Julia for robotics: simulation and real-time control in a high-level programming language

Conference Paper · May 2019

DOI: 10.1109/ICRA.2019.8793875

2 authors, including:

Twan Koolen
Massachusetts Institute of Technology
28 PUBLICATIONS **2,668** CITATIONS

Julia for robotics: simulation and real-time control in a high-level programming language

Twan Koolen[1] and Robin Deits[1]

***Abstract*— Robotics applications often suffer from the 'two-language problem', requiring a low-level language for performance-sensitive components and a high-level language for interactivity and experimentation, which tends to increase software complexity. We demonstrate the use of the Julia programming language to solve this problem by being fast enough for online control of a humanoid robot and flexible enough for prototyping. We present several Julia packages developed by the authors, which together enable roughly 2× realtime simulation of the Boston Dynamics Atlas humanoid robot balancing on flat ground using a quadratic-programming-based controller. Benchmarks show a sufficiently low variation in control frequency to make deployment on the physical robot feasible. We also show that Julia's naturally generic programming style results in versatile packages that are easy to compose and adapt to a wide variety of computational tasks in robotics.**

I. Introduction

Modern robotics requires performing a vast array of computational tasks, including online control, motion planning, dynamic simulation, controller synthesis, and stability analysis. These tasks in turn place a wide range of constraints on the software packages used to perform them. For example, online control tasks often have hard real-time constraints, where it is critical not to exceed a given time budget (*e.g.*, one millisecond) per control cycle. Typical online controllers for humanoid robots require evaluating quantities related to the dynamics of the robot subject to these real-time constraints. In contrast, controller synthesis is done just once, offline, but consumes the dynamics of the robot in symbolic form, requiring computation with an entirely different set of data types. In addition to this large variety of computational tasks, there is a wide range of robotics software users. Users may include students taking their first course in robotics, as well as experts who require high performance and rapid prototyping.

Currently, a standard approach to satisfying all of these requirements is to write software packages in two programming languages. Typically, a 'fast' programming language like C or C++ is used to implement the lower-level parts of a software package, while a dynamic, high-level language like Python or MATLAB is used either to write parts that do not have stringent performance requirements, or to provide bindings to the low-level parts. This approach provides ease of use for novice programmers and productivity for rapid prototyping, while retaining the option for power users to directly interact with the lower-level parts. Examples of software packages relevant to robotics that take this approach include Bullet [1], Drake [2], and Tensorflow [3].[1]

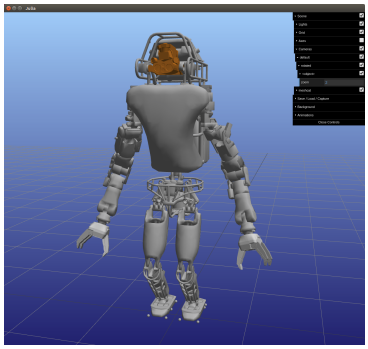

Fig. 1. Visualization of Atlas using the MeshCat.jl Julia package.

The two-language approach has some drawbacks, however. First, users who start out prototyping their robotics application in the 'high-level' language often need to duplicate their efforts by porting code to the lower-level language once they are satisfied with their prototype, a tedious and error-prone process. Second, in our experience, maintaining a software package written in multiple programming languages tends to cause a large amount of developer overhead: the code bases for each of the two languages have to be kept in sync, and a build system and documentation have to be maintained for each language.

This so-called 'two-language problem' exists in many fields with numerical computing needs, as recognized by the developers of the Julia programming language [5]. Julia is a relatively new high-level language that promises the high programmer productivity and interactivity of a dynamic language like Python in addition to C-like performance. In this paper, we provide evidence that supports this claim in the robotics domain. We present a number of Julia software packages written by the authors, and demonstrate that they can be used to implement a typical quadratic-programming (QP)-based low-level balancing controller for the humanoid robot Atlas (*e.g.,* [6], [7], [8], [9]) with low jitter in control frequency. In addition, we present a new robotics

[1]Twan Koolen and Robin Deits are with the Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA. {tkoolen, rdeits}@mit.edu.

[1]A notable exception to this pattern is the nearly exclusive use of Java by IHMC [4].

simulation environment written almost entirely in Julia.

We will focus on the following packages:

- RigidBodyDynamics.jl, a pure-Julia rigid body dynamics library;
- MeshCat.jl, a web-based, remotely controlled visualizer (see Fig. 1);
- RigidBodySim.jl, a simulation environment;
- Parametron.jl, a framework for formulating and efficiently solving instances of a parameterized family of convex optimization problems;
- QPControl.jl, an implementation of a QP-based low-level control algorithm for floating-base robots.

