Simulation Tools for Model-Based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX

Tom Erez, Yuval Tassa and Emanuel Todorov.

Abstract— **There is growing need for software tools that can accurately simulate the complex dynamics of modern robots. While a number of candidates exist, the field is fragmented. It is difficult to select the best tool for a given project, or to predict how much effort will be needed and what the ultimate simulation performance will be. Here we introduce new quantitative measures of simulation performance, focusing on the numerical challenges that are typical for robotics as opposed to multi-body dynamics and gaming. We then present extensive simulation results, obtained within a new software framework for instantiating the same model in multiple engines and running side-by-side comparisons. Overall we find that each engine performs best on the type of system it was designed and optimized for: MuJoCo wins the robotics-related tests, while the gaming engines win the gaming-related tests without a clear leader among them. The simulations are illustrated in the accompanying movie.**

I. Introduction

As robots become more complex and dynamically-capable, scripted movements and hand-tuned servo controllers are no longer sufficient to achieve high levels of performance, especially in uncertain environments. Instead there is growing need for model-based approaches in multiple areas of robotics. These include testing and validation of candidate control strategies before executing them on the robot; dynamically-consistent state estimation and model-predictive control which rely on faster-than-realtime internal simulation; system identification which is by definition model-based; and emerging machine learning techniques that need large datasets of state transitions which are difficult to obtain from the physical system.

Despite this need for fast and accurate simulation and the availability of raw computing power to make it possible, the existing simulation tools remain a limiting factor. Early work in robot dynamics gave rise to a range of efficient recursive algorithms [1], implemented perhaps most notably in SD/FAST [2] as well as the MATLAB Robotics Toolbox [3]. That phase focused on smooth multi-joint dynamics, largely leaving contact dynamics for future work. Yet contacts are the primary means by which robots interact with their environment. Accurate contact modeling and simulation remains an active area of research. Spring-damper models of contact dynamics were replaced by impulse-based velocity-stepping methods [4], [5], [6], [7], different flavors of which lie at the core of most physics engines used today. However many of these engines are aimed at graphics and animation, where it is often sufficient to achieve visually-plausible simulation, reducing the motivation to pursue the more elusive goal of physically-accurate simulation. Simulating contact dynamics with a velocity-stepping method is in itself problematic because it calls for solving NP-hard problems at each simulation step. Consequently much recent effort in this area has focused on developing convex approximations that yield similar contact behavior while being more tractable computationally [8], [9], [10], [11], further complicating the question of physical accuracy.

The notion of a physics engine in multi-body dynamics and gaming dates back to MathEngine, and indeed many of the engines used today can trace their origins back to it, in terms of methodology if not code. These engines use a Cartesian representation, maintaining 6 degrees of freedom (DOF) for each body, and imposing joints as equality constraints in $6N$-dimensional space where N is the number of bodies. This approach is natural for systems with few or no joints constraints. However a robot with N links has configuration space whose dimensionality is much closer to N that $6N$. Thus the preferred approach in robotics is to work in generalized/joint coordinates – which is both more efficient computationally, and more accurate because joint constraints cannot be violated. This is one reason why SD/FAST, which is actually older than MathEngine and its descendants, remains popular in robotics. It does not handle contact dynamics, and so a number of research groups have developed their own contact solvers around the smooth multi-joint simulation provided by SD/FAST.

The inadequacy of the Cartesian approach to heavily-constrained systems such as robots is now well recognized. This is currently prompting a new wave of simulators that aim to combine the best of the earlier approaches: efficient recursive algorithms in joint space, and modern velocity-stepping methods for contact dynamics. This category includes MuJoCo [12] and DART [13] (formerly RTQL8), as well as additions to PhysX [14], Bullet [15] and Havok [16] that utilize some form of joint-space representations. ODE [17] is the only engine in our comparison that does not yet have such functionality.

Are any of these engines suitable for wide adoption in robotics, not just for kinematics and visualization, but also for dynamics? A recent survey [18] found this field to be rather fragmented: while high-level visualization and modeling packages such as Gazebo and V-Rep are reasonably popular, none of the physics engines surveyed stood out. The most extensive usability test in this regard was the

University of Washington, Departments of Applied Mathematics and Computer Science & Engineering. {etom,tassa}@google.com, todorov@cs.washington.edu

This work was supported by the NSF and DARPA. Thanks to Erwin Coumans, John Hsu and Kenneth Bodin for suggestions and criticism.

DARPA Virtual Robotics Challenge (VRC), where a real-time simulation of the Atlas robot was implemented in ODE and accessed by the research teams through Gazebo. While the VRC was eventually a success, it is notable that for at least 6 months out of the 9-month program the simulation was borderline usable. Grasping with a high-DOF hand remained problematic even longer. And this was done by an experienced and well-funded development team, using one of the more mature physics engines available, with extensive help from a number of robotics experts participating in the VRC. Hopefully the experience gained in the VRC will transfer to other robots and tasks. Still, one has to wonder, how much effort is required for an individual research group to develop a realistic dynamic simulation of their robot.

