the opposite of engines such as ODE – where joints constrain motion instead of enabling motion. MuJoCo has four primitive joint types: `slide`, `hinge`, `ball` which is represented with a quaternion, and `free` joint which is a 3D translation followed by a 3D quaternion rotation. The latter joint type is used to define a floating base. A unique feature of MuJoCo is that the primitive joint types can be composed into more complex joints, without having to define intermediate dummy bodies.

c) DOF: Degrees of freedom (DOFs) are closely related to joints, but are not in one-to-one correspondence because ball and free joints have multiple dofs. Since quaternions are 4D vectors (with unit length) while their velocities are 3D vectors, we often have more positional coordinates than velocity coordinates. DOFs have damping, maximum velocity, armature inertia. DOFs can also have friction which

is handled by the impulse solver, together with contacts, joint limits and tendon limits.

d) Geom: Geoms are massless geometric objects. Their primary use in the engine is collision detection as well as tendon wrapping. However they can also be used during model construction to specify the inertial properties of the body to which they belong. The supported geom types are `plane`, `sphere`, `capsule`, `ellipsoid`, `box`, `cone`, `mesh`. Collision detection uses dedicated pair-wise functions when possible, and otherwise defaults to a general-purpose convex collider (implemented by libccd). Non-convex meshes can be rendered but are not used in collision detection; instead the user should decompose them into convex meshes.

e) Site: Sites are points of interest (along with 3D frames) defined in the local frames of the bodies, and thus moving with the bodies. They are used in the engine to route tendons and apply certain types of forces, but can also be used by the user's program to encode sensor locations etc.

f) Constraint: Constraints can be used to create loop joints or impose any other kinematic equality constraint (i.e. a constraint that only depends on position). MuJoCo has several predefined types of constraints: 3D position constraint forcing two points on two bodies to coincide (effectively creating another ball joint), joint angle constraint allowing joint angles to be coupled through polynomial functions, tendon length constraint specifying that the length of a given tendon must remain constant, as well as arbitrary user-defined constraints implemented via callbacks.

g) Tendon: Tendons define spatial paths that can be used for actuation and also for imposing inequality or equality constraints; for example one can simulate a marionette or a tensegrity structure by specifying maximum tendon lengths. The tendon path is the shortest path that passes through a sequence of specified sites or wraps around specified geoms.

h) Actuator: Actuators have control inputs, optional activation states (used to model muscles and pneumatic cylinders) with suitable first-order dynamics, gains that can be fixed or depend on position and velocity (so as to model the force-length-velocity properties of muscles). They can transmit forces to the multi-joint mechanism by acting directly on the joints, pulling on tendons, or acting via slider-crank mechanisms converting linear to angular motion.

IV. Timing tests

A. Performance on smooth dynamics compared to SD/FAST

We measured the speed of multi-joint dynamics simulation in the absence of contacts or equality constraints. We constructed four identical models in MuJoCo and SD/FAST. This was done not only for speed comparisons, but also for debugging MuJoCo and making sure the results are numerically correct. There were small differences on the order of machine precision – which is because the different implementations of the algorithms have somewhat different round-off patterns.

The Isolated model consists of isolated rigid bodies connected to the world with various joints; here M is block-diagonal. The Chain model is a long chain of bodies, thus M is dense. The Hand and Humanoid have sparse M.

Table 1 shows the number of dynamics evaluations per second in a single thread. Results are averaged over 100 runs on four different Intel processors: X9650 (3 GHz), X5570 (2.93 GHz), i7 940 (2.93 GHz), X5860 (3.33 GHz). The columns "MuJoCo" and "SD/FAST" involve computing M and $\mathbf{b}$ as well as the forward kinematics and the Jacobians of all rigid bodies. The last column adds sparse factorization of M followed by back-substitution; this is only tested in MuJoCo because SD/FAST does not provide sparse factorization.

