Judo: A User-Friendly Open-Source Package for Sampling-Based Model Predictive Control

Albert H. Li[*,†], Brandon Hung[†], Jan Brüdigam[†], Aaron D. Ames[*]
Jiuguang Wang[†], Simon Le Cleac'h[†,‡], and Preston Culbertson[§,†,‡]

***Abstract*— Sampling-based model predictive control (MPC) is experiencing a resurgence in robotics following both recent hardware successes and advancements in parallelized physics simulation. However, to build on this progress, the robotics community needs to develop shared tools for prototyping, benchmarking, and deploying sampling-based controllers. We introduce `judo`, a software package designed to address this need. To facilitate rapid prototyping and evaluation, `judo` provides robust implementations of common sampling-based MPC algorithms and a comprehensive suite of benchmark tasks. It emphasizes usability with simple but extensible interfaces for controller and task definitions, asynchronous execution for straightforward simulation-to-hardware transfer, and a highly customizable interactive GUI for tuning controllers interactively. While the high-level library is written in Python, `judo` leverages MuJoCo as its physics backend to achieve real-time performance. We present example benchmarking results using `judo` to compare standard sampling-based controllers across its tasks. We also provide real-world case studies in deploying `judo` on hardware for two contact-rich tasks: in-hand cube rotation and quadrupedal loco-manipulation. Code available at https://anonymous.4open.science/r/judo_anonymous.**

I. Introduction

Recent advances in parallel model-based simulation tools have shown the effectiveness of *sampling-based* algorithms like predictive sampling [1], the cross-entropy method (CEM) [2], model predictive path integral control (MPPI) [3], and more for generating rich, dynamic plans for a wide variety of tasks in real time with limited resources. Subsequent work has leveraged these controllers for real-world tasks like quadrupedal and bipedal locomotion [4], [5], [6] and dexterous, in-hand object reorientation [7]. However, existing approaches each rely on specialized software stacks to implement and deploy controllers, reducing the *frictionless reproducibility* [8] needed for the rapid development and evaluation of new methods.

To address this, we present `judo`, an open-source library for developing, tuning, and deploying sampling-based MPC algorithms. `judo` gives users simple, high-performance tools for rapidly prototyping tasks and controllers, and for easily deploying controllers from simulation to hardware. The design of `judo` is motivated by three core goals:

1. Maximize Research Velocity. Echoing MuJoCo MPC (MJPC) [1], `judo`'s main goal is to accelerate research in sampling-based MPC. To achieve this, `judo` is implemented in Python, offering an effective balance of rapid prototyping and performance suitable for fast development. Its simple yet extensible interfaces are designed to facilitate extension of (and contribution to) the codebase. An integrated GUI provides real-time visualization and interactive parameter tuning, significantly shortening the development loop.

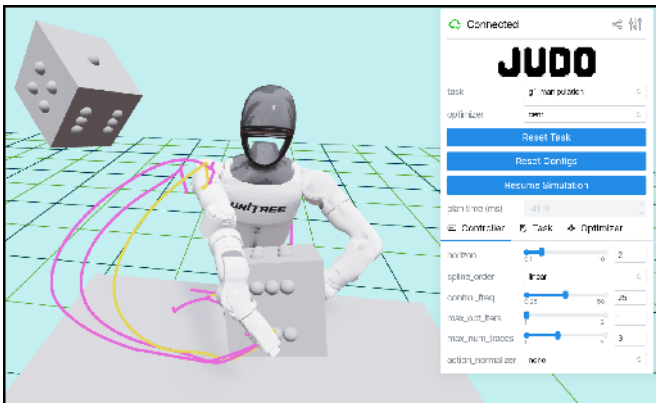

Fig. 1: The `judo` interface. The GUI is interactive, allowing users to tune parameters in real time. Dropdown menus allow switching between different tasks and controllers with ease.

2. Provide Accurate and Fast Simulation. While user-friendliness is a priority, high-performance simulation is also crucial for effective controller design and real-time applications. `judo` utilizes MuJoCo as its primary modeling backend, enabling simulation of complex scenes, geometries, and dynamics, and leverages MuJoCo's highly-optimized, multithreaded forward dynamics for online search. Crucially, `judo` is designed with a modular simulation backend interface to allow researchers to integrate alternative physics engines and custom dynamics models as needed.

3. Streamline Sim-to-Real Transfer. `judo` is engineered to minimize sim-to-real gaps and simplify the transition to hardware. The core application architecture separates control logic from physics simulation, running them in asynchronous threads. This design allows for straightforward hardware deployment: researchers can develop a specific hardware interface and substitute it for the simulation component with minimal changes to their controller code. This maximizes code reuse and provides an accurate preview of the control stack's performance prior to deployment on hardware.

Thus, this work has three contributions: (1) the open-source software package `judo`, (2) a comprehensive suite of benchmark tasks (mostly contact-rich) along with implementations of four sampling algorithms, and (3) demonstrations

* Caltech, † RAI Institute, § Cornell, ‡ Equal Advising.

of judo's deployment on two distinct hardware platforms.

II. Related Work

A. Control for Contact-Rich Tasks

Many important robotics tasks are *contact-rich*, exhibiting discontinuous, hybrid, or numerically stiff dynamics. These characteristics complicate modeling and control: methods designed for smooth systems, like gradient-based trajectory optimization, often require careful relaxations or extensions to be effective, motivating sampling-based approaches (among others) that can more naturally handle non-smooth dynamics.

