Annual Review of Control, Robotics, and Autonomous Systems

The Role of Physics-Based Simulators in Robotics

C. Karen Liu[1] and Dan Negrut[2]

[1]Department of Computer Science, Stanford University, Stanford, California 94305, USA; email: karenliu@cs.stanford.edu

[2]Department of Mechanical Engineering, University of Wisconsin–Madison, Madison, Wisconsin 53706, USA; email: negrut@wisc.edu

Annu. Rev. Control Robot. Auton. Syst. 2021. 4:35–58

First published as a Review in Advance on November 13, 2020

The *Annual Review of Control, Robotics, and Autonomous Systems* is online at control.annualreviews.org

https://doi.org/10.1146/annurev-control-072220-093055

Keywords

robot simulation, testing through simulation, virtual prototyping of robots

Abstract

Physics-based simulation provides an accelerated and safe avenue for developing, verifying, and testing robotic control algorithms and prototype designs. In the quest to leverage machine learning for developing AI-enabled robots, physics-based simulation can generate a wealth of labeled training data in a short amount of time. Physics-based simulation also creates an ideal proving ground for developing intelligent robots that can both learn from their mistakes and be verifiable. This article provides an overview of the use of simulation in robotics, emphasizing how robots (with sensing and actuation components), the environment they operate in, and the humans they interact with are simulated in practice. It concludes with an overview of existing tools for simulation in robotics and a short discussion of aspects that limit the role that simulation plays today in intelligent robot design.

1. INTRODUCTION

Computer simulation is increasingly relied upon to predict the outcome of real-world phenomena and to explore the response and performance of physical systems that are too complex to be investigated via analytical solutions. For instance, NASA simulated the "seven minutes of terror" millions of times to ensure that the Curiosity rover had a less than 1.7% risk of failure when it landed on Mars. The recent breakthrough in simulating the life cycle of *Mycoplasma genitalium*, the world's smallest free-living bacterium, demonstrated how computers can be used to expand our understanding of cellular function, presenting a stepping stone toward computer-aided design in bioengineering and disease treatment. Physics-based simulation has become mainstream in science and engineering, with adoption accelerated by the rapid growth in computational power over the last three decades. The highly successful Intel Pentium chip from 1993 had 3.1 million transistors; a GPU card in 2020 has more than 100 billion. Almost all these transistors are organized either as processing units or as cache memories, providing a staggering amount of compute power—trillions of arithmetic operations per second—at a low cost. Multidisciplinary in nature and computationally intensive in its numerical solution, simulation in robotics is well positioned to benefit from these hardware advances.

As argued in this article, simulation in robotics has deep roots in computer graphics and video gaming. Early computer graphics researchers saw the potential of physics simulation as a generic approach to creating realistic visual content. They envisioned simulation as a tool capable of transforming numbers and vectors produced by numerical processes into images, providing intuitive ways for artists to generate and tweak a wide range of phenomena in 3D worlds. Ideally, this happened fast enough to facilitate real-time or interactive applications, such as gaming and user-in-the-loop design tools. This concept of a visualizable, tunable, and fast simulation tool resonates well with researchers and practitioners in robotics. A visualizable simulator allows engineers to examine and understand how a robotic system might behave early in its life cycle, without the need to first build costly physical systems. A tunable simulator can be instrumental in choosing among competing solutions to identify trade-offs among task efficiency, safety, and end product costs, allowing the rapid design of components and control algorithms as well as the evaluation of safety and efficiency across a wide range of realistic and relevant scenarios. A sufficiently fast simulator can also shorten the development cycle and reduce costs. When an accurate or adaptive model is available, a fast simulator can enable online prediction and online planning, as employed in model predictive control (1). Generic physics engines with these properties have become increasingly instrumental in robot design and control (**Figure 1**).

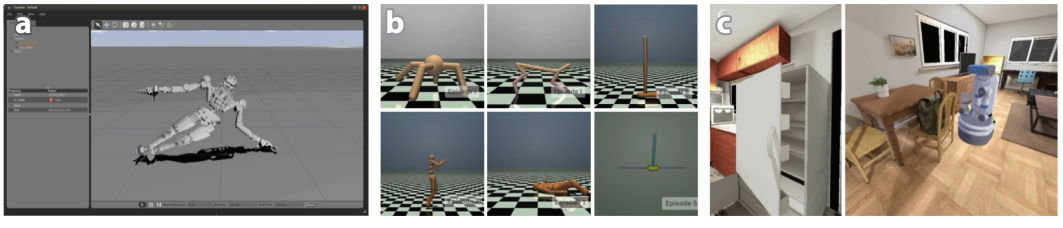

Figure 1

Examples of physics engines used in robot design and control. (*a*) The first DARPA Robotics Challenge was a software competition carried out in a virtual environment using physics-based simulation as the first proving ground for developing biped robot controllers. (*b*) OpenAI Gym (**https://gym.openai.com**) is a toolkit for developing reinforcement learning algorithms to control physically simulated agents. (*c*) iGibson (the Interactive Gibson Environment) (2) is a virtual environment reconstructed from real indoor scenes, providing visual rendering and physics simulation to train simulated robots for navigation and manipulation tasks.

The rise of deep learning has further augmented the importance of physics simulation in the era of AI. The last few years marked a manifest surge in interest for AI-enabled robots that operate in dynamic and unstructured environments and might interact with humans. These robots are expected in the future to drive us around on crowded streets, physically assist healthcare providers, teach young learners in schools, manage underwater oil spills, and execute rescue missions in adverse environments. AI is poised to endow a new generation of robots with mobility and decision-making skills. However, learning approaches demand a large amount of training data encapsulating physical interaction between the robot and its environment. Unlike other types of data, such as images and text, acquiring robot interaction data requires careful instrumentation in physical experiments, which can be challenging and risky to the robots and experimenters. For many physical tasks, the sheer quantity of training data needed for the learning algorithms is simply infeasible to acquire in the real world. In combination with the power of cloud computing, physics simulation can provide a virtual world in which robots generate training data at low cost, gain experience with a wider range of scenarios, and operate in a risk-free manner.

Computer simulation can be a versatile tool that is instrumental in developing next-generation robots that venture outside controlled environments and operate in complex real-world scenarios. Presently, this tool is not very effective, as many designs produced in simulation fail to deliver in the real world—the so-called sim-to-reality gap. We do not yet know the complex interplay of multiple physical regimes in the real world at the level we can simulate it; or, if we do, the physics-based simulation is too slow to be useful. Recent advances in machine learning and statistical techniques open the possibility of using a large amount of measured data to replace or augment physics models. However, such approaches are in their infancy, and principled methodologies to design and validate data-driven models remain to be identified. Simulating complex real-world environments is further challenged by the fragmented landscape of existing efforts in physics simulation. Many available physics engines bring unique aspects to the table, but they do not work together due to the lack of unified modeling abstraction and hierarchies. Finally, we do not know how accurate is accurate enough. Impractical demands for highly accurate models often make downstream control design tasks intractable, and we do not know for certain that high-fidelity physics simulation will lead to effective control when operating in the real world. Establishing a lower bound on physical fidelity for the simulation tools is challenging but critical for creating learning environments, producing training data, designing better controllers, improving mechanical performance, and auditing for safety purposes.

Against this backdrop, the article is organized as follows: Section 2 provides an overview of the concept of physics-based simulation. Section 3 focuses on what it takes to simulate robots. Section 4 discusses existing software and tools. Finally, Section 5 rounds off the article with a summary of several steps expected to augment the role that simulation plays in designing intelligent robots.

2. WHAT IS A PHYSICS-BASED SIMULATOR?

In the context of this article, simulation is the process of using a computer to approximate how a dynamic system (here a robot) evolves in time. To this end, if the computer relies on the laws of physics to predict the behavior of the dynamic system of interest, then the simulation is called physics based. In robotics, the laws of physics that come into play most often are mass and momentum conservation. The laws of optics might also come into play if sensor simulation is required, which turns out to be the case for autonomous vehicle (AV) simulation in particular. These laws bring along, or are framed in terms of, mathematical equations, e.g., $F = m \cdot a$. Formulating and then solving the collection of equations associated with the dynamic system of interest is what computers are extremely good at. Because of the sheer count (billions of equations for granular

terrains, for instance) or high complexity level, these equations require a specialized numerical method that produces an approximation of their solution. The software code that poses the equations and solves them is called a dynamics engine.

In practice, a physics-based simulator is more likely to be successful in predicting how a dynamic system will respond to external excitations and will do so over a broad spectrum of scenarios and applications. Simulators that cut corners in the equation formulation or numerical solution phases will predict the system dynamics poorly, or they might provide acceptable predictions but only over a narrow set of use scenarios.

2.1. The Momentum Equation

Robot simulation draws on the momentum balance equation, which leads to the following second-order ordinary differential equation (ODE) that governs the robot's dynamics (3, 4):

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} = \mathbf{f}(t, \mathbf{q}, \dot{\mathbf{q}}). \quad 1.$$

Equation 1 is a rehashing of $F = m \cdot a$ (i.e., mass times acceleration equals force). Since robots are made up of several bodies that move in 3D space, the mass m of a particle is replaced by the generalized mass $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{N\times N}$, which depends on N generalized coordinates $\mathbf{q} \equiv [\mathbf{q}(1), \ldots, \mathbf{q}(N)]^{\mathrm{T}}$. Knowing the value of $\mathbf{q}$ on a time grid t_0, $t_0 + \Delta t$, $t_0 + 2\Delta t, \ldots$ allows one to locate and orient each robot component or part over time. Finding how $\mathbf{q}$ changes in time under the agency of a generalized force $\mathbf{f}(t, \mathbf{q}, \dot{\mathbf{q}})$ is the job of the dynamics engine. Both external forces, such as contact force or gravity, and internal actuation are factored into the equations of motion via $\mathbf{f}$. The unknown quantities of the ODE that the dynamics engine needs to solve for are the accelerations $\ddot{\mathbf{q}}$, velocities $\dot{\mathbf{q}}$, and generalized coordinates $\mathbf{q}$.

