Efficient long-horizon planning and learning for locomotion and object manipulation

Victor Dhédin*
TUM, Germany
victor.dhedin@tum.de

Huaijiang Zhu*
NYU, USA
hzhu@nyu.edu

Ludovic Righetti
NYU, USA
ludovic.righetti@nyu.edu

Majid Khadiv
TUM, Germany
majid.khadiv@tum.de

Abstract: Locomotion and manipulation are difficult problems in robotics, as they involve a long-horizon decision-making problem that involves a combination of discrete and continuous decision variables. While simple end-to-end imitation and reinforcement learning have shown promise in the past few years, they generally struggle with problems that need reasoning over a long horizon, e.g., stacking objects and locomotion over highly constrained environments. In this paper, we propose a structured approach to learning long-horizon locomotion and manipulation problems. Our approach uses Monte-Carlo tree search (MCTS) to efficiently search over discrete decision variables (e.g., whcih surface to contact next) and structure-exploiting gradient-based trajectory optimization for checking the feasibility of the candidate contact plans. Since the whole process is still time-cnnsuming and cannot be done for real-time control, we propose to leverage imitation learning (in particular diffusion models) to learn a policy that can reactively generate new feasible contact sequences. We tested our whole pipeline on quadrupedal locomotion on stepping stones and fine object manipulation and show that this framework can reach real-time rates.

Keywords: contact-rich manipulation and locomotion, Monte-Carlo tree search, Diffusion models, multi-modal imitation learning

1 Introduction

The hybrid nature of intermittent contacts with the environment makes locomotion and object manipulation much more challenging problems than autonomous flying and driving. To control these systems, a controller should simultaneously decide over both continuous (e.g., contact forces) and discrete (e.g., which surface patch to make contact with next) decision variables. The two dominant approaches to control these robots in multi-contact scenarios are optimal control (OC) and reinforcement learning (RL). OC uses a forward model of the system and finds a locally optimal control input by minimizing a performance cost over a finite/infinite horizon into the future [3]. Similarly, RL finds an optimal policy (usually local) by maximizing the expected reward, but by drawing samples from rolling out control policies and interactively improving it [4]. In the past few years, both approaches have shown great success in the control of locomotion [5, 6, 7, 8, 9] and manipulation [10, 11, 12, 13, 14] systems.

OC approaches can take the constraints of the robot and the environment to ensure safety [15]. However, they need to perform intense computation at run-time and are not able to offload some of the

* Equal contribution, Most of the content in this work has been presented in [1, 2].

8th Conference on Robot Learning (CoRL 2024), Munich, Germany.

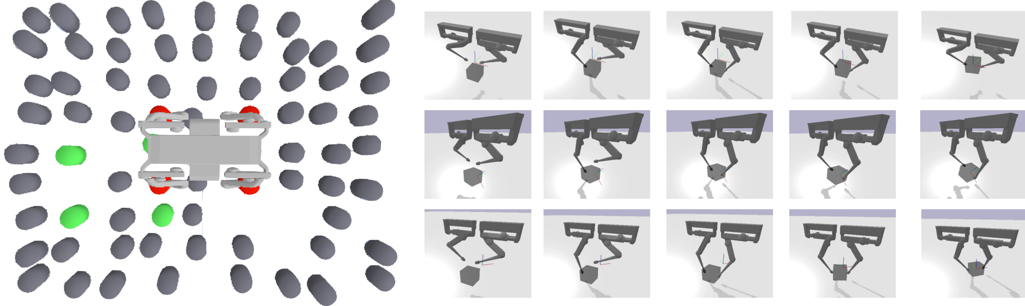


Figure 1: Left) Locomotion on challenging stepping stones, Right) contact-rich manipulation

computation for repetitive tasks. Furthermore, they do not use the data from the execution of these controllers in the real world or realistic simulation environments to improve the performance. State-of-the-art deep RL (DRL) approaches, on the other hand, can learn a policy offline in simulation which highly reduces run-time computation [16]. Furthermore, it is straightforward to randomize the uncertain parts of the models and perception pipeline to generate robust behaviors. However, these algorithms suffer from another set of fundamental problems: they require heavy reward shaping for each single task and no transfer of knowledge from one task to the next one happens; they are highly sample-inefficient which makes generation of large amounts of useful samples highly inefficient; they normally struggle with the long-horizon sequential problems with sparse rewards.

In this paper, we propose to combine the benefits of OC and DRL with supervised learning to devise a framework that efficiently generate new behaviours for robots with intermittent contact with the environment. In particular, we propose to use Monte-Carlo tree search (MCTS) to efficiently search over the discrete parts of decisions (which end-effector goes to contact and with wich surface). Given the candidates for a contact sequence, we perform a gradient-based trajectory optimization (open-loop or close-loop) to generate whole-body trajectories for the robot. We also show how supervised learning can be leveraged to speed up the whole pipeline.

2 Contact Planning using MCTS

We formalize our contact planning problem as a Markov decision process (MDP), where the state $\boldsymbol{s}$ represents the location of end-effectors in contact which we denote by a discrete index, and the action $\boldsymbol{a}$ selects the next locations for each end-effectors and brings the system to a new state $\boldsymbol{s}' = f(\boldsymbol{s}, \boldsymbol{a})$. Each state is evaluated by a reward function $r(\boldsymbol{s})$ that specifies its associated immediate reward. To solve this problem, MCTS creates a search tree $\mathcal{T} = (\mathcal{V}.\mathcal{E})$ where the set of nodes $\mathcal{V}$ contains the visited states and the set of edges contains the visited transitions $(\boldsymbol{s} \xrightarrow{\boldsymbol{a}} \boldsymbol{s}')$. Each transition maintains the state-action value $Q(\boldsymbol{s}, \boldsymbol{a})$ and the number of visits $N(\boldsymbol{s}, \boldsymbol{a})$. MCTS grows this search tree iteratively by the following steps.