Contact dynamics are often modeled using linear complementarity problems (LCPs), which are nonconvex and non-smooth. To use LCPs with gradient-based methods, prior work has modeled contact as a feasibility complementarity program, approximating their gradients using the implicit function theorem on intermediate solutions [9], [10]. Others have instead approximated the full LCP contact dynamics with a linear complementarity system solvable with ADMM to discover local contact sequences online [11], [12].

Other works [13], [14] model contact quasi-dynamically, eliminating the need to model velocities and computing a convex relaxation of the dynamics, allowing gradient-based planning via sensitivity analysis. Besides simplifying the dynamics, this approach allows "force at a distance" via a log barrier relaxation, which other works have done for contact mode discovery in manipulation and locomotion [15], [16].

Ultimately, the prior approaches hinge on how the problem is relaxed to permit gradient-based MPC to solve the task: by smoothing the dynamics, softening the contact constraints, or gradient approximation. Much existing work in contact-implicit MPC was driven by open-source differentiable simulators/solvers [17], [18], [19], [20], [21] that each make strong assumptions about the structure of the dynamics and MPC problem in tandem with these relaxations, exploiting problem structure for computational tractability.

This canonical tradeoff between problem exactness and algorithmic compatibility motivates the use of sampling-based MPC in contact-rich tasks. First, in principle, any simulator can be used for zero-order optimization with no/little relaxation, yielding more realistic solutions. Second, it is often significantly cheaper to perform only forward simulation rather than also computing or estimating gradients, which permits more real-time solutions. Third, sampling-based methods offer much greater flexibility in terms of search space and reward functions, allowing searches over discrete variables as well as non-differentiable or discontinuous reward functions. Lastly, little to no problem structure must be exploited to garner these benefits, which allows very fast prototyping and deployment for contact-rich tasks, often with higher performance than gradient-based methods. In particular, recent works have shown that sampling-based MPC can match reinforcement learning in tasks like cube rotation [7] and can achieve agile locomotion [4], [5], [6] on hardware.

Library	Lang	GUI	Sim	Multi-Planner	Async
MPPI-Generic [28]	C++	✗	Manual	✗	✗
mppi-isaac [29]	Python	✓	Isaac Gym	✗	✗
pytorch_mppi [30]	Python	✗	Manual	✗	✗
DIAL-MPC [5]	Python	✗	MJX	✗	✓
MJPC [1]	C++	✓	MuJoCo	✓	✓
hydrax [27]	Python	✗	MJX	✓	✓
Judo (ours)	Python	✓	MuJoCo+	✓	✓

TABLE I: Comparison of sampling-based MPC libraries

B. Sampling-Based MPC in Robotics

A major milestone for modern sampling-based MPC was the development of MPPI [3] for autonomous drifting using an onboard GPU. With the advent of modern deep learning, sampling-based MPC has also become standard for online planning in model-based reinforcement learning [22], [23], [24] when using data-driven dynamics. However, as discussed previously, the emergence of fast, parallel physics simulation has renewed interest in applying sampling-based MPC directly to full-order analytic models [4], [5], [6], [7].

Standard, open-source tooling has accelerated progress in reinforcement learning [25], [26], and a similar need exists for sampling-based MPC. Several libraries already target this domain (Table I), but most are narrow in scope: they may support only one planning algorithm (e.g., MPPI), provide minimal interfaces that require users to implement dynamics manually, or include only a restricted set of benchmark tasks. In short, most libraries for sampling-based MPC provide specialized implementations for particular applications, rather than general-purpose tooling for broader research.

However, two recent libraries stand out as general frameworks for research and prototyping. MJPC [1] first demonstrated the effectiveness of combining MuJoCo with sampling-based MPC and introduced GUI-based interactive tuning. Despite simulating full-order physics, MJPC achieves high performance through its native C++ implementation, but at a cost: it is structured as a standalone application, making integration with other libraries, particularly Python research codebases, difficult. hydrax [27] extends this template by adopting GPU-accelerated MJX as its backend. Thus, it can perform more parallel rollouts at once, but it also runs slower than realtime on contact-rich tasks. In comparison, judo is installable via standard Python tooling and integrates seamlessly with Python research workflows while retaining the speed of MuJoCo's multithreaded CPU rollouts.

III. The judo Package

This section details judo's design decisions. Code available at https://anonymous.4open.science/r/judo_anonymous. The open-source version will be released upon acceptance.

A. The Controller Interface

We assume sampling-based MPC controllers to have the structure shown in Alg. 1. The nominal control signal that is executed on the system is parameterized by the knots θ of some control spline, which we interpolate at time t.

