MPX: Model Predictive Control in JAX

Lorenzo Amatucci, Giulio Turrisi, Victor Barasuol, Claudio Semini

***Abstract*—In this work, we present MPX, a Python-based library for Model Predictive Control (MPC) that leverages GPU acceleration through JAX. The library is designed for speed and ease of use thanks to its JAX-based implementation: users can write simple Python code that is just-in-time compiled to achieve high-performance execution on hardware accelerators. The framework enables whole-body MPC at sufficiently high update rates for deployment on real robotic systems. System dynamics are seamlessly integrated through MuJoCo XLA. MPX has been developed with learning applications in mind; its GPU compatibility enables efficient parallelization over large batches of environments, making it well-suited for training scenarios with MPC in the loop. Additionally, the solver is fully differentiable, unlocking new possibilities for tightly integrated learning approaches. The related code can be found at https://github.com/iit-DLSLab/mpx.**

I. INTRODUCTION

MPC has become a central tool in robotics, enabling the generation of complex and dynamically consistent behaviors by solving an Optimal Control Problem (OCP) online [1], [2]. Its appeal lies in its interpretability, flexibility, and ability to explicitly handle system constraints and objectives, making it particularly effective for challenging tasks such as legged locomotion and whole-body control. However, these advantages come at a cost: MPC relies on repeatedly solving computationally demanding optimization problems at high frequency, which often limits its scalability and ease of deployment.

Modern structure-exploiting solvers tailored to OCPs, such as acados [3], crocoddyl [4], and ocs2 [5], enable fast replanning and have supported impressive results on real hardware [1], [2], [6]. Their efficiency comes from exploiting the stage-wise sparsity of the OCP, typically through Riccati-like recursions or related factorizations of the resulting KKT system. This reduces the dependence on the prediction horizon to linear time, but the dominant per-stage linear algebra still scales cubically with the size of the state and control blocks. As a consequence, increasing model fidelity, horizon length, or system dimension quickly becomes expensive, limiting the number of nodes, the richness of the model, and the number of coupled agents that can be considered online [7].

Recent advances in hardware acceleration, particularly through GPUs, have opened new opportunities to address this limitation. Unlike CPUs, GPUs are designed for high-throughput linear algebra and make it possible to exploit parallelism at several levels of the optimal control pipeline: across independent solves, along the prediction horizon, and inside the matrix operations that dominate each stage of the optimization. Prior GPU efforts have explored different pieces of this picture, for example, by translating symbolic optimal control expressions to GPU code [8], accelerating QP subproblems [9], or designing GPU-first solvers for nonlinear trajectory optimization and batched optimal control [10], [11], [12]. Among these efforts, We introduced in [7] a GPU-accelerated nonlinear MPC framework based on a primal-dual LQR solver in JAX, showing that temporal and state-space parallelization can reduce the computational complexity of the OCP solution from $\mathcal{O}(N(n+m)^3)$ to $\mathcal{O}(\log^2(n)\log N + \log^2(m))$, with n the state dimension, m the control dimension, and N the horizon length, following the finding in [13]. This scaling is particularly attractive in robotics, where long horizons and high-dimensional models are often needed to obtain robust whole-body behaviors, centralized multi-robot controllers, and large batched simulations for training.

At the same time, the integration of MPC with Reinforcement Learning (RL) has gained increasing attention. Combining model-based control with data-driven approaches can significantly improve robustness, sample efficiency, and performance. Recent work has shown that evaluating many MPC instances directly on the GPU makes MPC-in-the-loop learning practical at a scale that is difficult to achieve with CPU-only solvers [14], [15]. This highlights a broader trend: GPU-native MPC is not only about reducing solve times, but also about enabling tighter integration between optimization and learning.

In this work, we introduce MPX, a Python-based library for MPC that is both GPU-native and user-friendly. An early version of MPX was released with the publication of our previous paper [7]. Built on top of JAX, MPX allows users to define optimal control problems in simple, expressive Python while relying on just-in-time compilation to obtain efficient GPU-executable code.

