Language to Rewards for Robotic Skill Synthesis

Anonymous Author(s)
Affiliation
Address
email

Abstract: Large language models (LLMs) have demonstrated exciting progress in acquiring diverse new capabilities through in-context learning, ranging from logical reasoning to code-writing. Robotics researchers have also explored using LLMs to advance the capabilities of robotic control. However, since low-level robot actions are hardware-dependent and underrepresented in LLM training corpora, existing efforts in applying LLMs to robotics have largely treated LLMs as semantic planners or relied on human-engineered control primitives to interface with the robot. On the other hand, reward functions are shown to be flexible representations that can be optimized for control policies to achieve diverse tasks. Our key insight is that just as LLMs can generate code or high-level task plans from language instructions, their rich context enables specifying flexible reward functions. In this work, we introduce a new paradigm that harnesses this realization by utilizing LLMs to define reward parameters that can be optimized and accomplish variety of robotic tasks. As a specific instantiation of this method, we propose using a real-time behavior synthesis tool, MuJoCo MPC, to optimize the robot actions based on the provided reward. Using reward as the intermediate interface generated by LLMs, we can effectively bridge the gap between high-level language instructions or corrections to low-level robot actions. Meanwhile, combining this with the real-time optimizer MuJoCo MPC empowers an interactive behavior creation experience where users can immediately observe the results and provide feedback to the system. To systematically evaluate the performance of our proposed method, we designed a total of 17 tasks for a simulated quadruped robot and a dexterous manipulator robot. We demonstrate that our proposed method reliably tackles 90% of the designed tasks, while a baseline using primitive skills as the interface with Code-as-policies achieves 50% of the tasks.

Keywords: Large language model (LLM), Low-level skill learning, Legged locomotion, Dexterous manipulation

1 Introduction

Empowered by recent advancements in language modeling, the utilization of large language models (LLMs) pretrained on extensive internet data [1, 2] has revolutionized the ability to interpret user inputs in natural language and accomplish a wide array of tasks using just a few words. These LLMs exhibit remarkable adaptability to new contexts (such as APIs [3], task descriptions [4], or textual feedback [5]), allowing for tasks ranging from logical reasoning [6, 7] to code generation [8] with minimal hand-crafted examples.

These diverse applications have extended to the field of robotics as well, where substantial progress has been made in using LLMs to drive robot behaviors [3, 5, 4, 9, 10, 11]: from step-by-step planning [4, 9, 12], goal-oriented dialogue [10, 11], to robot-code-writing agents [3, 13]. While these methods impart new modes of compositional generalization, they focus on using language to concatenate together new behaviors from an existing library of control primitives that are either manually engineered or learned a priori. Despite having internal knowledge about robot motions, LLMs struggle with directly outputting low-level robot commands due to the limited availability of relevant training data (Fig. 1). As a result, the expression of these methods are bottlenecked by the breadth of the

Submitted to the 7th Conference on Robot Learning (CoRL 2023). Do not distribute.

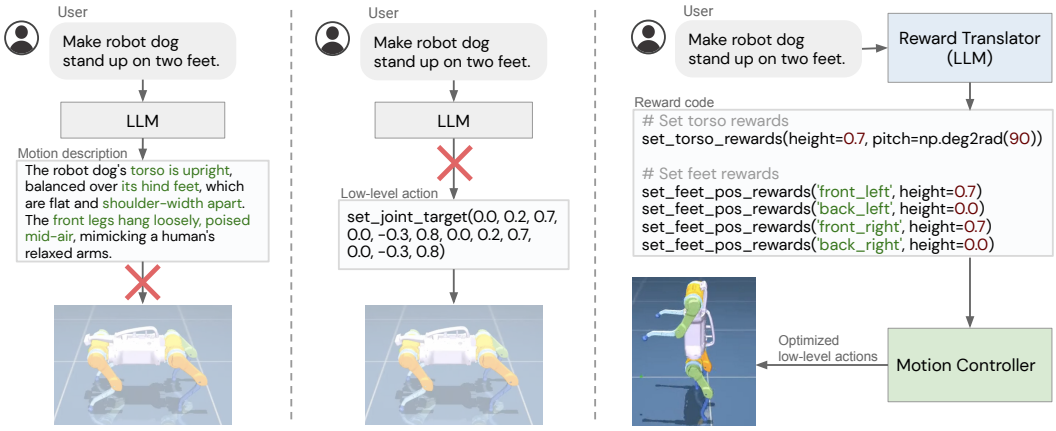


Figure 1: LLMs have some internal knowledge about robot motions, but cannot directly translate them into robot actions (left). Low-level action code can be executed on robots, but LLMs know little about them (mid). We attempt to bridge this gap, by proposing a system (right) consisting of the Reward Translator that interprets the user input and transform it into a reward specification. The reward specification is then consumed by a Motion Controller that interactively synthesizes a robot motion which optimizes the given reward.

available primitives, the design of which often requires extensive expert knowledge or massive data collection [14, 15, 16].

To tackle these challenges, we need to operate at a level of abstraction that allows harnessing the intuitive and interactive capabilities offered by LLMs. Our key insight is to leverage reward functions as an interface that bridges the gap between language and low-level robot actions. This is motivated by the fact that language instructions from humans often tend to describe behavioral outcomes instead of low-level behavioral details (e.g. "robot standing up" versus "applying 15 Nm to hip motor"), and therefore we posit that it would be easier to connect instructions to rewards than low-level actions given the richness of semantics in rewards. In addition, reward terms are usually modular and compositional, which enables concise representations of complex behaviors, goals, and constraints. This modularity further creates an opportunity for the user to interactively steer the robot behavior. However, in many previous works in reinforcement learning (RL) or model predictive control (MPC), manual reward design requires extensive domain expertise [17, 18]. While reward design can be automated, these techniques are sample-inefficient and still requires manual specification of an objective indicator function for each task [19]. This points to a missing link between the reward structures and task specification which is often in natural language. As such, we propose to utilize LLMs to automatically generate rewards, and leverage online optimization techniques to solve them. Concretely, we explore the code-writing capabilities of LLMs to translate task semantics to reward functions, and use MuJoCo MPC, a real-time optimization tool to synthesize robot behavior in real-time [20]. Thus reward functions generated by LLMs can enable non-technical users to generate and steer novel and intricate robot behaviors without the need for vast amounts of data nor the expertise to engineer low-level primitives.

The idea of grounding language to reward has been explored by prior work for extracting user preferences and task knowledge [21, 22, 23, 24, 25, 26]. Despite they demonstrated promising results, they lack the ability to synthesize low-level robot actions in response to language instructions or corrections on-the-fly. With our proposed method, we can enable multi-step human-in-the-loop interactions, where the human actively engages in a dialogue with the LLM to guide the generation of rewards and, consequently, robot behaviors (Fig. 1).

Across a span of 17 control problems on a simulated quadruped robot and a dexterous manipulator robot, we show that this formulation delivers diverse and challenging locomotion and manipulation skills. Examples include getting a quadruped robot to stand up, asking it to do a moonwalk, or tasking a manipulator with dexterous hand to open a faucet. We perform a large-scale evaluation to systematically measure the overall performance of our proposed method. We compare our method to a baseline that uses a fixed set of primitive skills and an alternative formulation of grounding

language to reward. We show that our proposed formulation can cover more diverse skills and is more stable in solving individual skills.