How are these unknowns computed by the dynamics engine? We set off to solve Equation 1 using the symplectic Euler integration formula (5) since it is simple and captures the essence of the algorithm. Given a set of initial conditions—i.e., having a specification of where the robot is (via $\mathbf{q}_0$) and what its velocity is (via $\dot{\mathbf{q}}_0$)—the velocity of the robot after Δt seconds will be $\dot{\mathbf{q}}_1 = \dot{\mathbf{q}}_0 + \Delta t\ \ddot{\mathbf{q}}_0$, and its new position will be $\mathbf{q}_1 = \mathbf{q}_0 + \Delta t\ \dot{\mathbf{q}}_1$. Then, $2\Delta t$ seconds into the simulation, the robot's velocity will be $\dot{\mathbf{q}}_2 = \dot{\mathbf{q}}_1 + \Delta t\ \ddot{\mathbf{q}}_1$, and its position will be $\mathbf{q}_2 = \mathbf{q}_1 + \Delta t\ \dot{\mathbf{q}}_2$. Thus, recursively applying this formula, one propagates the dynamics of the robot forward in time, producing a numerical solution at a collection of stations $\Delta t, 2\Delta t, \ldots, n_{\text{steps}}\Delta t = T_{\text{final}}$. What is left is to compute $\ddot{\mathbf{q}}_0$ at the beginning of the simulation, then $\ddot{\mathbf{q}}_1$ at time Δt, $\ddot{\mathbf{q}}_2$ at time $2\Delta t$, and so on. Finding the accelerations is as simple as solving a linear system $\mathbf{Ax} = \mathbf{b}$, where, using Equation 1, $\mathbf{A} = \mathbf{M}(\mathbf{q}_0)$, $\mathbf{x} = \ddot{\mathbf{q}}_0$, and $\mathbf{b} = \mathbf{f}(t_0, \mathbf{q}_0, \dot{\mathbf{q}}_0)$. Since the generalized mass matrix is typically symmetric and positive definite, one can use, for instance, the Cholesky algorithm (6) to compute $\ddot{\mathbf{q}}_0$. Computing, Δt seconds into the simulation, the generalized acceleration $\ddot{\mathbf{q}}_1$ once $\dot{\mathbf{q}}_1$ and $\mathbf{q}_1$ are available follows the same pattern and requires the solution of another linear system $\mathbf{Ax} = \mathbf{b}$. Note that $\mathbf{A}$ and $\mathbf{b}$ change with each time step.

The simple form of the equations of motion in Equation 1 is encountered only for robots whose topology is a tree with no closed loops. For tree-like mechanisms, a child element is positioned relative to its parent element; this is the so-called recursive formulation (4), which uses a minimal (or reduced) set of generalized coordinates $\mathbf{q}$. When one poses the equations of motion in the recursive formulation, the ensuing ODE in Equation 1 is small; the solution will be fast, not noisy, and good for controls. However, recovering the reaction forces in joints requires nontrivial computations, and the recursive formulation is challenging to comprehend and not simple to implement and parallelize.

2.2. Dynamic Systems with Constraints

For more complex systems, such as AVs and rovers, the bodies that make up the robot contain closed loops, and a minimal set of generalized coordinates cannot be produced. It might also be the case that the set of generalized coordinates $\mathbf{q}$ selected is not minimal, e.g., when using Cartesian coordinates to formulate the equations of motion. In this case, the equations of motion read (7)

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} = \mathbf{f}(t, \mathbf{q}, \dot{\mathbf{q}}) - \nabla_{\mathbf{q}}^{\mathrm{T}}\mathbf{g}(\mathbf{q}, t)\,\boldsymbol{\lambda}, \qquad 2a.$$

$$\mathbf{g}(\mathbf{q}, t) = \mathbf{0}_{M\times 1}. \qquad 2b.$$

The salient point is that the generalized coordinates in $\mathbf{q}$ are no longer independent, an observation captured in Equation 2b in the form of M kinematic constraints that are posed as nonlinear equations in $\mathbf{q}$ and t. Each of the M kinematic constraints has one associated Lagrange multiplier, which is a measure, or proxy, for the internal (reaction) force required to enforce the satisfaction of the kinematic constraint. The individual multipliers are collected in $\boldsymbol{\lambda} \in \mathbb{R}^M$, which is used to compute the constraint reaction force $-\nabla_{\mathbf{q}}\mathbf{g}\,\boldsymbol{\lambda}$ in Equation 2a, where $\nabla_{\mathbf{q}}$ is the gradient (with respect to $\mathbf{q}$) operator. While Equation 2a dictates how $\mathbf{q}$ changes in time, Equation 2b qualifies these changes so that they are compatible with the constraints present in the system. This qualification happens via the constraint reaction force induced by the Lagrange multipliers $\boldsymbol{\lambda}$. From an applied mathematics point of view, the problem in Equation 2 takes the form of a set of differential algebraic equations (DAEs) (5), which are different and more difficult to solve than ODEs (8). Knowing $\mathbf{q}$ and $\dot{\mathbf{q}}$, one can immediately evaluate the generalized mass matrix $\mathbf{M}(\mathbf{q})$ and the generalized forces $\mathbf{f}(t, \mathbf{q}, \dot{\mathbf{q}})$, that is, the torques and forces acting on the robot (the case of friction and contact forces, which can violate this assumption, is discussed in Section 3.1.5). Likewise, $\mathbf{g}(\mathbf{q}, t)$ and the gradient $\nabla_{\mathbf{q}}\mathbf{g}$ can be evaluated once $\mathbf{q}$ and $\dot{\mathbf{q}}$ are known. In the end, the unknown quantities that the dynamics engine needs to solve for are the Lagrange multipliers $\boldsymbol{\lambda}$ as well as the accelerations $\ddot{\mathbf{q}}$, velocities $\dot{\mathbf{q}}$, and generalized coordinates $\mathbf{q}$. The presence of kinematic constraints in Equation 2 complicates the process of solving for these unknowns. There are two widely used approaches: the DAE-to-ODE strategy and the DAE head-on strategy.

2.2.1. Reducing differential algebraic equations to ordinary differential equations. This approach reduces the DAE problem to an ODE and then solves it using the symplectic Euler algorithm or another explicit or implicit numerical integration algorithm. For instance, Wehage & Haug (9) selected a subset of so-called independent generalized coordinates $\hat{\mathbf{q}}$ out of all the generalized coordinates stored in $\mathbf{q}$. The remaining ones, $\bar{\mathbf{q}}$, are called dependent generalized coordinates: $\mathbf{q} = \hat{\mathbf{q}} \cup \bar{\mathbf{q}}$. The solution strategy is as follows: At time step t_l, one computes $\ddot{\mathbf{q}}_l$ but uses a numerical integration method to compute $\dot{\hat{\mathbf{q}}}_l$ and $\hat{\mathbf{q}}_l$. Then, using Equation 2b, one solves for $\bar{\mathbf{q}}_l$, and using the condition $\dot{\mathbf{g}}(\mathbf{q}, t) = \mathbf{0}$ (the time derivative of Equation 2b), one solves for $\dot{\bar{\mathbf{q}}}_l$ given $\mathbf{q}_l$ and $\dot{\hat{\mathbf{q}}}_l$.

The coordinate partitioning approach described above is predicated on the availability of $\ddot{\mathbf{q}}_l$. To compute $\ddot{\mathbf{q}}_l$ along with the unknown Lagrange multipliers, one first takes a time derivative of the kinematic constraints in Equation 2b to obtain $\dot{\mathbf{g}}(\mathbf{q}_l, t_l) = \nabla_{\mathbf{q}}^{\mathrm{T}}\dot{\mathbf{q}}_l + \mathbf{g}_t = \mathbf{0}$; then, a second time derivative yields $\ddot{\mathbf{g}}(\mathbf{q}_l, t_l) = \nabla_{\mathbf{q}}^{\mathrm{T}}\ddot{\mathbf{q}}_l + \boldsymbol{\gamma}_l = \mathbf{0}$, where $\boldsymbol{\gamma}_l$ is defined as the sum of all terms that do not depend on $\ddot{\mathbf{q}}_l$ in the expression of $\ddot{\mathbf{g}}(\mathbf{q}_l, t_l)$ (7). The unknown acceleration and Lagrange multiplier at time t_l are computed as the solution of the following symmetric linear system (10):

$$\begin{bmatrix} \mathbf{M}(\mathbf{q}_l) & \nabla_{\mathbf{q}}(\mathbf{q}_l) \\ \nabla_{\mathbf{q}}^{\mathrm{T}}(\mathbf{q}_l) & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_l \\ \boldsymbol{\lambda}_l \end{bmatrix} = \begin{bmatrix} \mathbf{f}(t_l, \mathbf{q}_l, \dot{\mathbf{q}}_l) \\ -\boldsymbol{\gamma}_l \end{bmatrix}. \qquad 3.$$

Consider a mechanical system with $N = 7$ generalized coordinates (Equation 2a) constrained by $M = 4$ equations (Equation 2b). The number of independent generalized coordinates is $7 - 4 = 3$, which also happens to be the degree-of-freedom count for the system. Wehage & Haug (9) discussed how to choose these three independent generalized coordinates: Assume they are the second, fourth, and seventh components of $\mathbf{q}$, i.e., $\hat{\mathbf{q}} = \left[\mathbf{q}(2),\ \mathbf{q}(4),\ \mathbf{q}(7)\right]^{\mathrm{T}}$. Once the acceleration $\ddot{\mathbf{q}}_l$ is available (Equation 3), one integrates forward in time $\ddot{\hat{\mathbf{q}}}_{l+1}$ to get $\dot{\mathbf{q}}_{l+1}(2)$, $\dot{\mathbf{q}}_{l+1}(4)$, and $\dot{\mathbf{q}}_{l+1}(7)$, and then once again to get $\mathbf{q}_{l+1}(2)$, $\mathbf{q}_{l+1}(4)$, and $\mathbf{q}_{l+1}(7)$. Then, Equation 2b is used to recover $\mathbf{q}_{l+1}(1)$, $\mathbf{q}_{l+1}(3)$, $\mathbf{q}_{l+1}(5)$, and $\mathbf{q}_{l+1}(6)$; this calls for the solution of a set of nonlinear algebraic equations. What is left to compute the state of the system at t_{l+1} is $\dot{\mathbf{q}}_{l+1}(1)$, $\dot{\mathbf{q}}_{l+1}(3)$, $\dot{\mathbf{q}}_{l+1}(5)$, and $\dot{\mathbf{q}}_{l+1}(6)$, using a 4×4 system of linear equations associated with the condition $\dot{\mathbf{g}}(\mathbf{q}_{l+1}, t_{l+1}) = \mathbf{0}$.

Two observations are in order. First, one fact that slightly complicates the numerical solution is that during an analysis, the set of independent generalized coordinates can change. This partitioning $\mathbf{q} = \hat{\mathbf{q}} \cup \bar{\mathbf{q}}$ is continuously vetted so as to ensure that $\hat{\mathbf{q}}$ can parameterize the underlying manifold on which the system dynamics evolves (11; see also 9). Second, there are more sophisticated approaches that reduce the DAE problem to an ODE problem beyond simply choosing a subset $\hat{\mathbf{q}}$ of $\mathbf{q}$. They are similar in spirit, though: The evolution of the system is expressed in terms of a set of time-dependent parameters (think of them as hyper-generalized coordinates) equal in number to the degree-of-freedom count. This choice of the hyper-generalized coordinates is what differentiates these methods (see, e.g., 12, 13).