The main goal of this paper is to compare multiple physics engines on identical model systems and characterize their speed, accuracy and stability. We hope that our tests will help roboticists assess the performance they are likely to achieve if they were to invest time in developing a detailed simulation, and to choose the engine that best suits their needs. Such comparisons are very much needed, which was also one of the conclusions of another comparison study [18]. Earlier work along these lines [19] is becoming outdated given the rapid development of physics simulation software. More recent engine comparisons [20], [21] were done from the viewpoint of multi-body dynamics, and the tests there are mostly complementary to ours. These papers also describe physics abstraction frameworks whose goals are similar to our framework described in Section II. The Open Source Robotics Foundation (OSRF) is currently working on a comparison of ODE, Bullet and DART – which are the engines integrated in Gazebo [22]. Accuracy tests of the Vortex engine (but not other engines) are described in [23]. Even though all these tests are related, what distinguishes the present work is the emphasis on more challenging model systems that can stress-test the engines. In contrast, most previous comparisons focused on simpler systems where analytical benchmarks are available. The rationale for our benchmarks is presented in Section III.

It should be noted that we are the developers of the MuJoCo engine which is included in the comparisons, and indeed a secondary goal of this paper is to identify the strengths and weaknesses of MuJoCo. We have made every effort to be objective, and will share our engine comparison code so that others can validate and extend our results (contact the first author if interested.) Such comparisons have the potential to be contentious, especially since all developers have invested a lot of effort in their software. This is why we have avoided subjective discussions such as ease-of-use and feature completeness, and have instead focused on reproducible quantitative measurements. A related question is, why not propose benchmarks which the engine developers can implement in their software, perhaps as part of a joint effort? This would be very useful if it happened, but the most likely outcome is for such a proposal to be ignored. In contrast, a benchmark proposal accompanied by extensive simulation results is likely to have more impact.

II. Software Framework

In order to compare different physics engines, we created a basic interface that has three functions:

- Create a simulation instance in a specified state.
- Advance the simulation one time step forward, given an input of joint torques.
- Output the current state in a uniform format.

We implemented this interface using the API of the different engines; see Appendix for the engine-specific implementation details. We record the position (translation and rotation) and velocity (linear and angular) of every rigid body. The data is then used to compute features such as momentum, energy, joint dislocation, joint limit violation and contact penetration. The data is also used for visualization.

In order to allow models to be edited once and used in all engines, we chose to specify the models in XML. Since most of the engines we consider do not include a built-in parser for any model definition markup languages, we took a different route: we parse the model description in one engine, extract the resulting body-joint configuration, and use this information to create the simulation world in every other engine, using the respective APIs to place bodies in their global position, attach them with properly-oriented constraints, and connect collision elements to the different bodies. We chose MuJoCo and its native MJCF file format to provide this service since it has a simple and compact data representation format. But a different format such as URDF could have also been used to obtain identical results.

In order to create a uniform playing field for the comparison, we identified a limited set of common features that are supported by all engines and are relevant for a robotics application. In particular, we restricted our models in the following ways:

Joints: Only hinge joints are allowed between parent and child. This excludes other common joints such as slide, ball, and universal joints. For engines that use joint coordinates, we also allow a free joint that represents a floating body (which is the default in Cartesian coordinates).

Collision: Only sphere and capsule geometries are supported for collision, as well as a fixed ground plane. This excludes other convex shapes (such as boxes) and mesh collisions, which are a complex topic that warrants its own comparison framework.

Contact: We restrict contact dynamics to the basic case of Coulomb friction, excluding features like restitution and rolling friction.

Multibody Dynamics: The only forces that act on the bodies are gravity, joint constraint forces, actuation torques applied at the joints, and joint stiffness and damping. We exclude (as much as the APIs allow; see Appendix) features like body-level damping (which is unrealistic) and joint friction (which is not supported by all engines).

In the name of uniformity we are not testing every engine-specific feature, even if some of the omitted features may offer an advantage for robotics applications.

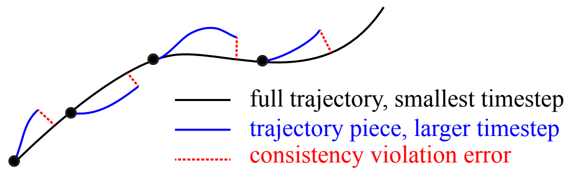
full trajectory, smallest timestep
trajectory piece, larger timestep
consistency violation error

Fig. 1. Illustration of how we measure consistency violation errors. The red dotted lines represent deviation, in our case in the body positions relative to the reference trajectory.

III. Tests and Metrics

We seek to develop universal tests and performance metrics that can be applied to any model system. This is in contrast with benchmarks such as single rigid-body motion or energy and momentum conservation which rely on analytical solutions. Although we include the latter when applicable, typical robotic systems do not preserve energy, momentum, symplectic forms or any other known quantity. Furthermore it is unclear if the ability to do well on simple systems generalizes to more complex systems, where the primary challenges are multiple interacting contacts, poor conditioning of large matrices and other scaling-related phenomena.