2 Related Work

Prior work towards grounding natural language to low-level robotic control has a long history [27], and has explored various ways of combining language with priors and intermediate representations. Here we discuss prior work that reason about language instructions to generate robot actions, code, or rewards. We will then discuss work focused on responding to interactive human feedback such as language corrections.

Language to Actions. Directly predicting low-level control actions based on a language instruction has been studied using various robot learning frameworks. Early work in the language community studied mapping templated language to controllers with temporal logic [28] or learning a parser to motion primitives [29], while more recent work utilize end-to-end models that produce actions conditioned on natural language descriptions. Instruction following approaches are well-studied in navigation, but often rely on low-dimensional actions navigating from one node of a navigation graph to another [30]. To extend the end-to-end approaches to manipulation, the policies often utilize latent embeddings of language commands as multitask input context, and can be trained with behavioral cloning [14, 31, 16], offline reinforcement learning [32], goal-conditioned reinforcement learning [33], or in a shared autonomy paradigm [34]. While end-to-end trained policies can be performant, they require significant amount of data in the form of offline datasets or online environment interaction as well as significant amount of computation in the form of gradient-based backpropagation. In contrast, we study a less data hungry approach where low-level actions are not directly produced by an end-to-end policy but instead by an optimal controller.

Language to Code. Code generation models have been widely studied both in and outside robotics context [35, 8, 36]. The capability of those models range from solving coding competition questions [37] and benchmarks [38], to drawing simple figures [39], generating policies that solve 2D tasks [40], and complex instruction following tasks [3]. Aside from being able to produce code (structured formats), LLMs that are pretrained on code also appear to do generally better on reasoning tasks. In this work, we study LLMs for generating code for reward functions, and show that the expression of the rewards can enable learning low-level policies that are more expressive and can more accurately generate the policy code outright.

Language to Rewards. The idea of translating natural language instructions to rewards has been explored by several prior work [25, 22, 24, 41, 21, 42, 26]. A common strategy in this direction is to train domain-specific reward models that map language instructions to reward values [22, 21, 41] or constraints [24]. Despite the fact that these methods can achieve challenging language conditioned robotic tasks such as object pushing [24], and drawer opening [41], they require a considerable amount of language-labeled data to train the reward model. Recent works investigated using LLMs directly as a reward function for inferring user intentions in negotiation games or collaborative human-AI interaction games [25, 26]. By leveraging LLMs to assign reward values during RL training, these works demonstrate training agents that are aligned with user intentions and preferences without the need for explicit reward modeling. However, these works do not capture a parameterization of the reward function, and instead only receive reward values of rollouts when training RL policies, which requires a large number of queries to LLMs during training. In contrast, we are interested in learning the full parameterization of the reward function that can then be optimized.

Incorporating Iterative Human Feedback. Correcting plans with iterative language feedback has also been explored in the past. Broad et al. enable efficient online corrections using distributed correspondence graphs to ground language [43]. However, this work relies on a semantic parser with pre-defined mappings to ground language corrections and using a motion planner that relies on hand-designed primitives. More end-to-end approaches have also demonstrated learning a language correction conditioned policy, but these works are similarly data hungry and thus fall back to shared autonomy to reduce complexity [44]. Later work explore how language corrections can help define composable cost functions similar to our work using a trajectory optimizer [24]. However, the trajectory optimizer requires substantial knowledge about the environment. To address this shortcoming, followup works integrate language corrections to directly modify the waypoints of a trajectory in both 2D and 3D environments using extensive datasets of paired corrections and demonstrations

[45, 46]. In contrast to these prior work, we demonstrate a flexible and data-efficient approach that leverages LLMs to allow for multi-step correction of reward functions based on human feedback.

Automated Reward Designs. TBW

3 Grounding Language to Actions Using Rewards

3.1 Background and Reward Interface

Our system takes a user instruction in natural language and synthesizes corresponding robot motions by leveraging reward function as the interface to communicate with low-level controllers. We define the reward function in the context of Markov Decision Process (MDP), commonly used to formulate robot control problems: (S, A, R, P, p_0), where S is the state space, A is the action space, $R : S \times A \mapsto \mathbb{R}$ is the reward function, $P : S \times A \mapsto S$ is the dynamics equation, and p_0 is the initial state distribution. Given a reward function R, an optimal controller finds a sequence of actions $\mathbf{a}_{1:H} = \{\mathbf{a}_1, \ldots, \mathbf{a}_H\}$ that maximizes the expected accumulated reward: $J(\mathbf{a}_{1:H}) = \mathbb{E}_{\tau=(\mathbf{s}_0,\mathbf{a}_0,\ldots,\mathbf{s}_H)} \sum_{t=0}^{H} R(\mathbf{s}_t, \mathbf{a}_t)$, where H is the rollout horizon.

In this work, we assume the reward to be the sum of a set of individual terms:

$$R(\mathbf{s}, \mathbf{a}) = -\sum_{i=0}^{M} w_i \cdot \mathrm{n}_i\big(r_i(\mathbf{s}, \mathbf{a}, \psi_i)\big), \tag{1}$$

where $w \in \mathbf{R}_+$ is a non-negative weight, $\mathrm{n}(\cdot) : \mathbf{R} \to \mathbf{R}_+$ is a twice-differentiable norm that takes its minimum at 0, $r \in \mathbf{R}$ is a residual term that achieves optimality when $r = 0$, and ψ_i is the parameters of the i_{th} residual term. For example, if we want to have the robot raise its body height h to a desired height, we may design a residual term $r_h(h, \psi) = h - \psi$, where the reward parameter ψ denotes the desired height, and use the l2 norm to construct the final reward function: $R_h = -w||r_h||_2$. In principle, one may design task-specific residual terms that can solve particular controller tasks. However, designing these residuals can be time-consuming, requires domain expertise, and may not generalize to novel tasks. In this work, we use a set of generic and simple residual terms, and leverage the power of LLMs to compose different terms to generate complex behaviors. The full set of residual terms used in this work can be found in the Appendix 6.2.

In order to bridge language and robot actions through reward functions, our system consists of two key components (as shown in Fig. 1 right): i) a *Reward Translator* , built upon pre-trained large language models (LLMs) [10], that interacts with the user through natural language to understand user's intents and modulates all reward parameters ψ and weights w, and ii) a *Motion Controller* , based on MuJoCo MPC (MJPC) [20], that takes the generated reward and interactively synthesizes the robot motion by optimizing for the optimal action sequence $\mathbf{a}_{1:H}$. Below we provide more details on the design of Reward Translator and Motion Controller .

3.2 Reward Translator

Inspired by recent progress on Large Language Models (LLMs), we propose to build the Reward Translator based on LLMs to map user interactions to reward functions corresponding to the desired robot motion. As reward tuning is highly domain-specific and requires expert knowledge, it is unsurprising that LLMs trained on generic language datasets (e.g. [1]) cannot directly generate a reward for a specific robot morphology. Instead, we explore the in-context learning ability of LLMs to achieve this goal, inspired by prior work that demonstrated the ability of LLMs to perform a variety of skills in-context [2, 47]. Furthermore, we decompose the problem of language to reward into two stages: motion description and reward coding task, as illustrated in Fig. 2.