2.2.2. Differential algebraic equation head-on strategy. This approach emerged before the DAE-to-ODE strategy (14, 15) and draws on implicit integration formulas. For demonstration purposes, without any loss of generality, the discussion here uses backward Euler, the simplest implicit algorithm. Assuming for the moment that $\ddot{\mathbf{q}}_1$ is available, the velocity at t_1 is computed as

$$\dot{\mathbf{q}}_1 = \dot{\mathbf{q}}_0 + \Delta t\ \ddot{\mathbf{q}}_1. \tag{4a}$$

Similarly, the new position configuration will be $\mathbf{q}_1 = \mathbf{q}_0 + \Delta t\ \dot{\mathbf{q}}_1$, or, equivalently,

$$\mathbf{q}_1 = \mathbf{q}_0 + \Delta t\ \dot{\mathbf{q}}_0 + (\Delta t)^2\ \ddot{\mathbf{q}}_1. \tag{4b}$$

Posing the problem at hand at time t_1 yields

$$\mathbf{M}(\mathbf{q}_1)\ddot{\mathbf{q}}_1 + \nabla_{\mathbf{q}}\mathbf{g}(\mathbf{q}_1)\ \boldsymbol{\lambda}_1 = \mathbf{f}\,(t_1, \mathbf{q}_1, \dot{\mathbf{q}}_1), \tag{4c}$$

$$\mathbf{g}(\mathbf{q}_1, t_1) = \mathbf{0}. \tag{4d}$$

The important observation is that were $\ddot{\mathbf{q}}_1$ known, one could immediately compute $\mathbf{q}_1$ and $\dot{\mathbf{q}}_1$ given the state of the system at time t_0. As such, in Equation 4c, one should keep in mind that the generalized mass matrix depends on $\ddot{\mathbf{q}}_1$; $\mathbf{q}_1$ is only a proxy for $\ddot{\mathbf{q}}_1$, as seen in Equation 4b. By the same token, Equation 4c references $\dot{\mathbf{q}}_1$ in the expression of the applied force $\mathbf{f}$. In reality, $\mathbf{f}$ depends only on the accelerations $\ddot{\mathbf{q}}_1$ owing to Equations 4a and 4b. In the end, in Equations 4c and 4d, one has a well-posed nonlinear system with an equal number of equations and unknowns: for equations, N momentum balance equations and M bilateral constraints, and for unknowns, N generalized accelerations $\ddot{\mathbf{q}}_1$ and M Lagrange multipliers $\boldsymbol{\lambda}_1$ associated with the constraint forces. Upon solving this system of nonlinear algebraic equations, one is in a position to immediately evaluate the position and velocity at t_1. Applying this idea recursively, one advances the time evolution of the robot, one Δt at a time. (For more details on this approach, see, e.g., Reference 16; for a survey, see Reference 17; and for a reference contribution, see Reference 5.)

How do these solution methods compare against each other? For the example above with $\mathbf{q} \in \mathbb{R}^7$ and $M = 4$, the DAE-to-ODE strategy will solve a linear system in 11 unknowns (see Equation 3) to recover the accelerations and the Lagrange multipliers. Then it will solve one nonlinear system with 4 unknowns to recover the dependent generalized positions, and finally a linear system with 4 unknowns to recover the dependent velocities. The DAE head-on strategy will solve a nonlinear system with 11 unknowns ($\ddot{\mathbf{q}}_I$ and $\boldsymbol{\lambda}_I$). In general, the DAE head-on strategy is fast and friendly to parallel computing. However, the solutions are often noisy, particularly in accelerations. Moreover, changes in time step Δt during the simulation lead to jolts in the solution.

Regardless of which approach one selects, rigid-body robots can typically be simulated in real time—i.e., one second of robot dynamics can be simulated in less than one second of compute time. This was not the case one to two decades ago, but advances in CPU hardware (e.g., large caches, multicore architectures, instruction-level parallelism, and high clock frequencies) have made real-time robotics simulation a reality.

2.3. Accounting for Uncertainty

There is uncertainty in the value of many parameters associated with the physical robot—masses, moments of inertia, spring stiffness, actuator parameters, joint compliance, elastic material properties, and so on. This uncertainty carries over to the model used to simulate the robot. Several analysis types can be used to manage and account for uncertainty: (*a*) gauging the sensitivity of the simulation results with respect to the model parameters and their uncertainty, (*b*) model calibration, and (*c*) quantifying how uncertainty in parameters leads to uncertainty in robot behavior. These three tasks can be approached in a frequentist (e.g., 18) or Bayesian (e.g., 19) framework.

Sensitivity analysis (the first type listed above) can pursue various ends and be accomplished by many means (20). One type of sensitivity analysis relevant for the problem at hand is variance based, which relatively ranks the importance of a model parameter based on the expected reduction in model output statistical variance, should one know the value of that parameter with certainty. For instance, if gauging the sensitivity indicates that precisely knowing the mass of a robot part significantly reduces variance in results, one might spend extra effort (time and/or money) to have a sharper input value for this parameter.

In contrast to the mass of a component, some parameters cannot be measured directly to produce sharp values for them. Model calibration (the second type listed above) is invoked to supply values for such model parameters, which include the friction coefficient μ, Young's modulus E, and so on. In this case, sensors placed on the robot collect data, the ground truth. Then, the Bayesian inference machinery (see, e.g., 21) could be set in motion to produce distributions for μ, E, and so on. To that end, before seeing the ground truth data, we provide expert insights into how these model parameters are distributed, and based on a Markov chain Monte Carlo process, these prior distributions are modified in light of the collected data. The outcome is a set of posterior distributions that learned from the measured data and can subsequently be used to yield data-informed parameter choices.

Regardless of whether one performs sensitivity analysis and then precisely measures the mass of a quadruped robot component or does model calibration and produces data-informed model parameters, inevitably there is still uncertainty in the model. The third type of analysis, which is less expensive than sensitivity analysis, aims to understand how the uncertainty in the value of the model parameters is reflected in the response of the system. This analysis is Monte Carlo based (see, e.g., 18) and therefore calls for running many simulations. This leads to a key observation: For the simulation engine to be useful, it must be fast.

3. SIMULATION IN ROBOTICS: WHAT GETS SIMULATED?

Simulating a robot in action is a puzzle with several pieces. Simulating the actual, mechanical contraption represents the beginning. The environment the robot interacts with must also be simulated; for example, a robot can move over a patch of deformable terrain and sink into it to the point where it gets stuck. Being able to re-create in simulation what an actual robot perceives, through sensing, is critical since most often the robots are simulated to test control policies. Finally, for many reasons (safety, ethics, desire to run millions of scenarios, range of subjects, etc.), human–computer interaction requires simulating the human element. This section discusses these robot simulation components.

3.1. World Simulation

The majority of the scenarios of interest in robotics can be simulated if the dynamics engine can capture the physics of four types of elements: rigid bodies, compliant (deformable) bodies, fluids, and terrains or granular material.

3.1.1. Simulating rigid bodies. Beyond modeling the robot, rigid bodies are also used to generate the virtual world in which the robot operates. The simulator will solve at t_0, then at t_1, then at t_2, and so on, using Equation 2 to produce the position, velocity, and acceleration of each rigid body, be it a robot part or some component of the virtual world around it. Typically, the actual geometry or shape of a rigid body is not relevant—all that is needed is its mass and mass moment of inertia. The geometry does come into play in two cases: when the body collides with other bodies, and when one wants to visualize how the body moves as time passes. For collision detection, which is a hard computational geometry problem (22), a relatively coarse mesh often suffices for capturing a collision or contact between bodies. For visualization, one usually draws on a fine mesh that is used in conjunction with textures that provide realism to make the scene photo-realistic.

3.1.2. Simulating soft bodies. Most robots are represented in simulation as a collection of rigid bodies connected through articulations (joints). The rigid-body assumption is violated for slender robots; likewise, robots that involve soft components bring to the fore large strains that cannot be neglected in the solution. Two classes of solutions emerged over time. One is more expeditious but assumes small deformations (flutter) and strains about a reference rigid body that travels or rotates in the 3D space: flex body = rigid body + small deformation on top. The other unleashes the full power of the nonlinear finite element method, where nonlinearities stem from three sources: large deformations and displacements or rotations, boundary conditions (friction, contact, and impact), and nonlinear material (soft material or padded gripper fingers).

The first approach is commonly known in the multibody dynamics community as the floating frame of reference (FFR) formulation. For rigid bodies, the location of a point P on body i at time t would be the position of the center of mass of body i plus a relative offset that takes into account the particular position of P on the body: $\mathbf{r}_P(t) = \mathbf{r}_i(t) + \mathbf{r}_{i,P}(t)$. In the FFR formulation, the position is further qualified to take into account an elastic deformation, which becomes an unknown in the problem: $\mathbf{r}_P(t) = \mathbf{r}_i(t) + \mathbf{r}_{i,P}(t) + \mathbf{e}_{i,P}(t)$. The cornerstone of the approach is the observation that, given the small magnitude of these excursions away from the rigid body, one obtains a linear problem that governs the time evolution of $\mathbf{e}_{i,P}$. This linear problem is subsequently simplified using techniques such as model truncation, component mode synthesis, condensation, proper orthogonal decomposition, and so on (23). On the upside, the FFR approach is easy to implement, and because it adds only a modest number of unknowns to the underlying rigid-body problem, it

does not lead to long simulation times. On the downside, it works only for small deformations of the compliant body and obscures physical insights into the solution. For instance, although it is possible to handle friction and contact forces that act at random and unknown locations on the body, doing so is not straightforward, because one is dealing not with an actual body but with some mathematical set of coordinates used in the model reduction.

A second class of solution employs the classical, full-blown nonlinear finite element analysis (FEA) methodology (24, 25). Note that linear FEA does not suffice owing to the three sources of nonlinearity mentioned above. On the upside, the FEA approach is versatile and, unlike the FFR approach, is a general and robust solution. On the downside, the execution times are long and the software implementation is complex.

3.1.3. Simulating terrain. As robots venture outside, they operate on uneven terrain that might also be deformable. If the terrain is simply uneven but otherwise rigid, the simulation is relatively simple: One can consider the terrain to be just a very large rigid body on which a robot moves. As such, knowing how to handle friction, contact, and impact suffices. When the terrain is deformable, there are three solution paths, drawing on semiempirical approaches, continuum formulations, and discrete representations, respectively.

Semiempirical approaches are anchored by seminal work reported in References 26–28. Upon carrying out a bevameter test (a uniaxial pressure–sinkage test and symmetric plate shear test) in the field, one obtains a set of parameters that are subsequently used in a simple model of the vehicle–terrain interaction (29). This methodology quickly produces reasonable results for vehicles moving in a straight line (no cornering forces) on rigid tires. The Bekker–Wong approach to modeling vehicle–terrain interaction targets heavy military vehicles rather than light robots. As such, it has been adjusted over time to meet the specific needs of robotics (see, e.g., 30, 31). For granular material only, a recent approach that bypasses the Bekker–Wong methodology is anchored by the so-called resistive force theory (32, 33), which maintains the spirit of quickly producing reasonably accurate results.

Continuum approaches have been used for terrains made up of silt or clays or covered in snow. The solution is FEA based (34, 35) and uses visco-elasto-plastic models, which are versatile but laborious to establish and implement in code (unless one falls back on commercial solutions), require parameters that are relatively difficult to produce, and computationally tend to be extremely taxing. The use of these models in robotics thus far has been limited.

Finally, discrete approaches have been relied upon heavily in simulating robots operating on granular terrain (see, e.g., 36). The methodology is anchored by the so-called discrete element method (see 37), in which the bed of granular material is modeled using a collection of discrete elements whose dynamics is individually tracked as the robot moves over the granular bed. This method is laborious, and alleviating the computational burden led to solutions that typically use larger particles than are encountered in the physical test and softer particles that allow for larger simulation steps.