These packages should provide a solid foundation for others roboticists to build on.

The remainder of the paper is structured as follows. Section II provides an overview of the Julia language and an initial assessment of its suitability for robotics applications. Section III presents the packages we have developed. Section IV lists benchmark results that demonstrate the feasibility of Julia for online control of the humanoid robot Atlas. Section V concludes the paper.

II. Julia for robotics

This section first provides a general overview of Julia (II-A), followed by an assessment of its usability as a programming language for robotics applications (II-B).

A. Overview of Julia

Although version 1.0 of Julia has been released only very recently, the language has been under development since 2009 [5]. Julia is distributed under the MIT license, open source, and available free of charge. Julia is a general-purpose programming language with a strong focus on numerical computing, evidenced for example by an extensive linear algebra module as part of the standard library. Under the hood, Julia uses LLVM [10] to generate native machine code. Julia is garbage-collected and just-in-time (JIT)-compiled.

The Julia developers refer to the language as being 'optionally typed': the field types of data structures may either be omitted or fully specified, with the latter option typically resulting in better performance. Julia has a rich type system that includes parametric types (similar to C++ template classes) and strong type inference capabilities that obviate the need to manually annotate the type of every variable (similar to C++'s `auto` keyword).

Julia is *not* object-oriented, in the sense that types do not contain associated methods. Instead, functions are defined 'outside' the types, optionally using multiple overloaded methods that accept different sets of argument types. At a call site, the method with the most specific applicable type signature is selected based on the types of all of the input arguments, an approach referred to as multiple dispatch. Multiple dispatch in Julia is efficient enough that it is used throughout the language and ecosystem, with even the primitive arithmetic operations implemented in terms of multiple dispatch. This provides a single framework allowing user-defined types and functions to behave as performantly and naturally as any built into the language.

Julia provides convenient features common to high-level languages, such as built-in support for list comprehensions, generators, and parallelism. Julia also features excellent support for functional programming, with functions represented as first-class objects and efficient higher-order functions that are used extensively throughout the standard library. Julia borrows an important feature from LISP: Julia code is represented as a data structure in the language itself. This enables powerful metaprogramming features, as it is straightforward to generate Julia code in Julia without requiring a preprocessor or a separate template system.

Julia's LinearAlgebra module, part of its standard library, provides both low-level bindings to BLAS/LAPACK libraries, as well as high-level abstractions built on top of these bindings. Julia ships with OpenBLAS [11] by default, but can optionally be built with Intel's MKL [12] (also used by NumPy and MATLAB). Doing so can result in modest performance improvements. StaticArrays.jl [13], a third-party Julia package, provides stack-allocated fixed-size arrays, similar to the Eigen C++ library [14].

The Julia community has so far developed over 1900 registered Julia packages, and has adopted unit testing and continuous integration extensively.

B. Suitability for robotics applications

Our assessment will focus on two main areas: 1) usability in terms of developer productivity and prototyping capabilities, and 2) performance characteristics for high-rate online control.

It is hard objectively measure whether Julia delivers on its promise of high developer productivity and suitability for prototyping. Instead, we will resort to listing features that we think improve or detract from the user experience. Positive features include Julia's REPL (read-eval-print-loop) for interactive use and the ability to use Julia from Jupyter notebooks [15] for exploratory programming and demonstrations. Developer overhead related to supporting multiple operating systems is minimal compared to C++. Julia has a built-in package manager, which facilitates composing various third-party packages with user code and avoids much of the overhead associated with maintaining a build system in other languages. Several plotting packages are available. Finally, Julia's documentation functionality combined with the Documenter.jl package [16] makes it easy to generate and deploy documentation. The main impediments to developer productivity and interactivity are relatively long compilation times and the limited caching of compilation results between Julia sessions, resulting in frequent recompilation to native code. The Revise.jl package [17] may be used to partially mitigate this issue.

Performance characteristics for online control may be assessed more objectively. In addition to average throughput, a major requirement for high-rate online control is low variation in the rate at which the controller produces its output (jitter) [18]. Ideally, to provide so-called hard realtime guarantees, there should be a guaranteed upper bound on jitter.