At the core of the judo API are *tasks* and *optimizers*, which are sub-components of the above controller. The task

```python
# Example Task
@dataclass
class CartpoleConfig(TaskConfig):
    w_vert: float = 10.0
    w_ctr: float = 10.0
    w_vel: float = 0.1
    w_ctrl: float = 0.1

class Cartpole(Task[CartpoleConfig]):
    def __init__(self, model_path=XML_PATH):
        super().__init__(model_path, sim_model_path=XML_PATH)
        self.reset()

    def reward(self, states, sensors, controls, config, system_metadata=None):
        x, y, vel = states[..., 0], states[..., 1], states[..., 2:]
        vertical_rew = -config.w_vert * smooth_l1_norm(np.cos(y) - 1, 0.01).sum(-1)
        centered_rew = -config.w_ctr * smooth_l1_norm(x, 0.1).sum(-1)
        velocity_rew = -config.w_vel * quadratic_norm(vel).sum(-1)
        control_rew = -config.w_ctrl * quadratic_norm(controls).sum(-1)
        return vertical_rew + centered_rew + velocity_rew + control_rew

    def reset(self) -> None:
        self.data.qpos = np.array([1.0, np.pi]) + np.random.randn(2)
        self.data.qvel = 1e-1 * np.random.randn(2)
        mujoco.mj_forward(self.model, self.data)

# Example Optimizer
@dataclass
class PredictiveSamplingConfig(OptimizerConfig):
    sigma: float = 0.05

class PredictiveSampling(Optimizer[PredictiveSamplingConfig]):
    def __init__(self, config: PredictiveSamplingConfig, nu: int) -> None:
        super().__init__(config, nu)

    def sample_control_knots(self, nominal_knots):
        nn = self.num_nodes
        nr = self.num_rollouts
        sigma = self.config.sigma
        noised_knots = nominal_knots + sigma * np.random.randn(nr - 1, nn, self.nu)
        return np.concatenate([nominal_knots, noised_knots])

    def update_nominal_knots(self, sampled_knots, rewards):
        i_best = rewards.argmax()
        return sampled_knots[i_best]
```

Listing 1: Minimal example implementing a cartpole `Task` along with a predictive sampling `Optimizer` in `judo`. The simple base API allows rapid development and iteration while exposing a rich set of additional features for advanced users.

specifies the reward and system dynamics, allowing us to evaluate rollouts. The optimizer specifies the procedure for (i) sampling new control knots and (ii) updating the nominal policy given the samples and their corresponding rewards. An example task and optimizer are shown in Listing 1.

In addition to the high-level API, we provide users with several starter tasks and implementations of canonical sampling-based control algorithms, including predictive sampling, CEM, and MPPI. Finally, we provide an extensible abstract interface for allowing alternative rollout backends.

B. The GUI

The `judo` GUI is built on `viser` [31], a customizable browser-based visualizer. In `judo`, the fields of registered task or optimizer config objects are automatically parsed into the appropriate GUI elements. For example, consider the dummy optimizer config in Fig. 3. Integer and float fields are parsed as sliders, booleans as checkboxes, literals as dropdown menus, and `numpy` arrays as folders with subsliders. `judo` also implements a flexible decorator-based interface for finer control of the range and step size of sliders.

To visualize optimization iterates, `judo` allows the user to specify particular locations as "traces" in the model XML

Algorithm 1: Sampling-based MPC [7]

Input: θ, N, planner-specific parameters.
while *planning* **do**
  $x_0 \leftarrow \hat{x}(t)$; `// estimate curr state`
  for $i = 1$ *to* N ; `// multi-threaded`
  do
    $U^{(i)} \sim \pi_\theta(U)$; `// sample controls`
    $J^{(i)} \leftarrow J\left(U^{(i)}; x_0\right)$; `// eval rollout`
  $\theta \leftarrow$ `update_params` $\left(U^{(1:N)}, J^{(1:N)}\right)$;
  $u(t) \leftarrow$ `get_action`(θ, t) ; `// asynchronous`

descriptions. At runtime, the highest-performing traces are shown in the visualizer over the rollout horizon, giving a fast, qualitative understanding of the optimizer's internal state.

We emphasize that in everyday use, we find the GUI to be one of the most crucial elements of `judo`. It allows the user to perform live tuning of parameters, compare algorithms against each other, and examine an optimizer's performance across multiple tasks. Among open-source sampling-based MPC libraries, `judo` is the only one that provides a general-