Model (DOF)	SD/FAST	MuJoCo	with $M\backslash\tau$
Isolated (34)	$122,000$	$151,000$	$133,000$
Chain (31)	$128,000$	$103,000$	$41,000$
Hand (32)	$89,000$	$101,000$	$76,000$
Humanoid (29)	$130,000$	$123,000$	$75,000$

Table 1: Number of smooth dynamics evaluations per second in a single thread, rounded to $1,000$.

The key observation here is that MuJoCo is quite comparable to SD/FAST. This was a pleasant surprise because, despite all our efforts to optimize the code, SD/FAST generates model-specific C code which we had expected to be faster. But apparently modern compilers (Visual Studio 2010 in this case) manage to blur the distinction. Note that as expected, the Hand and Humanoid models with sparse M are faster than the Chain model which has dense M, yet are slower than the Isolated model with block-diagonal M.

B. Overall performance in trajectory optimization

We also tested the overall performance of MuJoCo in the context of trajectory optimization through inverse dynamics [3]. The dynamical system is a 3D humanoid with 18 DOFs, shown in Fig. 3C. The trajectory has 30 time steps and limit-cycle topology. There are 3 contact points on each foot. Table 2 shows the time (in seconds) needed to evaluate the cost of the trajectory along with the cost gradient and Hessian. The derivative computations are based on central finite differences. We show results for two types of finite-difference clouds: quadratic and linear. The latter has $2n$ rather than n^2 states. These tests were run on a PC with two 6-core Intel X5860 processors (3.33 GHz), with Hyper-threading enabled. This yields 24 virtual cores, so we compare 1 vs. 24 threads.

Threads	States	6 contacts	4 contacts	0 contacts
1	42930	1.096	0.742	0.245
1	3150	0.071	0.046	0.014
24	42930	0.111	0.075	0.027
24	3150	0.008	0.006	0.003

Table 2: CPU time (in seconds) for one evaluation of the trajectory cost, gradient and Hessian. The number of states where the dynamics are evaluated corresponds to different Hessian approximations.

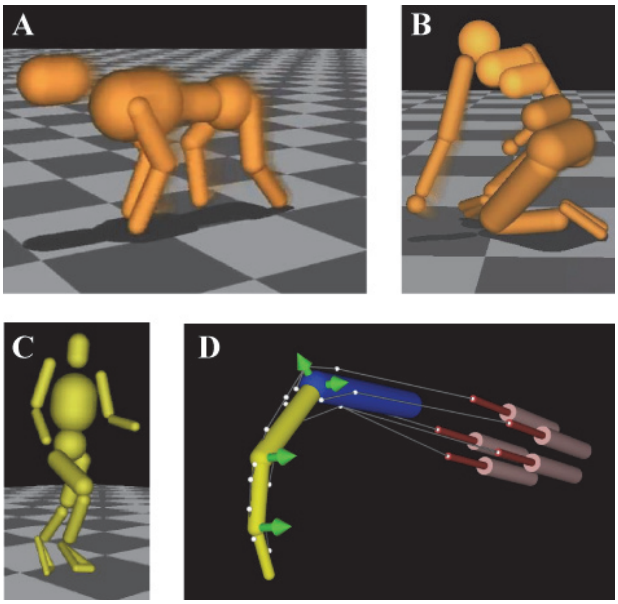


Fig. 3. Illustration of projects where we have used MuJoCo for control synthesis and modeling. A,B are from [9]. C is from [3]. D is from [13].

Threads	6 contacts	4 contacts	0 contacts
1	42,000	63,000	200,000
24	390,000	549,000	1,320,000

Table 3: This is the inverse of Table 2. Here we show the number of dynamics evaluations per second.

The results are quite remarkable. One a single desktop machine, we are able to run nearly $400,000$ evaluations per second including contact dynamics. Using the Gauss-Newton approximation to the Hessian (i.e. linear could), this corresponds to nearly 100 quasi-Newton steps of trajectory optimization per second. With fewer active contacts the speed is much higher. Note that these results do not take into account the time needed to implement the actual minimization method, however since the Hessian is very sparse, this time is a small fraction of the dynamics evaluations. Note also that here we used an interior-point method for solving the convex optimization problem (6). Our new projected methods appear to be faster, although we have not yet done careful testing.