At first glance, the fact that Julia is a garbage-collected language seems a major roadblock on the path to its adoption for online control. Julia employs a stop-the-world garbage collector (GC), meaning that no useful work is done while garbage is collected. Although garbage collectors with hard real-time guarantees do exist in other languages [19], [20], Julia's GC provides no such guarantees. Furthermore, a trivial benchmark shows that running a full garbage collector sweep without generating any garbage takes around 46 ms, which would upper-bound the achievable control rate to a mere 22 Hz.[2]

This implies that, currently, Julia's garbage collector should be disabled for online control purposes, which in turn implies that dynamic memory allocation should be avoided so as to not run out of memory. Hence, our current approach is to completely avoid dynamic allocation in code that is meant to be run in low-level control loops (after an initial preallocation phase). This includes evaluation of the dynamics, setting up and solving quadratic programs, and network communication. This is a serious constraint, but it should be noted that it is nontrivial to provide hard realtime constraints in the presence of dynamic allocation in any language, requiring *e.g.* a specialized memory allocator in C++ [21]. Avoiding dynamic allocation also has the added benefit that it tends to improve performance.

Given the constraint that dynamic allocation should be avoided, Julia has clear benefits over JVM-based languages like Java. The standard JVM implementation currently only provides the guarantee that instances of a predefined set of primitive types are stack-allocated. Julia additionally guarantees that immutable data structures (recursively) composed of such types are stack-allocated. This feature is used extensively throughout the Julia robotics code, allowing points, transforms, twists, and other fixed-size quantities to be freely constructed and used without requiring pre-allocation.

Julia's JIT compilation model is also more suitable to realtime control than Java's. Whereas Java's JIT compiler optimizes hot code at runtime [22], a potential source of jitter, Julia by default guarantees that functions are compiled to native code the first time they are called with a given set of argument types, providing more predictable behavior. While specialized JVM implementations with real-time JIT compilation and garbage collection capability are available [20], these solutions tend to have much lower throughput and/or a prohibitively high price tag for research robotics [4].

[2]We do note that soft realtime applications that perform dynamic memory allocation may be feasible, since an incremental sweep takes only ~78 μs and a full sweep is typically not run unless the program allocates significantly.

```
using RigidBodyDynamics
mechanism = parse_urdf("atlas.urdf", floating=true)
state = MechanismState(mechanism)
rand!(state)
result = DynamicsResult(mechanism)
dynamics!(result, state)
```

Fig. 2. RigidBodyDynamics.jl usage example: loading a mechanism from a URDF and evaluating the dynamics at a random state.

Julia's memory footprint is nontrivial, which may be an issue for robots with limited computing resources. A fresh Julia 1.1.0 session uses approximately 79 MB of RAM on Linux, increasing to 153 MB after importing the RigidBodyDynamics.jl package.

The PackageCompiler.jl package [23] can be used to create standalone executables and dynamic libraries from Julia code, potentially very useful for deployment to an embedded platform as well as to improve productivity by eliminating the need to recompile to native code. However, it is at an early stage of development and currently produces relatively big binaries.

III. Packages

This section presents a set of robotics-related Julia packages developed by the authors, which together form a foundation for simulation and control (III-A–III-E). All of the presented packages are MIT-licensed. Each package provides Jupyter notebooks containing usage examples. Some of the packages are part of the JuliaRobotics GitHub organization [24], which also incorporates packages for state estimation, SLAM, and parameter estimation (not discussed in this paper). Section III-F lists some other relevant packages and Section III-G discusses the current state of dissemination. Links to the packages can be found at `github.com/tkoolen/julia-robotics-paper-code`.

A. RigidBodyDynamics.jl

RigidBodyDynamics.jl is a dynamics library written in pure Julia. See Fig. 2 for a basic usage example. The library implements Featherstone-style recursive algorithms for computing dynamics-related quantities [25]. RigidBodyDynamics.jl is similar in scope to RBDL [26], RobCoGen [27], and Pinocchio [28], [29], and forms a building block for applications such as simulation, trajectory optimization, and model-based control.

The main design goals for the library were to be 1) user-friendly, 2) performant, and 3) generic, in the sense that the algorithms can be called with inputs of any (suitable) scalar types, not just floating point types. Features of RigidBodyDynamics.jl include:

- singularity-free algorithms using redundant orientation representations [30];

- support for closed-loop mechanisms (loop joints), including Baumgarte constraint stabilization [31];
- a specialized Munthe-Kaas-style differential equation integrator for correct numerical integration of dynamics on manifolds that arise from *e.g.* quaternion-parameterized floating joints [32];
- a parser and writer for the URDF format [33]
- various methods for composing mechanisms and manipulating their kinematic graphs, such as conversion to a maximal coordinates representation with all spanning tree joints converted to loop joints;
- low-overhead reference frame annotations for dynamical quantities, with optional checks that ensure that reference frames match before *e.g.* adding two twists, a feature borrowed from [34];
- automatic reuse of intermediate computation results, so that *e.g.* computing the mass matrix and a Jacobian in a certain configuration is faster than computing computing each separately.