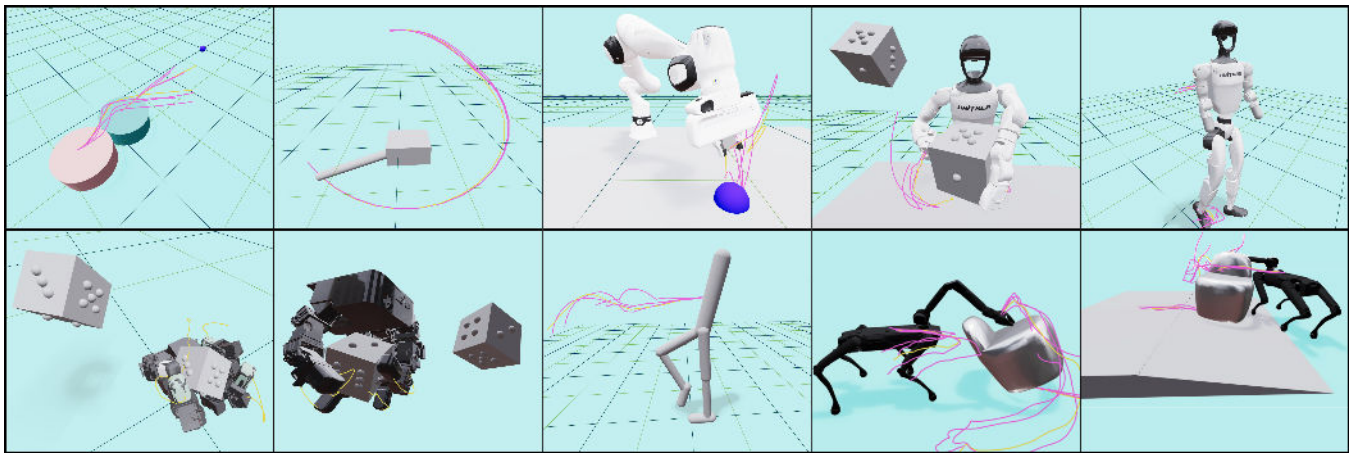

Fig. 2: The tasks in the `judo` suite. Every task except the cartpole is contact-rich, and there is a wide range of task difficulty, model complexity/morphology, and application area. For task descriptions, see Table II.

```python
@slider("num2", 0.0, 10.0, 0.1)
@dataclass
class DummyOptimizerConfig(OptimizerConfig):
    num1: int = 42  # default slider
    num2: float = 3.14  # custom slider
    num3: float = 2.71  # default slider
    checkbox: bool = True
    options: Literal["opt1", "opt2"] = "opt1"
    arr: np.ndarray = np_1d_field(
        np.array([1.0, 2.0]),
        names=["field1", "field2"],
        mins=[0.0, 1.0],
        maxs=[10.0, 20.0],
        steps=[0.1, 0.2],
    )
```

Fig. 3: A config and the corresponding automatically-generated GUI elements. Note the `slider` decorator, which allows fine-grained control over slider limits and step size.

purpose, research-friendly GUI for user interaction and tuning; MJPC includes a GUI, but extending it requires custom C++ development for each task.

C. Asynchronous Operation

The `judo` stack is modular, consisting of a "system" node (e.g., the simulator or the nodes required to run a hardware stack), a visualizer node, and a controller node. These nodes asynchronously communicate with each other, where the interprocess communication (IPC) is handled by `dora` [32], a "middleware" built on Rust with a Python interface.

We find that `dora` is substantially easier to install, set up, and use than other frameworks like ROS2, and the entire communication graph is natively specified in a standardized YAML format. Moreover, `dora` allows zero-copy reads of array data between nodes on the same device, which includes any array built on the `dlpack` specification as well as `pytorch` arrays on both CPU and GPU. If the simulation node is swapped for hardware nodes, then the same visualizer and controller nodes can be used, which facilitates low-friction transfer between simulation and hardware.

However, in the spirit of modularity, the core `judo` API is completely agnostic to user's choice of middleware. That is, one can swap `dora` for any other IPC library, and still easily use the same core functions of `judo`. To emphasize this point, the hardware demonstrations of `judo` in Sec. IV are developed using ROS2 and native CycloneDDS respectively.

D. Task Suite

We provide 10 starter tasks[1] in `judo` as examples which can be built upon by end-users or extended to custom tasks (see Fig. 2). The main focus is contact-rich tasks, allowing us to highlight some desirable qualities of sampling-based MPC algorithms. The task suite features a wide range of difficulty, from simple tasks like Cartpole or Walker, to more challenging tasks with a variety of robot morphologies, like pick and place, loco-manipulation, or upside-down cube rotation. We emphasize that the user is far from limited by the tasks that are provided, whose models, rewards, parameters, etc. can be easily modified. When possible, the task models/parameters are taken from standard sources, such as MuJoCo Menagerie [33].

E. Performance Benchmarks

In addition to `judo`'s large task suite, we also provide the ability to benchmark the performance of all available algorithms for every task. For cross-platform fairness, the

[1]The two Spot loco-manipulation tasks will be released upon acceptance, due to proprietary weights in the hybrid RL/sampling controller.

Task	Alg	Avg Reward	Avg Episode Len (s)	Successes	Failures	Metrics	Notes
Cartpole	PS	-12.38	4.54	10/10	N/A	N/A	Success upon upright and low velocity.
	CEM	-13.54	4.09	10/10	N/A	N/A	
	MPPI	-12.79	4.36	10/10	N/A	N/A	
	CMAES	-13.14	4.44	10/10	N/A	N/A	
Cylinder Push	PS	-0.17	17.02	8/10	N/A	N/A	Success upon pushing target cylinder close to goal in 30s.
	CEM	-0.15	17.55	10/10	N/A	N/A	
	MPPI	-0.23	12.15	10/10	N/A	N/A	
	CMAES	-0.24	13.27	10/10	N/A	N/A	
FR3 Pick	PS	-1.03	23.06	5/10	N/A	N/A	Success upon placing the cube and homing the arm.
	CEM	-1.03	16.87	9/10	N/A	N/A	
	MPPI	-1.10	21.48	7/10	N/A	N/A	
	CMAES	-0.96	26.29	3/10	N/A	N/A	
G1 Manipulation	PS	-2.45	30.00	N/A	0/10	total_rots: 55	Failure if cube becomes unreachable, tries to maximize number of cube rotations in 30s.
	CEM	-2.34	30.00	N/A	0/10	total_rots: 67	
	MPPI	-3.15	30.00	N/A	0/10	total_rots: 47	
	CMAES	-2.52	30.00	N/A	0/10	total_rots: 26	
G1 Stand	PS	-9.90	30.00	N/A	N/A	pct_stand: 0.76	Dropped from 2m, tries to stay standing over 30s. If falls, tries to get up. pct_stand is the ratio of time the robot is standing.
	CEM	-1.27	30.00	N/A	N/A	pct_stand: 1.00	
	MPPI	-5.64	30.00	N/A	N/A	pct_stand: 0.88	
	CMAES	-5.85	30.00	N/A	N/A	pct_stand: 0.88	
LEAP Cube	PS	-0.16	60.00	N/A	0/10	total_rots: 96	Failure if cube drops, tries to maximize number of cube rotations in 60s.
	CEM	-0.14	55.67	N/A	2/10	total_rots: 93	
	MPPI	-0.15	50.92	N/A	2/10	total_rots: 81	
	CMAES	-0.15	55.63	N/A	2/10	total_rots: 30	
LEAP Cube Down	PS	-0.14	19.74	N/A	10/10	tot_rots: 15	Same as Leap Cube but facing down.
	CEM	-0.12	45.29	N/A	5/10	total_rots: 34	
	MPPI	-0.14	30.26	N/A	9/10	total_rots: 18	
	CMAES	-0.12	35.06	N/A	8/10	total_rots: 19	
Walker	PS	-0.21	10.00	N/A	N/A	avg_vel: 0.79	Classic MuJoCo walker. Episode lasts 10s.
	CEM	-0.24	10.00	N/A	N/A	avg_vel: 0.77	
	MPPI	-0.50	10.00	N/A	N/A	avg_vel: 0.75	
	CMAES	-0.23	10.00	N/A	N/A	avg_vel: 0.77	
Spot Uprighting	PS	5.4814	9.65	9/10	N/A	N/A	Spot manipulates a chair into an upright orientation within 30s.
	CEM	15.5829	2.32	10/10	N/A	N/A	
	MPPI	17.6788	3.40	10/10	N/A	N/A	
	CMAES	14.7396	2.09	10/10	N/A	N/A	
Spot Ramp	PS	-509.05	16.82	10/10	N/A	N/A	Spot pushes a chair to the top of a ramp within 120s.
	CEM	-515.39	13.05	10/10	N/A	N/A	
	MPPI	-484.66	14.56	10/10	N/A	N/A	
	CMAES	-463.55	15.11	10/10	N/A	N/A	