V. Summary and future work

We described our new physics engine designed for model-based control, including algorithms, modeling conventions and timing results. The engine has already been used successfully in a number of projects, as illustrated in Fig. 3.

In terms of smooth multi-joint dynamics, single-threaded MuJoCo is comparable to SD/FAST. The contact-related tests we reported are very encouraging. We have not yet performed systematic comparisons to gaming engines, however the existing literature indicates that gaming engines are substantially slower when used in the same context.

The code is thread-safe and is already multi-threaded. The next step is to implement a cluster version, where a central dispatcher will send subsets of states to individual machines and assemble the results. The cluster version should yield another order of magnitude speedup. In addition we plan to port the key pieces of code to OpenCL and run it on GPUs. This has actually been a design goal from day one, and to this end we have written all run-time code in C and avoided external libraries.

With regard to simulation accuracy, we compared the smooth portion of the dynamics to the output of SD/FAST and found the differences to be within the margin of round-off errors. The accuracy of the contact dynamics has not yet been validated, however our implicit complementarity method has a built-in accuracy check – when the residual is 0 we know that the problem has been solved exactly. The convex solver should have similar accuracy to the one studied in [1], which produced results indistinguishable from LCP solvers but much faster. Of course our implementation is different so this needs to be tested.

MuJoCo was developed to enable our research in model-based control. The experience so far indicates that it is a very useful and widely applicable tool, that can accelerate progress in robotic control. Thus we have decided to make it publicly available. It will be free for non-profit research. GPU-enabled and cluster versions, as well as additional functionality regarding numerical optimization will be released subsequently. The website will be `www.mujoco.org`

Acknowledgments

This work was supported in part by the US National Science Foundation and the US National Institutes of Health.

References

[1] E. Drumwright and Shell. Modeling contact friction and joint friction in dynamic robotic simulation using the principle of maximum dissipation. *International Workshop on the Algorithmic Foundations of Robotics*, 2010.

[2] E. Drumwright and D. Shell. A robust and tractable contact model for dynamic robotics simulation. *Proc. of ACM Symposium on Applied Computing*, 2009.

[3] T. Erez and E. Todorov. Trajectory optimization for domains with contacts using inverse dynamics. *IROS*, 2012.

[4] R. Featherstone. *Rigid Body Dynamics Algorithms*. Springer, 2008.

[5] M. de Lasa J. Wang and A. Hertzmann. Optimizing walking controllers for uncertain inputs and environments. *ACM Transactions on Graphics*, 2010.

[6] D. Kaufman, S. Sueda, D. James, and D. Pai. Staggered projections for frictional contact in multibody systems. *ACM Transactions on Graphics (SIGGRAPH AISA)*, 164:1–11, 2008.

[7] K. Sims. Evolving virtual creatures. SIGGRAPH, 1994.

[8] D. Stewart and J. Trinkle. An implicit time-stepping scheme for rigid-body dynamics with inelastic collisions and coulomb friction. *International Journal Numerical Methods Engineering*, 39:2673–2691, 1996.

[9] Y. Tassa, T. Erez, and E. Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. *IROS*, 2012.

[10] Y. Tassa and E. Todorov. Stochastic complementarity for local control of discontinuous dynamics. *Robotics: Science and Systems*, 2010.

[11] E. Todorov. Implicit nonlinear complementarity: A new approach to contact dynamics. *ICRA*, 2010.

[12] E. Todorov. A convex, smooth and invertible contact model for trajectory optimization. *ICRA*, 2011.

[13] J. Xu, V. Kumar, Y. Matsuoka, and E. Todorov. Design of an anthropomorphic robotic finger with biomimetic artificial joints. *IEEE BioRob*, 2012.