RigidBodyDynamics.jl includes algorithms for computing the mass matrix (composite rigid body algorithm), inverse dynamics (recursive Newton-Euler), the constraint Jacobian for closed-loop mechanisms, and forward dynamics (linear solve using a Cholesky decomposition of the mass matrix, employing Julia's LAPACK backend).

Support for mechanisms with contact is currently limited to penalty-based point-to-halfspace contacts using a Hunt-Crossley-Hertz contact model with (stateful) viscoelastic Coulomb friction [35], [25].

The library is competitive with RBDL in terms of performance, as demonstrated by the benchmark results in Section IV-A. RigidBodyDynamics.jl's few dependencies include StaticArrays.jl for fixed-size matrix functionality and LightXML.jl for URDF parsing.

A notable design decision was to implement the algorithms in world frame, as opposed to the conventional choice of using a body-frame implementation [25]. This allows for better reuse of intermediate results, since quantities transformed to a common coordinate system are in a sense more valuable than quantities expressed in body frame. It also allows the joints to be processed out-of-order in some of the data passes performed in the main algorithms. We exploit this fact by processing all joints *of the same type* in separate loops, which leads to significant performance improvements over an approach that uses branching or virtual functions (in C++). Although not currently implemented, this fact could potentially be exploited further using parallelism.

RigidBodyDynamics.jl also exploits Julia's metaprogramming and the ability to switch back and forth between running and compiling code, by generating specialized code for the specific joint types present in a given mechanism, an approach reminiscent of RobCoGen [27], but all in the same language.

Similar to Pinocchio [28] and Drake [2], RigidBodyDynamics.jl's dynamics algorithms are generic (type-parameterized or templated), meaning that they may be called with non-numeric input arguments. Tested examples include evaluation of the dynamics in symbolic form using the Julia bindings for SymPy [36], automatic differentiation using ForwardDiff.jl [37], rigorous interval propagation using IntervalArithmetic.jl [38], and uncertainty propagation using Measurements.jl [39]. See Section III-C for some examples.

B. MeshCat.jl

The MeshCat visualizer is designed to allow lightweight, composable visualization of robots in 3D environments. MeshCat represents a scene as a tree of geometries with associated transforms (i.e. an acyclic scene graph [40]), allowing the visualizer to mirror the tree structure of a robot mechanism. Composability is ensured by allowing multiple robots to coexist within a single scene simply by occupying different branches of the scene tree. Cameras and light sources also occupy nodes in the scene tree, allowing for unified control of the robot and the visualizer itself.

To avoid requiring any binary dependencies, the UI component of MeshCat is implemented in JavaScript using the Three.js library [41], allowing the 3D visualization to run entirely within a standard web browser. The accompanying Julia package, MeshCat.jl, provides an abstraction for communication with the visualizer from Julia and integrates with the existing ecosystem of geometry tools in Julia. The MeshCatMechanisms.jl package [42] further extends MeshCat, allowing entire RigidBodyDynamics.jl mechanisms to be automatically added to the MeshCat scene tree, allowing real-time visualization and animation of robot motions.

C. RigidBodySim.jl

RigidBodySim.jl is a simulation environment that uses RigidBodyDynamics.jl for evaluation of the dynamics, MeshCat.jl for (optional) 3D visualization, and packages from the DifferentialEquations.jl ecosystem [43] for numerical integration of the ordinary differential equations. Fig. 3 shows the RigidBodySim.jl user interface during a simulation of a mechanism inspired by Theo Jansen's Strandbeest [44] walking on flat ground.

The fact that both RigidBodyDynamics.jl and DifferentialEquations.jl are designed to work with arbitrary input types leads to some unique features not present in more conventional simulators like Gazebo [33]. For example, it is easy to use integrators for stiff differential equations that expect Jacobians of the dynamics, such as the `Rodas4P` integrator from DifferentialEquations.jl, since the integrator can simply evaluate the dynamics with an input type that implements forward-mode automatic differentiation, provided by the ForwardDiff.jl package [37]. Dedicated algorithms can compute these Jacobians even faster [29], but having this work out of the box showcases the power of Julia's support for generic programming.