One candidate for such a universal test is simulation stability in the sense of not blowing up (which is obvious when it happens). Unfortunately gaming engines are known for sacrificing physical accuracy to gain stability – by introducing artificial damping, or even more drastic measures such as ignoring Coriolis forces as in Havok and PhysX. On the other hand any sensible measure of accuracy will detect instability, thus we will focus on accuracy here.

We adopt a measure of physical accuracy that evaluates *self-consistency*. A simulation can deviate from reality for two types of reasons: numerical integration errors, and model errors. Addressing the latter type of error involves system identification – which is generally a harder problem than simulation, and is outside the scope of the present comparison. Numerical integration errors on the other hand can be quantified in a universal way, which is what our measure does. The idea is simple. Numerical integrators become exact in the limit $h \rightarrow 0$ where h is the timestep. Therefore we can integrate the system with very small h, in our case 1/64 msec, and obtain an engine-specific reference trajectory. Then we can increase h and check how much the results deviate from this reference. This has to be done carefully because once the state deviates from the reference, further deviations may actually be physically correct (especially if the system is near-chaotic). So we should measure deviations over small integration intervals. This procedure is illustrated in Figure 1; consistency violation is the average of the red dotted lines.

A related consideration is the speed-accuracy trade-off inherent in every physics engine. A numerical integrator of order n has global truncation error proportional to h^n. Velocity-stepping schemes for contact simulation, implemented in all the engines we are comparing, rely on Euler integration which has fixed order $n = 1$. Nevertheless we can improve accuracy by reducing h; in the case of MuJoCo we can also increase n which is done by sub-dividing the timestep in Runge-Kutta fashion. In addition some engines have iterative contact solvers with a user-adjustable number of iterations. All these parameters have the same general effect: improved accuracy at the expense of simulation speed. The use of RK in conjunction with velocity-stepping is not trivial, since velocity-stepping does not define an ordinary differential equation. However one can measure the change in velocity over an integration step, treat this change as a proxy for the acceleration, and apply the RK equations as usual. This is a general approach, and in theory it could be applied to any of the other engines considered here, but in practice it requires full control of the system state and no hidden variables, which may be difficult to implement.

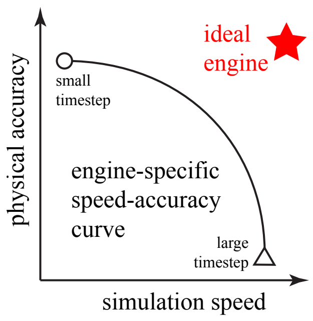

Fig. 2. Format of speed-accuracy tradeoff plots below.

Different parameter settings correspond to different points along the speed-accuracy curve (Figure 2). This is an example of a Pareto front in multi-objective optimization: we cannot maximize both of our conflicting criteria simultaneously. Therefore in order to evaluate an engine we need to characterize its entire speed-accuracy curve, and in particular how close it gets to the ideal-yet-unachievable top-right corner. Most of our results are presented in this format (figure 3), where the X-axis represents simulation speed expressed as a realtime factor, and the Y-axis represents different measures of accuracy – including the consistency measure defined above, as well as energy and momentum conservation when applicable. In addition to speed-accuracy plots, we provide raw CPU timing results independent of accuracy (figure 3).

Finally, one might ask how well these engines perform in the context of specific robotic tasks. Unfortunately the answer depends not only on the engine, but also on the interaction between the engine and the specific controller used to accomplish the task. Some controllers such as the SIMBICON [24] walking controller are so robust that the choice of engine makes little difference [25]. Indeed the latter study is an engine comparison that did not find significant differences (between ODE, Bullet, Havok and Vortex). But in general such controller robustness cannot be expected. Here we prefer to avoid tests that depend on elaborate controllers. Instead we selected a task which is fundamental to robotics and at the same time can be achieved with a naive controller. This task is grasping an object rigidly and moving it around. Our grasping controller is simply a PD controller with constant reference positions for the finger joints.

