`frax`: Fast Robot Kinematics and Dynamics in JAX

Daniel Morton and Marco Pavone

***Abstract*—In robot control, planning, and learning, there is a need for rigid-body dynamics libraries that are highly performant, easy to use, and compatible with CPUs and accelerators. While existing libraries often excel at either low-latency CPU execution or high-throughput GPU workloads, few provide a unified framework that targets multiple architectures without compromising performance or ease-of-use. To address this, we introduce `frax`, a JAX-based library for robot kinematics and dynamics, providing a high-performance, pure-Python interface across CPU, GPU, and TPU. Via a fully-vectorized approach to robot dynamics, `frax` enables efficient real-time control and parallelization, while supporting automatic differentiation for optimization-based methods. On CPU, `frax` achieves low-microsecond computation times suitable for kilohertz control rates, outperforming common libraries in Python and approaching optimized C++ implementations. On GPU, the same code scales to thousands of instances, reaching upwards of 100 million dynamics evaluations per second. We validate performance on a Franka Panda manipulator and a Unitree G1 humanoid, and release `frax` as an open-source library.**

I. Introduction

As robots move faster and out of purely kinematic, quasi-static settings, an understanding of dynamics is critical for computing or learning real-world-feasible planners, controllers, and safety filters. Many strong libraries exist for computing robot dynamics, with Pinocchio [1] typically noted as the fastest, though closely followed by alternatives [2–5]. Similarly, physics engines and simulators are often used in control and planning to extract dynamics properties, on CPU [6–8] and GPU [9–12].

There have also been a number of fast libraries for motion planning and inverse kinematics, including those that use GPU acceleration [13], single instruction / multiple data (SIMD) operations [14], or native JAX implementations [15]. Others focus on differential inverse kinematics, with Python interfaces to compiled libraries [16, 17].

Recent work has also focused extensively on analytical and accelerated computation of gradients of rigid body dynamics algorithms [18–20], with applications to nonlinear trajectory optimization, MPC, and optimal control more broadly [21–23].

Most of the leading dynamics libraries are written in C++, and while they often provide Python bindings, this can lead to added development challenges and overhead when integrating into Python-based workflows. The best libraries on CPU cannot be directly run on GPU, and vice-versa. Likewise, even if large-scale parallelization on GPU is not needed, C++-based libraries often lead to difficulties when using automatic differentiation (AD) in Python, whether in machine learning, robot planning, or control [24]. An ideal library would run across devices, be easy to prototype with, deploy efficiently in high-performance settings, and be compatible with AD libraries in Python, such as JAX.

Daniel Morton was supported by a NASA Space Technology Graduate Research Opportunity

Daniel Morton and Marco Pavone are with the Departments of Mechanical Engineering and Aeronautics & Astronautics, Stanford University, Stanford, CA 94305. {dmorton, pavone}@stanford.edu

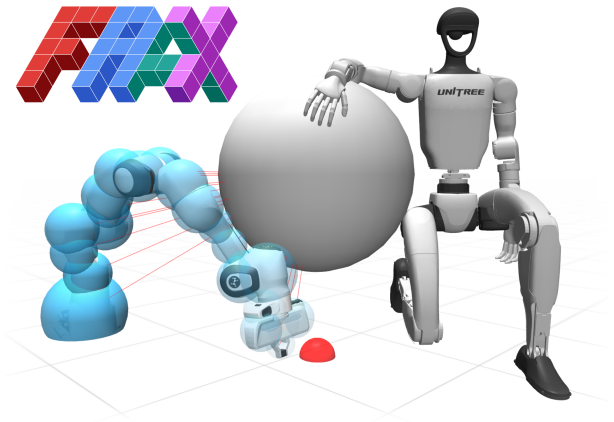

Fig. 1. **`frax` is a high-performance library for robot kinematics and dynamics on CPU, GPU, and TPU, supporting manipulators, humanoids, and more**. Via JAX, `frax` enables fast robot control and planning with automatic differentiation through arbitrary functions of the kinematics and dynamics. Shown above (Franka Panda): collision and singularity avoidance in an optimization-based inverse dynamics controller, fully through `jax.jvp`.