Beyond ease of use, MPX is designed to support high-performance whole-body MPC suitable for real robotic systems. By leveraging GPU acceleration and parallelization, the framework scales favorably with respect to horizon length, system dimension, and batch size, making it attractive not only for rich whole-body models but also for centralized controllers and large-scale learning pipelines. Importantly, MPX is also built with learning in mind. Its fully differentiable implementation and native support for batching allow thousands of MPC instances to be evaluated in parallel, making it particularly well suited for RL, imitation learning, and other data-driven settings in which the controller remains inside the loop rather than being replaced by a purely black-box policy.

To support a wide range of use cases, the library includes two solvers: a primal-dual LQR-based method derived from our previous work, [7], designed to fully exploit GPU par-

The authors are with the Dynamic Legged Systems Laboratory, Istituto Italiano di Tecnologia (IIT), Genova, Italy. Email: name.surname@iit.it

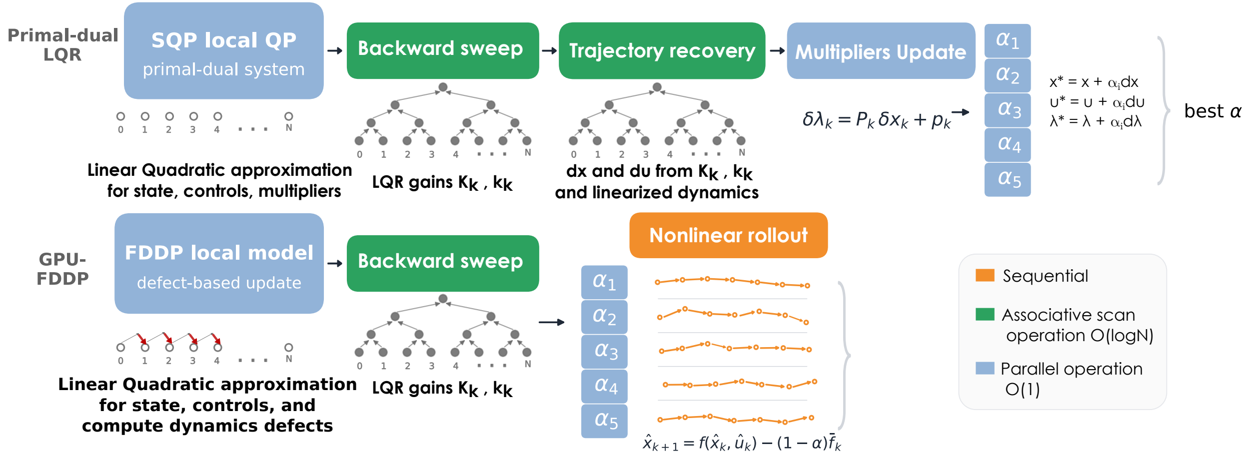

Fig. 1: Scheme of resolution strategies of both solvers, highlighted in light blue the fully parallelized operation, in green the operation that uses associative scans to reduce the computational complexity, and in orange the sequential operations.

allelism, and a GPU-adapted implementation of Feasibility-Driven Differential Dynamic Programming (FDDP) inspired by Crocoddyl [4]. Having both solvers within the same interface makes it possible to compare optimal control strategies without changing the surrounding modeling workflow.

In summary, this work contributes:

- MPX, a simple and flexible GPU-native Python library for rapid prototyping of MPC.
- a unified implementation of two complementary solver backends, namely a primal-dual LQR solver and a GPU-adapted FDDP solver, both compatible with batching and differentiable programming.

II. GPU-Accelerated Solvers

In this section, we are going to briefly introduce both GPU-accelerated solvers provided in MPX. Both target the same nonlinear OCP problem, but they differ in the way the search direction is built and in the way feasibility is handled during the iterations. The first solver is the primal-dual LQR method that we introduced in [7]. The second one is a GPU-oriented re-implementation of Feasibility-Driven Differential Dynamic Programming (FDDP), shipped with crocoddyl [4]. As will be justified later, the goal of including both is practical: the primal-dual solver is typically the fastest option, while FDDP shows better convergence. A brief overview of the solver's structures is presented in Fig. 1.