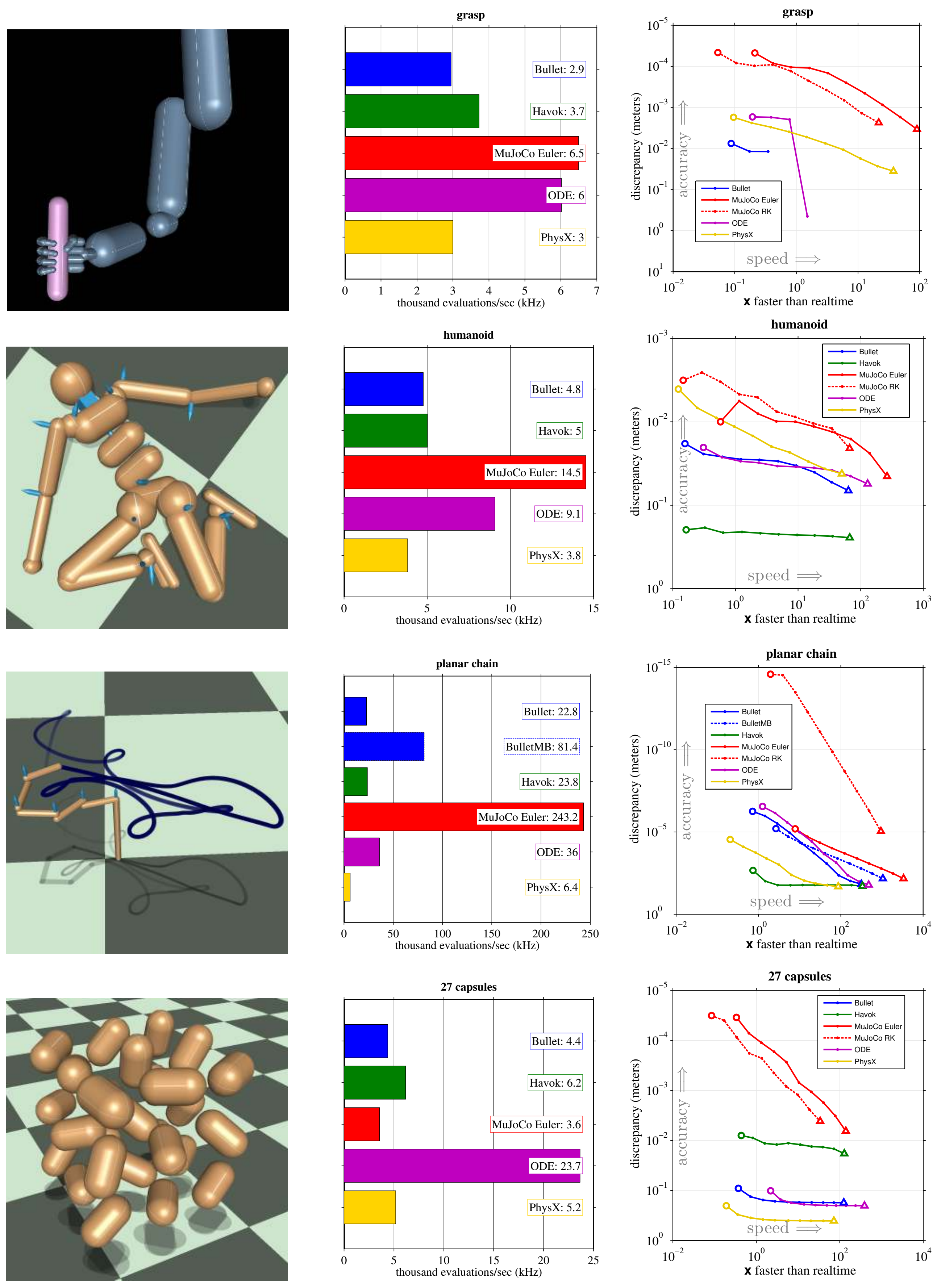

Fig. 3. Each row corresponds to a different test system. The first column illustrates the dynamical system. The second column shows raw speed as thousands of evaluations per second for each engine. The third column shows the speed-accuracy trade-off in terms of our consistency measure.

After grasping is achieved, PD controllers in the arm as well as gravity make the arm swing around, while the hand hopefully maintains the grasp. Another reason we focus on grasping is because even though it is not difficult with a physical system, in the VRC it proved difficult to accomplish in simulation. The quantitative measure of performance is straightforward: for each engine we find the largest timestep h for which the object remains in the hand during the entire simulation.

In summary, we will characterize raw CPU speed and the above speed-accuracy trade-off in several simulated systems, energy and momentum conservation when applicable, and grasp stability in the hand-object system.

IV. Simulations and Results

A. Model systems

The first column of Figure 3 illustrates the four model systems used in our main comparison:

Grasping: A 35-DOF robotic arm, modeled after the Shadow Hand robot, grasping a capsule using fixed spring-dampers. Set-points for the spring correspond to a clenched-fist configuration. Spring-damper parameters are respectively K = 0.4 Nm/rad and B = 0.005 Nms/rad. This system has large mass ratios and involves many simultaneous contacts, making dynamic simulation difficult. At the same time it is the kind of system many roboticists would like to model.

Humanoid: A 25-DOF humanoid model, which falls on the floor and wiggles due to sinusoidal open-loop torques applied to its joints. This (and the remaining) model systems are not performing a specific task, but nevertheless are useful for speed and accuracy measurements.

Planar Chain: A 5-DOF planar kinematic chain is composed of five bodies and five frictionless hinge joints (represented by the light blue arrows in the Figure). The bodies are shifted in the vertical plane to avoid collisions. This system preserves kinetic energy and angular momentum. It is initialized with non-zero joint velocities, after which the simulation unfolds under the passive dynamics. The resulting complex path of the tip is shown with the dark blue line.

27 Capsules: Randomly-oriented capsules are allowed to fall onto the floor. This system has $27 \cdot 6 = 162$ DOFs and is similar to the object-stacking demos often used in gaming engines. It is not directly relevant to robotics, but we include it to illustrate performance on the kind of system that gaming engines are optimized for. See also Section V.