A. Contributions

To address this, we present `frax`, a pure-Python library for fast computation of robot kinematics and dynamics. Due to its vectorized JAX implementation of rigid-body dynamics algorithms, `frax` natively supports just-in-time (JIT) compilation, compatibility with CPU, GPU, and TPU, and automatic differentiation. In comparisons to common dynamics libraries with Python interfaces (Pinocchio, MuJoCo), `frax` is the fastest library for writing inverse-kinematic and inverse-dynamic controllers, coming close to matching the raw performance of optimized C++. The *same* code can then be deployed across thousands of parallel instances, meeting or exceeding the performance of libraries for GPU-based simulation (MJX, BRAX). `frax`'s codebase can be found at `https://github.com/danielpmorton/frax`.

II. Kinematics and Dynamics Preliminaries

We consider a robot modeled as a kinematic tree with generalized coordinates $\mathbf{q}$. The *forward kinematics* $FK(\mathbf{q})$ defines the transformation $\mathbf{T}$ of a link, joint, or other reference frame i with respect to the 0th (root) frame, denoted as ${}^0\mathbf{T}_i$. *Inverse kinematics* (IK) seeks joint configurations or velocities that achieve a desired task-space pose or motion, either *globally* (solving for $\mathbf{q}$) or *differentially* (solving for $\dot{\mathbf{q}}$).

TABLE I
FEATURE AND COMPATIBILITY COMPARISON FOR COMMON DYNAMICS LIBRARIES

	CPU	GPU/TPU	Automatic Differentiation	Interface	Compilation	Best For
Pinocchio (C++)	✓	✗	✗†	C++	✓	Fast dynamics and analytical derivatives on CPU
MuJoCo (C++)	✓	✗	✗	C++	✓	Fast simulation and dynamics on CPU
Pinocchio (Python)	✓	✗	✗	Python	✗*	Controller prototyping on CPU
MuJoCo (Python)	✓	✗	✗	Python	✗*	Simulation and controller prototyping on CPU
BRAX	✓	✓	✓	Python	✓	GPU/TPU-parallelized simulation and RL
MJX	✓	✓	✓	Python	✓	GPU/TPU-parallelized simulation and RL
`frax`	✓	✓	✓	Python	✓	Fast controller design on CPU, GPU, and TPU

† Compatible with external AD libraries such as CppAD
* Python bindings to a compiled library

Inverse dynamics (ID) computes the torque required to achieve a joint acceleration $\ddot{\mathbf{q}}$ given the joint state $[\mathbf{q}, \dot{\mathbf{q}}]$ and any external contacts or applied forces,

$$\mathbf{\Gamma} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) + \mathbf{J}^T(\mathbf{q})\mathbf{f} \tag{1}$$

Here, $\mathbf{\Gamma}$ is the vector of generalized joint torques, $\mathbf{M}(\mathbf{q})$ is the joint-space mass matrix, $\mathbf{c}(\mathbf{q}, \dot{\mathbf{q}})$ is the vector of centrifugal and Coriolis forces, $\mathbf{g}(\mathbf{q})$ is the gravity vector, and $\mathbf{J}$ is the Jacobian for any constraints or external forces $\mathbf{f}$.

The corresponding *forward dynamics* (FD) computes the acceleration of the robot $\ddot{\mathbf{q}}$ resulting from the applied joint torques and external forces, for a given joint state $[\mathbf{q}, \dot{\mathbf{q}}]$,

$$\ddot{\mathbf{q}} = \mathbf{M}^{-1}(\mathbf{q})\left[\mathbf{\Gamma} - \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) - \mathbf{g}(\mathbf{q}) - \mathbf{J}^T(\mathbf{q})\mathbf{f}\right] \tag{2}$$

Rigid body dynamics algorithms are commonly based on Featherstone's formulation [25], namely, the Composite Rigid Body Algorithm (CRBA), the Recursive Newton-Euler Algorithm (RNEA), and the Articulated Body Algorithm (ABA). CRBA computes the joint-space mass matrix $\mathbf{M}$; RNEA computes the inverse dynamics (Eq. 1) or individual terms such as $\mathbf{c}$ or $\mathbf{g}$; and ABA computes the forward dynamics as a lower-complexity alternative to Eq. 2. These methods rely on *spatial algebra*, expressing motion and force in 6-D vector spaces. Using the notation of spatial algebra, we can construct the spatial axes $\mathbf{S}$, inertias $\mathbf{I}$, velocities $\mathbf{V}$, accelerations $\mathbf{A}$, forces $\mathbf{F}$, and for CRBA, the composite inertias $\mathbf{C}$.

III. VECTORIZED RIGID-BODY DYNAMICS

While most dynamics libraries implement these algorithms through a recursive traversal of the kinematic tree, `frax` instead adopts a fully-vectorized formulation that prioritizes fine-grained parallelism, simplified tracing for automatic differentiation, and fast JIT compilation. Algorithms 1 and 2 provide an overview of the primary vectorized dynamics methods used in `frax`. Notation is based on spatial algebra and array broadcasting; a full overview is available in the Appendix (Table III).