A. Primal-dual LQR

The primal-dual solver follows an Sequential Quadratic Programming (SQP) viewpoint. At each iteration, the nonlinear OCP is approximated by a local quadratic problem, and the resulting search direction is computed by solving a primal-dual system that includes both the state-control update and the associated Lagrange multipliers [7]. In a classical CPU implementation, this step is carried out through a sequential backward pass along the horizon. In MPX, this is one of the main points where GPU parallelization is exploited.

More specifically, the backward pass is reformulated using associative scans, following the temporal-parallel dynamic programming introduced by [13]. This allows information to be propagated across the horizon in logarithmic time rather than sequentially. The same idea is then used again to reconstruct the state trajectory in the forward pass. Once the local control law has been computed, the multiplier update is also performed in parallel over the horizon. Finally, the line search is evaluated over a fixed set of candidate step sizes simultaneously on the GPU, following the filter-style globalization adopted in [16]. In practice, the GPU potentials are exploited in the backward sweep, in the recovery of the trajectory, in the multiplier update, and in the line search.

This solver is the most lightweight of the two because it works directly on the primal-dual SQP subproblem and uses a linear rollout associated with the local model. For this reason, it is generally the fastest solver in the library and is often the preferred option for simpler tasks.

B. GPU-FDDP

The second solver follows the FDDP philosophy introduced in Crocoddyl [4]. As a key difference with respect to the primal-dual solver, FDDP reasons in terms of dynamics defects, namely the gaps between consecutive shooting nodes, instead of explicitly updating Lagrange multipliers.

In the original FDDP formulation, the backward pass relies on the sequential Riccati recursion. In MPX, we keep the same high-level FDDP logic, but we re-implement the backward pass using the same GPU-friendly associative-scan machinery adopted for the primal-dual solver. This means that the computation of the local gains can be parallelized across the horizon, rather than executed node by node. The line search is also adapted to the GPU: the Goldstein-based acceptance test used in FDDP is evaluated in parallel for several candidate step sizes, which avoids a sequential search and keeps the solver compatible with batched execution.

The other main distinction with respect to the primal-dual solver lies in the forward pass. While the primal-dual method applies the update through the SQP linear rollout, FDDP performs a nonlinear rollout that cannot be parallelized as before. This usually makes each iteration more expensive, but

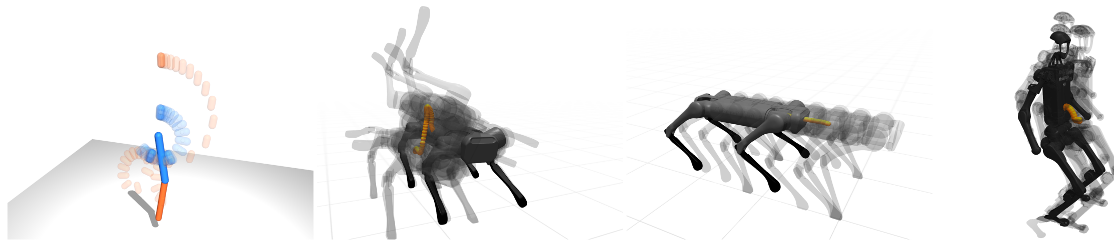

Fig. 2: Snapshot of the task tested in Fig., from left to right: an acrobot performing a swing up motion, a quadruped performing a barrel roll, a quadruped walking forward, and a humanoid jumping forward. The shadowed robots show the optimized trajectory, while the yellow line is the base trajectory.

as we could test empirically made the solver more forgiving when the optimization landscape is more challenging. For this reason, although FDDP is generally slower to compute, it can converge more reliably in situations where the faster primal-dual update shows less robustness.

III. Results

In this section, we compare the two solvers in terms of convergence and computational cost. As discussed in Sec. II, Primal-Dual LQR and FDDP mainly differ in how they handle the multiple-shooting formulation and in the rollout used to update the trajectory. These design choices lead to different behaviors: Primal-Dual LQR is typically cheaper per iteration, whereas FDDP can be more robust. To evaluate the solvers, we consider four tasks:

- **Acrobot swing-up.** The acrobot is a double pendulum with two degrees of freedom, the first passive, and the second one, connecting the two links, actuated. This system represents a non-trivial control problem, since the optimizer must first build momentum through a sequence of swings before reaching the upright equilibrium.
- **Quadruped barrel roll.** In this task, the goal is to generate a feasible barrel-roll trajectory for a quadruped robot with a predefined contact sequence. Compared with the acrobot, the system is significantly more complex, as the robot has a floating base and 12 actuated joints. The motion is particularly challenging because the robot must build angular momentum before takeoff and then reconfigure its legs in flight to reduce the moment of inertia around the rotation axis.
- **Quadruped trot.** This task uses the same quadruped model as before, but it is substantially easier. The robot is asked to track a desired base trajectory, while the contact sequence is predefined, and the foot swing references are generated heuristically.
- **Humanoid jump.** In the last task, we consider a humanoid robot with a floating base and 19 actuated joints. The robot is required to perform a forward jump with predefined contact timing. Although this motion is less demanding than the barrel roll, the optimizer needs to use the arms to stabilize takeoff and landing.

Figure 2 collects representative snapshots of the four tasks. For all of them, the solvers are initialized from a dynamically infeasible pair of inputs and state, while the initial state is randomized across runs. In Fig.3a-3b, we report the evolution of both the cost and the dynamics violation over the solver iterations. The solid curves represent the average trend, while the faded lines correspond to the individual runs. Both quantities are normalized with respect to the warm start.

Because the four tasks have different difficulties, we group them into two categories. The first group includes the acrobot swing-up and the quadruped barrel roll. In both these cases, the warm start is far from a feasible trajectory, and the solver must first recover dynamic consistency before refining the motion. As a result, these tasks require a substantially larger number of iterations to converge. This distinction is particularly clear in the dynamics-violation plots, where the defects decay more slowly than in the other two tasks. In fact, in the second group with the quadruped trot and the humanoid jump, both solvers reach dynamically feasible trajectories in a few iterations.

The trends in Fig. 3a-3b show a clear difference between the two groups. On the more challenging tasks, FDDP reaches low dynamics violation and low normalized cost in fewer iterations than Primal-Dual LQR. This behavior is consistent with what is reported by [4]. On the simpler tasks, by contrast, the two solvers perform similarly, with comparable reductions in both cost and defect. Overall, these results suggest that FDDP is advantageous when the optimization landscape is more challenging. In contrast, Primal-Dual LQR remains attractive when a good warm start is available, and a lower per-iteration computational cost is desired. As shown in Fig. 4, FDDP exhibits a noticeably larger iteration time across all tasks. This is expected, since each FDDP iteration requires a sequential nonlinear rollout, which limits the GPU exploitation.

The timings reported in the Fig. 4 were recorded with 64-bit floating-point precision enabled. This setting improves numerical accuracy at the level of individual operations and is important to ensure stable convergence on tasks with long horizons or more unstable dynamics. It should, however, be noted that such precision is not always necessary when the solver is used in a standard MPC setting, where shorter horizons are typically considered. For this reason, MPX uses 32-bit floating-point precision by default. In this condition, the iteration time of both solvers is approximately halved, enabling real-time torque control of a quadruped robot [7].

Finally, we highlight the role of MPX as a bridge between learning and classical control. The solver is fully differentiable, thanks to JAX [17] automatic differentiation, and multiple

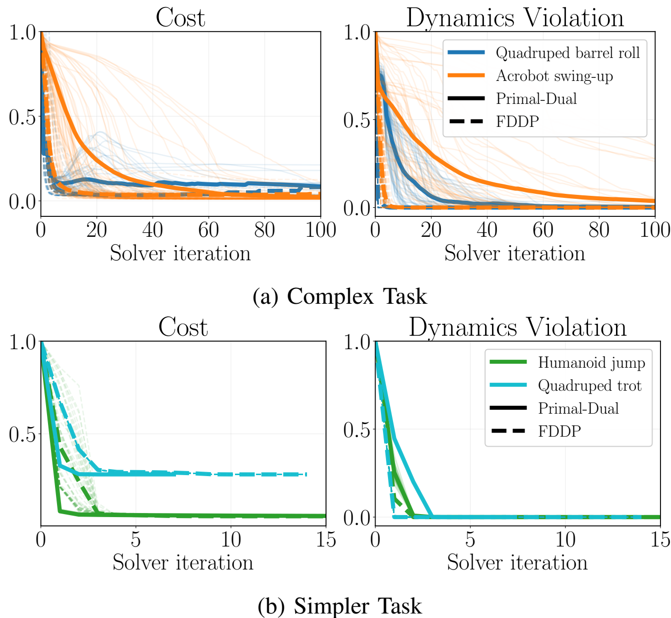

Fig. 3: Trend of the cost and dynamics violation of the solver solution over the number of iterations. a) Quadruped barrel roll and acrobot swing up motion b) Quadruped trotting and humanoid forward jumping.