Selection: Start from the root node (initial state) and select successively a child until a leaf (node that has not been expanded yet or terminal state) has been reached. If all the children of a node have already been expanded, a child is selected according to its Upper Confidence Bound (UCB) (1) that balances exploration and exploitation during the search.
Expansion: Unless the selected state from the previous step is a terminal state, its successor states are added to the tree by enumerating all possible actions. The corresponding state-action pairs are initialized with $Q(\boldsymbol{s}, \boldsymbol{a}) = 0$ and $N(\boldsymbol{s}, \boldsymbol{a}) = 0$.
Simulation: From one of the successor states, random actions are performed for a predefined number of steps to create a simulation rollout. The reward r is evaluated at the end of the simulation.
Back-propagation: The reward is then back-propagated to update the state-action value $Q(\boldsymbol{s}, \boldsymbol{a}) = Q(\boldsymbol{s}, \boldsymbol{a}) + r$ and the number of visits $N(\boldsymbol{s}, \boldsymbol{a}) = N(\boldsymbol{s}, \boldsymbol{a}) + 1$ for all the states along the node selected in the selection and expansion steps.

As shown schematically in Fig. 1, in our problem formulation, the MCTS is given the initial and final desired locations of all the robot end-effectors and is asked to compute the feasible set of contacts

that results in a successful motion. To balance between the exploration of un-visited and visited states, we consider the upper confidence bound (UCB) [17]. In the **Selection** phase, MCTS selects the action with the highest UCB score

$$U(\boldsymbol{s}, \boldsymbol{a}) = \frac{Q(\boldsymbol{s}, \boldsymbol{a})}{N(\boldsymbol{s}, \boldsymbol{a})} + c\sqrt{\frac{\log N(\boldsymbol{s})}{N(\boldsymbol{s}, \boldsymbol{a})}} \quad (1)$$

where c is a coefficient to balance exploration against exploitation and $N(\boldsymbol{s}) = \sum_{\boldsymbol{a}} N(\boldsymbol{s}, \boldsymbol{a})$ is the total number of visits for a node. The reward function that is used to update the state-action value is

$$r(\boldsymbol{s}) = W\sigma(\frac{1}{N_e}\sum_{j=1}^{n_e}(1 - \frac{||\mathbf{c}_W^j - {}^g\mathbf{c}_W^j||_2}{d_{max}})) \quad (2)$$

where $\mathbf{c}_W^j$ is the contact location in world frame of the jth end-effector at state $\boldsymbol{s}$, ${}^g\mathbf{c}_W^j$ is the desired goal location for the jth end-effector in world frame, and N_e is the total number of end-effectors. d_{max} is the maximum distance between two contact patches in the map. $\sigma : [0, 1] \rightarrow [0, 1]$ is a function that shapes the reward. $W \in \{-1, 1\}$ is a success indicator, by default set to 1 when the goal state is not reached.

To check the feasibility of the contact sequence, we takes the biconvex structure of the dynamics between the interaction forces and the center of mass [9, 1] into account to develop an efficient solver based on alternating direction method of multipliers (ADMM). Please refer to Appendix A for the heuristics used for locomotion and manipulation problems.

3 Supervised learning

While efficient, our MCTS together with the OC-based feasibility check cannot be run in real-time. To enable the robot to reactively select the next feasible contact given the current ones, we learn a neural network to imitate the MCTS. We have slightly different supervised learning settings for locomotion and manipulation that we outline in the following of this section.

3.1 Learning locomotion policies

While MCTS admits a natural extension of a learnable value function and action probability prior [18, 19], for the locomotion problem, we decided not to adopt this methodology despite its success in game-play for two reasons. First, in contrast to generic game-play, locomotion tasks on different maps (e.g. varying locations and numbers of stepping stones) are likely to have different states and action space; an MCTS trained on a specific map does not generalize to other environment maps. Second, some other sensory inputs that are not modeled in the MCTS state space may provide additional information (eg. base velocity) on *if* and *where* to make the next contact. Therefore, we take a direct action imitation learning perspective and treat the MCTS as an algorithmic demonstrator, whose behavior will be cloned by a neural network policy. We collect data of the dynamically feasible solutions discovered by the MCTS together with the contact locations of the simulation environment. This dataset is then used to train a neural network in a supervised fashion. A schematic structure of the network is shown in Fig. 2-left.

Our learning problem structure is a selection procedure as the policy should ideally return contact locations that are given as input. While this can be achieved using a projection function, some network architectures are suited to this task such as the Pointer-Network architecture (6) that we consider as a candidate. Additionally, our dataset is multi-modal as MCTS provides different contact sequences for the same start and goal contact locations in a given environment. It is not possible to represent such multi-modal data distribution with a conventional uni-modal policy class as the model could collapse to one of the modes or an average over several modes (see Fig 3). Therefore, we consider Denoising diffusion probabilistic models (DDPMs) [20] (cf. 6), as another potential candidate, since they are theoretically grounded to handle multi-modality [21] and is practically verified for some robotic applications [22] [23]. We also consider multi-layer perceptron (MLP)

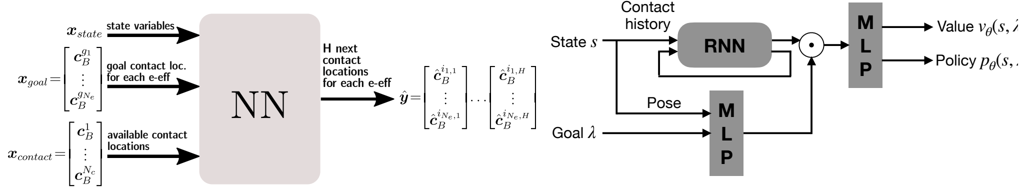


Figure 2: left) Final policy network for the locomotion problem. For the NN block, we used MLP, Pointer network and diffusion model, right) schematic diagram of the policy-value network architecture for manipulation

architecture as a baseline. Please refer to Appendix B for further discussion on the structure of the Pointer-Networks and DDPMs.