B. Raw Timing

The second column of Figure 3 illustrates the raw speed of the engines. All tests were performed on an i7-3930K processor running Windows 8.1. We used high-resolution timers to time the main step function of each engine, excluding any overhead from our framework. The data were then converted into number of evaluations per second, shown in kHz.

We see that on systems relevant to robotics, MuJoCo is the fastest engine in the comparison, sometimes by a wide margin. In the gaming scenario however (i.e. the capsule test) ODE wins by a large margin, while MuJoCo is the slowest. The conclusion here is simple: each engine is good at simulating the type of system it was designed and optimized for. MuJoCo was designed for robotics while the other engines were designed for gaming. In particular, MuJoCo represents the system in joint coordinates and performs all computations and numerical integration in that representation, while the other engines use Cartesian coordinates.

Note BulletMB in the Planar Chain system. This is the recent articulated version of Bullet, which we could only get to work reliably on systems without contact such as the chain model. While it is 3 times slower than MuJoCo, it still outperforms all the engines that use Cartesian coordinates by a significant margin in this test.

C. Consistency

The third column of Figure 3 shows the consistency results, which we believe are the most informative regarding overall engine accuracy. The speed-accuracy curve for each engine is obtained by running the simulation at many different values of the timestep h, from 1/64 msec to 32 msec increasing by a factor of 2. For each value of h and for each engine we also measure the CPU time it takes to execute a single update (this was the raw timing data discussed above), and then compute the corresponding realtime factor. Consistency is measured as explained earlier, averaging over 10 trajectory pieces in each case.

MuJoCo outperforms all other engines by orders of magnitude (note the log scale), even on the capsule test where it is the slowest. The largest difference is seen for the planar chain, where the test is won not so much by the engine but by the RK integrator. On the other systems however the RK integrator does not seem to help.

The least consistent engines were PhysX and Havok – which is interesting because they are probably the most heavily optimized gaming engines in our comparison. This result confirms the common knowledge that gaming optimizations to do not target accuracy but rather stability.

Note the grasp model where ODE and Bullet have partial speed-accuracy curves. This is because for larger timesteps they go unstable on this system, and therefore we could not measure consistency in a meaningful way.

D. Grasp stability

The grasp model was the most challenging. We ran the simulation at different values of the timestep h and recorded the largest value for which the object was still in the hand at the end of the simulation. The results in milliseconds are:

engine	**max timestep (ms)**
Bullet	1/32
MuJoCo Euler	16
MuJoCo RK	16
ODE	1/4
PhysX	2

Recall that the timestep values we are testing are spaced logarithmically, so these measurements are only accurate up to a factor of 2, but still the picture is clear. MuJoCo

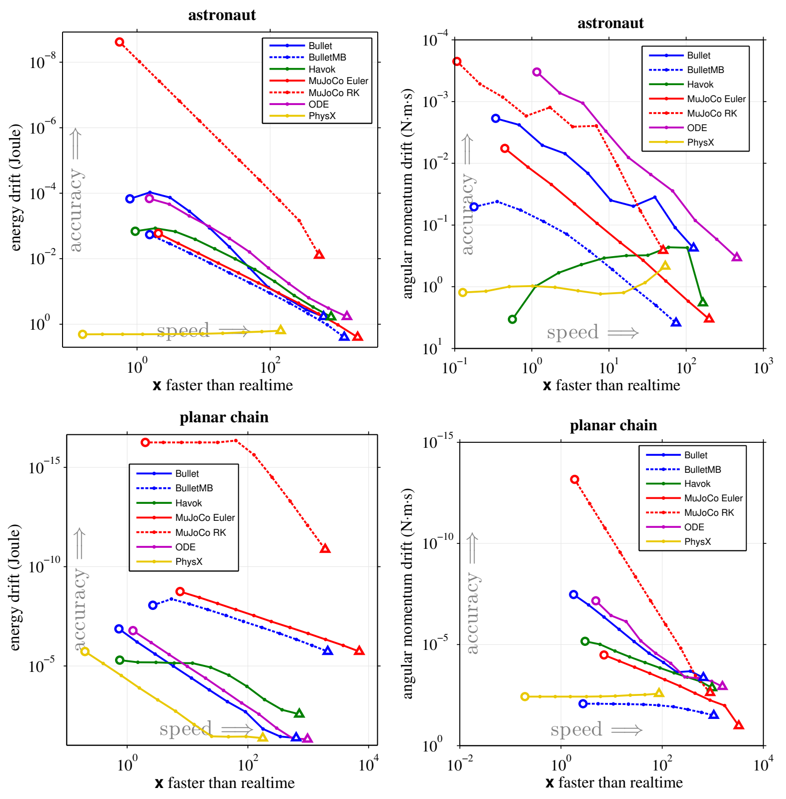

Fig. 4. Top row: conservation of kinetic energy (left), angular momentum (center) and linear momentum (right) for the floating humanoid, a.k.a 'astronaut'. In all cases ground contact and gravity are disabled. For the energy conservation case, joint limits, self-contacts and actuators are also disabled. Bottom row: energy and angular momentum conservation for the planar chain. Note that this system can apply forces (but not torques) to the world and so it does not preserve linear momentum.

produces qualitatively accurate simulation (in terms of not dropping the object) with large time steps. PhysX is a distant second, but still significantly better than the remaining engines. Note that all engines manage to complete the test at small timesteps – so their underlying physics model indeed predicts that the object should remain in the hand. But as the timestep increases they effectively simulate a different physics model which can no longer hold the object.