3.1.4. Simulating fluid dynamics. Should the robot move in or handle (pour or transport) water or some other incompressible fluid, the motion of the fluid is governed by the momentum balance (Navier–Stokes) and mass conservation equations (38):

$$\frac{\mathrm{D}(\rho \mathbf{u})}{\mathrm{D}t} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}_\mathrm{b}, \qquad 5a.$$

$$\nabla \cdot \mathbf{u} = \mathbf{0}. \qquad 5b.$$

The momentum balance equation dictates the change over time in the location of each infinitesimal volume of fluid. This change is qualified by a constraint: The conservation of mass, stated in Equation 5b, constrains the motion of the fluid so as to prevent it from bunching in one spot only to deplete somewhere else. In Equation 5a, ρ is the density of the fluid; the term on the left of the equation represents the change in momentum for an infinitesimal volume of fluid. The term on the right has forcing terms that are responsible for the momentum change: The first is the pressure gradient, the second is a viscous force that depends on the viscosity coefficient μ and velocity Laplacian, and the last is the external (or applied) force per unit volume. For the motion of a fluid, the velocity $\mathbf{u}$ and pressure p are not known and must be computed in the solution process; the other quantities are known. The most common computational fluid dynamics method to approximate the solution of the partial differential equation (PDE) shown in Equation 5 is the smoothed particle hydrodynamics (SPH) method (39) or a similar approach that is Lagrangian based, i.e., that follows the trajectory of a collection of fluid particles to infer the motion of the aggregate. Compared with Eulerian (grid) methods, Lagrangian methods are known for trading off accuracy for gains in speed and versatility. The weakly compressible SPH formulation (40) uses interpolation to approximate an unknown variable at location P using values of that variable as reported by the SPH particles in the neighborhood of P. These interpolation-based approximations are used to express the action of the ∇ and ∇^2 operators, effectively reducing the PDE to an ODE problem. The solution of the latter was discussed in conjunction with the solid-body dynamics.

3.1.5. Simulating contact and friction. Handling frictional contact in robotics is as important as it is difficult. From a high vantage point, there are two main approaches to compute the contact normal force F_n and the tangential friction force F_f. Computing F_n and F_f requires at each time step a collision detection process, which is costly when the geometries of the bodies in contact are nontrivial, e.g., when using meshes to specify complex shapes (22).

3.1.5.1. The penalty approach. The main actor in this approach is δ_n, the local and normal deformation at the interface between the two bodies, A and B, in mutual contact. In general, the normal force is computed as $F_n = k_n\delta_n^\alpha + k_c\dot{\delta}_n$, where k_n and k_c are material- and shape-dependent parameters, and $1 \leq \alpha \leq 2$ (41). For any penalty-based solution, the collision detection should be highly accurate since the normal stiffness k_n assumes large values. As an order of magnitude, for steel, $k_n \approx 1 \times 10^7$ N/m; this stiffness comes multiplied by a local deformation of the order 1×10^{-6} m. Implicitly, the collision detection should be accurate within less than one micron, which is a tall order.

Since in most robotics applications the bodies are treated as rigid to save computational time, the penalty approach leads to a paradox: How can rigid bodies produce a deformation δ_n? Here, δ_n is interpreted not as a deformation but rather as an overlap of the geometries of A and B. This hand-waving associated with the penalty approach also permeates the computation of F_f. In this case, a second deformation δ_t is associated with a fictitious stiff spring, acting in the tangent plane and with ends connected to the contact points on A and B. This fictitious spring saturates when its internal load reaches μF_n. The tangent plane changes in time as the location and pose of A and B change, and with this comes a change in δ_t. This means that for each contact, the algorithm carries along δ_t as a state and updates its value to compute $F_f = \min(k_t\delta_t, \mu F_n)$, where the stiffness k_t is an empirical model parameter, and μ is the friction coefficient that assumes one of two constant values depending on the slip regime (static or kinetic).

The penalty approach does not increase the size of the problem at hand: The number of equations in Equation 2 does not increase no matter how many contact events are in play. Additionally,

the approach is simple to comprehend and easy to implement, and it scales well (all system contacts can be computed at the same time, in parallel). However, the approach is finicky to work with (it requires a significant amount of tuning), places tight constraints on the step size (because of the stiff fictitious springs), and requires values for many parameters (k_n, k_t, α, etc.).

3.1.5.2. The complementarity approach. In this approach, a signed distance Φ between bodies A and B is monitored, and the condition stating that the bodies should not penetrate is posed as $\Phi \geq 0$ (42). This is a kinematic constraint, intrinsically tied to the geometry of the bodies in contact. When $\Phi = 0$, there is a kinematic constraint present in the problem (similar in spirit to the ones in Equation 2b), and along with it there is a Lagrange multiplier $\lambda_{AB} \geq 0$. The positive value of λ_{AB} captures the fact that a contact produces a force that prevents bodies from penetration and should not seek to keep them together. Note that $\Phi \geq 0$ and $\lambda_{AB} \geq 0$, yet their product is always zero. This is because a normal force λ_{AB} only crops up if $\Phi = 0$ (bodies A and B touch each other); this represents a complementarity condition, $0 \leq \Phi \perp \lambda_{AB} \geq 0$, hence the name of the approach.

The computation of the friction force relies on a complementarity condition as well: $0 \leq v_t \perp (\mu F_n - F_f) \geq 0$, where v_t is the slip speed between A and B. This condition and the observation that whenever $0 < v_t$, the friction force F_f acts in the opposite direction of the relative slip, are the ingredients required by the numerical algorithm to compute F_n and F_t. In addition to not bringing δ_n or k_n into the discussion, the complementarity approach does not need δ_t or k_t. This is good—there are fewer variables to keep track of (δ_n and δ_t) and fewer model parameters (k_n and k_t) to come up with. Additionally, the collision detection algorithm can be sloppier since there are no δ_n and δ_t to worry about. Moreover, since there are no stiff springs required by the model, the simulation can proceed with large Δt without rendering the numerical integration unstable.

The complementarity approach does have its own problems, though. First, the implementation is complex. Second, computing F_n and F_t leads to a coupled problem that brings together all friction and normal forces for the entire system. For robotics, this is not a major limitation, since the number of contact events at any given time is small, perhaps less than 15–20 contacts. However, for granular terrains or other systems that involve large numbers of contact events, the fact that the normal and friction forces become unknowns adds millions of new variables to the problem. Finally, and perhaps most vexing, there is a lack of uniqueness in the solution of the problem that produces F_n and F_t. In fairness, this is tied to the rigid-body assumption, not exactly to the complementarity approach. Nonetheless, since the latter builds off the former, the approach, in certain conditions that cannot be foreseen, can produce, equally well, several different distributions for the normal and friction forces. The simplest way to understand this is to think of a symmetric four-legged stool weighing 100 N. Under a rigid-body assumption, the four legs could take four 25-N loads; or 30-N, 20-N, 30-N, and 20-N loads; or 40-N, 10-N, 40-N, and 10-N loads; and so on. This redundancy is common in packed granular materials but can arise in robotics as well. Nonetheless, this is disconcerting since it can compromise smoothness, which confuses the control algorithms. The penalty approach does not display this lack of uniqueness since there is local deformation (δ_n and δ_t), which lifts the indeterminacy.

3.2. Subsystem Simulation

This section concentrates on sensor and actuation simulation. Mainstream computer-aided engineering is rarely, if ever, concerned with sensor simulation, which is a prerequisite for intelligent robot analysis.

3.2.1. Simulating sensors. The extensive use of inexpensive sensors by today's robots is what differentiates them from older designs. Robots have been used for decades in industrial settings, but their catalog of actions is limited since they operate in structured and tightly controlled environments, such as the assembly line of a car manufacturing plant; concepts such as fog, dim light, and bicycle riders fall outside these robots' ontology. Having mobile robots operate in a hospital, in a plant populated with other mobile robots and workers, or in a warehouse with a topology that changes based on the day of the week requires sensors such as cameras, radars, lidars, infrared sensors, and inertial measurement units (IMUs).

In its basic task, sensor simulation is the process of generating synthetic information that is similar to the information that comes out of an actual sensor. If a vehicle drives down Main Street and its camera is on, data start streaming out of that sensor. If a virtual vehicle moves down a 3D Main Street replica generated using meshes, textures, appropriate light sources, and so on, as in a video game, a stream of data should come out of the virtual camera that resembles the data produced by the actual camera. A common misconception is that the data produced by the simulated camera should represent something that is pleasing to the eye, but the human eye is a sensor that is quite different from a camera's complementary metal–oxide–semiconductor (CMOS) sensor. A CMOS sensor can be expensive or cheap, produce large amounts of data or not, be noisy or crisp, and so on. Another way to put this is that there is a difference between the Unity (**https://unity3d.com**) or Unreal Engine (**https://www.unrealengine.com**) image we see on a screen, which is pleasing, and what it means for a camera sensor to be realistic (the issue of sensor-realism versus photo-realism).

Perhaps, in an end-to-end control strategy, a high-resolution camera is not actually needed. This point is made in **Figure 2**, in which a camera sensor is used for end-to-end learning for autonomous off-road convoy navigation (43). **Figure 2*a*** shows a frame from an animation intended to produce something that is pleasing to the eye, for use in a presentation, and **Figure 2*b*** shows a low-resolution image fed to a neural net for inference; the latter is the type of image the actual camera produces and is meant for vehicle control, not human consumption. Generating the **Figure 2*b*** image in simulation is nontrivial, and it starts with a photon leaving the vehicle in front and hitting the lens of the camera attached to the ego vehicle. When moving through the physical lens, there is lens distortion, vignetting, flare, and so on. After the lens comes the sensor,

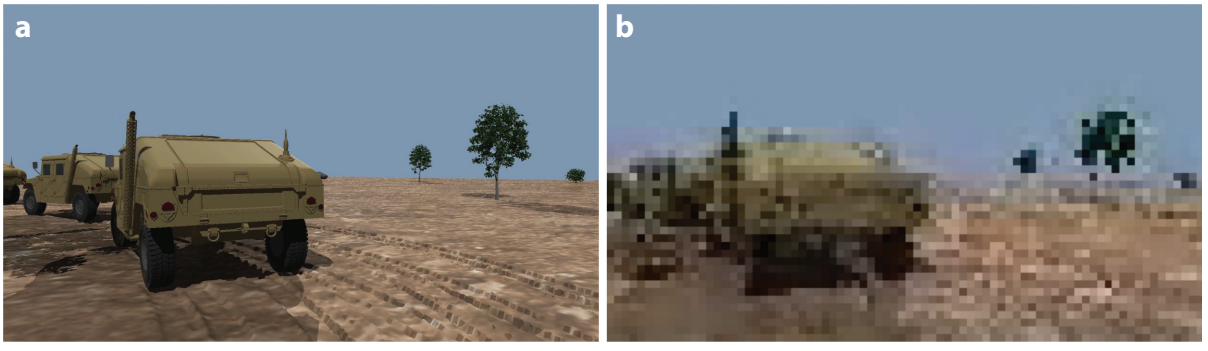

Figure 2

(*a*) Rendering of a frame used to generate a movie for human consumption. The snapshot is from the fourth vehicle in a convoy of four vehicles. (*b*) An 80 × 45–pixel replica of the frame in panel *a* simulated as being produced by a camera sensor attached to a vehicle in a platooning maneuver that relies on an end-to-end deep learning policy for driving in formation (43).

which introduces artifacts tied to quantum efficiency, dark noise, and so on (44), depending on the nature of the sensor [CMOS, charge-coupled device (CCD), etc.]. The data coming out of the sensor are color-blind and are passed into an image signal processor, which further massages the data for demosaicing, gamma correction, compression, and so on (45). Eventually, the data output is encoded, which might discard information from the data. This entire sensing process needs to be regenerated in simulation, leading to what is called physics-based camera simulation (46). Data-driven approaches are poised to make a mark in sensor simulation; with enough training, one can expect to augment or replace the data produced by the sensor simulator pipeline. Rather than spending time offline discovering and implementing sophisticated methodologies and then online running complex algorithms, one can use simpler sensor simulation approaches (perhaps non–physics based) and enhance them through machine learning. Such data-driven approaches are not yet available in the literature, but the building blocks are (47). A thorough discussion of sensing models for the different sensors (cameras, lidars, IMUs, etc.) falls outside the scope of this review; Elmquist & Negrut (48) discussed this topic further.