Motion Description In the first stage, we design a *Motion Descriptor* LLM that interprets and expands the user input into a natural language description of the desired robot motion following a pre-defined template (see example in Fig. 2). Although it is possible for LLMs to directly generate reasonable reward functions for relatively simple task, it often fails for tasks that necessitates complex reasoning. On the other hand, as observed in Fig. 1 left, LLMs can describe complex motions in detailed natural language successfully.

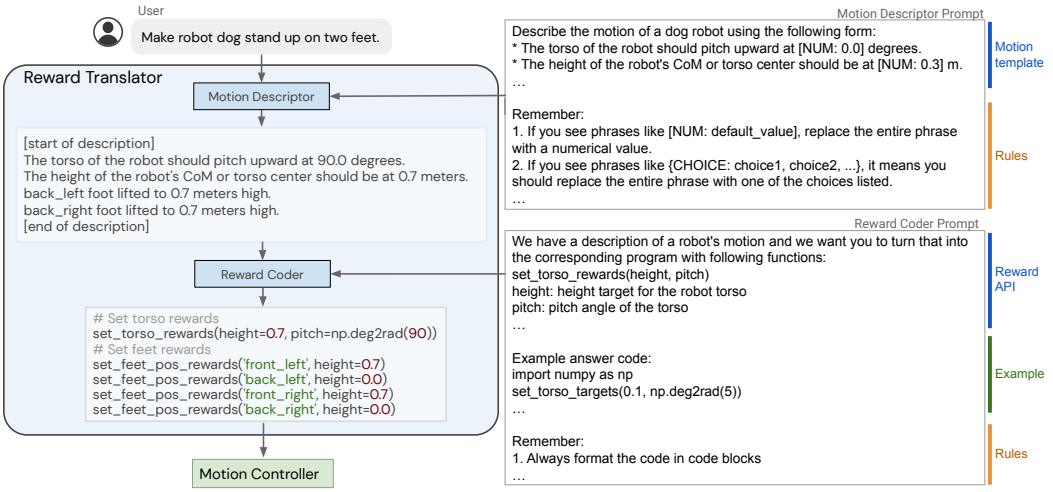


Figure 2: Detailed dataflow of the Reward Translator . A *Motion Descriptor* LLM takes the user input and describe the user-specified motion in natural language, and a *Reward Coder* translates the motion into the reward parameters.

Inspired by this observation, we design a template that describes common movements of a robot (see Fig. 2 top right for an example of the template and the prompt for the LLM) to effectively harness LLMs' internal knowledge about motions. The role of the *Motion Descriptor* is to complete the provided template (e.g., replacing certain elements such as `CHOICE` and `NUM` in the example. This helps the *Motion Descriptor* produce more structured and predictable outputs and improves stability of the overall system. In addition, as we are describing the motion in natural language, we do not need to provide any specific examples in the prompt and can rely entirely on LLMs to generate the result.

Reward Coding In the second stage, we translate the generated motion description into the reward function using a second LLM. We formulate the problem of language to reward function as a code-writing task to benefit from the LLMs' knowledge of coding and code structure, thus we name the second LLM the *Reward Coder*. We design a prompt for instructing the LLM to generate reward specifying code (see example in Fig. 2 bottom right). The prompt consists of three parts: i) description of the reward APIs that the LLM can call to specify different parameters of the reward function, ii) an example response that we expect the *Reward Coder* to produce, and iii) the constraints and rules that the *Reward Coder* needs to follow. Note that the example is to demonstrate to the LLM how the response should look like, instead of teaching it how to perform a specific task. As such, the *Reward Coder* needs to specify the reward parameters based on its own knowledge about the motion from the natural language description.

3.3 Motion Controller

The Motion Controller needs to map the reward function generated by the Reward Translator to low-level robot actions $\mathbf{a}_{1:H}$ that maximize the accumulated reward $J(\mathbf{a}_{1:H})$. There are a few possible ways to achieve this, including using reinforcement learning (RL), offline trajectory optimization, or, as in this work, receding horizon trajectory optimization, i.e., model predictive control (MPC). At each control step, MPC plans a sequence of optimized actions $\mathbf{a}_{1:H}$ and sends to the robot. The robot applies the action corresponding to its current timestamp, advances to the next step, and sends the updated robot states to the MJPC planner to initiate the next planning cycle. The frequent re-planning in MPC empowers its robustness to uncertainties in the system and, importantly, enables interactive motion synthesis and correction. Specifically, we use an open-source implementation based on the MuJoCo simulator [48], MJPC [20]. MJPC has demonstrated the interactive creation of diverse behaviors such as legged locomotion, grasping, and finger-gaiting while supporting multiple planning algorithms, such as iLQG and Predictive Sampling. Following the observation by Howell et al [20], second-order planners such as iLQG produces smoother and more accurate actions while zeroth-order planners such as Predictive Sampling is better at exploring non-smooth

optimization landscape, we use iLQG for legged locomotion tasks, while use Predictive Sampling for manipulation tasks in this work.

4 Experiments

We design experiments to answer the following questions:

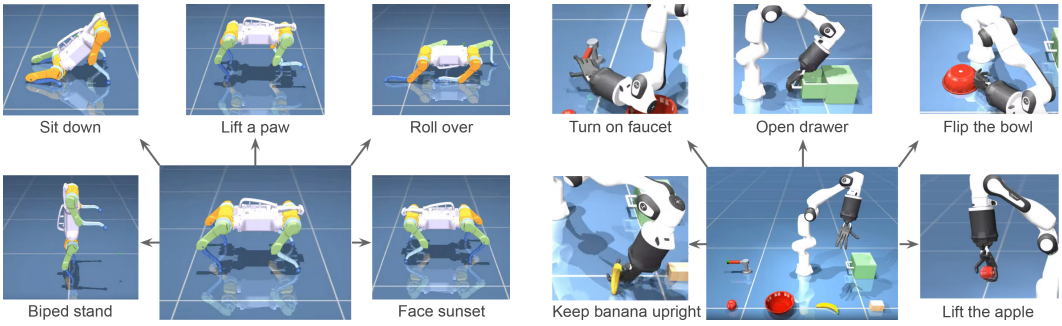
1) Is our proposed method, by combining LLMs and MJPC, able to generate diverse and complex robot motions through natural language interface?

2) Does interfacing with the reward function result in a more expressive pipeline than interfacing directly with low-level actions or primitive skills?

3) Is the Motion Descriptor necessary for achieving reliable performance for the system?

4.1 Experiment Setup

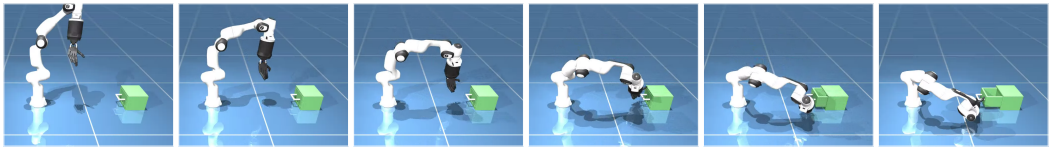
We evaluate our approach on two simulated robotic systems: a quadruped robot, and a dexterous robot manipulator (Fig. 3). Both robots are modeled and simulated in MuJoCo MPC [20]. In all experiments we use GPT-4 as the underlying LLM module [49]. Here we describe the key setups of each robot. More details regarding the full prompts and reward function can be found in the Appendix.