3.2 Learning manipulation policies

One key difference between our manipulation task and generic game-play is that our dataset is highly imbalanced—many contact sequences explored by the MCTS are dynamically infeasible, resulting in zero reward. Directly training on such a dataset leads to underestimation of the value function. Instead, we first train on the dataset $\mathcal{D}$ a binary classifier $C_\phi(s)$ with logistic regression where dynamically feasible samples are given more weights. At inference time, a state is only fed into the policy-value network if the classifier labels it as dynamically feasible; otherwise, we output zero value $v_\theta(s) = 0$ and uniformly distributed action probability $p_\theta(s,a) = 1/|\mathcal{A}(s)|$. This feasibility classifier screens out dynamically infeasible contact sequences before the MCTS completes the search, thus greatly improves search efficiency.

Note that each MCTS instance only searches for the contact sequence for a given object motion ξ, thus the rewards are motion-specific. If we were to learn from the data collected for this object motion only, it is unlikely that the network would generalize to other motions. Thus, we generate multiple object motions in the training phase and additionally input an intermediate goal variable to the network. It is defined as the difference between the current desired pose and the one h steps in the future $\lambda(t) = q(t+h) \ominus q(t)$, where $\ominus$ denotes subtraction in $\mathrm{SE}(3)$. Figure 2-right depicts the policy-value network architecture: it takes as inputs the state s and the goal λ, and outputs the goal-conditioned value $v_\theta(s,\lambda)$ and action probabilities $p_\theta(s,\lambda)$. Since the sequence of contact surfaces has varying lengths, we use a RNN to encode this information and concatenate it with the pose and the goal processed by a MLP. Due to the usage of the feasibility classifier mentioned above, we only train our policy-value network on a subset $\mathcal{D}_+ \subseteq \mathcal{D}$ with positive samples $\mathcal{V}_+ = \{s \in \mathcal{V} | \bar{v}(s) > 0\}$ to avoid underestimating the value function.

4 Results and Discussion

In this section we present the results for multiple gaits of Solo12 quadruped robot [24] on highly constrained random stepping stone environments. Then, we present simulation and experiment results for contact-rich object manipulation problem.

4.1 Locomotion results

In this section, we first compare the capability of different neural network architectures to handle multi-modality in a simple example scenario (4.1.1). Then, we present the result of using the diffusion model to learn a policy that can output contact plans from the state of the robot and the environment (4.1.2). Finally, we show the effectiveness of our learned policy in selecting the feasible contact patches as the environment and goal change dynamically.

4.1.1 Choice of network architecture

To systematically compare the performance of different neural network architectures, we design a simple learning problem. In this problem, the dataset, *multimod*, consists of two different contact sequences to reach the same goal (in diagonal) starting from the same position in the same non-randomized environment. One plan goes first straight and left, the other one goes left and then straight to reach the goal. We record 50 randomized runs for each plan for a total of 700 samples in the dataset (procedure described in Appendix B).

Since we perturb the states while collecting data, the dataset is not strictly multi-modal. We would like to evaluate how this partial multi-modality affects the trained policy to reproduce the variety of contact plans of the dataset. Generating diverse contact plans is beneficial as some might be more relevant in specific states. We compare 3 different model architectures presented in Sec. 3.1 on this dataset. The training parameters are detailed in the Appendix B. As shown in Fig. 3, only the diffusion model was able to reproduce the two possible modes of the solution. On 20 simulations, the diffusion model reached the goal by going 12 times first to the right and 8 times first up (we used a different seed each time to generate the initial noise). MLP and Pointer-Network collapsed to one of the solutions, as can be seen on the base trajectories in Fig. 3. As we would like to benefit from the multiple feasible solutions of MCTS for a given goal, we focus on diffusion models for the navigation task in the following section. Note that, for the diffusion model, the projection error $||\hat{\boldsymbol{c}}_B^{i_j,h} - p_{\mathcal{C}}(\hat{\boldsymbol{c}}_B^{i_j,h})||_2$ is 5.4 mm on average and at a maximum 2.2 cm for the predicted jump locations, which is approximately 3 times less than the half distance between two stepping stones.

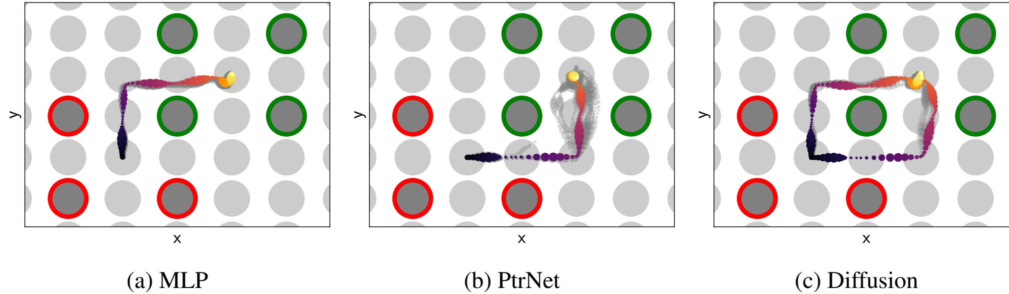
4.1.2 Learning to navigate on stepping stones

In this subsection, we aim to qualitatively show that the learned policy can reactively generate feasible contact plans when the environment or the goals change on the fly. To do that, we generated $N_{\text{env}} = 80$ different environments, but this time with less randomization as we focus on the ability of the policy to plan and select the right contact locations: $N_{\text{removed}} = 12$, $\alpha_x = \alpha_y = \alpha_h = 0$. We expect the results to be reproducible in a randomized environment with a larger dataset. The number of goals for each environment N_{goals} is 8, $N_{\text{paths}} = 3$ and $N_{\text{repeat}} = 5$. Our training dataset, *multigoal*, has 52224 samples in total. Our test environment has been generated in the same conditions on $N_{\text{env}} = 20$ different environment. We tested our policy on test goals for which MCTS found a feasible contact plan. Our training procedure and hyperparameters are similar to [23] and are detailed in the appendix.

As done in [23], we used a DDIM [25] approach to decouple the number of denoising iterations in training and inference. Taking a similar number of steps in the forward and backward diffusion process usually leads to better results but is time-consuming. For 15 steps in the backward process and above, the policy achieved approximately a similar success rate (our policy was trained on $T = 50$ diffusion steps). Therefore, we used 15 denoising iterations in the experiments. It takes 70 ms for the policy to be evaluated on an AMD Ryzen 5 5625u CPU.