3.2.2. Simulating actuation. Actuation simulation seeks to map the command from the controller to the robot actuation expressed as a generalized force in Equation 1. The actuation mechanisms vary with the hardware, but most robots are actuated by independent rotor motors, which generate torques around joints. Many actuator models in robotics assume an ideal DC motor controlled by pulse-width modulation signals received from the controller. These signals modulate the supplied voltage, which determines the armature current of the actuator based on the armature resistance. For an ideal motor, the torque generated by the actuator is linear in the level of current, and the armature resistance and the coefficient of the linear function are typically parameters provided by the motor manufacturer. In reality, the linear torque–current relation does not hold true everywhere in the current domain—the torque saturates as the current increases. For applications that require highly accurate actuation simulation, one needs to carefully construct a nonlinear function that characterizes the torque–current relation through system identification.

In practice, most control algorithms operate in an abstract control space instead of directly commanding the supplied voltage or current. The most common abstraction is to directly command the joint torque. This torque-based control is trivial to simulate because it assumes that the actuator is able to achieve the target torque instantaneously, as long as the target torque is within predefined bounds and does not require too sudden of a change. Many robot platforms provide further abstraction in the control space, such as position- or velocity-based control. Position-based control allows the controller to command a desired configuration of the robot and use it as the rest position of a proportional–derivative controller to generate force. Alternatively, one can compute the generalized force required to achieve the desired configuration via inverse dynamics. Similarly, velocity-based control commands a desired velocity, and the actuator internally converts the target velocity to the required generalized force using inverse dynamics.

3.3. Human Simulation

Robots have the potential to provide adaptable and intelligent assistance to individuals, but robotic research involving humans presents numerous challenges, including the risks of rough contact or collision. Mobile robots and AVs can benefit from training in environments populated with ambulating humans and learning to avoid colliding with them. Healthcare robots, on the other hand, need to embrace physical contact and learn to utilize it in order to enable the daily living activities of humans. Likewise, wearable robotic devices and exoskeletons often need to apply large amounts of force to humans to support their body weight. An immediate concern in developing

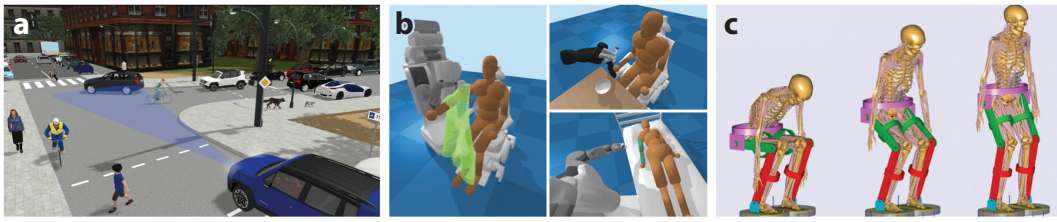

Figure 3

Three examples of human simulation: (*a*) DYNA4 (49), (*b*) Assistive Gym (50), and (*c*) the AnyBody Modeling System (51). Learning from physically simulated humans and environments presents a promising alternative that might enable robots to explore and learn from vast amounts of data more freely than they can in the real world. For example, autonomous vehicles can be trained in an environment populated with simulated pedestrians in the street. Likewise, healthcare robots and exoskeletons can safely make and learn from mistakes without putting real people at risk.

such an autonomous and powered robotic device is the safety of human users during the early development phase, when the control policies are still largely suboptimal (**Figure 3**).

Learning from physically simulated humans and environments presents a promising alternative that might enable robots to explore and learn from vast amounts of data more freely than they can in the real world. When compared with real-world robotic systems, physics simulation allows robots to safely make and learn from mistakes without putting real people at risk. It can also parallelize data collection to perform thousands of human–robot trials in a few hours and provide models of people representing a wide spectrum of human body shapes, weights, physical capabilities, and impairments.

What makes modeling human motion difficult is that it must simulate both the mechanics of the human body and its control. How do we develop such a human model that predicts the responses of real humans under external stimuli? Researchers in biomechanics and computer animation are interested in understanding the underlying dynamics and control mechanisms of natural human motions by re-creating and predicting them in the simulated world. The first step toward a computational model for human motion is to define a parameterized space of human motion with governing laws expressed as differential equations.

A natural starting point is to model the human skeletal system as articulated rigid bodies, a compact representation for functional, volitional movements of the human body. An articulated rigid-body system in the generalized coordinates is governed by Lagrange's equations of motion:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{g}(\mathbf{q}) + \mathbf{J}_{\mathrm{e}}^{\mathrm{T}}(\mathbf{q})\mathbf{f}_{\mathrm{e}} + \boldsymbol{\tau}, \qquad 6.$$

where $\mathbf{q} \in \mathbb{R}^N$ are the system's generalized coordinates used to determine the pose of the human via the orientation and location of each component of the human model. We denote the mass matrix, Coriolis force, and gravitational force as $\mathbf{M}$, $\mathbf{c}$, and $\mathbf{g}$, respectively, all in generalized coordinates. The external forces, $\mathbf{f}_{\mathrm{e}}$, such as the contact force, are described in the Cartesian space and impressed onto the generalized coordinates via the Jacobian matrix $\mathbf{J}_{\mathrm{e}}$. In addition to external gravitational and contact forces, humans also generate an internal force $\boldsymbol{\tau}$ to actuate their movements. The simplest actuation mechanism is to allow joints to directly generate torques, as if they were electric motors capable of producing angular acceleration between bones. However, the real human skeletal system does not move by itself—the bones are pulled by muscles, which are activated by neural excitation. A more accurate representation of human motion includes the internal states of the muscles and incorporates muscle contraction and neural activation as part of the equations of motion:

$$\boldsymbol{\tau} = \mathbf{J}_{\mathrm{m}}^{\mathrm{T}}(\mathbf{q})\mathbf{f}_{\mathrm{m}}(\mathbf{q}, \mathbf{l}, \dot{\mathbf{l}}, \mathbf{a}), \qquad 7.$$

where the muscle force $\mathbf{f}_m$ is computed by muscle contraction dynamics that depends on the current joint configuration $\mathbf{q}$, the muscle length $\mathbf{l}$ and its time derivative $\dot{\mathbf{l}}$, and the muscle activation $\mathbf{a}$. The muscle force $\mathbf{f}_m$ is impressed onto the generalized coordinates via the Jacobian matrix $\mathbf{J}_m$. Note that $\mathbf{J}_m$ and $\mathbf{J}_e$ are different when the points of application for $\mathbf{f}_m$ and for $\mathbf{f}_e$ are not the same. The muscle length $\mathbf{l}$ is governed by an ODE of musculotendon dynamics, $\dot{\mathbf{l}} = \mathbf{h}(\mathbf{q}, \mathbf{l}, \mathbf{a})$, which can be integrated to obtain the muscle length in the next time step: $\mathbf{l}_{t+1} = \mathbf{l}_t + \Delta t\ \mathbf{h}(\mathbf{q}_t, \mathbf{l}_t, \mathbf{a}_t)$. Seth et al. (52) provided details about the musculotendon dynamics defining the derivative function $\mathbf{h}$.

Each muscle activation a is also governed by an ODE:

$$\dot{a} = \frac{u-a}{k}, \quad \text{where } k = \begin{cases} k_A(0.5+1.5a) & \text{if } u > a, \\ \frac{k_D}{0.5+1.5a} & \text{otherwise.} \end{cases} \tag{8.}$$

The neural excitation $u \in [0, 1]$ is a scalar that controls muscle activation in the human musculoskeletal system, and k_A and k_D represent the time it takes to activate and deactivate a muscle fiber, respectively. The muscle activation ODE regulates the activation speed based on the current activation level. For example, if the current activation of a muscle fiber is already high, then the activation speed is slower, but the deactivation speed is higher.

While the dynamics and control mechanisms of a single muscle are well understood, modeling the whole human body using such a bottom-up approach is simply intractable—the human body consists of 206 bones driven by more than 600 muscles and tendons, which in turn are controlled by billions of neurons. In addition, Equation 7 depends on a large number of model parameters, such as the inertial, material, and geometrical properties of each muscle fiber, which are difficult to identify accurately. It is also important to note that numerous simplifications and assumptions have been made in modeling the geometry, dynamics, and physiological properties of the musculoskeletal system.

If modeling the human musculoskeletal system is daunting, controlling such a system to create natural and functional movements is perhaps even more complicated. The human musculoskeletal system is uniquely perplexing because we have redundant control variables (i.e., billions of neurons) and yet our dynamic system is underactuated. One can start by dividing the degrees of freedom into two sets, $\mathbf{q} = \mathbf{q}_0 \cup \mathbf{q}_1$, where $\mathbf{q}_0 \in \mathbb{R}^6$ contains the 6D global translation and orientation of the entire human system, and $\mathbf{q}_1$ stores the remaining degrees of freedom. The equations of motion (Equation 6) can then also be divided into two sets:

$$\begin{bmatrix} \mathbf{M}_{00} & \mathbf{M}_{01} \\ \mathbf{M}_{01}^T & \mathbf{M}_{11} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \ddot{\mathbf{q}}_1 \end{bmatrix} + \begin{bmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \end{bmatrix} + \begin{bmatrix} \mathbf{J}_0^T \\ \mathbf{J}_1^T \end{bmatrix} \mathbf{f}_c + \begin{bmatrix} \mathbf{0} \\ \boldsymbol{\tau}_1 \end{bmatrix}, \tag{9.}$$

where the zero entries in the forcing term reflect the fact that the global translation and orientation of the entire body, $\mathbf{q}_0$, are unactuated. To control $\mathbf{q}_0$, we must utilize external forces by creating and manipulating contacts $\mathbf{f}_c$ with the environment:

$$\ddot{\mathbf{q}}_0 = -\mathbf{M}_{00}^{-1}\mathbf{M}_{01}\ddot{\mathbf{q}}_1 - \mathbf{M}_{00}^{-1}\mathbf{c}_0 + \mathbf{M}_{00}^{-1}\mathbf{g}_0 + \mathbf{M}_{00}^{-1}\mathbf{J}_0^T\mathbf{f}_c. \tag{10a.}$$

Obviously, we cannot directly control the contact force $\mathbf{f}_c$. Instead, we need to use the controllable degrees of freedom $\boldsymbol{\tau}_1$ to manipulate the contact force indirectly, which makes the control problem substantially more difficult.