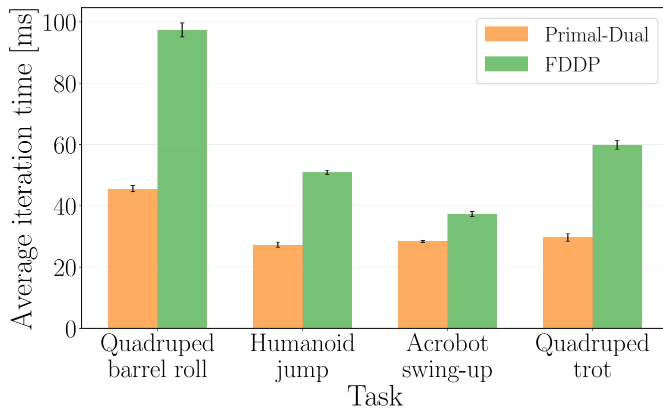

Fig. 4: Average iteration solving times for the different tasks of both solvers. The reported times are obtained with 64-bit floating point operations enabled.

problem instances can be parallelized with essentially a single line of code. In Fig. 5, we compare MPX against the CusADi-based kinodynamic formulation of [14]. While the solver architecture itself cannot be directly compared, MPX achieves a lower solve time when using the same kinodynamic model, with a solving time of 383.6 ms, compared to 504.6 ms for the CusADi implementation, corresponding to a 1.32× speedup. Interestingly, even the richer whole-body formulation remains in the same range for MPX, with a solve time of 660.5 ms, showing that a more expressive model can still be handled efficiently within the same framework. Beyond raw solving time, MPX also provides a more flexible workflow, since the solver can be modified directly at the JAX level without relying

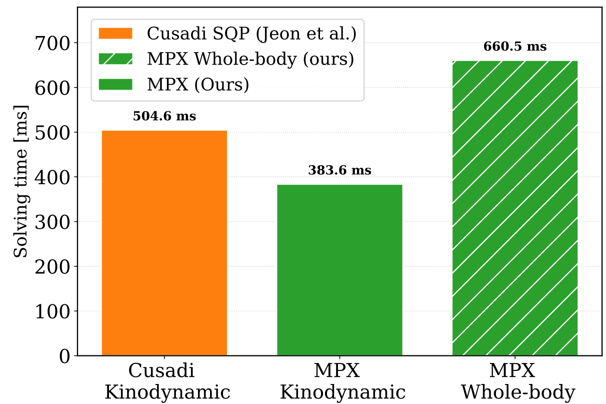

Fig. 5: Solving time for a batch of 1000 humanoid robots. In orange, the solving time report in [14] using CUSADI. In green are the results using MPX with the primal-dual solver. Solid green uses the same kinodynamic model as [14], while the dashed bar uses the whole-body formulation.

on a heavy symbolic CUDA code-generation pipeline.

IV. Conclusion

In this paper, we presented MPX, a modular JAX-based library for MPC. MPX currently provides two solver backends, namely Primal-Dual LQR and GPU-FDDP, both designed to exploit GPU parallelization as much as possible. In particular, the Primal-Dual LQR method from [7] showed remarkable performance in terms of solving time. We showed that our implementation is competitive when compared with recent GPU-based MPC implementations, showing that MPX can serve as both a practical and high-performance framework for nonlinear MPC. Beyond the solvers themselves, MPX also provides the utilities needed to build complete MPC pipelines. Although not covered in detail here due to space limitations, the library includes tools such as gait timers, reference generators, and wrapper classes for directly running receding-horizon control on quadruped and humanoid platforms. Several ready-to-use examples are already available in the repository, including Go2, Aliengo, H1, and Talos. At the current stage, one limitation of MPX is that only dynamic constraints are handled explicitly by the solver. Other equality and inequality constraints must still be introduced indirectly through barrier or penalty terms in the cost function. Another important improvement will be the implementation of a dedicated gradient computation based on the implicit function theorem [18]. While JAX automatic differentiation makes the current framework flexible and easy to use, differentiating through the full solver can become computationally expensive and, in some cases, numerically less robust. An implicit differentiation scheme would make the learning-oriented use of MPX more efficient and scalable. Overall, MPX is designed as a bridge between high-performance optimal control and modern learning pipelines. We believe that combining ease of use, modularity, and GPU efficiency in a single framework can help make nonlinear MPC more accessible for both robotics research and real-world applications.