(a) Quadruped robot

(b) Dexterous manipulator robot

(c) Quadruped robot doing bipedal stand



(d) Dexterous manipulation robot opening a drawer.

Figure 3: The two robots used in our experiments and sampled tasks. (a) a Quadruped robot with 12 DoFs. (b) a dexterous manipulator robot with 27 DoFs. (c)(d) shows successful rollouts of sequences produced by our algorithm.

Quadruped Robot In this example, we demonstrate using our system to command a four legged robot (Fig. 3 (a)) to perform a variety of motor skills. The quadruped robot has 12 joints, 3 on each leg. Quadruped robots have been demonstrated to perform a large variety of skills including locomotion [39], hopping [18], biped standing [50, 51], etc. We apply our system to the quadruped robot to perform a similar suite of skills while only using natural language as input.

Dexterous Manipulator In the second example, we demonstrate our system on a dexterous manipulator robot. The robot consists of a 7 DoF Franka Emika arm and a 20 DoF shadow hand as the end-effector (as shown in Fig. 3 (b)). This creates a large action space, making it challenging to manually train a controller to directly perform tasks using this robot.

4.2 Baselines

We compare our proposed system to two baseline methods: i) an ablation of our approach that only uses *Reward Coder* without having access to the *Motion Descriptor*, and ii) Code-as-Policies [3] where the LLM generates a plan for the robot motion using a set of pre-defined robot primitive skills instead of reward functions. For the Code-as-Policies (CaP) baseline, we design the primitive skills based on common commands available to the robot.

For the quadruped robot, we use the following three primitive skills:

- `head_towards(direction)` specifies a target heading direction `direction` for the robot to reach.
- `walk(forward_speed, sideway_speed, turning_speed)` controls the robot to walk and turn in different directions. This is a common interface used in quadruped robots to navigate in different environments.
- `set_joint_poses(leg_name, joint_angles)` directly sets the joint positions for each DoF on the robot. To help the LLMs understand the joint angles, we provide a set of examples in the prompt.

For the dexterous manipulator robot, we use three primitive skills to control the robot motion and also a function to get access to the position of an object in the scene:

- `end_effector_to(position)` moves the center of the robot hand's palm to the given `position`.
- `end_effector_open()` opens the hand of the robot by extending all fingers.
- `end_effector_close()` closes the hand to form a grasping pose.
- `get_object_position(obj_name)` gets the position of a certain object in the scene.
- `get_joint_position(joint_name)` gets the position of a certain joint in the scene.

4.3 Tasks

We design nine tasks for the quadruped robot and seven tasks for the dexterous manipulator to evaluate the performance of our system in synthesizing different types of desired behaviors for the quadruped robot. Table 1 and Table 2 shows the list of tasks as well as their input instructions and Fig. 3 shows some samples of the tasks. Videos of all the tasks can be found at the project website [1].

For the quadruped robot, the tasks can be categorized into four types: *1) Heading direction control*, where the system needs to interpret indirect instructions about the robot's heading direction and to control the robot to face the right direction (e.g., identify the direction of sunrise or sunset). *2) Body pose control*, where we evaluate the ability of the system to understand and process commands to have the robot reach different body poses, inspired by common commands issued by human to dogs such as sit and roll over. *3) Limb control*, where we task the system to lift particular foot of the robot. Furthermore, we also test the ability of the system to take additional instructions to modify an existing skill, such turn in place with lifted feet. *4) Locomotion styles*, where we evaluate our proposed system in generating different locomotion styles. In particular, we design a challenging task of having the quadruped stand up on two back feet to walk in a bipedal mode.

For the dexterous manipulator, we design tasks to test its ability to achieve different types of interactions with objects such as lifting, moving, and re-orienting. We test the system on a diverse set of objects with significantly different shapes and sizes (Fig. 3) for each task. We further include two tasks that involves interacting with articulated objects of different joint types.

[1] ????

Task	Instructions
Facing sunrise	It's early in the morning, make the robot head towards the sun.
Facing sunset	It's late in the afternoon, make the robot head towards the sunset.
Sit down	Sit down low to ground with torso flat.
Roll Over	I want the robot to roll by 180 degrees.
Spin	Spin fast.
Lift one paw	I want the robot to lift its front right paw in the air.
Lift paw higher	I want the robot to lift its front right paw in the air. Lift it even higher.
Spin with lifted paws	Lift front left paw. Good, now lift diagonal paw as well. Good, in addition I want the robot to spin fast.
Stand up on two feet	Make the robot stand upright on two back feet like a human.

Table 1: List of tasks used in evaluation for the quadruped robot.

Task	Instructions
Touch object with hand	Touch the {`object`}
Lift object	Lift the {`object`} to $0.5\,\mathrm{m}$
Move object	Move the {`object_a`} to {`object_b`}
Upright object	Place the {`object`} upright
Flip object	Flip the {`object`}
Lift two objects	Lift the {`object_a`} and {`object_b`} at the same time.
Turn on the faucet	Turn on the faucet.
Open the drawer	Open the drawer.

Table 2: List of tasks used in evaluation for the dexterous manipulation.

4.4 Evaluation results

For each task and method considered, we generate 10 responses from Reward Translator , each evaluated in MJPC for 50 times, thus we measure the end-to-end stability of the full pipeline. Fig. 4 shows the results for both robots. Our proposed approach achieves significantly higher success rate for $11/17$ task categories and comparable performance for the rest tasks, showing the effectiveness and reliability of the proposed method. When compared to the CaP baseline, our method achieves better success rate in almost all tasks. This is due to that CaP can perform well on tasks that can be expressed by the given primitives (e.g. Touch object) or very close to the given examples in prompt (e.g. Sit down), but fails to generalize to novel low-level skills. On the other hand, using Reward Coder only can achieve success on some tasks but fails in ones that requires more reasoning. For example, when asked to lift an object higher, the Reward Coder only baseline often forget to task the robot hand to get closer to the object and only design the reward for encouraging the object to be higher, i.e. it does not shape the reward correctly.

To further understand the overall performance of different systems, we also show the pass rate in Fig. 4 right, which is a standard metric for analyzing code generation performance [8]. For each point in the plot, it represents the percentage of tasks the system can solve, given that it can generate N pieces of code for each task and pick the best performing one. As such, the pass rate curve measures the stability of the system (the more flat it is, the more stable the system is) as well as the task coverage of the system (the converged point represents how many tasks the system can solve given sufficient number of trials). It is clear from the result that for both embodiments, using reward as the interface empowers LLMs to solve more tasks more reliably, and the use of Structured Motion Description further boosts the system performance significantly.