It is an open question whether this model system can be tuned to work better in the gaming engines. The experience with grasping in the VRC suggests that at least for ODE, it will be difficult to go above 1 msec even with the implicit damping implemented by OSRF (which is not yet part of the official ODE codebase and so is not used in our tests).

E. Energy and Momentum Conservation

By removing the ground plane and disabling gravity, the humanoid model (now called 'astronaut') preserves momentum. Further disabling joint limits, contacts and actuators (but initializing with non-zero joint velocities) allows for energy conservation. The planar chain preserves energy and angular momentum (but not linear momentum) so it can be tested without any modification.

Figure 4 shows the measured energy and momentum drift, again expressed as a speed-accuracy curve. In the energy conserving systems the RK integrator wins by orders of magnitude, as it should. The angular momentum picture is more mixed. In the astronaut test ODE performs very well despite using Euler integration. This is probably because it implements semi-implicit integration of Coriolis forces [26].

The linear momentum conservation test (top-right) shows a genuine advantage of simulation in Cartesian coordinates. Constant linear momentum means that the system state remains on a manifold which is a linear subspace in Cartesian coordinates, and so numerical integration has no reason to accumulate errors. Indeed, the Cartesian engines are actually more accurate with larger time steps. The same manifold is curved in joint coordinates. The RK integrator manages to deal with this curvature to some extent, but even it cannot compete with Cartesian engines on this test.

V. Large-scale simulation in joint space

Thus far we focused on models where the number of rigid bodies and DOFs is modest and the computational challenges come from complex kinematic structures that propagate interaction forces instantaneously, large mass ratios, multiple simultaneous contacts. Such systems are typical for present-day robotics. Our tests showed that simulation in joint coordinates (used in MuJoCo) has significant advantages over the Cartesian coordinates preferred in gaming engines. Yet there is a different class of systems where Cartesian coordinates are advantageous – namely systems that consist of a large number of floating bodies. The computational challenges there have to do with scaling, often demonstrated in simulations such as box-stacking that are not directly relevant to robotics. Nevertheless one can imagine future applications involving robots in crowds, many interacting robots, robots manipulating many movable objects – in which case scaling to larger systems becomes essential. These applications will require engines that can simultaneously meet both types of challenges, combining the advantages of joint and Cartesian coordinates.

There are already efforts under way to produce such engines. PhysX, Bullet and Havok now have the option to use joint coordinates, even though the PhysX implementation turned out not to be using real joint coordinates (see appendix), and the Bullet implementation is not yet ready (we have not tested the Havok joint-coordinate option). MuJoCo on the other hand has a sparse solver designed to be scalable. This feature set was not used in the systematic comparisons above, and we only demonstrate it in this section.

The model system here consists of $N = (25, 50, 75, ...250)$ capsules. It is simulated for 10 sec with a timestep of 10 msec, Euler integration. The initial and final configurations for the $N = 250$ system are shown in the top of Figure 5. The bottom of the figure compares the CPU time for MuJoCo's (still experimental) sparse solver and the dense solver used in the rest of the paper. The CPU time in the absence of contacts is also shown in both plots. We see that the sparse solver has linear scaling, and for the largest system it is 76 times faster than the dense solver – whose scaling is between quadratic and cubic. Preliminary tests show that MuJoCo's sparse solver in this setting is about 2 times slower than gaming engines, however it does have linear scaling.

The poor scaling of the dense solver is not surprising. Indeed a recent comparison [22] of ODE, Bullet and DART found that DART (which also uses joint coordinates) has the same problem. It arises from the fact that both the joint-space inertia M and the contact Jacobian J in this case are very sparse, and yet the inverse contact inertia $A = JM^{-1}J^T$ which is the Hessian of the underlying LCP or QP tends to be dense. So the key is to avoid working with A directly and instead use Hessian-free methods. This requires elaborate indexing machinery which is unnecessary in Cartesian coordinates – and so gaming engines have an advantage for such systems even if a sparse solver is used.

VI. Summary

We introduced a speed-accuracy measure of simulator performance, applicable to complex systems where analytical benchmarks are not available. We also characterized performance in terms of energy and momentum conservation when applicable, as well as grasp stability. None of the engines being compared was uniformly better than all others. MuJoCo was both the fastest and the most accurate on constrained systems relevant to robotics, and was capable of stable grasping at a much larger time step. On systems composed of many disconnected bodies it was the slowest in term of raw CPU speed (while ODE was the fastest), however it remained the most accurate overall. Semi-implicit integration of Coriolis forces (in ODE) significantly improved performance in the presence of rotating floating bodies. Runge-Kutta integration (in MuJoCo) outperformed semi-implicit Euler integration by orders of magnitude for smooth dynamics, but its advantages were lost in the presence of contact dynamics. Simulation in Cartesian coordinates yielded better linear momentum conservation compared to joint coordinates.