REFERENCES

[1] C. Mastalli, S. Chhatoi, T. Corbéres, S. Tonneau, and S. Vijayakumar, "Inverse-dynamics mpc via nullspace resolution," *IEEE Transactions on Robotics*, vol. 39, no. 4, pp. 3222–3241, 2023.

[2] R. Grandia, F. Jenelten, S. Yang, F. Farshidian, and M. Hutter, "Perceptive locomotion through nonlinear model-predictive control," *IEEE Transactions on Robotics*, vol. 39, no. 5, pp. 3402–3421, 2023.

[3] V. Robin, F. Gianluca, K. Dimitris, F. Jonathan, D. Niels, Z. Andrea, N. Branimir, A. Thivaharan, Q. Rien, and D. Moritz, "acados a modular open-source framework for fast embedded optimal control," 2022.

[4] C. Mastalli, R. Budhiraja, W. Merkt, G. Saurel, B. Hammoud, M. Naveau, J. Carpentier, L. Righetti, S. Vijayakumar, and N. Mansard, "Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control," in *IEEE ICRA*, 2020.

[5] F. Farshidian *et al.*, "OCS2: An open source library for optimal control of switched systems."

[6] L. Amatucci, G. Turrisi, A. Bratta, V. Barasuol, and C. Semini, "Accelerating model predictive control for legged robots through distributed optimization," in *IEEE/RSJ IROS*, 2024, pp. 12 734–12 741.

[7] L. Amatucci, J. Sousa-Pinto, G. Turrisi, D. Orban, V. Barasuol, and C. Semini, "Primal-dual ilqr for gpu-accelerated learning and control in legged robots," *IEEE Robotics and Automation Letters*, vol. 11, no. 1, pp. 1010–1017, 2026.

[8] S. H. Jeon, S. Hong, H. J. Lee, C. Khazoom, and S. Kim, "Cusadi: A gpu parallelization framework for symbolic expressions and optimal control," *IEEE Robotics and Automation Letters*, vol. 10, no. 2, pp. 899–906, 2025.

[9] A. L. Bishop, J. Zhang, S. Gurumurthy, K. Tracy, and Z. Manchester, "Relu-qp: A gpu-accelerated quadratic programming solver for model-predictive control," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 13 285–13 292.

[10] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2024.

[11] A. Du, E. Adabag, G. Bravo, and B. Plancher, "Gato: Gpu-accelerated and batched trajectory optimization for scalable edge model predictive control," *arXiv preprint arXiv:2510.07625*, 2025.

[12] I. Tsikelis and E. Mingo Hoffman, "Massively parallel sequential quadratic programming with inverse dynamics," *arXiv preprint*, 2026.

[13] S. Särkkä and A. F. García-Fernández, "Temporal parallelization of dynamic programming and linear quadratic control," *IEEE Transactions on Automatic Control*, vol. 68, no. 2, pp. 851–866, 2023.

[14] S. H. Jeon, H. J. Lee, S. Hong, and S. Kim, "Residual mpc: Blending reinforcement learning with gpu-parallelized model predictive control," *arXiv preprint arXiv:2510.12717*, 2025.

[15] E. Adabag, M. Greiff, J. Subosits, and T. Lew, "Differentiable model predictive control on the gpu," *arXiv preprint arXiv:2510.06179*, 2025.

[16] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, 2006.

[17] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs."

[18] B. Amos, I. Jimenez, J. Sacks, B. Boots, and J. Z. Kolter, "Differentiable MPC for End-to-end Planning and Control," 2018.