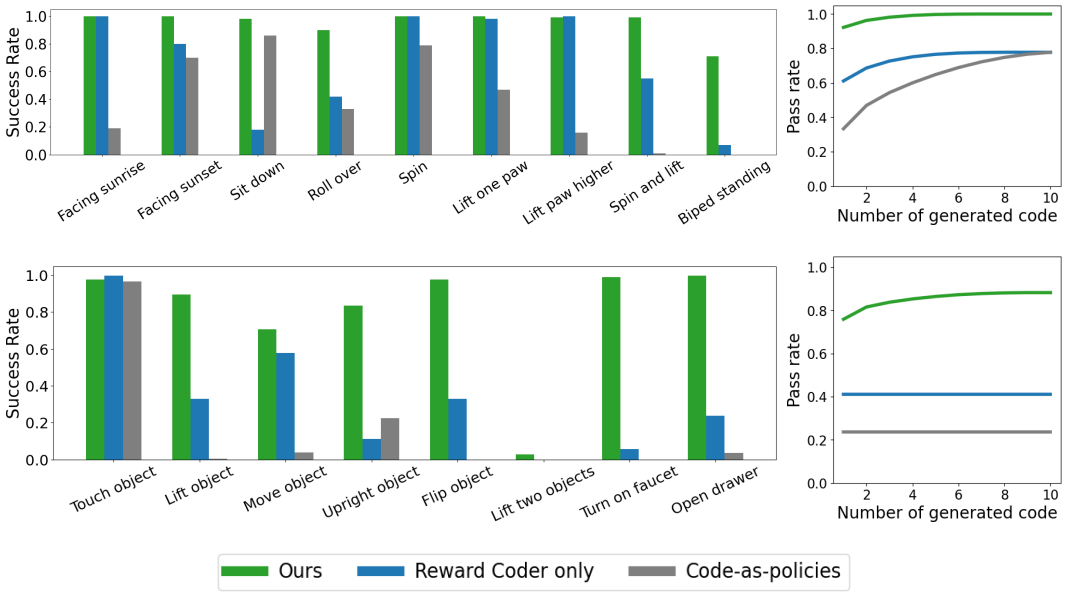
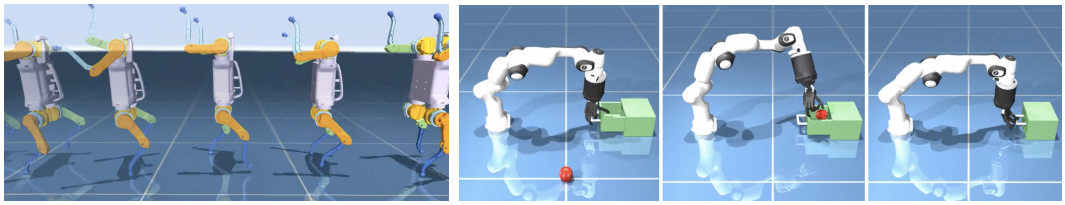


Figure 4: Comparison of our method and alternative methods in terms of pass rate: if we generate N pieces of code for each task and pick the best performing one, what's the percentage of tasks that the system can successfully tackle.

4.5 Interactive Motion Synthesis Results

One added benefit of using a real time motion synthesizing tool like MuJoCo MPC is that humans can observe the motion being synthesized in real time, and provide feedback. We showcase two examples where we teach the robot to perform complex tasks through multiple rounds of interactions, following human language corrections. In the first example, we task the quadruped robot to stand up and perform a moon-walk skill (Figure 5a). We give four instructions to achieve the task, each corrects the quadruped's motion to be closer to the desired movement. This showcase that users can interactively shape the behavior of the robot in natural language. In the second example, Results of the interactive results are best viewed in the supplementary video.



(a) The quadruped perform a moon-walk.

(b) The manipulator places an apple in the drawer.

Figure 5: The two interactive examples using our proposed system.

5 Discussion

Conclusion. In this work, we investigate a new paradigm for interfacing an LLM with a robot through reward functions, powered by a low-level model predictive control tool, MuJoCo MPC. Using reward function as the interface enables LLMs to work in a semantic-rich space that play to the strengths of LLMs, while ensures the expressiveness of the resulting controller. To further improve the performance of the system, we propose to use a structured motion description template to better extract internal knowledge about robot motions from LLMs. We evaluate our proposed system on two simulated robotic platforms: a quadruped robot and a dexterous manipulator robot. We apply our approach to both robots to acquire a wide variety of skills. Compared to alternative methods that do not use reward as the interface, or do not use the structured motion description, our

method achieves significantly better performance in terms of stability and the number of tasks it can solve.

Limitations and Future Work. Though we show that our system can obtain a diverse set of skills through natural language interactions, there are a few limitations. First, we currently design templates of structured motion descriptions for each type of robot morphology, which requires manual work. An interesting future direction is to unify or automate the template design to make the system easily extendable to novel robot morphologies. Second, our method currently relies on language as the interaction interface with human users. As such, it can be challenging to design tasks that are not easily described in language (e.g., "walk gracefully"). One potential way to mitigate this issue is to extend the system to multi-modal inputs to allow richer forms of user interactions (e.g., by showing a video of the desirable behavior). Thirdly, we currently use pre-defined reward terms whose weights and parameters are modulated by the LLMs. Constraining the reward design space helps improve stability of the system while sacrifices some flexibility. For example, our current design does not support time-varying rewards and would require re-designing the prompt to support that. Enabling LLMs to reliably design reward functions from scratch is thus an important and fruitful research direction.

References

[1] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

[4] A. Zeng, A. Wong, S. Welker, K. Choromanski, F. Tombari, A. Purohit, M. Ryoo, V. Sindhwani, J. Lee, V. Vanhoucke, et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

[5] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[6] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[7] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[9] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[10] S. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor. Chatgpt for robotics: Design principles and model abilities. *Microsoft Auton. Syst. Robot. Res*, 2:20, 2023.

[11] C. Snell, S. Yang, J. Fu, Y. Su, and S. Levine. Context-aware language modeling for goal-oriented dialogue systems. *arXiv preprint arXiv:2204.10198*, 2022.

[12] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[13] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

[14] E. Jang, A. Irpan, M. Khansari, D. Kappler, F. Ebert, C. Lynch, S. Levine, and C. Finn. BC-z: Zero-shot task generalization with robotic imitation learning. In *5th Annual Conference on Robot Learning*, 2021. URL https://openreview.net/forum?id=8kbp23tSGYv.

[15] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*, 2022.

[16] C. Lynch, A. Wahid, J. Tompson, T. Ding, J. Betker, R. Baruch, T. Armstrong, and P. Florence. Interactive language: Talking to robots in real time. *arXiv preprint arXiv:2210.06407*, 2022.

[17] J. Lee, J. Hwangbo, and M. Hutter. Robust recovery controller for a quadrupedal robot using deep reinforcement learning. *arXiv preprint arXiv:1901.07517*, 2019.

[18] J. Siekmann, Y. Godse, A. Fern, and J. Hurst. Sim-to-real learning of all common bipedal gaits via periodic reward composition. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7309–7315. IEEE, 2021.

[19] H.-T. L. Chiang, A. Faust, M. Fiser, and A. Francis. Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*, 4(2):2007–2014, 2019.