The underactuation challenge is not due to the lack of control variables. In fact, our musculoskeletal system is largely redundant—we have nearly 50 muscles to control just six degrees of freedom on the leg. The redundancy serves many mechanical purposes (e.g., increasing stability), but the highly nonlinear mapping from muscle activation $\mathbf{a} \in \mathbb{R}^m$ to independent joint torques $\boldsymbol{\tau}_1 \in \mathbb{R}^{N-6}$, where $m \gg N$, increases the difficulty of control:

$$\ddot{\mathbf{q}}_1 = -\mathbf{M}_{11}^{-1}\mathbf{M}_{01}\ddot{\mathbf{q}}_0 - \mathbf{M}_{11}^{-1}\mathbf{c}_1 + \mathbf{M}_{11}^{-1}\mathbf{g}_1 + \mathbf{M}_{11}^{-1}\mathbf{J}_1^T\mathbf{f}_c + \mathbf{M}_{11}^{-1}\mathbf{J}_M^T\mathbf{f}_m(\mathbf{q}, \mathbf{l}, \dot{\mathbf{l}}, \mathbf{a}). \tag{10b.}$$

While the road to modeling, simulating, and controlling the human physiological system is extremely arduous, over the last few decades, the field of computer animation has been quite successful in developing shortcuts to re-create natural human movements. One common belief is that we can synthesize the observable aspects of human movement without needing to fully understand human biological systems. As such, it is common for computer animation techniques to make whatever simplifications and assumptions are necessary to achieve realistic human movements. When applying these techniques to human–robot interaction applications, it is critical to first understand the key trade-offs that characterize the research landscape in computer animation.

3.3.1. The trade-off between optimality and coverage of state space. One way to understand the complex landscape of computer animation is through the lens of optimal control theory. The problem of human motion synthesis can be generally formulated as a PDE, and the solution to the PDE is a controller that maps states to actions. Most PDEs that address complex and continuous human movements do not have classic solutions where the controller is optimal everywhere in the state space. The need to simplify the PDE results in a trade-off between optimally solving a trajectory from a single state and suboptimally solving a wide range of states. Earlier work on manually designing controllers for specific tasks, such as locomotion, did not attempt to pursue optimality but yielded reasonable actions for a wide range of states (53). At the other side of the spectrum, trajectory optimization seeks the optimal action along one specific trajectory in the state space. The solution of trajectory optimization, while optimal, deteriorates quickly when the agent deviates from the trajectory due to perturbations in the environment (54).

A deep reinforcement learning approach strives to reach a balance between optimality and a wider coverage of the state space by maximizing the long-term reward for a set of states during stochastic exploration. Applying deep reinforcement learning to human motion synthesis has been actively explored in the recent years, and great achievements have been made in locomotion (55), manipulation (56), and other highly dynamic tasks (57, 58). Understanding the trade-off between a single optimal trajectory and a suboptimal policy for a wide range of states is paramount to a robotic application. For example, if the role of the human is simply that of a moving obstacle in a navigation application, tracking an optimal trajectory in an open-loop fashion is likely to be sufficient. On the other hand, physical human–robot interaction, such as assistive dressing or feeding, might require a policy that can handle unexpected perturbations in the testing environment. In that case, the coverage of state space that can be controlled reasonably well is more important than finding the true optimal solution.

3.3.2. The trade-off between functionality and naturalness. One major difference between a human motion controller and a robot controller is that human motion must be not only functional but also visually natural. Designing an objective function that balances these two often conflicting goals is intricate and usually done in a trial-and-error fashion. For example, a locomotion policy that is concerned only with task goals tends to produce highly energetic and asymmetric motion that is unlike any biological movements we see in the real world. If we simply dial up the weight for the energy cost in the objective function, the motion could look smoother, but the agent might simply stand still to save energy and fail to achieve the goal of locomotion.

Frustrated by endless tweaking, some researchers argue that naturalness in motion should be achieved not through control but through modeling. Using muscles and tendons to drive the skeletal system has long been hypothesized as the key element to producing human-like movements, as opposed to directly controlling torque actuated at joints. Modeling the musculotendon system essentially changes the boundary of the action space and the calculation of the cost of control,

such that the controller yields more human-like actions (59–61). However, the complexity of the musculotendon model comes at a steep computation and implementation cost, rendering many downstream motor learning tasks infeasible. While modeling the musculotendon system for the purpose of visual naturalness of human motion might not be necessary in robotic applications, estimating the biomechanical or physiological state of human motion, such as the realistic range of motion or fatigue level, could be essential for evaluating the effectiveness and safety of a robot that physically interacts with humans (62).

3.3.3. The trade-off between data and domain knowledge. Using real human motion data to inform the model is a logical approach when our prior knowledge is lacking and model parameters are uncertain. In computer animation, motion data are utilized in many different ways, from informing reward functions to selecting features in the state space to customizing the action space (63, 64). Motion data can also be used to train a dynamic model that predicts future human movement conditioned on high-level commands, bypassing the need for physics simulation. The controllers that effectively utilize motion data tend to create more human-like movements, but the dependency on the data also limits the transfer of the controller to new situations, different tasks, or unexpected perturbations in the environment. On the other hand, developing a controller without the aid of motion data is significantly more challenging. Without the desired bias induced by the motion data, the controller might start off very far away from the solution and can easily fall into one of many undesired local minima. Prior work has utilized domain knowledge to design objective functions or curriculum learning to mitigate this issue (65), but when applied to complex motor skills, these approaches usually cannot be exempted from costly computation, initial guess tweaking, or objective function shaping.

If the generalization of human movements is not expected in a robotic application, then data-driven approaches and imitation learning are viable ways to achieve realistic and functional human movements. However, if the robot frequently perturbs the human's movement during (physical) interaction, as with wearable robotic devices, a data-driven controller that mimics the motion data is likely to be suboptimal. Moreover, the motion data for many human–robot interaction scenarios cannot be captured if a functional robot controller does not yet exist. This presents a chicken-and-egg problem because the robot controller depends on the data, which in turn depend on the availability of a functional robot controller.

The three trade-offs presented are not meant to be viewed as orthogonal axes. The use of data has an obvious impact on the trade-off between functionalities and naturalness. Likewise, the trade-off between optimality and state-space coverage also depends on the amount of high-quality data available. The takeaway is that the field of computer animation can provide many practical solutions to the field of robotics, but it does not have a panacea for synthesizing general human movements. Knowing the demands of the robotic application and the trade-off each method brings is the first step toward an effective solution.

4. SIMULATION IN ROBOTICS SOLUTIONS

Simulation in robotics is used mostly for mechanical (body) design and controls (mind) design. The body design is concerned with whether a candidate solution can physically accomplish one or more tasks—reach a high drawer in a cabinet, lift a certain weight in a warehouse, not break in two months, and so on—while meeting certain constraints. The mind design is concerned with enabling the robot to safely and efficiently execute its tasks. For both body and mind design, simulation is used to optimize a candidate solution, quantify uncertainty, and assess safety. The concept of a robot, and by implication the simulation thereof, is open to interpretation. One can

argue that AVs are actually sophisticated robots. The discussion here touches but does not focus on AV simulation. When discussing AVs, we emphasize off-road AVs; on-road AV simulation tools have been reviewed elsewhere (66, 67).

We list below, alphabetically and with comments, entries in two categories: dynamics engines and platforms, the latter of which integrate dynamics engines with sensing support, virtual world modeling, advanced rendering, user interaction support, and so on, providing a one-stop shopping solution for robotics simulation. The list is not comprehensive; a solution is listed if it is broadly used or open source. References 68–70 provide head-to-head comparisons of several of the dynamics engines discussed below.

4.1. Dynamics Engines

Bullet (**http://pybullet.org**), a widely used engine in the movie and gaming industry, places equal emphasis on speed and accuracy. It supports rigid-body dynamics (primarily via an extended set of generalized coordinates, and more recently also via reduced coordinates), friction, and contact (complementarity). It has a collision detection engine that itself is widely used by other engines. Flexible bodies are approximated via lumped-mass systems (mass–spring–damper elements). It is used for traditional robotics simulation—manipulation, robotic arms, limbed robots, and so on (71)—and anchors Google's push into robotics.

Chrono (72) is a multiphysics engine that emphasizes accuracy; its strength is the simulation of wheeled and tracked robots. It supports rigid and flexible (nonlinear FEA) body dynamics, friction and contact handling (penalty and complementarity), and deformable terrain simulation at several levels: empirical (expeditious but less accurate), continuum (a middle-ground solution), and discrete element (accurate but slow). It has GPU-implemented modules for fluid–solid interaction (Navier–Stokes based, with SPH) and granular dynamics.

DART (Dynamic Animation and Robotics Toolkit) (73) places equal emphasis on speed and accuracy. Its API design seeks to reduce implementation overhead for researchers in the robotics and computer graphics communities. In contrast to many popular physics engines that view the simulator as a black box, DART gives full access to internal kinematic and dynamic quantities and allows users to easily specify or request data computed by the engine. In addition, it provides flexibility to extend the API for embedding user-provided classes into DART data structures.

MuJoCo (Multi-Joint Dynamics with Contact) (74) places equal emphasis on speed and accuracy. It focuses on classical robotics applications, such as multiple link arms, fingers for grasping, and walking bipeds. It has become very popular owing to its speed, accuracy, and robotics-centric user experience. It supports rigid-body dynamics (reduced set of generalized coordinates) and friction and contact (complementarity). Flexible bodies are approximated using a lumped-mass approach (similar in nature to Bullet).

Newton Dynamics (**http://newtondynamics.com**) emphasizes speed, as it is widely used in video gaming. It supports rigid-body dynamics, collision detection, and friction and contact (penalty).

ODE (Open Dynamics Engine) is a physics engine that emphasizes speed over accuracy. It is used for video game development and was relied upon in the early virtual robotics competitions by virtue of being the default simulation engine in Gazebo. It supports rigid-body dynamics (extended set of generalized coordinates) and a collision detection engine.

PhysX is a multiphysics engine that emphasizes speed over accuracy and is primarily designed for video gaming. It is NVIDIA's flagship simulation solution; the company has gone to great lengths to improve its accuracy, as it is nowadays relied upon in robotics (Isaac; see below) and AV simulation (75). It handles rigid-body dynamics and particle-based simulation for scenes

with multiple bodies that collide, break, and so on. The SPH-type particle is used in expeditious simulation of fluids.

4.2. Platforms

Unity (**https://unity3d.com**) and Unreal Engine (**https://www.unrealengine.com**) are the two most versatile platforms for video game development. They have out-of-the-box support for physics simulation, graphics rendering, scripting, collision detection, and more. They are not robotics simulation platforms per se, but they anchor several platforms, as discussed below. Neither is open source; it is possible to gain access to the code, but owing to the software complexity and dependencies associated with these massive codes, augmenting the software to improve the dynamics engine (both Unity and Unreal Engine use NVIDIA's PhysX) or sensing support is not simple. Moreover, such investments are not guaranteed to be portable to the next version of the gaming engine. This leads to a conundrum: Building a robotics simulation platform off a gaming environment provides significant advantages for virtual world content modeling, but the physics and sensing simulation support is limited, and improving it is daunting.