The main difference between `frax`'s vectorized approach and traditional recursive approaches comes from the use of an ancestor mask $\mathbf{U} \in \{0,1\}^{n \times n}$ to encode the tree structure. Specifically, $\mathbf{U}_{ij} = 1$ if $j \in \text{Anc}(i)$ (we consider $i \in \text{Anc}(i)$), and for a topological sort of the robot joints, $\mathbf{U}$ is lower triangular. This construction allows for summations up and down the tree to be performed with matrix multiplication, assuming spatial values are expressed in a common reference frame. For CRBA, $\mathbf{U}$ also allows for vectorized computation of the entries of $\mathbf{M}$, while masking out any terms corresponding to non-ancestral pairs. Similar operators have appeared in older work, such as [26, 27], but are uncommon in modern libraries.

Algorithm 1 Vectorized RNEA$(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \mathbf{a}_g, {}^0\mathbf{F}_{\text{ext},1:n}) \rightarrow \mathbf{\Gamma}$

Precompute: $\mathbf{U}$ ▷ Ancestor mask
Prepare: ${}^0\mathbf{S}_{1:n}, {}^0\mathbf{I}_{1:n}$
${}^0\mathbf{V}_{1:n} = \mathbf{U}({}^0\mathbf{S}_i\dot{\mathbf{q}}_i)_{i=1}^n$
${}^0\mathbf{A}_{1:n} = (\mathbf{a}_g)_{i=1}^n + \mathbf{U}({}^0\mathbf{S}_i\ddot{\mathbf{q}}_i + {}^0\mathbf{V}_i \times {}^0\mathbf{S}_i\dot{\mathbf{q}}_i)_{i=1}^n$
${}^0\mathbf{F}_{1:n} = \mathbf{U}^T({}^0\mathbf{I}_i{}^0\mathbf{A}_i + {}^0\mathbf{V}_i \times^* {}^0\mathbf{I}_i{}^0\mathbf{V}_i - {}^0\mathbf{F}_{\text{ext},i})_{i=1}^n$
$\mathbf{\Gamma} = ({}^0\mathbf{S}_i \cdot {}^0\mathbf{F}_i)_{i=1}^n$

Algorithm 2 Vectorized CRBA$(\mathbf{q}) \rightarrow \mathbf{M}$

Precompute: $\mathbf{U}$ ▷ Ancestor mask
Prepare: ${}^0\mathbf{S}_{1:n}, {}^0\mathbf{I}_{1:n}$
${}^0\mathbf{C}_{1:n} = \mathbf{U}^T\ {}^0\mathbf{I}_{1:n}$
$\mathbf{M}_{\text{lower}} = \mathbf{U} \odot ({}^0\mathbf{S}_i^T{}^0\mathbf{C}_i{}^0\mathbf{S}_j)_{i,j=1,1}^{n,n}$
$\mathbf{M} = \text{Symmetrize}(\mathbf{M}_{\text{lower}})$

A. Complexity Tradeoffs

Crucially, by avoiding a recursive loop-based structure, we intentionally *increase* the complexity of RNEA from $O(n)$ to $O(n^2)$, and from $O(nd)$ to $O(n^2)$ for CRBA. Here, n is the number of degrees of freedom (DOFs) and d is the maximum depth of the kinematic tree. However, the overhead of redundant $O(n^2)$ computations is vastly outweighed by the performance gains from the XLA compiler's ability to make use of fine-grained parallelism within the CPU, GPU, or TPU.

In general, this aversion to loops typically leads to modest improvements to the compute time for a single evaluation on CPU, but significant improvements in JIT compilation, GPU performance, and automatic differentiation performance (on both CPU and GPU). While CPUs can efficiently handle tree traversals, GPUs favor operations with fewer sequential dependencies; thus, we take this approach to maintain strong performance across platforms. And, when applying automatic differentiation to the robot dynamics, the vectorized form