[20] T. Howell, N. Gileadi, S. Tunyasuvunakool, K. Zakka, T. Erez, and Y. Tassa. Predictive Sampling: Real-time Behaviour Synthesis with MuJoCo. dec 2022. doi:10.48550/arXiv.2212.00541. URL https://arxiv.org/abs/2212.00541.

[21] P. Goyal, S. Niekum, and R. J. Mooney. Using natural language for reward shaping in reinforcement learning. *arXiv preprint arXiv:1903.02020*, 2019.

[22] J. Lin, D. Fried, D. Klein, and A. Dragan. Inferring rewards from language in context. *arXiv preprint arXiv:2204.02515*, 2022.

[23] L. Fan, G. Wang, Y. Jiang, A. Mandlekar, Y. Yang, H. Zhu, A. Tang, D.-A. Huang, Y. Zhu, and A. Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=rc8o_j8I8PX.

[24] P. Sharma, B. Sundaralingam, V. Blukis, C. Paxton, T. Hermans, A. Torralba, J. Andreas, and D. Fox. Correcting robot plans with natural language feedback. In *Robotics: Science and Systems (RSS)*, 2022.

[25] M. Kwon, S. M. Xie, K. Bullard, and D. Sadigh. Reward design with language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[26] H. Hu and D. Sadigh. Language instructed reinforcement learning for human-ai coordination. In *40th International Conference on Machine Learning (ICML)*, 2023.

[27] S. Tellex, N. Gopalan, H. Kress-Gazit, and C. Matuszek. Robots that use language. *Annual Review of Control, Robotics, and Autonomous Systems*, 3:25–55, 2020.

[28] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[29] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox. Learning to parse natural language commands to a robot control system. In *Experimental robotics: the 13th international symposium on experimental robotics*, pages 403–415. Springer, 2013.

[30] A. Ku, P. Anderson, R. Patel, E. Ie, and J. Baldridge. Room-across-room: Multilingual vision-and-language navigation with dense spatiotemporal grounding. *arXiv preprint arXiv:2010.07954*, 2020.

[31] O. Mees, J. Borja-Diaz, and W. Burgard. Grounding language with visual affordances over unstructured data. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, London, UK, 2023.

[32] F. Ebert, Y. Yang, K. Schmeckpeper, B. Bucher, G. Georgakis, K. Daniilidis, C. Finn, and S. Levine. Bridge data: Boosting generalization of robotic skills with cross-domain datasets. *arXiv preprint arXiv:2109.13396*, 2021.

[33] J. Fu, A. Korattikara, S. Levine, and S. Guadarrama. From language to goals: Inverse reinforcement learning for vision-based instruction following. *arXiv preprint arXiv:1902.07742*, 2019.

[34] S. Karamcheti, M. Srivastava, P. Liang, and D. Sadigh. Lila: Language-informed latent actions. In *Proceedings of the 5th Conference on Robot Learning (CoRL)*, 2021.

[35] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv:2108.07732*, 2021.

[36] K. Ellis, C. Wong, M. Nye, M. Sable-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv:2006.08381*, 2020.

[37] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[38] F. Alet, J. Lopez-Contreras, J. Koppel, M. Nye, A. Solar-Lezama, T. Lozano-Perez, L. Kaelbling, and J. Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In *ICML*, 2021.

[39] L. Tian, K. Ellis, M. Kryven, and J. Tenenbaum. Learning abstract structure for drawing by efficient motor program induction. *NeurIPS*, 2020.

[40] D. Trivedi, J. Zhang, S.-H. Sun, and J. J. Lim. Learning to synthesize programs as interpretable and generalizable policies. *NeurIPS*, 2021.

[41] S. Nair, E. Mitchell, K. Chen, b. ichter, S. Savarese, and C. Finn. Learning language-conditioned robot behavior from offline data and crowd-sourced annotation. In A. Faust, D. Hsu, and G. Neumann, editors, *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 1303–1315. PMLR, 08–11 Nov 2022. URL https://proceedings.mlr.press/v164/nair22a.html.

[42] D. Bahdanau, F. Hill, J. Leike, E. Hughes, A. Hosseini, P. Kohli, and E. Grefenstette. Learning to understand goal specifications by modelling reward. *arXiv preprint arXiv:1806.01946*, 2018.

[43] A. Broad, J. Arkin, N. D. Ratliff, T. M. Howard, and B. Argall. Real-time natural language corrections for assistive robotic manipulators. *International Journal of Robotics Research (IJRR)*, 36:684–698, 2017.

[44] Y. Cui, S. Karamcheti, R. Palleti, N. Shivakumar, P. Liang, and D. Sadigh. "no, to the right"–online language corrections for robotic manipulation via shared autonomy. *arXiv preprint arXiv:2301.02555*, 2023.

[45] A. F. C. Bucker, L. F. C. Figueredo, S. Haddadin, A. Kapoor, S. Ma, and R. Bonatti. Reshaping robot trajectories using natural language commands: A study of multi-modal data alignment using transformers. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 978–984, 2022.

[46] A. F. C. Bucker, L. F. C. Figueredo, S. Haddadin, A. Kapoor, S. Ma, S. Vemprala, and R. Bonatti. Latte: Language trajectory transformer. *arXiv preprint arXiv:2208.02918*, 2022.

[47] N. Ziems, W. Yu, Z. Zhang, and M. Jiang. Large language models are built-in autoregressive search engines. *arXiv preprint arXiv:2305.09612*, 2023.

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

[49] OpenAI. Gpt-4 technical report. *arXiv*, 2023.

[50] L. Smith, J. C. Kew, T. Li, L. Luu, X. B. Peng, S. Ha, J. Tan, and S. Levine. Learning and adapting agile locomotion skills by transferring experience. *arXiv preprint arXiv:2304.09834*, 2023.

[51] Y. Fuchioka, Z. Xie, and M. van de Panne. Opt-mimic: Imitation of optimized trajectories for dynamic quadruped behaviors. *arXiv preprint arXiv:2210.01247*, 2022.