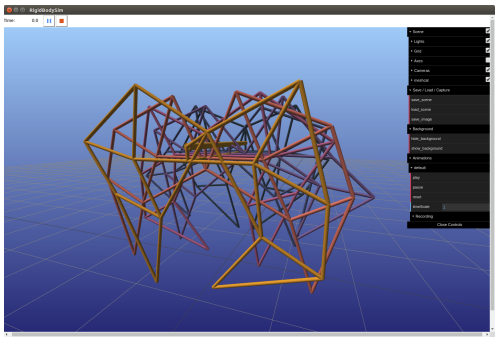

Fig. 3. Strandbeest model walking on flat ground, simulated using RigidBodySim.jl and visualized using MeshCat.jl. The model has 135 joints, including 42 loop joints and a floating joint, and simulates at around 58% of realtime. Initial resolution of the loop joint constraints was done using a gradient-based method (LBFGS) from the optimization package Optim.jl [45], which again uses ForwardDiff.jl to compute Jacobians of the kinematics. Available at `github.com/rdeits/StrandbeestRobot.jl`.

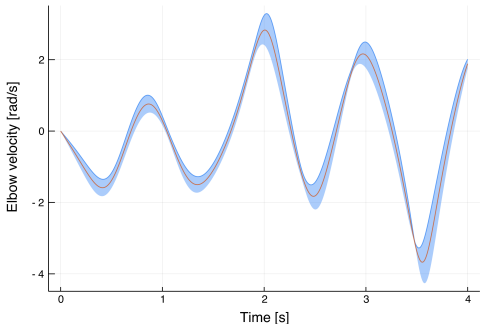

Fig. 4. Uncertainty propagation based on linear error propagation theory using Measurements.jl, an example of using RigidBodySim.jl with nonstandard input types. An unactuated double pendulum was simulated starting from initial joint angles $\mathbf{q} = (q_{\text{shoulder}}, q_{\text{elbow}}) = (0.4 \pm 0.05, 0.5 \pm 0.05)$ radians (mean $\pm$ standard deviation) and zero initial velocity. The red line shows the propagated mean of the elbow joint velocity, while the blue area shows the propagated standard deviation.

Another feature enabled by pervasive generic programming is uncertainty propagation through a simulation. The Measurements.jl package provides the `Measurement` type, which implements error propagation based on linear error propagation theory [39]. Fig. 4 shows the result of a simulation started from an uncertain initial state represented using `Measurements`.

RigidBodySim.jl inherits most of its features from its dependencies and composition with third-party packages such as ForwardDiff.jl and Measurements.jl that provide non-standard input types. Version 1.0.0 of RigidBodySim.jl consists of only 760 lines of Julia code, making it a good example of the ease with which Julia packages can be combined to create useful applications.

RigidBodySim.jl features a `PeriodicController` type, which can be used to simulate a digital controller running at a fixed rate even when using a variable time step integrator. This functionality was used to perform controller-in-the-loop simulations of the Atlas robot.

D. Parametron.jl

Parametron.jl is a framework for formulating and efficiently solving instances of a parameterized family of convex optimization problems. It is inspired by software packages such as YALMIP [46], CVX [47], and Julia's JuMP [48], which take a high-level formulation of an optimization problem and transform it to the appropriate low-level description expected by one of a number of supported solvers. Parametron.jl was built on top of MathOptInterface.jl, the same solver interface used by JuMP. However, Parametron.jl was written specifically for the task of efficiently solving optimization problems with a shared sparsity structure in a loop, by hooking into the solvers' problem modification and warm-starting functionality. In this sense, Parametron.jl is akin to CVXGEN [49]. Parametron.jl currently supports formulating continuous and mixed-integer linear and quadratic programs. With a suitable problem formulation and solver, Parametron.jl can solve optimization problems with zero dynamic memory allocation.

E. QPControl.jl

QPControl.jl implements tools to build QP-based controllers in the style of [8] and [9], using RigidBodyDynamics.jl and Parametron.jl to set up and solve the QPs. QPControl.jl exploits Parametron.jl's support for efficient problem modification, updating only the coefficients of constraints corresponding to the robot's current state rather than rebuilding the entire optimization problem at each control step. QPControl.jl provides an example momentum-based balancing controller implementation similar to [8] and uses OSQP [50] to solve the resulting QPs, but it can also be used to implement other formulations and connect with other solvers. QPControl.jl has been used to construct general model-predictive control (MPC) optimizations, including mixed-integer MPC for systems with contact [51]. We are currently developing a humanoid walking controller similar to [8] on top of QPControl.jl. See `github.com/tkoolen/julia-robotics-paper-code` for an early flat-ground walking simulation.

F. Other relevant packages

Additional tools for robotics in Julia include:

- LCMCore.jl: a Julia interface to the LCM library [52], which handles the communication between the robot's control, planning, and perception processes;
- HumanoidLCMSim.jl: a framework for simulating humanoid robots like Atlas, using separate simulation and control processes to mimic the real-time communication with the physical robot over LCM;
- RobotOS.jl (not developed by the authors): provides a Julia interface to the ROS ecosystem [53].