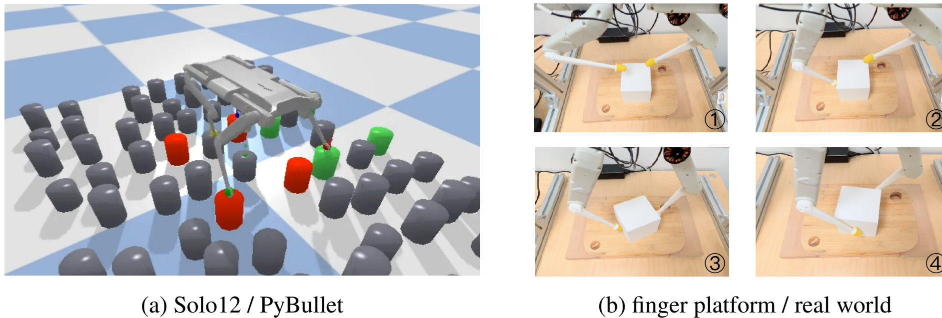
Static environments Our policy has been trained to output different contact patches for the next jump as our dataset contains multi-modal samples. Therefore, by combination, the final full-length contact plan is likely to differ from the ones of MCTS. This is confirmed as 40% of the successful contact plans were not in the training dataset when replaying on the training environment. However, since a maximum of 3 different paths are recorded for each goal in a given environment, it could lead to a poorer success rate as the policy might end up out of the training distribution. When replayed in the same conditions 20 times for a goal in a corner of the environment, our trained policy reached the goal all the times in 14 different ways. Some examples can be seen in the accompanying video. This confirms the results of Sec. 4.1.1 with a policy trained on a more diverse dataset.

Dynamic environments Now, we proceed to evaluate how well the policy can be used in a dynamic environment. To do that, we randomly remove two contact locations from the environment before each jump while the robot is reaching a goal. The removed stepping stones are chosen among



(a) MLP (b) PtrNet (c) Diffusion

Figure 3: Base position (x, y) for 20 randomized runs with models trained on the *multimod* dataset. Starting contact locations are circled in red and goal contact locations in green. The base position is recorded every 20ms. The size of the circles represents the height of the base. Only the diffusion model 3c is able to reproduce both two contact sequences of the training dataset.



(a) Solo12 / PyBullet (b) finger platform / real world

Figure 4: a) Solo12 navigation on stepping stones [2], b) real-world demonstration of object manipulation with a two-finger platform [1].

the ones found by MCTS. This way, it is more likely to remove a stepping stone that would have been initially selected, which shows the ability of the policy to replan reactively. This task is challenging, as removing stepping stones could make the robot jump into a position that is bound to fail or that is out of the training data distribution. Our policy sporadically succeeded on this task (22% of the time on 50 trials on a goal in diagonal). Additionally, our trained policy was also able to perform navigation to reach some user-defined changing goals in a new environment with 12 removed stepping stones.

5 Manipulation results

We conduct experiments in simulation and on real hardware to show that our method 1) is capable of finding high quality dynamically feasible solutions much faster than a MIQP baseline, 2) scales to long-horizon tasks even when trained only on data collected from shorter-horizon tasks. Throughout all experiments, we consider a manipulator composed of two modular robot fingers similar to the ones used in [26] and a $10\,\text{cm} \times 10\,\text{cm} \times 10\,\text{cm}$ cube with mass $m = 0.5\,\text{kg}$ on an infinitely large plane. The cube and the plane have the same friction coefficient $\mu = \mu_e = 0.8$. We consider one contact surface for each face of the cube except for the bottom one; each contact surface is a $8\,\text{cm} \times 8\,\text{cm}$ square to avoid contact locations at the corner. The object motion trajectory is generated with spline interpolation in SE(3) between the initial object pose and a desired pose parameterized as the following primitives:

- S on the xy-plane by $-10\,\text{cm}$ to $10\,\text{cm}$
- SC on the xy-plane by $-5\,\text{cm}$ to $5\,\text{cm}$ with a rotation about the z-axis by $-45°$ to $45°$
- R about the z-axis by $-90°$ to $90°$
- L along the z-axis by $0\,\text{cm}$ to $10\,\text{cm}$, and

Table 1: Task performance for object motion primitives. Values ≤ 0.005 are rounded to zero.

	Method	**Time** [s]		**Error** [N, N $\cdot$ m]	
		Mean	Worst	Mean	Worst
S	MIQP	0.65	0.79	$0.66, \mathbf{0.00}$	$2.90, \mathbf{0.00}$
	MCTS	$\mathbf{0.10}$	$\mathbf{0.18}$	$\mathbf{0.00}, \mathbf{0.00}$	$\mathbf{0.00}, \mathbf{0.00}$
	MCTS^U	0.24	1.23	$\mathbf{0.00}, 0.03$	$0.06, 1.14$
L	MIQP	0.25	0.51	$6.29, \mathbf{0.00}$	$6.87, \mathbf{0.00}$
	MCTS	$\mathbf{0.15}$	$\mathbf{0.23}$	$\mathbf{0.24}, 0.05$	$\mathbf{0.86}, 0.18$
	MCTS^U	0.53	2.23	$0.53, 0.11$	$0.88, 0.35$
R	MIQP	4.83	29.46	$8.36, 16.64$	$30.72, 45.57$
	MCTS	$\mathbf{0.12}$	$\mathbf{0.27}$	$\mathbf{0.00}, \mathbf{0.00}$	$\mathbf{0.00}, \mathbf{0.00}$
	MCTS^U	0.41	1.22	$\mathbf{0.00}, \mathbf{0.00}$	$\mathbf{0.00}, \mathbf{0.00}$
SC	MIQP	2.19	4.41	$11.73, 22.39$	$49.61, 88.27$
	MCTS	$\mathbf{0.11}$	$\mathbf{0.24}$	$\mathbf{0.00}, \mathbf{0.00}$	$\mathbf{0.00}, \mathbf{0.00}$
	MCTS^U	0.20	0.81	$\mathbf{0.00}, \mathbf{0.00}$	$0.03, 0.07$
P	MIQP	6.69	50.41	$7.65, 15.31$	$26.85, 53.65$
	MCTS	$\mathbf{0.15}$	$\mathbf{0.34}$	$\mathbf{0.01}, \mathbf{1.23}$	$\mathbf{0.23}, \mathbf{14.01}$
	MCTS^U	0.17	0.45	$\mathbf{0.01}, 1.46$	$0.33, 19.55$

- P about the y-axis by $0°$ to $45°$.