6 Appendix

6.1 Full Prompts

Figure 6, Figure 7, and Figure 8 shows the full prompts used in this work for the quadruped robot. Figure 9, Figure 10, and Figure 11 shows the full prompts used in this work for the dexterous manipulator robot.

```
Describe the motion of a dog robot using the following form:

[start of description]
* This {CHOICE: [is, is not]} a new task.
* The torso of the robot should roll by [NUM: 0.0] degrees towards right, the torso should pitch upward at [NUM: 0.0] degrees.
* The height of the robot's CoM or torso center should be at [NUM: 0.3] meters.
* The robot should {CHOICE: [face certain direction, turn at certain speed]}. If facing certain direction, it should be facing {CHOICE: [east, south, north, west]}. If turning, it should turn at [NUM: 0.0] degrees/s.
* The robot should {CHOICE: [go to a certain location, move at certain speed]}. If going to certain location, it should go to (x=[NUM: 0.0], y=[NUM: 0.0]). If moving at certain speed, it should move forward at [NUM: 0.0]m/s and sideways at [NUM: 0.0]m/s (positive means left).
* [optional] front_left foot lifted to [NUM: 0.0] meters high.
* [optional] back_left foot lifted to [NUM: 0.0] meters high.
* [optional] front_right foot lifted to [NUM: 0.0] meters high.
* [optional] back_right foot lifted to [NUM: 0.0] meters high.
* [optional] front_left foot extend forward by [NUM: 0.0] meters.
* [optional] back_left foot extend forward by [NUM: 0.0] meters.
* [optional] front_right foot extend forward by [NUM: 0.0] meters.
* [optional] back_right foot extend forward by [NUM: 0.0] meters.
* [optional] front_left foot shifts inward laterally by [NUM: 0.0] meters.
* [optional] back_left foot shifts inward laterally by [NUM: 0.0] meters.
* [optional] front_right foot shifts inward laterally by [NUM: 0.0] meters.
* [optional] back_right foot shifts inward laterally by [NUM: 0.0] meters.
* [optional] front_left foot steps on the ground at a frequency of [NUM: 0.0] Hz, during the stepping motion, the foot will move [NUM: 0.0] meters up and down, and [NUM: 0.0] meters forward and back, drawing a circle as if it's walking {CHOICE: forward, back}, spending [NUM: 0.0] portion of the time in the air vs gait cycle.
* [optional] back_left foot steps on the ground at a frequency of [NUM: 0.0] Hz, during the stepping motion, the foot will move [NUM: 0.0] meters up and down, and [NUM: 0.0] meters forward and back, drawing a circle as if it's walking {CHOICE: forward, back}, spending [NUM: 0.0] portion of the time in the air vs gait cycle.
* [optional] front_right foot steps on the ground at a frequency of [NUM: 0.0] Hz, during the stepping motion, the foot will move [NUM: 0.0] meters up and down, and [NUM: 0.0] meters forward and back, drawing a circle as if it's walking {CHOICE: forward, back}, spending [NUM: 0.0] portion of the time in the air vs gait cycle.
* [optional] back_right foot steps on the ground at a frequency of [NUM: 0.0] Hz, during the stepping motion, the foot will move [NUM: 0.0] meters up and down, and [NUM: 0.0] meters forward and back, drawing a circle as if it's walking {CHOICE: forward, back}, spending [NUM: 0.0] portion of the time in the air vs gait cycle.
* [optional] The phase offsets for the four legs should be front_left: [NUM: 0.0], back_left: [NUM: 0.0], front_right: [NUM: 0.0], back_right: [NUM: 0.0].

[end of description]

Rules:
1. If you see phrases like [NUM: default_value], replace the entire phrase with a numerical value. If you see [PNUM: default_value], replace it with a positive, non-zero numerical value.
2. If you see phrases like {CHOICE: [choice1, choice2, ...]}, it means you should replace the entire phrase with one of the choices listed. Be sure to replace all of them. If you are not sure about the value, just use your best judgement.
3. Phase offset is between [0, 1]. So if two legs' phase offset differs by 0 or 1 they are moving in synchronous. If they have phase offset difference of 0.5, they are moving opposite in the gait cycle.
4. The portion of air vs the gait cycle is between [0, 1]. So if it's 0, it means the foot will always stay on the ground, and if it's 1 it means the foot will always be in the air.
5. I will tell you a behavior/skill/task that I want the quadruped to perform and you will provide the full description of the quadruped motion, even if you may only need to change a few lines. Always start the description with [start of description] and end it with [end of description].
6. We can assume that the robot has a good low-level controller that maintains balance and stability as long as it's in a reasonable pose.
7. You can assume that the robot is capable of doing anything, even for the most challenging task.
8. The robot is about 0.3m high in CoM or torso center when it's standing on all four feet with horizontal body. It's about 0.65m high when it stand upright on two feet with vertical body. When the robot's torso/body is flat and parallel to the ground, the pitch and roll angles are both 0.
9. Holding a foot 0.0m in the air is the same as saying it should maintain contact with the ground.
10. Do not add additional descriptions not shown above. Only use the bullet points given in the template.
11. If a bullet point is marked [optional], do NOT add it unless it's absolutely needed.
12. Use as few bullet points as possible. Be concise.

If you understand, just say Yes. Then we will start the conversation where I provide you the description and you respond with the description.
```

Figure 6: Full prompt of Motion Descriptor used for quadruped.

6.2 Reward functions used in our experiments

In this work we use a set of generic reward functions for each embodiment that the LLMs can modulate. More specifically, we design a set of residual terms as in Equation 1 that are optimized

```
We have a description of a robot's motion and we want you to turn that into the corresponding program with following functions:

def set_torso_targets(target_torso_height, target_torso_pitch, target_torso_roll, target_torso_location_xy, target_torso_velocity_xy, target_torso_heading, target_turning_speed)
target_torso_height: how high the torso wants to reach. When the robot is standing on all four feet in a normal standing pose, the torso is about 0.3m high.
target_torso_pitch: How much the torso should tilt up from a horizontal pose in radians. A positive number means robot is looking up, e.g. if the angle is 0.5*pi the robot will be looking upward, if the angel is 0, then robot will be looking forward.
target_torso_velocity_xy: target torso moving velocity in local space, x is forward velocity, y is sideway velocity (positive means left).
target_torso_heading: the desired direction that the robot should face towards. The value of target_torso_heading is in the range of 0 to 2*pi, where 0 and 2*pi both mean East, pi being West, etc.
target_turning_speed: the desired turning speed of the torso in radians per second.
Remember:
one of target_torso_location_xy and target_torso_velocity_xy must be None.
one of target_torso_heading and target_turning_speed must be None.
No other inputs can be None.

def set_feet_pos_parameters(feet_name, lift_height, extend_forward, move_inward)
feet_name is one of ('front_left', 'back_left', 'front_right', 'back_right').
lift_height: how high should the foot be lifted in the air. If is None, disable this term. If it's set to 0, the foot will touch the ground.
extend_forward: how much should the foot extend forward. If is None, disable this term.
move_inward: how much should the foot move inward. If is None, disable this term.

def set_feet_stepping_parameters(feet_name, stepping_frequency, air_ratio, phase_offset, swing_up_down, swing_forward_back, should_activate)
feet_name is one of ('front_left', 'rear_left', 'front_right', 'rear_right').
air_ratio (value from 0 to 1) describes how much time the foot spends in the air versus the whole gait cycle. If it's 0 the foot will always stay on ground, and if it's 1 it'll always stay in the air.
phase_offset (value from 0 to 1) describes how the timing of the stepping motion differs between
different feet. For example, if the phase_offset between two legs differs by 0.5, it means
one leg will start the stepping motion in the middle of the stepping motion cycle of the other leg.
swing_up_down is how much the foot swings vertical during the motion cycle.
swing_forward_back is how much the foot swings horizontally during the motion cycle.
If swing_forward_back is positive, the foot would look like it's going forward, if it's negative, the foot will look like it's going backward.
If should_activate is False, the leg will not follow the stepping motion.

def execute_plan(plan_duration=2)
This function sends the parameters to the robot and execute the plan for `plan_duration` seconds, default to be 2

Example answer code:
import numpy as np  # import numpy because we are using it below

reset_reward() # This is a new task so reset reward; otherwise we don't need it
set_torso_targets(0.1, np.deg2rad(5), np.deg2rad(15), (2, 3), None, None, np.deg2rad(10))

set_feet_pos_parameters('front_left', 0.1, 0.1, None)
set_feet_pos_parameters('back_left', None, None, 0.15)
set_feet_pos_parameters('front_right', None, None, None)
set_feet_pos_parameters('back_right', 0.0, 0.0, None)
set_feet_stepping_parameters('front_right', 2.0, 0.5, 0.2, 0.1, -0.05, True)
set_feet_stepping_parameters('back_left', 3.0, 0.7, 0.1, 0.1, 0.05, True)
set_feet_stepping_parameters('front_left', 0.0, 0.0, 0.0, 0.0, 0.0, False)
set_feet_stepping_parameters('back_right', 0.0, 0.0, 0.0, 0.0, 0.0, False)

execute_plan(4)

Remember:
1. Always format the code in code blocks. In your response all four functions above: set_torso_targets, set_feet_pos_parameters, execute_plan, should be called at least once.
2. Do not invent new functions or classes. The only allowed functions you can call are the ones listed above. Do not leave unimplemented code blocks in your response.
3. The only allowed library is numpy. Do not import or use any other library. If you use np, be sure to import numpy.
4. If you are not sure what value to use, just use your best judge. Do not use None for anything.
5. Do not calculate the position or direction of any object (except for the ones provided above). Just use a number directly based on your best guess.
6. For set_torso_targets, only the last four arguments (target_torso_location_xy, target_torso_velocity_xy, target_torso_heading, target_turning_speed) can be None. Do not set None for any other arguments.
7. Don't forget to call execute_plan at the end.

If you understand, simply say Yes. Then we will start the conversation where I provide you the description and you respond with the code.
```