TABLE II: Performance benchmarks in simulation on the `judo` task suite for all algorithms using task-specific defaults for 10 episodes per task.

benchmarks are conducted synchronously and deterministically via random seeding and all algorithms receive the same initial conditions per episode. See Table II for benchmark results using reasonable task-specific controller and optimizer settings for all tasks in the suite. In particular, we chose default values such that each task could run reasonably well with only 8 or 16 threads, runnable on most consumer-grade hardware. Each task's results should be independently interpreted, and could consist of success/failure conditions and auxiliary measures of performance besides reward. For each task, `judo` exposes the ability to track custom metrics and specify success and failure conditions so it is simple for users to update the benchmarks with new tasks.

F. Algorithmic Features

Besides the basic sampling-based MPC optimizers, `judo` has also implemented a few additional algorithmic features.

Action Space Normalization. In some systems, the scale of each dimension of the control input may be very different. In such cases, an isotropic sampling distribution may yield poor convergence (e.g., for algorithms like MPPI).

To remedy this, `judo` provides two opt-in modes for action-space normalization. First, the *min-max* normalizer assumes that each control input is bounded and normalizes the sampling scale to the interval $[-1, 1]$ along each dimension. That is, for dimension i:

$$\tilde{u}_i = \frac{u_i - u_i^{\text{lb}}}{u_i^{\text{ub}} - u_i^{\text{lb}}}, \tag{1}$$

where $u_i \in [u_i^{\text{lb}}, u_i^{\text{ub}}]$, and $\tilde{u}_i$ is the new normalized control input over which we sample.

Second, the *running Gaussian* normalizer keeps running statistics of the values along each dimension and normalizes the sampling space along each dimension by the mean and standard deviation. Mathematically, we have

$$\tilde{u}_i = \frac{u_i - m_i}{s_i}, \tag{2}$$

where (m_i, s_i) are the running mean and standard deviation of dimension i.

Noise Ramping. Another challenge in sampling-based optimization is managing the tradeoff between converging to a local solution (requires low noise) and exploring to find a globally-optimal solution (requires high noise). One solution in an MPC-style framework where optimization iterates far in the future are not executed immediately is to imbue the decision variables with a higher noise level the farther they are in the future, which we call "noise ramping"

(and resembles the "action-level annealing" of [5]). We find that even a simple linear ramping schedule greatly improves performance across all optimizers. That is:

$$u_i[k] \sim \mathcal{N}(\mu_i[k], k \cdot \gamma_{\text{ramp}} \cdot \sigma_i[k]), \tag{3}$$

where k represents the spline index along time and γ_{ramp} is the linear noise ramping factor.

Adjustable Optimizer Iterations. The quality of the MPC solution at a particular time step has variance governed by the number of available samples. Often, an optimizer can perform multiple sampling iterations at each step while still operating at a real-time rate, depending on the complexity of the model. We expose the option to adjust the number of iterations in the GUI, and find this improves the performance of highly unstable tasks, e.g., for legged systems.

G. Computational Speed

In the sampling-based MPC loop, the computational bottleneck is the nominal control knot update (and in particular, the multi-threaded rollouts of a model over a prediction horizon). To evaluate whether `judo` is viable for real-time control, we provide speed benchmarks comparing `hydrax`, MJPC, and `judo` on two tasks that are shared between the frameworks: the Walker task and the LEAP Cube task. The Walker task is a standard MuJoCo task with simple geometries that we use to sanity check speed. The LEAP Cube task is significantly more complex, involving a LEAP hand [34] rotating a cube in its hand. Because this task is extremely contact-rich, we use it to stress test rollout speed under challenging numerical conditions (see Sec. IV-A).

For each library, we used the same model descriptions and settings for the predictive sampling optimizer. We compare to `hydrax` on both CPU and GPU, using the recently-released `warp` backend for maximum speed; MJPC was timed in pure C++. Before timing, all tasks were warmed up for 1000 steps such that timings were done on "typical" states. All timings were run on an AMD EPYC 9354 32-Core Processor.