G. Dissemination

While the presented packages are very new, they have already been used by third parties to develop a novel

approach to control and parameter estimation [54], a trajectory optimization library [55], and soft contact models [56], as well as to teach a class at Boise State.

IV. Benchmarks

This section provides benchmark results for the dynamics algorithms in RigidBodyDynamics.jl (IV-A) and the balancing controller from QPControl.jl (IV-B). All results were obtained on a desktop machine with an Intel Core i7-6950X CPU @ 3.00GHz. The code used to generate these results can be found at `github.com/tkoolen/julia-robotics-paper-code`.

A. Dynamics algorithms

Table I shows benchmark results for some of the dynamics-related algorithms implemented in RigidBodyDynamics.jl, compared to RBDL [26]. RBDL is a reasonably well-optimized C++ library based on Eigen [14], a fast linear algebra library. RBDL was chosen as a representative example; other libraries may be faster. Results refer to a floating-base model of the version of Atlas that was used during the DARPA Robotics Challenge finals. For RBDL, we used version 2.6.0 compiled with g++ 7.3.0 using the CMake 'Release' build type, with Eigen 3.3.4. For RigidBodyDynamics.jl, we used RigidBodyDynamics.jl 1.4.0 on Julia 1.1.0 with flags `-O3` and `--check-bounds=no` and StaticArrays 0.10.2. Version information for other dependencies may be found in the associated code repository. These timings show that RigidBodyDynamics.jl is competitive with a state-of-the-art C++ implementation in terms of raw performance.

RBDL and RigidBodyDynamics.jl use the same algorithms for the mass matrix, dynamics bias, and inverse dynamics, but RBDL implements the articulated body algorithm for forward dynamics, which is not yet implemented in RigidBodyDynamics.jl. This explains the performance gap for forward dynamics. The 'Mass matrix + dynamics bias' entry is meant to demonstrate performance gains made by reuse of intermediate computation results (present in both libraries). Performance gains by reuse are somewhat higher for RigidBodyDynamics.jl compared to RBDL (35.4% vs. 13.7%), possibly due to the world-frame implementation in RigidBodyDynamics.jl (see Section III-A). Any fixed joints present in the URDF were removed by lumping together the inertias of bodies attached via fixed joints. The 'Momentum matrix' entry refers to computation of the centroidal momentum matrix [57]. RBDL does not implement an algorithm for this, but Wensing *et al.* report a runtime of 63.2 μs for their implementation of the algorithm used in RigidBodyDynamics.jl and 10.5 μs when the mass matrix is already given [58], albeit on a machine that is likely slower.

After an initial preallocation phase, none of these algorithms perform any dynamic memory allocation, thereby completely sidestepping performance penalties and jitter induced by the garbage collector.

TABLE I
Benchmark timings for dynamics-related algorithms.

Algorithm	RBDL	RigidBodyDynamics.jl
Mass matrix	8.46 μs	5.79 μs
Dynamics bias	6.02 μs	6.87 μs
Inverse dynamics	6.00 μs	5.83 μs
Mass matrix + dyn. bias	12.49 μs	8.18 μs
Forward dynamics	12.13 μs	19.46 μs
Momentum matrix	N/A	5.42 μs

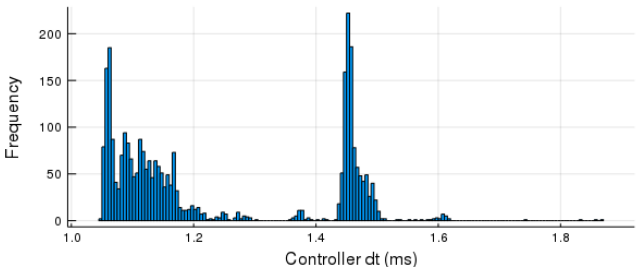

Fig. 5. Histogram of time difference between sending a state message to the controller and receiving an actuator command message from the controller.

B. Control

As noted in Section II-B, both high throughput and low jitter are important requirements for online control. Fig. 5 presents a a histogram of control times, including network communication over LCM, during a simulation of Atlas balancing using the controller from QPControl.jl. The data was collected during a 10-second simulation, with the simulation rate artificially slowed to approximately 1× realtime using periodic pauses. The controller was run at a fixed rate of 300 Hz (in terms of simulation time), which is sufficient for this type of controller [8], [9]. Note that the LCM communication layer is not a requirement, but it is representative of how we would currently control the physical robot.