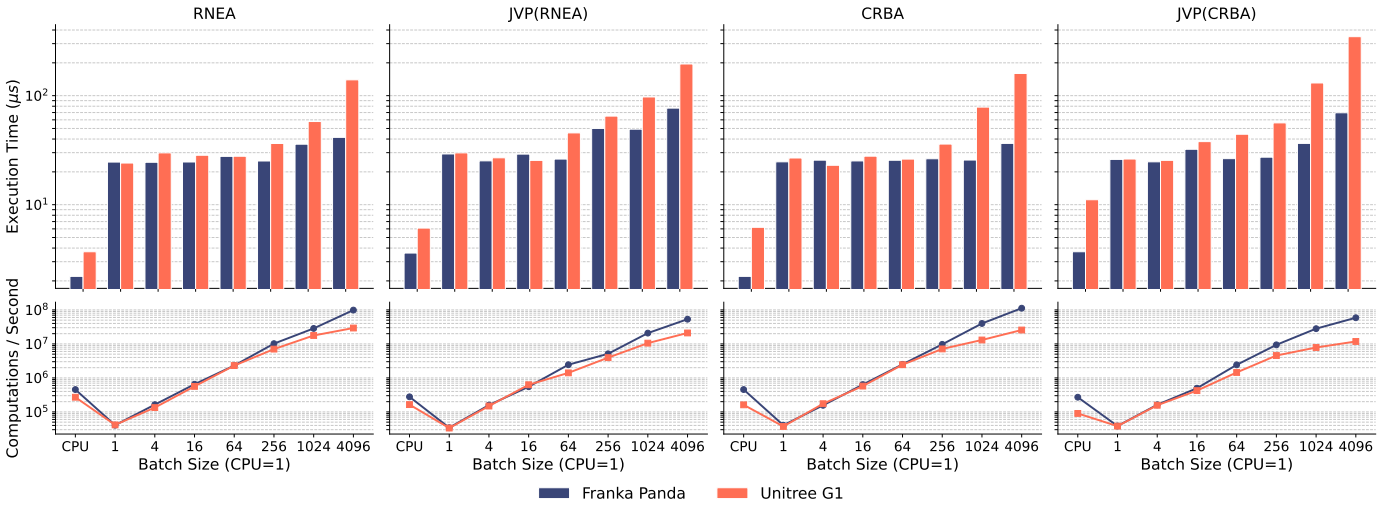

Fig. 2. **Rigid-body dynamics performance across batch sizes**. `frax`'s vectorized dynamics methods are designed for high performance on CPU and on GPU, including calls to automatic differentiation (`jax.jvp`) on these methods. All timing values for GPU include any I/O overhead.

avoids the cost of tracing gradients through long unrolled loops, for fast evaluation and compilation.

Note that only one loop is strictly required in `frax`, to compute the forward kinematics. All dynamics terms are evaluated purely with array broadcasting, as seen in Algorithms 1 and 2. And, in the special case of a serial kinematic chain, such as the Franka Panda, the FK loop can even be replaced with an `associative_scan`. In this case, both the kinematics and dynamics can be evaluated fully *loop-free*, in the sense that no Python `for` loops are unrolled in the compiled graph.

B. Batched Performance

In Fig. 2, we explore the trends in throughput across CPU and increasing batch sizes on GPU. For a single robot instance, CPU is extremely fast – much more so than on GPU, which experiences additional overhead cost, though it begins to match the total computations/second achievable once the batch size exceeds 4. As batch size increases, overhead is considerable up to approximately $N = 64$, after which the increased computational load begins to noticeably impact the total time for the batch. We also see strong performance even on very large batch sizes (4096) with throughput rates of upwards of 100 million computations per second. While this is not directly comparable with a GPU-accelerated simulator's "steps per second" metric, it does indicate that `frax`'s dynamics methods can be used in a highly-parallelized context without impeding the performance of an underlying training environment.

IV. Getting Started with FRAX

After installing either via `pip` or from source, `frax` supports any robot loaded via URDF, with specialized methods based on the kinematic structure (end-effector transforms, Jacobians, manipulability indices, ...) available via the Manipulator and Humanoid classes. `frax` has native support for the Franka Panda and Unitree G1, including tuned spherized collision models and self-collision pairs for both robots.

`frax` currently contains three examples, with more to be added: differential IK, operational space control, and safe operational space control with CBFs [24]. Each is a minimal example of designing an IK or ID controller with JAX, and can be fully wrapped in `jax.jit` for high performance. Figure 1 gives a visual representation of the CBF demo, including the spherized collision model for the Franka, an obstacle, and the end-effector target shown in red.