Task	Hydrax (CPU)	Hydrax (GPU)	MJPC	Judo (ours)
Walker	0.1326	0.2190	0.0026	0.0028
LEAP Cube	compile failed	0.6187	0.0324	0.0287

TABLE III: Avg. control update timing (s) over 1000 steps. The LEAP cube task failed to compile in under 1 hour for `hydrax` on CPU.

As expected, `hydrax` is many times slower on both tasks than MJPC and `judo` with full model fidelity on both CPU and GPU. Conversely, we find that `judo` has no performance degradation (and even outperforms MJPC on the Leap Cube task, which we attribute to inefficient noise sampling routines in MJPC). While the `judo` interface is written in Python, `judo` directly calls a multi-threaded rollout provided by MuJoCo, which is written natively in C and C++. Because rollout speed is by far the biggest bottleneck in sampling-based MPC, this eliminates the speed gap between using `judo` and C++ tooling like MJPC, even when the rest of the stack is written in pure Python.

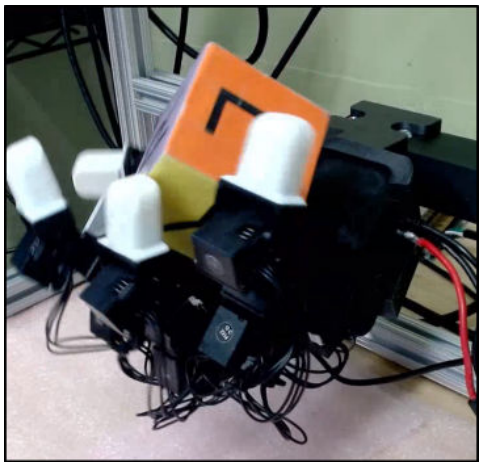

Fig. 4: Cube rotation setup: a multi-fingered hand rigidly mounted to a frame rotates a colored cube, whose pose is estimated with a vision-based keypoint tracker [7]. The goal is to manipulate the cube to as many target orientations as possible in sequence without dropping.

IV. Case Studies in Research Prototyping

To illustrate how `judo` supports the research development loop, we present two case studies on contact-rich hardware tasks. These examples *are not intended as algorithmic benchmarks,* but rather as demonstrations of how the tooling enables rapid iteration, integration, and deployment. First, we apply `judo` to the in-hand cube rotation task widely studied in dexterous manipulation [35], [36], [7]. Second, we use `judo` for quadrupedal loco-manipulation tasks in which a Spot robot employs both its body and mounted arm to manipulate objects in its environment.

A. Case Study 1: In-Hand Cube Rotation

Problem. We first consider the in-hand cube rotation task studied in DROP [7], where a dexterous hand rigidly mounted to a fixed frame must rotate a cube to successive target orientations without dropping (Fig. 4). The cube's pose is estimated using a vision-based keypoint tracker. The MPC algorithms were run on a workstation equipped with a Threadripper Pro 5995WX.

DROP optimized a reward consisting of two terms: a positional penalty encouraging the cube to remain within the hand's workspace, and a rotational penalty defined with respect to a static target orientation. While the authors demonstrated this reward was effective when using a large number of planning threads (120 on a server-grade CPU), they reported that performance degraded sharply at lower thread counts (e.g., 64). This strongly limits the practical applicability of the method for hardware setups without workstation-grade CPUs.

Development loop. To explore whether a more resource-efficient formulation could be found, we used `judo` to prototype a *tracking-based reward*. Instead of computing orientation error relative to a static target, we generated a short time-varying reference trajectory from the cube's current orientation to the goal via spherical linear interpolation at a desired angular velocity. The planner's reward then penalized deviation from this moving target, effectively transforming a

Fig. 5: A Spot robot uses its whole arm to upright a fallen chair (top). Spot pushes a chair up a ramp while making contact with multiple parts of its body (bottom). A low-level policy controls the locomotion, while `judo` samples high-level base velocity and arm commands for manipulation.

Method	Mean	Median
DROP Reward – 120 threads [7]	33.6	30.5
DROP Reward – 16 threads	0.0	0.0
Tracking Reward – 16 threads (ours)	21.2	20.0

TABLE IV: Performance on consecutive cube rotations.

long-horizon search into a sequence of local tracking problems. Designing and tuning this reward (including adjusting reward weights, optimizer noise variance, and the planning horizon) took roughly one day from ideation to deployment.

Results. We directly compared the static-goal reward from DROP with the tracking-based reward at 16 planning threads, a setting in which the baseline failed entirely. Over ten trials, the static-goal reward did not achieve a single successful rotation, while the tracking-based reward achieved an average of 21.2 consecutive rotations (median 20.0). For context, DROP with 120 threads achieved an average of 33.6 rotations. This indicates that modest reward shaping can substantially improve performance in low-resource regimes.

Role of the framework. This result was enabled by the rapid prototyping features of our system. Several aspects of the framework were critical:

- *Thread scalability experiments.* The number of planning threads is directly exposed in the GUI, allowing performance to be profiled across different thread counts without restarting the system.
- *Reward prototyping.* Reward functions could be swapped interactively in the GUI, enabling immediate comparison of static vs. tracking objectives.
- *System integration.* The task was integrated into an existing hardware stack via lightweight IPC wrappers. While `judo` ships with a default IPC backend, it can be used with other middleware (e.g., ROS2) with minimal changes, showcasing its flexibility to diverse research infrastructures.