During this run, the minimum, median, and maximum control times were 1.049 ms, 1.148 ms, and 1.869 ms, respectively. Although there is more jitter than we would like, all of the samples in this data-set would have made the 300 Hz deadline. We also note that this benchmark was performed on a standard desktop machine with no special precautions taken to mitigate sources of jitter (*e.g.*, no realtime kernel patches).

V. Conclusion

We demonstrated that Julia can be used to solve the two-language problem in the robotics domain, combining excellent performance with flexibility and interactivity. We presented a number of robotics-related Julia packages, together used to simulate Atlas balancing on flat ground at 2× realtime rate. Benchmarks showed that the rigid body dynamics package RigidBodyDynamics.jl is competitive with a state-of-the-art C++ implementation, and demonstrated the feasibility of using these packages for online control of a humanoid robot.

References

[1] E. Coumans and contributors, "Bullet Physics SDK: real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc." 2006.

[2] R. Tedrake and the Drake Development Team, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems," 2016. [Online]. Available: https://drake.mit.edu

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[4] J. Smith, D. Stephen, A. Lesman, and J. Pratt, "Real-time control of humanoid robots using OpenJDK," in *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 2014, p. 29.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: a fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[6] T. Koolen, J. Smith, G. Thomas, S. Bertrand, J. Carff, N. Mertins, D. Stephen, P. Abeles, J. Englsberger, S. Mccrory, *et al.*, "Summary of team ihmc's virtual robotics challenge entry," in *Humanoid Robots (Humanoids), 2013 13th IEEE-RAS International Conference on*. IEEE, 2013, pp. 307–314.

[7] S. Feng, E. Whitman, X. Xinjilefu, and C. G. Atkeson, "Optimization-based full body control for the darpa robotics challenge," *Journal of Field Robotics*, vol. 32, no. 2, pp. 293–312, 2015.

[8] T. Koolen, S. Bertrand, G. Thomas, T. De Boer, T. Wu, J. Smith, J. Englsberger, and J. Pratt, "Design of a momentum-based control framework and application to the humanoid robot atlas," *International Journal of Humanoid Robotics*, vol. 13, no. 01, p. 1650007, 2016.

[9] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous Robots*, vol. 40, no. 3, pp. 429–455, 2016.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[11] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 684–691.

[12] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel Xeon Phi*. Springer, 2014, pp. 167–188.

[13] JuliaArrays, "StaticArrays.jl," 2018. [Online]. Available: https://github.com/JuliaArrays/StaticArrays.jl

[14] G. Guennebaud, B. Jacob, P. Avery, A. Bachrach, S. Barthelemy, *et al.*, "Eigen v3," 2010. [Online]. Available: https://eigen.tuxfamily.org/

[15] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *ELPUB*, 2016, pp. 87–90.

[16] JuliaDocs, "Documenter.jl," 2018. [Online]. Available: https://github.com/JuliaDocs/Documenter.jl

[17] T. Holy, "Revise.jl," 2018. [Online]. Available: https://github.com/timholy/Revise.jl

[18] M. Törngren, "Fundamentals of implementing real-time control applications in distributed computer systems," *Real-time systems*, vol. 14, no. 3, pp. 219–250, 1998.

[19] D. F. Bacon, P. Cheng, and V. Rajan, "The metronome: A simpler approach to garbage collection in real-time systems," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2003, pp. 466–478.

[20] G. Bollella and J. Gosling, "The real-time specification for Java," *Computer*, vol. 33, no. 6, pp. 47–54, 2000.

[21] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*. IEEE, 2004, pp. 79–88.

[22] M. Paleczny, C. Vick, and C. Click, "The java hotspot (tm) server compiler," in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, vol. 1, no. S 1, 2001.

[23] L. Trevisani, S. Danisch, N. Daly, and J. Nash, "Revise.jl," 2018. [Online]. Available: https://github.com/JuliaLang/PackageCompiler.jl

[24] JuliaRobotics, "JuliaRobotics: Robotics powered by Julia," 2018. [Online]. Available: http://www.juliarobotics.org/

[25] R. Featherstone, *Rigid body dynamics algorithms*. Springer-Verlag New York Inc, 2008.

[26] M. L. Felis, "RBDL: an efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, vol. 41, no. 2, pp. 495–511, 2017.

[27] F. Marco, B. Jonas, D. G. Caldwell, and S. Claudio, "Robcogen: a code generator for efficient kinematics and dynamics of articulated robots, based on domain specific languages," *Journal of Software Engineering in Robotics*, vol. 7, no. 1, pp. 36–54, 2016.