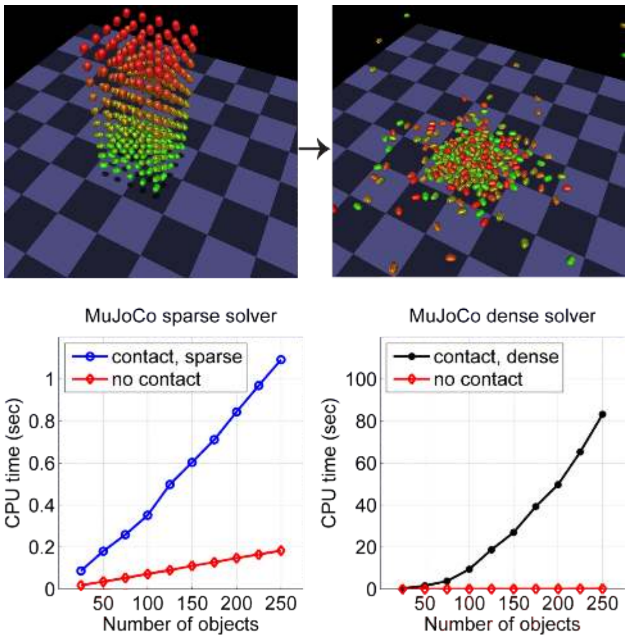

Fig. 5. Comparison of MuJoCo's sparse and dense contact solvers on many-body systems. Simulation time 10 sec; Euler integration with 10 msec timestep. The top plots show the initial and final configurations of the 250-capsule system. The bottom plots show the CPU time to complete each simulation. We also tested MuJoCo in the absence of contacts, where the results for the sparse and dense solvers are identical because the solver is not actually invoked. Note the different scales on the vertical axis.

Appendix: Implementation Details

1) ODE: ODE is an open-source physics engine. It is probably the engine most commonly used in robotics applications, most notably in the VRC. It is integrated with Gazebo and V-REP, as well as other robotics frameworks. ODE implements a sophisticated integrator for angular DOFs [26], a feature that contributes to its performance in the energy and angular momentum tests.

ODE has an iterative solver and an exact solver. As part of the VRC effort, OSRF has developed implicit damping for ODE (John Hsu, personal communication) but this has not yet been merged with the main version in the official repository, and therefore is not included in our comparisons. We were unable to run stable simulations with the exact solver, in contrast to the experience of others [27]. This may be because we are testing contact configurations where the underlying LCPs become harder to solve exactly. Therefore we only report results using the iterative solver. We used 50 iterations, which Stevens et al. [22] found to be necessary to achieve comparable accuracy to Bullet and DART.

We implemented our own PD controller in ODE, which was straightforward using `dJointGetHingeAngle` to measure hinge angle and `dJointGetHingeAngleRate` to measure hinge velocity.

2) Bullet: Bullet is another open-source physics engine that is also integrated with many of the popular robotics software platforms, including V-REP and Gazebo.

While Bullet has built-in functionality for spring-dampers at hinge joints, its damping functionality does not conform to the standard PD controller design pattern: it is impulse-based, and therefore uses abnormal units to specify damping (instead of using Nms/rad, the damper's parameter roughly corresponds to "what fraction of the velocity should be retained in the next timestep"). Impulse-based damping is likely to be more numerically stable and robust than explicit force-based damping; however, in order to maintain uniformity with the other engines, we implemented a PD controller using explicit damping torques. Since Bullet uses a Cartesian representation, equality constraint violations may lead to the rotation axes being different in two connected bodies. We measured the joint velocity by averaging the two rotation axes and comparing the angular velocities of the two connected bodies along the mean rotation axis. The joint angle was determined using `getHingeAngle`.

We implemented a comparison instance that uses the new Featherstone functionality in Bullet, but this functionality is not yet fully debugged and exhibits unexpected behavior in some simulations, so we were able to use this new functionality only in tests without contact.

3) PhysX: PhysX, together with Havok, is among the most widely used gaming engines. It also makes the most drastic compromises in terms of physical accuracy – in particular it ignores Coriolis forces [28]. This alone makes it unsuitable for robotics applications where accuracy is important, but we include it in our comparisons anyway.

While PhysX supports hinge joints, we used a constrained 6D joint instead for the benefit of using a built-in PD controller class `PxD6Drive`.

We also implemented an articulation-based instance, but we experienced several difficulties: first, articulation joints have 3 DOFs, so in order to create a hinge we had to impose artificially-tight swing limits. Furthermore, the articulation API does not offer a direct way to compute joint angles. We therefore do not include these results in our comparisons.

4) Havok: Havok also ignores Coriolis forces, and therefore is also unsuitable to applications where accuracy is important. Note however that a Havok engineer suggested a possible way to manually introduce such forces [29].