The xy-axes span the plane that the object is placed on and the z-axis points to the opposite gravity direction. Finally, the initial object position is randomly sampled on the xy-plane within a $[-5\,\mathrm{cm}, 5\,\mathrm{cm}]^2$ area centered at the origin and the initial orientation about the z-axis by $-90°$ to $90°$. Please refer to Appendix B for details about our training procedure, baselines, and evaluation metrics.

5.1 Single motion primitives

In this experiment, we consider object motion trajectories that consist of one single primitive. Each primitive has a desired pose uniformly randomly sampled from its respective parameter range described in Appendix B. Each trajectory consists of $T = 10$ time steps with step size $\Delta t = 0.1\,\mathrm{s}$; each contact persists as well at least $0.1\,\mathrm{s}$, hence a trajectory can admit at most 9 contact switches. We run 50 trials for each primitive to collect the performance statistics.

Table 1 shows that our method is capable of finding dynamically feasible solutions consistently faster than the MIQP baseline thanks to the MCTS formulation. Especially for primitives that require non-zero torques (R, SC, P), the MIQP baseline is not only an order of magnitude slower, but also produces solutions with large errors. We note that the force error can be reduced by letting the MIQP solver explore more feasible solutions, while the torque error remains high nonetheless. This might be due to its usage of the McCormick envelopes to approximate cross products, which not only introduces approximation error but also adds additional discrete variables. In contrast, thanks to the ADMM formulation, our method solves the original problem instead of a relaxed one and has thus near-zero average force and torque error. We also note that while the solutions found by the MIQP baseline are dynamically feasible, they are not necessarily kinematically feasible or collision-free since these conditions cannot be incorporated as linear constraints. While it is possible to collect multiple dynamically feasible solutions and pick the kinematically feasible one from them, it may further increase the computation time.

5.2 Long-horizon tasks

In the previous experiments, we have shown the effectiveness of our method for short motion primitives. Let us now consider tasks that last a longer period of time. First, we extend the primitive to $T = 30$ time steps by stipulating each contact persists for $d = 3$ steps. Note that there are still at most 9 contact switches for a single primitive. However, such extended primitives may be useful for tasks that require longer execution time but not necessarily more contact switches; for instance, sliding a cube for a long distance or rotating it very slowly. In the following experiments, we con-

Table 2: Task performance for object motions composed of SC primitives. Errors are computed only on successful trials. Values ≤ 0.005 are rounded to zero.

# SC	Method	Success rate	Time [s] Mean	Time [s] Worst	Error [N, N · m] Mean	Error [N, N · m] Worst
1	MIQP	94%	10.15	60.00	$3.40, 11.72$	$19.93, 41.68$
	MCTS	**100%**	**0.21**	**0.41**	**0.00, 0.00**	**0.00, 0.00**
	MCTS^U	**100%**	0.91	3.67	**0.00, 0.00**	$0.03, 0.07$
2	MIQP	42%	40.93	60.00	$4.96, 4.38$	$16.61, 22.54$
	MCTS	**100%**	**0.47**	**1.56**	**0.00, 0.00**	**0.01, 0.03**
	MCTS^U	**100%**	3.08	12.84	**0.00, 0.00**	$0.03, 0.07$
3	MIQP	0%	–	–	–	–
	MCTS	**100%**	**1.35**	**8.84**	**0.00, 0.00**	**0.01, 0.03**
	MCTS^U	90%	20.87	60.00	**0.00, 0.00**	$0.01, 0.04$

catenate such extended primitives to form a even longer trajectory. In particular, we consider the primitive SC as it represents typical planar repositioning/reorienting tasks.

Table 2 reports the performance metrics for each method. A task is considered failed if no dynamically feasible solution is found within $60\,\mathrm{s}$. We can immediately see that the trained MCTS efficiently solves all the tasks regardless of the trajectory length, while the MIQP baseline and the untrained MCTS struggle in long-horizon tasks (the errors decrease because they are computed only on successful trials). Indeed, the MIQP baseline cannot solve any tasks containing more than two primitives in $60\,\mathrm{s}$. Interestingly, even for the task with a single extended primitive, where the number of possible contact switches does not change compared to the previous tasks in Sec. 5.1, the MIQP baseline still need significantly more time to find a feasible solution. This could again be attributed to the McCormick envelopes as they add additional discrete variables to each time step regardless of the underlying number of contact switches.

Finally, we highlight that the MCTS training dataset only contains object motion trajectories consisting of at most two primitives. However, Table 2 shows that our method is able to efficiently solve the longer-horizon tasks without being explicitly trained on them as our MCTS formulation exploits the intrinsic compositionality of the task by learning a goal-conditioned policy-value network. Hence, we do not need to collect training data on large-scale, time-consuming problems as opposed to the learning-based MIP method proposed in [12].

5.3 Executing the contact plan

To validate the contact plans found by our method, we execute them in an open-loop fashion with a simple impedance controller in the PyBullet simulator [27] and on a real robot

$$\tau = J^{\mathsf{T}}\left(K(r_c^{\mathrm{w}} - r^{\mathrm{w}}) + D(\dot{r}_c^{\mathrm{w}} - \dot{r}^{\mathrm{w}}) + f_c^{\mathrm{w}}\right), \tag{3}$$

where J is the end-effector Jacobian; K, D are manually tuned gain matrices; $r^{\mathrm{w}}, \dot{r}^{\mathrm{w}}$ are the position and velocity of the end-effector and $f_c^{\mathrm{w}}, r_c^{\mathrm{w}}, \dot{r}_c^{\mathrm{w}}$ are the planed contact force, location and velocity, all expressed in the world frame. Fig. 4b shows an example of the contact plan execution of rotating a cube by $90°$. The robot is able to move the object towards the target pose if the object is placed at the desired initial position. We note that the same impedance is used for all tasks with different primitives.