A. Fast JVPs for Robot Control

As a unique benefit of using JAX, we can often replace a full set of Jacobian computations with an efficient, direct evaluation of a Jacobian-vector product (JVP). In general, a Jacobian computed via AD will be slower than an analytical form[1], *unless the Jacobian does not need to be computed in the first place*. As an example, in the CBF demo, we differentiate through the model of the Franka to form the safety constraints, which can be nontrivial functions of the kinematics and dynamics (e.g., the manipulability index). The Lie derivatives required to construct the CBF constraint, such as $L_f h(\mathbf{z}) = \frac{dh}{d\mathbf{z}} f(\mathbf{z})$, is itself a JVP, and thus $\frac{dh}{d\mathbf{z}}$ does not need to be computed in full. This can be particularly beneficial for high-DOF systems like the Unitree G1, while also being simple to derive and implement.

V. A Comparison of Dynamics Libraries

To evaluate `frax`'s performance, we consider the application of writing robot controllers, including IK-based (differential IK) and ID-based (operational space control) methods. For comparison, we include compiled C++ libraries (Pinocchio, MuJoCo), their Python APIs (with NumPy for intermediate computations), and JIT-compiled JAX libraries (MJX, BRAX). These tests reflect a typical workflow, where we need multiple kinematic and dynamic terms ($\mathbf{M}$, $\mathbf{c}$, $\mathbf{g}$, $\mathbf{J}$, ...).

[1] `frax` contains analytical Jacobians for many common functions of the robot kinematics, for this reason.

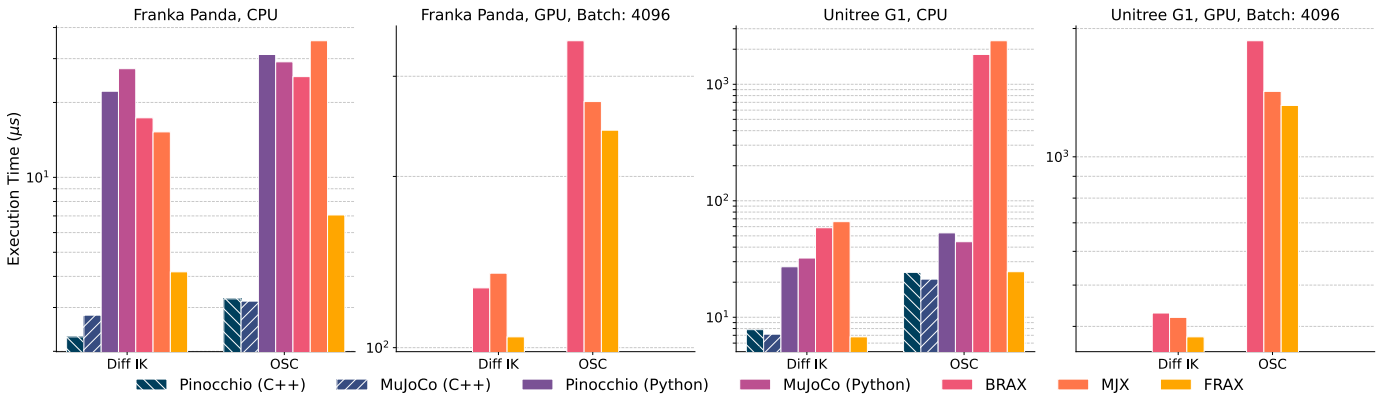

Fig. 3. **Compute timing for controllers written with `frax` and other common libraries**. On CPU, Python-based controllers are *up to 2-5x faster* with `frax` than Pinocchio or MuJoCo's bindings, coming close to matching their underlying performance on C++. And, on GPU, `frax` is on par with MJX or BRAX; enabling high-speed control for *both* single-robot deployment and massively-parallel training.

TABLE II
COMPUTE TIMES FOR FRAX CONTROLLERS ON CPU (μS)

Controller	Franka Panda	Unitree G1
Inverse kinematics [Diff. IK]	4.09	6.71
Inverse dynamics [OSC]	7.03	24.6

From Figure 3, we observe that existing JAX libraries for robot dynamics (MJX, BRAX) tend to optimize for GPU performance, leading to limited applicability for single-robot control and planning on CPU. This is especially noticeable for inverse dynamics methods on the G1, which can lead to significant spikes in controller compute times (on the order of 1 ms as opposed to 25 μs with `frax`).

In comparison to the Python APIs for Pinocchio and MuJoCo, `frax` fares very well, with around a 2-5x speedup, due to JAX's JIT compilation leading to reduced overhead, compiler fusions, and multithreaded linear algebra operations.