Overall, this case study illustrates how `judo` enables researchers to quickly prototype and evaluate new reward designs for challenging contact-rich tasks, even in resource-constrained settings.

B. Case Study 2: Spot Loco-Manipulation

Problem. Our second hardware case study emphasizes how users can use `judo` to design and prototype entirely new tasks[2]. This study considers two variations of a loco-manipulation task: using a Spot robot with a mounted arm to (i) upright a fallen chair (Fig. 5 top) and (ii) push a heavy chair on casters up a ramp (Fig. 5 bottom). The task requires whole-body contact to generate sufficient force against gravity and is representative of loco-manipulation problems with many contact modes. Experiments were conducted on a workstation equipped with a Threadripper Pro 5955WX.

Development loop. In `judo`, we combined a low-level RL locomotion policy (trained to track base velocities and arm joint angles) with a high-level sampling-based planner, illustrating that `judo` can be combined with other conventional methods in robotics. A multi-term reward incentivizes the robot to stay upright, remain close to the chair, and encourages the chair to reach an upright orientation. For the ramp task, this final term instead encourages moving the chair up the ramp toward the goal. Hyperparameters were first selected in simulation and directly deployed on hardware by toggling a single entrypoint flag. `judo`'s interactive GUI enabled real-time parameter tuning directly on the robot, making it possible to identify controller parameters robust to the unmodeled compliance of the chair.

Task	Avg Episode Len	Successes	Time Limit
Spot Uprighting	14.1s	7/10	30s
Spot Ramp	93.9s	5/10	120s

TABLE V: Performance of Spot loco-manipulation tasks on hardware.

Results. Using CEM for sampling with 17 threads and a rollout horizon of 3 seconds, Spot successfully pushed the 16 kg chair up the ramp in 5/10 trials and uprighted it in 7/10 trials within time limits (see Table V). Failures were mostly due to unmodeled compliance in the chair, highlighting sampling-based MPC's current limitations for addressing the sim-to-real gap on contact-rich tasks. Nonetheless, in all trials, Spot reliably employed its arms, legs, and body together to manipulate the chair, showcasing `judo`'s ability to generate highly contact-rich whole-body motions. We emphasize that due to the size and mass of the chair, whole-body solutions leveraging contact over many parts of Spot are necessary for task completion, motivating the choice of tools like RL or sampling-based MPC.

Role of the framework. This case study illustrates how `judo` supports research and prototyping with layered control architectures (RL locomotion + MPC planning). It enabled online parameter tuning during hardware trials (crucial for addressing modeling issues with the soft chair), and provides a sampling-based framework capable of handling the discontinuous contact dynamics inherent in loco-manipulation. Moreover, `judo`'s modular API allowed a seamless transfer between the simulation-only prototyping phase and hardware

[2]The two Spot tasks were developed on an older version of `judo` with the same core API but fewer features than described in this paper.

deployment phase, facilitating rapid exploration, tuning, and hypothesis testing.

V. Conclusion

We hope that `judo` can support rapid research on sampling-based MPC algorithms in the wider robotics community by providing a hackable, extensible, and expressive framework for algorithm development and study of complex tasks with a feature-rich API. Upcoming features include integration with learned controllers, examples of hardware usage, more tasks and optimizers in the core library, and a GPU-based rollout backend.

Limitations. While we are optimistic about `judo` and sampling-based MPC, several limitations remain. First, because `judo` currently relies on CPU-based simulation, it can only run hundreds of rollouts in parallel, far fewer than the thousands common in large-scale applications [3]. We expect learning-based techniques (e.g., generative proposals, value function-guided horizon reduction, and learned dynamics [37]) will be key to making online search tractable without workstation-grade compute. Second, like most model-based methods, performance depends on accurate models and geometries. Though MuJoCo offers fast rigid-body simulation, we find performance degrades without careful modeling, underscoring the persistent sim-to-real gap. Finally, as in our first case study, closed-loop results are sensitive to reward specification. These challenges are not unique to sampling-based MPC: RL suffers from the same dependence on simulation fidelity, reward design, and transferability. Indeed, the two paradigms are closely related, as both rely on trajectory rollouts, value estimation, and reward functions. We hope `judo` can serve as a common platform for exchanging ideas and "best practices" across these communities.

References

[1] Taylor Howell, Nimrod Gileadi, Saran Tunyasuvunakool, Kevin Zakka, Tom Erez, and Yuval Tassa. Predictive sampling: Real-time behaviour synthesis with mujoco, 2022.

[2] Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization, 1999.

[3] Grady Williams, Paul Drews, Brian Goldfain, James M. Rehg, and Evangelos A. Theodorou. Information-theoretic model predictive control: Theory and applications to autonomous driving. *IEEE Transactions on Robotics*, 34(6):1603–1622, 2018.

[4] Juan Alvarez-Padilla, John Z. Zhang, Sofia Kwok, John M. Dolan, and Zachary Manchester. Real-time whole-body control of legged robots with model-predictive path integral control, 2024.

[5] Haoru Xue, Chaoyi Pan, Zeji Yi, Guannan Qu, and Guanya Shi. Full-order sampling-based mpc for torque-level locomotion control via diffusion-style annealing, 2024.

[6] John Z. Zhang, Taylor A. Howell, Zeji Yi, Chaoyi Pan, Guanya Shi, Guannan Qu, Tom Erez, Yuval Tassa, and Zachary Manchester. Whole-body model-predictive control of legged robots with mujoco, 2025.