6 Conclusions and future work

In this paper, we presented a framework to efficiently search for non-trivial contact plans for long sequence of locomotion and manipulation problems. In our framework, we proposed a customized version of MCTS together with a our ADMM-based trajectory optimization to search for feasible solutions for contact planning. We showed that we can reliably find feasible solutions for different arrangements of the environments as well as initial and final conditions. Collecting feasible rollouts enabled us to collect a rich dataset and learn a control policy that can generate contact plans in real-time rates. In the future, we plan to explore to learn long-horizon of loco-manipulation tasks, e.g., manipulating objects with a humanoid robot in cluttered environments.

References

[1] H. Zhu, A. Meduri, and L. Righetti. Efficient object manipulation planning with monte carlo tree search. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10628–10635. IEEE, 2023.

[2] V. Dhédin, A. K. Chinnakkonda Ravi, A. Jordana, H. Zhu, A. Meduri, L. Righetti, B. Schölkopf, and M. Khadiv. Diffusion-based learning of contact plans for agile locomotion. In *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages arXiv–2403, 2024.

[3] S. V. Raković and W. S. Levine. *Handbook of model predictive control*. Springer, 2018.

[4] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] J. Siekmann, K. Green, J. Warila, A. Fern, and J. Hurst. Blind bipedal stair traversal via sim-to-real reinforcement learning. In *Robotics: Science and Systems*, 7 2021.

[6] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics*, 7(62):eabk2822, 2022.

[7] M. Bogdanovic, M. Khadiv, and L. Righetti. Model-free reinforcement learning for robust locomotion using demonstrations from trajectory optimization. *Frontiers in Robotics and AI*, 9, 2022.

[8] R. Grandia, F. Jenelten, S. Yang, F. Farshidian, and M. Hutter. Perceptive locomotion through nonlinear model predictive control. *IEEE Transactions on Robotics*, 2023.

[9] A. Meduri, P. Shah, J. Viereck, M. Khadiv, I. Havoutis, and L. Righetti. Biconmp: A nonlinear model predictive control framework for whole body motion planning. *IEEE Transactions on Robotics*, 2023.

[10] M. Toussaint. Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[11] B. Aceituno-Cabezas and A. Rodriguez. A global quasi-dynamic model for contact-trajectory optimization. In *Robotics: Science and Systems (RSS)*, 2020.

[12] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[13] A. Cauligi et al. Coco: Online mixed-integer control via supervised learning. *IEEE Robotics and Automation Letters*, 7(2):1447–1454, 2021.

[14] X. Lin, G. I. Fernandez, and D. W. Hong. Reduce: Reformulation of mixed integer programs using data from unsupervised clusters for learning efficient strategies. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 4459–4465. IEEE, 2022.

[15] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. Del Prete. Optimization-based control for dynamic legged robots. *IEEE Transactions on Robotics*, 2023.

[16] S. Ha, J. Lee, M. van de Panne, Z. Xie, W. Yu, and M. Khadiv. Learning-based legged locomotion; state of the art and future perspectives. *arXiv preprint arXiv:2406.01152*, 2024.

[17] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[18] D. Silver et al. Mastering the game of go without human knowledge. *Nature*, 550(7676): 354–359, 2017.

[19] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[20] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models, 2020.

[21] A. Block, A. Jadbabaie, D. Pfrommer, M. Simchowitz, and R. Tedrake. Provable guarantees for generative behavior cloning: Bridging low-level stability and high-level behavior, 2023.

[22] M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine. Planning with diffusion for flexible behavior synthesis, 2022.

[23] C. Chi, S. Feng, Y. Du, Z. Xu, E. Cousineau, B. Burchfiel, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion, 2023.

[24] F. Grimminger, A. Meduri, M. Khadiv, J. Viereck, M. Wüthrich, M. Naveau, V. Berenz, S. Heim, F. Widmaier, T. Flayols, et al. An open torque-controlled modular robot architecture for legged locomotion research. *IEEE Robotics and Automation Letters*, 5(2):3650–3657, 2020.

[25] J. Song, C. Meng, and S. Ermon. Denoising diffusion implicit models, 2022.

[26] M. Wüthrich et al. Trifinger: An open-source robot for learning dexterity. *arXiv preprint arXiv:2008.03596*, 2020.

[27] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`.

[28] Y. Labbé, S. Zagoruyko, I. Kalevatykh, I. Laptev, J. Carpentier, M. Aubry, and J. Sivic. Monte-carlo tree search for efficient visually guided rearrangement planning. *IEEE Robotics and Automation Letters*, 5(2):3715–3722, 2020.

[29] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks, 2017.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023.

[31] E. Perez, F. Strub, H. de Vries, V. Dumoulin, and A. Courville. Film: Visual reasoning with a general conditioning layer, 2017.

[32] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[34] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[35] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL `https://www.gurobi.com`.

Appendix A

In the following we outline the problem-specific heuristics used for locomotion and manipulation problems. Note that to check the dynamic feasiblity of the candidate contact sequence by MCTS, we use the nonlinear model predictive controller (NMPC) in [9] for locomotion and a similar trajectory optimization formulation for manipulaiton.

Locomotion heuristics

The following heuristics are used for the lcomotion problem: During the **Expansion** phase, to limit the search space, we additionally perform a simplified kinematic feasibility check to prune the sequences that are likely to be not reachable or cause self-collision for the legs of the robot. This check verifies that the size of the step taken by each foot is below a maximal step distance d_{step} and that the legs are not crossing. It is important to note that, similar to [28], since we can define a reward at each state that gives us a proxy of how close this state is to the end goal, we do not simulate a complete rollout to a terminal state as opposed to game-play (such as chess or go). Instead, we only take one action at the chosen state and back-propagate the reward from the resulting next state, hence a rollout depth of one in the **Simulation** phase. Moreover, we perform whole-body NMPC [9] using the found sequence of contact locations only when a terminal state is reached (all the end-effectors are at the desired goal surface patches). This way, we avoid performing whole-body NMPC for every rollout in the **Simulation** phase. This significantly reduces search time as whole-body NMPC is computationally intensive. If this contact sequence leads to dynamically feasible whole-body motion in physical simulation, W is set to 1, else to -1. Ultimately, we back-propagate the final reward. Overall, the MCTS searches for promising kinematically feasible contact sequences, and the NMPC evaluates their dynamic feasibility at the end.