For the best possible performance on CPU, a controller written in C++ and compiled directly with the C++ interfaces to Pinocchio and MuJoCo will beat out `frax`, but this gap can be quite close, as seen in the G1 OSC timings (Fig. 3). This C++ approach also loses the flexibility of Python, the easy automatic differentiation enabled via JAX, and the ability to run on GPU or TPU (see Table I). A brief summary of `frax`'s CPU performance on common IK and ID controllers is also available in Table II, and further discussion on performance is included in the Appendix.

On GPU, `frax` is on par (or slightly faster) than MJX or BRAX. All three were designed for good GPU compatibility with JAX as the backend, so this is to be expected. However, `frax` has the additional benefit of significantly reduced JIT compilation times — while JIT times are of much lower importance than "hot" calls in the control loop, for these tests, `frax` compiles in 1-2 seconds versus 6-12 for MJX/BRAX. For iterative development and tuning, this speedup can significantly reduce friction for designers.

VI. CONCLUSION

In this work, we presented `frax`, a high-performance JAX-based library for robot kinematics and dynamics. Through a vectorized form of rigid-body dynamics, `frax` is fast enough for real-time, single-robot control on CPU, and scales to thousands of parallel instances on GPU or TPU. Integration with JAX allows users to automatically differentiate through complex functions of the robot kinematics and dynamics, and easily prototype in Python, without sacrificing on performance.

A. Limitations and Future Work

`frax` is more minimal than other libraries like Pinocchio and MJX, but this is by design, resulting in high performance on a restricted setting. While C++ can lead to moderately faster control rates (and potentially further improved speed with code generation [1]), `frax` provides a much more flexible Python-based AD-compatible interface that is sufficiently fast for kilohertz-rate control on hardware [24]. And, while MJX can provide a full simulation environment for training RL policies, `frax` can complement these environments (e.g. a low-level controller for an end-effector action space), and is more flexible for other parallelized applications such as planning. Additionally, `frax` does not currently focus on solving global IK problems, which is best handled by other libraries [15].

Future directions for the library include (as an inexhaustive list) (1) support for collision geometries beyond just spheres and planes, especially capsules, boxes, and ellipsoids; (2) implementations of ABA, analytical derivatives of RNEA, and direct inversion of the mass matrix [1, 18][2]; and (3) improved support for more classes of robots (e.g. quadrupeds) and more built-in collision models for common robots. And, while we have mainly focused on `frax`'s application to robot control, the dynamics and automatic differentiation support will be beneficial for a wide range of future planners and nonlinear optimization methods in robotics.

[2]Initial experiments indicate that these methods are not well suited to vectorization, and often, it seems to be more practical and faster to just call a Cholesky-based inversion or solve method on the mass matrix.

References

[1] Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse, and Nicolas Mansard. The pinocchio c++ library – a fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. In *IEEE International Symposium on System Integrations (SII)*, 2019.

[2] Martin L. Felis. Rbdl: an efficient rigid-body dynamics library using recursive algorithms. *Autonomous Robots*, pages 1–17, 2016. ISSN 1573-7527. doi: 10.1007/s10514-016-9574-0. URL http://dx.doi.org/10.1007/s10514-016-9574-0.

[3] Marco Frigerio, Jonas Buchli, Darwin G Caldwell, and Claudio Semini. Robcogen: a code generator for efficient kinematics and dynamics of articulated robots, based on domain specific languages. *Journal of Software Engineering for Robotics (JOSER)*, 7(1):36–54, 2016.

[4] Twan Koolen and contributors. Rigidbodydynamics.jl, 2016. URL https://github.com/tkoolen/RigidBodyDynamics.jl.

[5] Sabrina M. Neuman, Twan Koolen, Jules Drean, Jason E. Miller, and Srinivas Devadas. Benchmarking and workload analysis of robot dynamics algorithms. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5235–5242, 2019. doi: 10.1109/IROS40897.2019.8967694.

[6] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.

[7] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics, 2019. URL https://drake.mit.edu.