AirSim (76) is Microsoft's open source platform for robotics simulation built off Unreal Engine. Originally developed for simulating flying drones, it now supports on-road mobility via PhysX. While the graphics are top-notch, the physics-based sensing simulation is limited.

Chrono (72, 77) is an open source platform developed in academia that is focused primarily on off-road mobility. It comes with a Python interface and predefined templates for easy instantiation of wheeled or tracked vehicles. It relies on parallel computing to simulate tens of high-fidelity vehicles operating on rigid terrain in real time (43). It is used by NASA and the US Army and Navy, in industry, and in academic projects concerned with robots operating primarily off-road (silt-like, snow-like, and granular deformable terrains; see **Figure 2**).

CoppeliaSim (formerly V-REP) (78) is a closed source, commercial multirobot simulation platform that exposes a rich set of dynamics engines: Bullet, MuJoCo, Newton Dynamics, and Vortex Studio (79). It simulates manipulation, robotic arms, manufacturing robots, factory automation, and so on, and it was designed to scale for multirobot applications. It has limited physics-based sensor support.

Gazebo (80; **http://gazebosim.org**) is likely the most widely used robotics simulation platform. It exposes several physics engines: Bullet, DART, ODE, and Simbody (81). For graphics, it uses Ogre3D (**http://www.ogre3d.org**) and OpenGL (**http://www.opengl.org**). The support for physics-based sensing simulation is fairly basic, although it can be user augmented by virtue of being open source. It is used for both traditional robotics applications and off-road mobility. It has been used for several robotics challenges, including the DARPA Robotics Challenge, Virtual RobotX, and competitions conducted by NASA and Toyota.

iGibson (the Interactive Gibson Environment) (2) and Habitat (82) are two platforms that simulate photo-realistic indoor scenes based on the real world for the development of robot controllers that use RGB images. The platforms provide very fast rendering of photo-realistic images, as they were designed to support data-intensive deep (reinforcement) learning. Both platforms use Bullet as their dynamics engine to enable physical interaction between the robot and the environment. iGibson was developed in academia, while Habitat was developed by Facebook.

Isaac (83) is NVIDIA's recent entry in the robotics simulation arena. Virtual world support comes via Unreal Engine, which under the hood embeds PhysX. There is support for human models that can interact with the robots. Three predefined worlds are available out of the box: a hospital, kitchen, and warehouse. The software is free but not open source. The level of sensor sophistication is unknown.

MAVS (Mississippi State University Autonomous Vehicle Simulator) is a closed source off-road simulation environment actively developed in academia (84). It provides an in-house-developed, sophisticated physics-based sensor simulation module (85, 86), has a ROS (Robot Operating System) bridge, and uses Chrono as its dynamics engine.

ROAMS (Rover Analysis, Modeling, and Simulation) (87) is a platform for rover design and testing developed by NASA's Jet Propulsion Lab. It relies on the in-house-developed simulation engine DARTS (Dynamics Algorithms for Real-Time Simulation) (88) to provide support for simulating instrument arms, terrain interaction, sensors, actuators, and power sources. It is not open source but is available for free for research purposes.

USARSim (Urban Search and Rescue Simulation) (89) is a platform that draws on Unreal Engine. It is open source and aimed at multilegged robot simulation. It is not under current development.

Webots (90) is a popular open source platform that draws on a customized version of the ODE dynamics engine; it relies on OpenGL for virtual world rendering. It has been actively developed for almost two decades and has a large collection of ready-to-use robot models, mostly of the indoor type, although there is limited support for AVs, flying drones, and tracked robots. It has a rich collection of sensors, catering mostly to indoor robotics.

5. CLOSING THOUGHTS: LOOKING AHEAD

In simulation, robots can be controlled to perform amazing tasks, and stunning movies can be generated as well. As has been poignantly observed, "Simulation is doomed to succeed"—and indeed, reality is often unkind to simulation-derived control policies. Yet if we discard simulation, what alternatives are there? Setting aside the case of robotic manipulators, the trial-and-error approach is expensive, sometimes dangerous, and sometimes impossible (e.g., testing a Mars rover). We believe that striving to steadily improve the simulation-to-real transfer rate is a worthwhile endeavor. One question is how a computer analysis can be used to decide whether a control policy derived in simulation will work in reality. While we do not know the answer to this question, we believe that increasing the fidelity of the simulation is key. Another school of thought is that robust policies can be obtained through randomization of model parameters (91). Instead of randomly changing parameters, one relatively unexplored approach would change the fidelity of the model, using several levels of resolutions, matryoshka-style (92). Some of these models can be exclusively data driven, an approach that is used in many applications, such as cars (93). Using statistical learning techniques, one could bypass simulation-specific hurdles such as model generation and calibration by constructing oracles that predict the next system state given its current one. Data-driven simulation could be used to methodically reduce model complexity through systematic dimensional reduction. However, such approaches for highly nonlinear and nonsmooth systems are in their infancy, and principled methodologies to design and validate data-driven models remain to be identified.

Building a simulation-in-robotics ecosystem is a multidisciplinary effort in which funding agencies can bring together disparate communities to work toward a worthwhile end. As shown in this article, simulation in robotics consists of much more than a dynamics engine that predicts the next state of a mechanical contraption. It also needs to simulate sensors, unstructured and time-dependent environments, and potentially the humans interacting with the robot. Relevant open problems that are beyond our current level of understanding will require basic research in a variety of areas, including physics-based modeling of frictional contact, impacts, soft or compliant bodies, fluid–solid interactions, and terramechanics, as well as expeditious numerical methods for DAEs, PDEs, FEA, and computational fluid dynamics. These problems are highly intertwined

and need to be addressed in a holistic way. We believe that a broad, multiagency initiative can stimulate fundamental research in relevant areas and at the same time foster its translation into open source simulation platforms. An initiative of this caliber would bring together, ideally in an international framework, research groups from academia and agile software development outfits from research laboratories and industry.

DISCLOSURE STATEMENT

C.K.L. is the inventor of the physics engine DART and has been a consultant for X Development LLC since 2019.

ACKNOWLEDGMENTS

C.K.L. was supported in part by National Science Foundation projects IIS-1514258 and IIS-1953008. D.N. was supported in part by National Science Foundation project CPS-1739869.

LITERATURE CITED

1. Mayne DQ, Rawlings JB, Rao CV, Scokaert PO. 2000. Constrained model predictive control: stability and optimality. *Automatica* 36:789–814
2. Xia F, Shen WB, Li C, Kasimbeg P, Tchapmi ME, et al. 2020. Interactive Gibson benchmark: a benchmark for interactive navigation in cluttered environments. *IEEE Robot. Autom. Lett.* 5:713–20
3. Kane TR, Levinson DA. 1983. The use of Kane's dynamical equations in robotics. *Int. J. Robot. Res.* 2:3–21
4. Featherstone R. 1987. *Robot Dynamics Algorithms*. Boston: Kluwer
5. Hairer E, Wanner G. 1996. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Berlin: Springer
6. Golub GH, Van Loan CF. 1980. *Matrix Computations*. Baltimore, MD: Johns Hopkins Univ. Press
7. Haug E. 1989. *Computer-Aided Kinematics and Dynamics of Mechanical Systems*, Vol. 1: *Basic Methods*. Englewood Cliffs, NJ: Prentice Hall
8. Petzold LR. 1982. Differential-algebraic equations are not ODE's. *SIAM J. Sci. Stat. Comput.* 3:367–84
9. Wehage RA, Haug EJ. 1982. Generalized coordinate partitioning for dimension reduction in analysis of constrained dynamic systems. *J. Mech. Des.* 104:247–55
10. Baraff D. 1996. Linear-time dynamics using Lagrange multipliers. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 137–46. New York: ACM
11. Marsden JE, Ratiu TS. 1994. *Introduction to Mechanics and Symmetry*. New York: Springer
12. Liang CD, Lance GM. 1987. A differentiable null-space method for constrained dynamic analysis. *ASME J. Mech. Transm. Autom. Des.* 109:405–10
13. Betsch P, Leyendecker S. 2006. The discrete null space method for the energy consistent integration of constrained mechanical systems. Part II: multibody dynamics. *Int. J. Numer. Methods Eng.* 67:499–552
14. Orlandea N, Chace MA, Calahan DA. 1977. A sparsity-oriented approach to the dynamic analysis and design of mechanical systems—part 1. *Trans. ASME J. Eng. Ind.* 99:773–79
15. Orlandea N, Calahan DA, Chace MA. 1977. A sparsity-oriented approach to the dynamic analysis and design of mechanical systems—part 2. *Trans. ASME J. Eng. Ind.* 99:780–84
16. Negrut D, Jay L, Khude N. 2009. A discussion of low-order numerical integration formulas for rigid and flexible multibody dynamics. *J. Comput. Nonlinear Dyn.* 4:021008
17. Bauchau OA, Laulusa A. 2008. Review of contemporary approaches for constraint enforcement in multibody systems. *J. Comput. Nonlinear Dyn.* 3:011005
18. Helton JC, Davis FJ. 2003. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliab. Eng. Syst. Saf.* 81:23–69
19. Kennedy MC, O'Hagan A. 2001. Bayesian calibration of computer models. *J. R. Stat. Soc. B* 63:425–64