[28] J. Carpentier, F. Valenza, N. Mansard, *et al.*, "Pinocchio: fast forward and inverse dynamics for poly-articulated systems," 2015.

[29] J. Carpentier and N. Mansard, "Analytical derivatives of rigid body dynamics algorithms," in *Robotics: Science and Systems (RSS 2018)*, 2018.

[30] V. Duindam and S. Stramigioli, "Singularity-free dynamic equations of open-chain mechanisms with general holonomic and nonholonomic joints," *IEEE Transactions on Robotics*, vol. 24, no. 3, pp. 517–526, 2008.

[31] J. Baumgarte, "Stabilization of constraints and integrals of motion in dynamical systems," *Computer methods in applied mechanics and engineering*, vol. 1, no. 1, pp. 1–16, 1972.

[32] A. Iserles, H. Z. Munthe-Kaas, S. P. Nørsett, and A. Zanna, "Lie-group methods," *Acta numerica*, vol. 9, pp. 215–365, 2000.

[33] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Proc. 2004 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, vol. 3. IEEE, 2004, pp. 2149–2154.

[34] P. D. Neuhaus, J. E. Pratt, and M. J. Johnson, "Comprehensive summary of the institute for human and machine cognition's experience with littledog," *The International Journal of Robotics Research*, vol. 30, no. 2, pp. 216–235, 2011.

[35] D. W. Marhefka and D. E. Orin, "A compliant contact model with nonlinear damping for simulation of robotic systems," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 29, no. 6, pp. 566–572, 1999.

[36] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, *et al.*, "Sympy: symbolic computing in Python," *PeerJ Computer Science*, vol. 3, p. e103, 2017.

[37] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv preprint arXiv:1607.07892*, 2016.

[38] L. Benet and D. P. Sanders, "Intervalarithmetic.jl," 2018. [Online]. Available: https://github.com/JuliaIntervals/IntervalArithmetic.jl

[39] M. Giordano, "Uncertainty propagation with functionally correlated quantities," *arXiv preprint arXiv:1610.08716*, 2016.

[40] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," *Communications of the ACM*, vol. 19, no. 10, pp. 547–554, 1976.

[41] J. Dirksen, *Learning Three. js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd, 2013.

[42] JuliaRobotics, "MeshCatMechanisms.jl," 2018. [Online]. Available: https://github.com/JuliaRobotics/MeshCatMechanisms.jl

[43] C. Rackauckas and Q. Nie, "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia," *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[44] T. Jansen. Strandbeest. Accessed 13 August, 2018. [Online]. Available: http://www.strandbeest.com
[45] P. K. Mogensen and A. N. Riseth, "Optim: A mathematical optimization package for julia," *Journal of Open Source Software*, vol. 3, no. 24, p. 615, 2018.
[46] J. Lofberg, "Yalmip: A toolbox for modeling and optimization in matlab," in *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*. IEEE, 2004, pp. 284–289.
[47] M. Grant, S. Boyd, and Y. Ye, "Cvx: Matlab software for disciplined convex programming," 2008.
[48] I. Dunning, J. Huchette, and M. Lubin, "JuMP: A modeling language for mathematical optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.
[49] J. Mattingley and S. Boyd, "Cvxgen: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.
[50] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An operator splitting solver for quadratic programs," *arXiv preprint arXiv:1711.08013*, 2017.
[51] R. Deits, T. Koolen, and R. Tedrake, "LVIS : Learning from Value Function Intervals for Contact-Aware Robot Controllers," Submitted to ICRA 2019., 2018.
[52] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight communications and marshalling," in *Intelligent robots and systems (IROS), 2010 IEEE/RSJ international conference on*. IEEE, 2010, pp. 4057–4062.
[53] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
[54] B. Landry, Z. Manchester, and M. Pavone, "A differentiable augmented lagrangian method for bilevel nonlinear optimization," *arXiv preprint arXiv:1902.03319*, 2019.
[55] B. Jackson, T. Howell, and Z. Manchester, "TrajectoryOptimization.jl," 2019. [Online]. Available: https://github.com/RoboticExplorationLab/TrajectoryOptimization.jl
[56] R. Elandt, "SoftContact.jl," 2019. [Online]. Available: https://github.com/ryanelandt/SoftContact.jl
[57] D. E. Orin and A. Goswami, "Centroidal momentum matrix of a humanoid robot: Structure and properties," *dynamics*, vol. 4, p. 6, 2008.
[58] P. M. Wensing and D. E. Orin, "Improved computation of the humanoid centroidal dynamics and application for whole-body control," *International Journal of Humanoid Robotics*, vol. 13, no. 01, p. 1550039, 2016.