[8] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

[9] Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, Antonio Serrano-Muñoz, Xinjie Yao, René Zurbrügg, Nikita Rudin, Lukasz Wawrzyniak, Milad Rakhsha, Alain Denzler, Eric Heiden, Ales Borovicka, Ossama Ahmed, Iretiayo Akinola, Abrar Anwar, Mark T. Carlson, Ji Yuan Feng, Animesh Garg, Renato Gasoto, Lionel Gulich, Yijie Guo, M. Gussert, Alex Hansen, Mihir Kulkarni, Chenran Li, Wei Liu, Viktor Makoviychuk, Grzegorz Malczyk, Hammad Mazhar, Masoud Moghani, Adithyavairavan Murali, Michael Noseworthy, Alexander Poddubny, Nathan Ratliff, Welf Rehberg, Clemens Schwarke, Ritvik Singh, James Latham Smith, Bingjie Tang, Ruchik Thaker, Matthew Trepte, Karl Van Wyk, Fangzhou Yu, Alex Millane, Vikram Ramasamy, Remo Steiner, Sangeeta Subramanian, Clemens Volk, CY Chen, Neel Jawale, Ashwin Varghese Kuruttukulam, Michael A. Lin, Ajay Mandlekar, Karsten Patzwaldt, John Welsh, Huihua Zhao, Fatima Anes, Jean-Francois Lafleche, Nicolas Moënne-Loccoz, Soowan Park, Rob Stepinski, Dirk Van Gelder, Chris Amevor, Jan Carius, Jumyung Chang, Anka He Chen, Pablo de Heras Ciechomski, Gilles Daviet, Mohammad Mohajerani, Julia von Muralt, Viktor Reutskyy, Michael Sauter, Simon Schirm, Eric L. Shi, Pierre Terdiman, Kenny Vilella, Tobias Widmer, Gordon Yeoman, Tiffany Chen, Sergey Grizan, Cathy Li, Lotus Li, Connor Smith, Rafael Wiltz, Kostas Alexis, Yan Chang, David Chu, Linxi "Jim" Fan, Farbod Farshidian, Ankur Handa, Spencer Huang, Marco Hutter, Yashraj Narang, Soha Pouya, Shiwei Sheng, Yuke Zhu, Miles Macklin, Adam Moravanszky, Philipp Reist, Yunrong Guo, David Hoeller, and Gavriel State. Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831*, 2025. URL https://arxiv.org/abs/2511.04831.

[10] The Newton Contributors. Newton: Gpu-accelerated physics simulation for robotics and simulation research, 2025. URL https://github.com/newton-physics/newton.

[11] Genesis Authors. Genesis: A generative and universal physics engine for robotics and beyond, December 2024. URL https://github.com/Genesis-Embodied-AI/Genesis.

[12] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

[13] Balakumar Sundaralingam, Siva Kumar Sastry Hari, Adam Fishman, Caelan Garrett, Karl Van Wyk, Valts Blukis, Alexander Millane, Helen Oleynikova, Ankur Handa, Fabio Ramos, Nathan Ratliff, and Dieter Fox. curobo: Parallelized collision-free minimum-jerk robot motion generation, 2023.

[14] Wil Thomason, Zachary Kingston, and Lydia E. Kavraki. Motions in microseconds via vectorized sampling-based planning. In *IEEE International Conference on Robotics and Automation*, pages 8749–8756, 2024. doi: 10.1109/ICRA57147.2024.10611190. URL http://arxiv.org/abs/2309.14545.

[15] Chung Min Kim*, Brent Yi*, Hongsuk Choi, Yi Ma, Ken Goldberg, and Angjoo Kanazawa. Pyroki: A modular toolkit for robot kinematic optimization. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2025. URL https://arxiv.org/abs/2505.03728.

[16] Stéphane Caron, Yann De Mont-Marin, Rohan Budhiraja, Seung Hyeon Bang, Ivan Domrachev, Simeon Nedelchev, Peter Du, Adrien Escande, Joris Vaillant, Bruce Wingo, Santosh Patapati, and Daniel San José Pro. Pink: Python inverse kinematics based on Pinocchio, 2026. URL https://github.com/stephane-caron/pink.

[17] Kevin Zakka. Mink: Python inverse kinematics based on MuJoCo, February 2026. URL https://github.com/kevinzakka/mink.

[18] Justin Carpentier and Nicolas Mansard. Analytical

derivatives of rigid body dynamics algorithms. In *Robotics: Science and systems (RSS 2018)*, 2018.

[19] Brian Plancher, Sabrina M. Neuman, Thomas Bourgeat, Scott Kuindersma, Srinivas Devadas, and Vijay Janapa Reddi. Accelerating robot dynamics gradients on a cpu, gpu, and fpga. *IEEE Robotics and Automation Letters*, 6 (2):2335–2342, 2021. doi: 10.1109/LRA.2021.3057845.

[20] Brian Plancher, Sabrina M. Neuman, Radhika Ghosal, Scott Kuindersma, and Vijay Janapa Reddi. Grid: Gpu-accelerated rigid body dynamics with analytical gradients. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 6253–6260, 2022. doi: 10.1109/ICRA46639.2022.9812384.