Havok does not support plane geometries, and therefore we implemented the ground plane using a big box, setting the collision margin (`setRadius`) to 0. The Havok API does not provide a way to query the angle of a hinge constraint; we tried to implement this feature in several ways, including following the example of ODE's codebase and advice from Havok engineers on the developers' forum [30], but none of these solutions worked well enough. Therefore, we currently have no working PD controller for Havok, and it is therefore excluded from the grasping test in the current version.

5) MuJoCo: MuJoCo is the engine we have developed and used extensively in our research over the past 5 years. MuJoCo has a built-in implementation of Hinge PD controllers that uses implicit damping. We present results for two versions of MuJoCo, using the semi-implicit Euler intrator vs. the 4th-order Runge-Kutta integrator.

References

[1] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, New York, 2008.

[2] M. Hollars, D. Rosenthal, and M. Sherman, "SD/FAST user's manual," *Symbolic Dynamics, Inc.*, 1994.

[3] P. Corke, "A robotics toolbox for MATLAB," *Robotics & Automation Magazine, IEEE*, vol. 3, no. 1, pp. 24–32, 1996.

[4] D. Baraff, "Fast contact force computation for nonpenetrating rigid bodies," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM, 1994, pp. 23–34.

[5] B. Mirtich and J. Canny, "Impulse-based simulation of rigid bodies," in *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM, 1995, pp. 181–ff.

[6] D. E. Stewart and J. C. Trinkle, "An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction," *International Journal for Numerical Methods in Engineering*, vol. 39, no. 15, pp. 2673–2691, 1996.

[7] M. Anitescu and F. A. Potra, "Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems," *Nonlinear Dynamics*, vol. 14, no. 3, pp. 231–247, Nov. 1997.

[8] M. Anitescu, "Optimization-based simulation of nonsmooth rigid multibody dynamics," *Math. Program. Ser A.*, vol. 105, 2006.

[9] D. Kaufman, S. Sueda, D. James, and D. Pai, "Staggered projections for frictional contact in multibody systems," *SIGGPAPH Asia*, 2008.

[10] E. Drumwright and D. A. Shell, "Modeling contact friction and joint friction in dynamic robotic simulation using the principle of maximum dissipation," in *Algorithmic Foundations of Robotics IX*, ser. Springer Tracts in Advanced Robotics, D. Hsu, V. Isler, J.-C. Latombe, and M. C. Lin, Eds. Springer Berlin Heidelberg, 2010, vol. 68.

[11] E. Todorov, "Analytically-invertible dynamics with contacts and constraints: Theory and implementation in MuJoCo," in *International Conference on Robotics and Automation (ICRA)*, 2014.

[12] "MuJoCo physics engine." [Online]. Available: www.mujoco.org

[13] "DART physics egnine." [Online]. Available: http://dartsim.github.io

[14] "PhysX physics engine." [Online]. Available: www.geforce.com/hardware/technology/physx

[15] "Bullet physics engine." [Online]. Available: www.bulletphysics.org

[16] "Havok physics engine." [Online]. Available: www.havok.com

[17] "Open dynamics engine." [Online]. Available: http://ode.org

[18] S. Ivaldi, V. Padois, and F. Nori, "Tools for dynamics simulation of robots: a survey based on user feedback," *arXiv:1402.7050 [cs]*, Feb. 2014, arXiv: 1402.7050.

[19] J. Lander and C. Hecker, "Product review of physics engines," *Gamasutra*, Sep. 2000.

[20] A. Boeing and T. Brunl, "Evaluation of real-time physics simulation systems." ACM Press, 2007, p. 281.

[21] A. Seugling and M. Rollin, "Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool," *Umea University Masters Thesis*, 2006.

[22] P. Stevens, "Physics accuracy testing for the gazebo simulator," 2014. [Online]. Available: http://nbviewer.ipython.org/urls/dl.dropboxusercontent.com/u/96660379/boxes.ipynb

[23] CM-LABS, "Vortex dynamics verification and validation, version 2.0," 2014.

[24] K. Yin, K. Loken, and M. van de Panne, "Simbicon: Simple biped locomotion control," *SIGGRAPH*, 2007.

[25] S. Giovanni and K. Yin, "Locotest: Deploying and evaluating physics-based locomotion on multiple simulation platforms," *Proc. Motion in Games*, 2011.

[26] C. Lacoursire, *Stabilizing gyroscopic forces in rigid multibody simulations*. Ume: Department of Computing Science, Ume University, 2006.

[27] P. Hämäläinen, S. Eriksson, E. Tanskanen, V. Kyrki, and J. Lehtinen, "Online motion synthesis using sequential monte carlo," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 51:1–51:12, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601218

[28] "Are euler's equations integrated for a pxrigiddynamic in pxscene::simulate(dt) ?" [Online]. Available: https://devtalk.nvidia.com/default/topic/525665/

[29] "support for gyroscopic (coriolis) forces?" [Online]. Available: https://software.intel.com/en-us/comment/1798511

[30] "How to get the angle of the hinge constraint." [Online]. Available: https://software.intel.com/en-us/forums/topic/293063