Manipulation heuristics

We apply the following heuristics to the object manipulation setting: Each contact surface can only be touched by at most one end-effector and each end-effector can touch at most one contact surface. While the downstream continuous optimization problem may have a small discretization step, for example $\Delta t = 0.1\,\mathrm{s}$, most manipulation tasks do not require decisions of contact switch at such a high resolution. Thus, we assume that an end-effector must remain on the same surface for d time steps, which means a trajectory of length T can admit at most $T/d - 1$ contact switches. For each end-effector c, a contact surface will only be considered if inverse kinematics can find a robot configuration that reaches the center of this surface within an error threshold of $2\,\mathrm{cm}$ and does not result in any undesired collision (e.g. between non-end-effector links and the object). We allow at most one end-effector to make or break the contact at each time step. Moreover, the end-effector can only break the contact if the desired object velocity and acceleration is zero.

Appendix B

In this Appendix, we present the details on the baselines and evaluation metrics for the comparisons.

Locomotion Setting

Dataset In our setup, a quadruped robot ($N_e = 4$) navigates in an environment with up to 81 stepping stones. Each stone provides the robot with a cylindrical patch of radius $4.4\,\mathrm{cm}$ and height $h = 10\,\mathrm{cm}$ to step onto. The stepping stones initially form a regular grid of spacing (e_x, e_y) so that the feet lay on 4 stepping stones in the initial configuration. From this, the environment is randomized. The position of each stone is then displaced by $\epsilon_x(\frac{e_x}{2} - r)$ with $\epsilon_x \sim \mathcal{U}(-\alpha_x, \alpha_x)$, $\alpha_x \in [0, 1]$ in the x direction (respectively for the direction y). Similarly, the randomized height equals $(1 + \epsilon_h)h$ with $\epsilon_h \sim \mathcal{U}(-\alpha_h, \alpha_h), \alpha_h \in [0, 1]$. Additionally, N_{removed} stones are randomly removed. The simulated environment can be seen in Fig. 1. Goals are also sampled randomly so

that the center of the 4 goal contact locations is within d^g_{min} and d^g_{max} of the initial robot position. In our experiments, H encodes the NMPC horizon, which in our problem is two jumps in the future ($H = 2$). The goal is to evaluate the learned policy before each jump and feed the selected contact locations to the NMPC. As state variables, we consider the position of the end-effectors, the current base linear and angular velocities (all expressed in base frame B).

To be independent of the global position of the robot and environment, we express all positions in the inputs and outputs to the network with respect to the current base frame B of the robot. The inputs, $\boldsymbol{x}$ to the network are the current 3D position of the available contact locations $\boldsymbol{x}_{\text{contact}}$, denoted by $\boldsymbol{c}^i_B \in \mathcal{C}$ with $i \in \{1, ..., N_c\}$ and $\mathcal{C}$ the set of available contact locations (N_c is the current number of available contact locations), $\boldsymbol{x}_{\text{state}}$ that includes some state variables (see Appendix A) and $\boldsymbol{x}_{\text{goal}}$, the final desired N_e end-effector locations (N_e is the number of end-effectors). The size of the input is equal to $3 \times (N_c + 2 \times (N_e + 1))$.

To collect a diverse dataset, we sample a random environment and run MCTS for a fixed maximum number of iterations. To collect diverse paths towards a goal, we keep up to N_{paths} different feasible paths for the same goal and environment. To cover a wider range of robot states, for each MCTS solution, we perturb the simulation N_{rand} times and add feasible solutions to the dataset. The randomization procedure consists of randomizing the initial state of the robot (position and velocity) as well as the contact locations inside the selected patches. The training data $(\boldsymbol{x}, \boldsymbol{y})$ (see Fig. **??**) are recorded at each jump ($\boldsymbol{y}$ are the contact locations of the next two jumps). We repeat the procedure on N_{env} different environments (set of stepping stones).

Pointer-Networks Pointer-Networks [29] take as input a sequence and output discrete indices, called *pointers*, that select elements from the input sequence. In this case, the projection $p_{\mathcal{C}}$ is not performed as the model directly outputs from the input set. The architecture is composed of 2 recurrent networks and an attention mechanism that operates on the past decoder's hidden states and all the encoder's hidden states. At each step, the output of the decoder is the index of the encoder's hidden state that has the maximum attention value with the past decoder's hidden state. This operation is repeated for as many times as needed. In our case 8 times (the next two contact locations for each four legs).

To make the model select only the contact patches from the input $\boldsymbol{x}$, $\boldsymbol{x}_{\text{contact}}$ is given as the input sequence while $[\boldsymbol{x}_{\text{state}}, \boldsymbol{x}_{\text{goal}}]$ is embedded and given as the first hidden state of the encoder. Like so, contact patches can be provided in any order and a different number which is not the case for instance for an MLP.

Diffusion models DDPMs are generative models that map samples from a latent random distribution to the data distribution in T steps by successive denoising of the original noise. For each intermediate step $t \in [1, T]$, one can sample a corrupted input $\boldsymbol{x}_t$ by adding noise $\boldsymbol{\epsilon}_t$ to a sample of the data $\boldsymbol{x}_0$. A variance schedule assigns an increasing noise level at each step t so that $\boldsymbol{x}_T$ can be seen as a pure random noise (usually from a Gaussian distribution). Those corrupted samples are used to train the diffusion model $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}$, parametrized by $\boldsymbol{\theta}$, to estimate the noise added in a supervised manner. It is done by minimizing the MSE loss between the actual sampled noise $\boldsymbol{\epsilon}_t$ and the estimated one $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_0 + \boldsymbol{\epsilon}_t, t)$. Minimizing the MSE loss leads to the minimization of the variational lower bound of the KL-divergence between the data distribution and the distribution of samples drawn from the DDPM [20].