[21] Carlos Mastalli, Rohan Budhiraja, Wolfgang Merkt, Guilhem Saurel, Bilal Hammoud, Maximilien Naveau, Justin Carpentier, Ludovic Righetti, Sethu Vijayakumar, and Nicolas Mansard. Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[22] Wilson Jallet, Antoine Bambade, Etienne Arlaud, Sarah El-Kazdadi, Nicolas Mansard, and Justin Carpentier. Proxddp: Proximal constrained trajectory optimization. *IEEE Transactions on Robotics*, 41:2605–2624, 2025. doi: 10.1109/TRO.2025.3554437.

[23] Andrea Del Prete, Nicolas Mansard, Oscar E Ramos, Olivier Stasse, and Francesco Nori. Implementing torque control with high-ratio gear boxes and without joint-torque sensors. In *Int. Journal of Humanoid Robotics*, page 1550044, 2016. URL https://hal.archives-ouvertes.fr/hal-01136936/document.

[24] Daniel Morton and Marco Pavone. Safe, task-consistent manipulation with operational space control barrier functions. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 187–194, 2025. doi: 10.1109/IROS60139.2025.11246389.

[25] Roy Featherstone. *Rigid body dynamics algorithms*. Springer, 2008.

[26] G. Rodriguez, K. Kreutz, and A. Jain. A spatial operator algebra for manipulator modeling and control. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 1374–1379 vol.3, 1989. doi: 10.1109/ROBOT.1989.100171.

[27] P. Fisette, B. Raucent, and J. C. Samin. Minimal dynamic characterization of tree-like multibody systems. *Nonlinear Dynamics*, 9(1):165–184, Feb 1996. ISSN 1573-269X. doi: 10.1007/BF01833299. URL https://doi.org/10.1007/BF01833299.

APPENDIX

TABLE III
NOTATION FOR ALGORITHMS 1 AND 2

Notation	Definition	Shape
$\mathbf{U}$	Ancestor mask: $\mathbf{U}_{ij} = 1$ if $j \in \text{Anc}(i)$ else 0	$n \times n$
$\mathbf{S}$	Spatial axes	$n \times 6$
$\mathbf{I}$	Spatial inertias	$n \times 6 \times 6$
$\mathbf{V}$	Spatial velocities	$n \times 6$
$\mathbf{A}$	Spatial accelerations	$n \times 6$
$\mathbf{F}$	Spatial forces	$n \times 6$
$\mathbf{C}$	Composite inertias	$n \times 6 \times 6$
$\mathbf{a}_g$	Spatial acceleration due to gravity	6
$\odot$	Hadamard (elementwise) product	—
$\times$	Spatial cross product (motion)	—
$\times^*$	Spatial cross product (force)	—
$(\cdot)_{i=1}^n$	Array broadcasting	—
${}^0(\cdot)$	Expressed in the root reference frame	—
$(\cdot)_{\text{lower}}$	Lower triangular	—

A. Timing

All timing values were recorded on a desktop PC with an i9-14900KF CPU, 64GB RAM, and an RTX 4090 GPU. Python methods were tested with `time.perf_counter` over 10,000 iterations, with `jax.block_until_ready()` called on any JAX functions to avoid timing the asynchronous dispatch. C++ methods were compiled with `g++ -O3` and timed via `clock_gettime()` with `CLOCK_MONOTONIC` as the source, as per [20].

B. JAX Configuration

On CPU, JAX is particularly sensitive to versions and configuration flags. We find that the best CPU performance can be achieved by using JAX version 0.4.30 and the following flags: `JAX_ENABLE_x64=1` and `XLA_FLAGS= "--xla_cpu_multi_thread_eigen=false intra_op_parallelism_threads=1"`. The XLA flags restrict the program to a single thread, which tends to work well for, e.g., robot controllers within a single ROS node. However, this may not be the best choice for all possible workloads with `frax`, and as such, it is left to the user to specify.

C. Evaluating Performance

In general, `frax` performs best when there are *many* operations (kinematics, dynamics, and otherwise) which can all be captured under a single JIT region, and this is where performance is most directly comparable to C++. In Sec. V we use controller design for benchmarking various libraries: this is a representative use-case that requires multiple operations, all of which can be computed with `frax` and JAX. As such, under JIT, the XLA compiler is able to optimize performance across the full controller evaluation. An alternative benchmark would be to compare performance on a *single* operation such as RNEA or CRBA, but this is a less typical use-case, and overhead would dominate the timing for JAX methods.