20. Borgonovo E, Plischke E. 2016. Sensitivity analysis: a review of recent advances. *Eur. J. Oper. Res.* 248:869–87
21. Hoffman MD, Gelman A. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15:1593–623
22. Ericson C. 2005. *Real Time Collision Detection*. San Francisco, CA: Morgan Kaufmann
23. Nowakowski C, Fehr J, Fischer M, Eberhard P. 2012. Model order reduction in elastic multibody systems using the floating frame of reference formulation. *IFAC Proc.* 45:40–48
24. Shabana AA, Yakoub RY. 2001. Three dimensional absolute nodal coordinate formulation for beam elements: theory. *ASME J. Mech. Des.* 123:606–13
25. Shabana AA. 2020. *Dynamics of Multibody Systems*. Cambridge, UK: Cambridge Univ. Press. 5th ed.
26. Bekker MG. 1969. *Introduction to Terrain-Vehicle Systems*. Ann Arbor: Univ. Mich. Press
27. Janosi Z, Hanamoto B. 1961. *The analytical determination of drawbar pull as a function of slip for tracked vehicles in deformable soils*. Paper presented at the 1st International Conference on the Mechanics of Soil-Vehicle Systems, Turin, It., June 15–23
28. Wong JY, Reece AR. 1967. Prediction of rigid wheel performance based on the analysis of soil-wheel stresses: part II. Performance of towed rigid wheels. *J. Terramech.* 4:7–25
29. Wong JY. 2008. *Theory of Ground Vehicles*. New York: Wiley & Sons
30. Ishigami G, Miwa A, Nagatani K, Yoshida K. 2007. Terramechanics-based model for steering maneuver of planetary exploration rovers on loose soil. *J. Field Robot.* 24:233–50
31. Krenn R, Gibbesch A. 2011. Soft soil contact modeling technique for multi-body system simulation. In *Trends in Computational Contact Mechanics*, ed. G Zavarise, P Wriggers, pp. 135–55. Berlin: Springer
32. Li C, Zhang T, Goldman D. 2013. A terradynamics of legged locomotion on granular media. *Bull. Am. Phys. Soc.* 58:1408–12
33. Agarwal S, Senatore C, Zhang T, Kingsbury M, Iagnemma K, et al. 2019. Modeling of the interaction of rigid wheels with dry granular media. *J. Terramech.* 85:1–14
34. Fervers CW. 2004. Improved FEM simulation model for tiresoil interaction. *J. Terramech.* 41:87–100
35. Chiroux R, Foster W Jr., Johnson C, Shoop S, Raper R. 2005. Three-dimensional finite element analysis of soil interaction with a rigid wheel. *Appl. Math. Comp.* 162:707–22
36. Knuth M, Johnson J, Hopkins M, Sullivan R, Moore J. 2012. Discrete element modeling of a Mars exploration rover wheel in granular material. *J. Terramech.* 49:27–36
37. Cundall P, Strack O. 1979. A discrete numerical model for granular assemblies. *Geotechnique* 29:47–65
38. Gurtin ME, Fried E, Anand L. 2010. *The Mechanics and Thermodynamics of Continua*. Cambridge, UK: Cambridge Univ. Press
39. Gingold RA, Monaghan JJ. 1982. Kernel estimates as a basis for general particle methods in hydrodynamics. *J. Comput. Phys.* 46:429–53
40. Becker M, Teschner M. 2007. Weakly compressible SPH for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 209–17. Goslar, Ger.: Eurographics Assoc.
41. Johnson KL. 1987. *Contact Mechanics*. Cambridge, UK: Cambridge Univ. Press
42. Stewart DE. 2000. Rigid-body dynamics with friction and impact. *SIAM Rev.* 42:3–39
43. Negrut D, Serban R, Elmquist A, Taves J, Young A, et al. 2020. *Enabling artificial intelligence studies in off-road mobility through physics-based simulation of multi-agent scenarios*. Paper presented at the 12th Ground Vehicle Systems Engineering and Technology Symposium, online, Nov. 3–5
44. Eur. Mach. Vis. Assoc. 2010. *Standard for characterization of image sensors and cameras*. Stand. 1288, Eur. Mach. Vis. Assoc., Barcelona, Spain
45. Guo S, Yan Z, Zhang K, Zuo W, Zhang L. 2019. Toward convolutional blind denoising of real photographs. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1712–22. Piscataway, NJ: IEEE
46. Farrell JE, Catrysse PB, Wandell BA. 2012. Digital camera simulation. *Appl. Opt.* 51:A80–90
47. Isola P, Zhu JY, Zhou T, Efros AA. 2017. Image-to-image translation with conditional adversarial networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1125–34. Piscataway, NJ: IEEE

48. Elmquist A, Negrut D. 2020. Methods and models for simulating autonomous vehicle sensors. *IEEE Trans. Intell. Veh.* 5:684–92
49. Vector. 2020. DYNA4 - virtual test driving. *Vector*. **https://www.vector.com/int/en/products/products-a-z/software/dyna4**
50. Erickson ZM, Gangaram V, Kapusta A, Liu CK, Kemp CC. 2020. Assistive Gym: a physics simulation framework for assistive robotics. In *2020 IEEE International Conference on Robotics and Automation*, pp. 10169–76. Piscataway, NJ: IEEE
51. AnyBody Technol. 2020. The AnyBody Modeling System. *AnyBody Technology*. **https://www.anybodytech.com/software**
52. Seth A, Sherman M, Reinbolt JA, Delp SL. 2011. OpenSim: a musculoskeletal modeling and simulation framework for *in silico* investigations and exchange. *Procedia IUTAM* 2:212–32
53. Yin K, Loken K, van de Panne M. 2007. SIMBICON: simple biped locomotion control. *ACM Trans. Graph.* 26:105
54. Liu CK, Hertzmann A, Popović Z. 2005. Learning physics-based motion style with nonlinear inverse optimization. *ACM Trans. Graph.* 24:1071–81
55. Peng XB, Berseth G, Yin K, van de Panne M. 2017. DeepLoco: dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Trans. Graph.* 36:41
56. Zhu H, Gupta A, Rajeswaran A, Levine S, Kumar V. 2019. Dexterous manipulation with deep reinforcement learning: efficient, general, and low-cost. In *2019 International Conference on Robotics and Automation*, pp. 3651–57. Piscataway, NJ: IEEE
57. Tan J, Gu Y, Liu CK, Turk G. 2014. Learning bicycle stunts. *ACM Trans. Graph.* 33:50
58. Liu L, Hodgins JK. 2018. Learning basketball dribbling skills using trajectory optimization and deep reinforcement learning. *ACM Trans. Graph.* 37:142
59. Jiang Y, Wouwe TV, Groote FD, Liu CK. 2019. Synthesis of biologically realistic human motion using joint torque actuation. *ACM Trans. Graph.* 38:72
60. Lee S, Park M, Lee K, Lee J. 2019. Scalable muscle-actuated human simulation and control. *ACM Trans. Graph.* 38:73
61. Nakada M, Zhou T, Chen H, Weiss T, Terzopoulos D. 2018. Deep learning of biomimetic sensorimotor control for biomechanical human animation. *ACM Trans. Graph.* 37:56
62. Jiang Y, Liu CK. 2018. Data-driven approach to simulating realistic human joint constraints. In *2018 IEEE International Conference on Robotics and Automation*, pp. 1098–103. Piscataway, NJ: IEEE
63. Peng XB, Abbeel P, Levine S, van de Panne M. 2018. DeepMimic: example-guided deep reinforcement learning of physics-based character skills. *ACM Trans. Graph.* 37:143
64. Liu L, Hodgins JK. 2017. Learning to schedule control fragments for physics-based characters using deep Q-learning. *ACM Trans. Graph.* 36:29
65. Yu W, Turk G, Liu CK. 2018. Learning symmetric and low-energy locomotion. *ACM Trans. Graph.* 37:144
66. Rosique F, Navarro PJ, Fernández C, Padilla A. 2019. A systematic review of perception system and simulators for autonomous vehicles research. *Sensors* 19:648
67. Kang Y, Yin H, Berger C. 2019. Test your self-driving algorithm: an overview of publicly available driving datasets and virtual testing environments. *IEEE Trans. Intell. Veh.* 4:171–85
68. Boeing A, Bräunl T. 2007. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, pp. 281–88. New York: ACM
69. Erez T, Tassa Y, Todorov E. 2015. Simulation tools for model-based robotics: comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *2015 IEEE International Conference on Robotics and Automation*, pp. 4397–404. Piscataway, NJ: IEEE
70. Collins J, Howard D, Leitner J. 2019. Quantifying the reality gap in robotic manipulation tasks. In *2019 International Conference on Robotics and Automation*, pp. 6706–12. Piscataway, NJ: IEEE
71. Peng XB, Coumans E, Zhang T, Lee TW, Tan J, Levine S. 2020. Learning agile robotic locomotion skills by imitating animals. arXiv:2004.00784 [cs.RO]
72. Tasora A, Serban R, Mazhar H, Pazouki A, Melanz D, et al. 2016. Chrono: an open source multi-physics dynamics engine. In *High Performance Computing in Science and Engineering*, ed. T Kozubek, pp. 19–49. Cham, Switz.: Springer

73. Lee J, Grey M, Ha S, Kunz T, Jain S, et al. 2018. DART: Dynamic Animation and Robotics Toolkit. *J. Open Source Softw.* 3:500
74. Todorov E, Erez T, Tassa Y. 2012. MuJoCo: a physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–33. Piscataway, NJ: IEEE
75. NVIDIA. 2020. NVIDIA DRIVE Constellation. *NVIDIA*. **https://www.nvidia.com/en-us/self-driving-cars/drive-constellation**
76. Shah S, Dey D, Lovett C, Kapoor A. 2018. AirSim: high-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, ed. M Hutter, R Siegwart, pp. 621–35. Cham, Switz.: Springer
77. Proj. Chrono Dev. Team. 2020. Chrono. *GitHub*. **https://github.com/projectchrono/chrono**
78. Rohmer E, Singh SP, Freese M. 2013. V-REP: a versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1321–26. Piscataway, NJ: IEEE
79. CM Labs. 2020. Vortex Studio. *CM Labs*. **https://www.cm-labs.com/vortex-studio**
80. Koenig NP, Howard A. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 4, pp. 2149–54. Piscataway, NJ: IEEE
81. Sherman MA, Seth A, Delp SL. 2011. Simbody: multibody dynamics for biomedical research. *Procedia IUTAM* 2:241–61
82. Savva M, Kadian A, Maksymets O, Zhao Y, Wijmans E, et al. 2019. Habitat: a platform for embodied AI research. In *2019 IEEE/CVF International Conference on Computer Vision*, pp. 9338–46. Piscataway, NJ: IEEE
83. NVIDIA. 2020. NVIDIA Isaac platform for robotics. *NVIDIA*. **https://www.nvidia.com/en-us/deep-learning-ai/industries/robotics**
84. Carruth DW. 2018. Simulation for training and testing intelligent systems. In *2018 World Symposium on Digital Intelligence for Systems and Machines*, pp. 101–6. Piscataway, NJ: IEEE
85. Goodin C, Doude M, Hudson C, Carruth D. 2018. Enabling off-road autonomous navigation-simulation of LIDAR in dense vegetation. *Electronics* 7:154
86. Goodin C, Carruth D, Doude M, Hudson C. 2019. Predicting the influence of rain on LIDAR in ADAS. *Electronics* 8:89
87. Jain A, Balaram J, Cameron J, Guineau J, Lim C, et al. 2004. Recent developments in the ROAMS planetary rover simulation environment. In *2004 IEEE Aerospace Conference Proceedings*, pp. 861–76. Piscataway, NJ: IEEE
88. Jain A. 2020. DARTS spacecraft dynamics simulator. *NASA Jet Propulsion Laboratory*. **https://dartslab.jpl.nasa.gov/DARTS**
89. Carpin S, Lewis M, Wang J, Balakirsky S, Scrapper C. 2007. USARSim: a robot simulator for research and education. In *2007 IEEE International Conference on Robotics and Automation*, pp. 1400–5. Piscataway, NJ: IEEE
90. Michel O. 2004. Cyberbotics Ltd. Webots^{TM}: professional mobile robot simulation. *Int. J. Adv. Robot. Syst.* 1:39–42
91. Peng XB, Andrychowicz M, Zaremba W, Abbeel P. 2018. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation*, pp. 3803–10. Piscataway, NJ: IEEE
92. Peherstorfer B, Willcox K, Gunzburger M. 2018. Survey of multifidelity methods in uncertainty propagation, inference, and optimization. *SIAM Rev.* 60:550–91
93. Bayarri MJ, Berger JO, Paulo R, Sacks J, Cafeo JA, et al. 2007. A framework for validation of computer models. *Technometrics* 49:138–54