[7] Albert H. Li, Preston Culbertson, Vince Kurtz, and Aaron D. Ames. DROP: Dexterous reorientation via online planning. In *2025 IEEE International Conference on Robotics and Automation*, 2025. Available at: `https://arxiv.org/abs/2409.14562`.

[8] David Donoho. Data science at the singularity, 2023.

[9] Simon Le Cleac'h, Mac Schwager, Zachary Manchester, Vikas Sindhwani, Pete Florence, and Sumeet Singh. Single-level differentiable contact simulation. *IEEE Robotics and Automation Letters*, 8(7):4012–4019, 2023.

[10] Simon Le Cleac'h, Taylor Howell, Shuo Yang, Chi-Yen Lee, John Zhang, Arun Bishop, Mac Schwager, and Zachary Manchester. Fast contact-implicit model-predictive control, 2023.

[11] Alp Aydinoglu and Michael Posa. Real-time multi-contact model predictive control via admm, 2022.

[12] William Yang and Michael Posa. Dynamic on-palm manipulation via controlled sliding. In *Proceedings of Robotics: Science and Systems*, July 2024.

[13] Tao Pang, H. J. Terry Suh, Lujie Yang, and Russ Tedrake. Global planning for contact-rich manipulation via local smoothing of quasi-dynamic contact models, 2023.

[14] H. J. Terry Suh, Tao Pang, Tong Zhao, and Russ Tedrake. Dexterous contact-rich manipulation via the contact trust region, 2025.

[15] Vince Kurtz, Alejandro Castro, Aykut Özgün Önol, and Hai Lin. Inverse dynamics trajectory optimization for contact-implicit model predictive control, 2023.

[16] Sergio A. Esteban, Vince Kurtz, Adrian B. Ghansah, and Aaron D. Ames. Reduced-order model guided contact-implicit model predictive control for humanoid locomotion. *arXiv preprint arXiv:2502.15630*, 2025.

[17] A. Howell Taylor, Simon Le Cleac'h, Zico Kolter, Mac Schwager, and Zachary Manchester. Dojo: A differentiable simulator for robotics. *arXiv preprint arXiv:2203.00806*, 2022.

[18] Taylor A. Howell, Kevin Tracy, Simon Le Cleac'h, and Zachary Manchester. Calipso: A differentiable solver for trajectory optimization with conic and complementarity constraints. In Aude Billard, Tamim Asfour, and Oussama Khatib, editors, *Robotics Research*, pages 504–521, Cham, 2023. Springer Nature Switzerland.

[19] Rhys Newbury, Jack Collins, Kerry He, Jiahe Pan, Ingmar Posner, David Howard, and Akansel Cosgun. A review of differentiable simulators, 2024.

[20] The MJX Developers. Mjx. `https://github.com/google-deepmind/mujoco/tree/main/mjx`, 2024.

[21] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics, 2019.

[22] Nicklas Hansen, Xiaolong Wang, and Hao Su. Temporal difference learning for model predictive control. In *ICML*, 2022.

[23] Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2786–2793, 2017.

[24] Anusha Nagabandi, Kurt Konoglie, Sergey Levine, and Vikash Kumar. Deep Dynamics Models for Learning Dexterous Manipulation. In *Conference on Robot Learning (CoRL)*, 2019.

[25] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[26] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. Rlbench: The robot learning benchmark & learning environment, 2019.

[27] Vince Kurtz. Hydrax: Sampling-based model predictive control on gpu with jax and mujoco mjx, 2024. https://github.com/vincekurtz/hydrax.

[28] Bogdan Vlahov, Jason Gibson, Manan Gandhi, and Evangelos A. Theodorou. Mppi-generic: A cuda library for stochastic optimization, 2024.

[29] Corrado Pezzato, Chadi Salmi, Max Spahn, Elia Trevisan, Javier Alonso-Mora, and Carlos Hernandez Corbato. Sampling-based model predictive control leveraging parallelizable physics simulations, 2023.

[30] University of Michigan ARM Lab. Pytorch mppi: Model predictive path integral with approximate dynamics. `https://github.com/UM-ARM-Lab/pytorch_mppi`, 2024.

[31] The NerfStudio Team. Viser: A real-time web visualizer for 3d content. `https://github.com/nerfstudio-project/viser`, 2023.

[32] Haixuan Xavier Tao and Philipp Oppermann. DORA: Dataflow-oriented robotic architecture. `https://github.com/dora-rs/dora`, 2025.

[33] Kevin Zakka, Yuval Tassa, and MuJoCo Menagerie Contributors. MuJoCo Menagerie: A collection of high-quality simulation models for MuJoCo, 2022.

[34] Kenneth Shaw, Ananye Agarwal, and Deepak Pathak. Leap hand: Low-cost, efficient, and anthropomorphic hand for robot learning, 2023.

[35] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.

[36] Ankur Handa, Arthur Allshire, Viktor Makoviychuk, Aleksei Petrenko, Ritvik Singh, Jingzhou Liu, Denys Makoviichuk, Karl Van Wyk, Alexander Zhurkevich, Balakumar Sundaralingam, Yashraj Narang, Jean-Francois Lafleche, Dieter Fox, and Gavriel State. Dextreme: Transfer of agile in-hand manipulation from simulation to reality, 2024.

[37] Jie Xu, Eric Heiden, Iretiayo Akinola, Dieter Fox, Miles Macklin, and Yashraj Narang. Neural robot dynamics. *CoRL*, 2025.