To sample with the trained model, noise is successively removed from a random sample $\boldsymbol{x}_T$ in the following way

$$\boldsymbol{x}_{t-1} = a_t \left(\boldsymbol{x}_t - b_t \, \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{z}$$

where $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and a_t, b_t, σ_t are computed according to the noise schedule.

U-Net-based architectures are widely used as DDPMs, especially for conditional image generation. Following [22], we consider a conditional U-Net1D with 1D convolutions applied on the input sequence length (end-effector dimension). The conditioning is done on both the denoising iteration

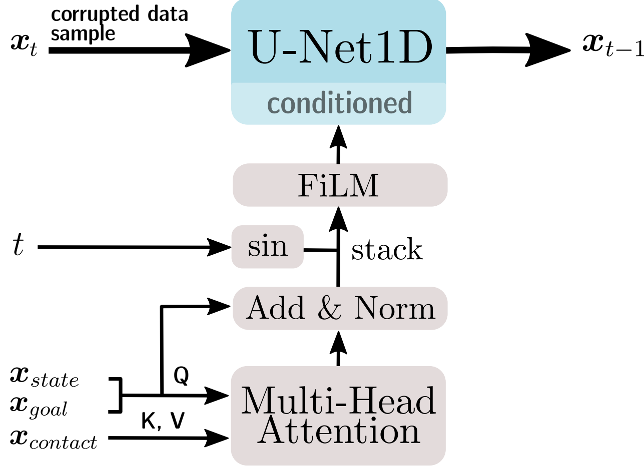


Figure 5: High-level description of the conditional U-Net1D with multi-head attention. *sin* refers to Sinusoidal positional embedding applied to t.

t of the diffusion process and the current input $\boldsymbol{x}$. Similarly to what has been done with the Pointer-Network in Sec. 6, the input $\boldsymbol{x}$ is split into two. We apply a multi-head attention layer [30] with $[\boldsymbol{x}_{\text{state}}, \boldsymbol{x}_{\text{goal}}]$ as query and $\boldsymbol{x}_{\text{contact}}$ as key and values. Thus, contact locations can be shuffled and provided in any number. The resulting embedding of $\boldsymbol{x}_{\text{state}}$ is concatenated to a sinusoidal position embedding [30] of t to form the input of a feature-wise linear modulation (FiLM) [31] layer, as proposed by [23]. The architecture is detailed in Fig. 5. Finally, the conditioning vector is added to all layers of the encoder and the decoder parts of the network, as done in [22].

MLP parameters We used a standard MLP with 4 hidden layers of latent dimension 64 with LeakyReLU activation. The size input dimension is 273 (81 stepping stones locations) and the size output dimension is 24. The total number of trainable parameters of this model is 35736.

Pointer-Network parameters Both the encoder and decoder are LSTMs with 2 layers and hidden dimension 32. The attention mechanism is also of hidden dimension 32. $[\boldsymbol{x}_{\text{state}}, \boldsymbol{x}_{\text{goal}}]$ is embedded through an MLP with 2 hidden layers of hidden dimensions 16 and 32 with PReLU activation. The total number of trainable parameters is 31521 which is comparable to the MLP architecture considered.

U-Net1D parameters We chose a U-Net with 3 layers with respectively a channel width equal to 64, 128 and 256 and a kernel size of 3 for the convolutions. Convolutions are sliding on the temporal/end-effector dimension. The sinusoidal embedding dimension is 32. The multi-head attention layer has only one head. We used a squared cosine noise schedule with $\beta_1 = 0.004$ and $\beta_T = 0.02$ as suggested by [20] for $T = 50$ training iterations. The model has in total 2639207 parameters.

Manipulation setting

Baselines We compare our method (**MCTS**) with two baselines:

- the **MIQP** baseline is implemented following [11]. We did not use the authors' open-source implementation as it was only implemented for 2D objects. But the same formulation can be directly extended to 3D. To facilitate a fair comparison, we also implemented all heuristics described in Sec **??** except the kinematic feasibility check. For the McCormick envelope relaxation of the cross product, we partition the contact location into 4 intervals and the contact force into 2 intervals. In all experiments, we terminate the MIQP solver at the first feasible solution instead of waiting for the global optimum which may take

extremely long time. In addition, we implement the constraint (**??**) as a penalty term in the cost function. This accelerates the MIQP solver significantly from our observation during the experiments. We note that our implementation has comparable computation time as reported in [11].

- the **MCTS**^{U} baseline represents an untrained model and constantly outputs zero values $v_\theta(s) = 0$ and uniform action probability $p_\theta(s, a) = 1/|\mathcal{A}(s)|$.

Training procedure We generate 300 object motion trajectories, each comprising two primitives with randomly sampled parameters. In particular, 200 trajectories are composed of two SC primitives; 50 trajectories of one SC and one L; 50 trajectories of one SC and one P. For the i-th trajectory, we let an untrained MCTS run until it evaluates 200 candidate contact sequences; then we construct the dataset $\mathcal{D} = \mathcal{D} \cup \{(\bar{v}(s), \bar{p}(s))|s \in \mathcal{V}_i\}$ where $\mathcal{V}_i$ contains all the states the MCTS visited for the i-th trajectory. The policy-value network and the value classifier are then trained via Adam [32] for 300 epochs.

Evaluation metrics We examine two metrics to evaluate the effectiveness and efficiency of our method:

- the **force and torque error** between the desired and the solution. The error is averaged over the horizon T and scaled by the object mass and inertia respectively. The smaller this error is, the better the solution tracks the desired object motion.
- The **computation time** needed to find the first dynamically feasible solution.

Software and hardware We conduct all experiments on a single GeForce RTX 2080 TI GPU and an Intel Xeon CPU at $3.7\,\mathrm{GHz}$ using Python and PyTorch. We model and solve the QP with CVXPY [33] and OSQP [34] and use Gurobi [35] for MIQP. All source code including the baseline can be found at `https://hzhu.io/contact-mcts`.