Figure 7: Full prompt of Reward Coder used for quadruped.

to reach zero by internally converting them to a l2 loss. Thus given a residual term $r(\cdot)$ a reward term can be recovered by $-||r(\cdot)||_2^2$. Below we describe the full set of residual terms we use in our experiments for each embodiment. For each term we select the weights for them to have about the same magnitude. The reward coder can adjust the parameters in each term and optionally set the weight to zero to disable a term.

6.2.1 Quadruped

Table 3 shows the residual terms used in the quadruped tasks. Note that for the foot-related terms, they are repeated for all four feet respectively. Furthermore, LLMs can optionally set the target foot positions $\mathbf{fp}$ directly or through a periodic function $max(asin(b2\pi + c), 0)$ where a is the magnitude of the motion, b is the frequency, and c is the phase offset.

6.2.2 Dexterous Manipulator

6.3 Details evaluation results for each task

Figure 13 and Figure 12.

```
We have a quadruped robot. It has 12 joints in total, three for each leg.
We can use the following functions to control its movements:

def set_target_joint_angles(leg_name, target_joint_angles)
leg_name is one of ('front_left', 'back_left', 'front_right', 'back_right').
target_joint_angles: a 3D vector that describes the target angle for the abduction/adduction, hip, and knee joint of the each leg.

def walk(forward_speed, sideway_speed, turning_speed)
forward_speed: how fast the robot should walk forward
sideway_speed: how fast the robot should walk sideways
turning_speed: how fast the robot should be turning (positive means turning right)

def head_towards(heading_direction)
heading_direction: target heading for the robot to reach, in the range of 0 to 2*pi, where 0 means East, 0.5pi means North, pi means West, and 1.5pi means South.

def execute_plan(plan_duration=10)
This function sends the parameters to the robot and execute the plan for `plan_duration` seconds, default to be 2

Details about joint angles of each leg:
abduction/adduction joint controls the upper leg to swinging inward/outward.
When it's positive, legs will swing outward (swing to the right for right legs and left for left legs).
When it's negative, legs will swing inward.

hip joint controls the upper leg to rotate around the shoulder.
When it's zero, the upper leg is parallel to the torso (hip is same height as shoulder), pointing backward.
When it's positive, the upper leg rotates downward so the knee is below the shoulder. When it's 0.5pi, it's perpendicular to the torso, pointing downward.
When it's negative, the upper leg rotates upward so the knee is higher than the shoulder.

knee joint controls the lower leg to rotate around the knee.
When it's zero, the lower leg is folded closer to the upper leg.
knee joint angle can only be positive. When it's 0.5pi, the lower leg is perpendicular to the upper leg. When it's pi, the lower leg is fully streching out and parallel to the upper leg.

Here are a few examples for setting the joint angles to make the robot reach a few key poses:
standing on all four feet:
set_target_joint_angles("front_left", [0, 1, 1.5])
set_target_joint_angles("back_left", [0, 0.75, 1.5])
set_target_joint_angles("front_right", [0, 1, 1.5])
set_target_joint_angles("back_right", [0, 0.75, 1.5])
execute_plan()

sit down on the floor:
set_target_joint_angles("front_left", [0, 0, 0])
set_target_joint_angles("back_left", [0, 0, 0])
set_target_joint_angles("front_right", [0, 0, 0])
set_target_joint_angles("back_right", [0, 0, 0])
execute_plan()

lift front left foot:
set_target_joint_angles("front_left", [0, 0.45, 0.35])
set_target_joint_angles("back_left", [0, 1, 1.5])
set_target_joint_angles("front_right", [0, 1.4, 1.5])
set_target_joint_angles("back_right", [0, 1, 1.5])
execute_plan()

lift back left foot:
set_target_joint_angles("front_left", [0, 0.5, 1.5])
set_target_joint_angles("back_left", [0, 0.45, 0.35])
set_target_joint_angles("front_right", [0, 0.5, 1.5])
set_target_joint_angles("back_right", [0, 0.5, 1.5])
execute_plan()

Remember:
1. Always start your response with [start analysis]. Provide your analysis of the problem within 100 words, then end it with [end analysis].
2. After analysis, start your code response, format the code in code blocks.
3. Do not invent new functions or classes. The only allowed functions you can call are the ones listed above. Do not leave unimplemented code blocks in your response.
4. The only allowed library is numpy. Do not import or use any other library. If you use np, be sure to import numpy.
5. If you are not sure what value to use, just use your best judge. Do not use None for anything.
6. Do not calculate the position or direction of any object (except for the ones provided above). Just use a number directly based on your best guess.
7. Write the code as concisely as possible and try not to define additional variables.
8. If you define a new function for the skill, be sure to call it somewhere.
9. Be sure to call execute_plan at the end.

If you understand, simply say Yes. Then we will start the conversation where I provide you the description and you respond with the code.
```

Figure 8: Full prompt of CaP+Primitive Skills baseline used for quadruped.

Residual Term	**Formulation**	**Default weight**
CoM X-Y position	$\lvert\mathbf{p}_{xy} - \bar{\mathbf{p}}_{xy}\rvert$	0.3
CoM height	$\mathbf{p}_z - \bar{\mathbf{p}}_z$	1.0
base yaw	$\mathbf{p}_{yaw} - \bar{\mathbf{p}}_{yaw}$	0.3
base pitch	$\mathbf{p}_{pitch} - \bar{\mathbf{p}}_{pitch}$	0.6
base roll	$\mathbf{p}_{roll} - \bar{\mathbf{p}}_{roll}$	0.1
forward velocity	$\dot{\mathbf{p}}_x - \bar{\dot{\mathbf{p}}}_x$	0.1
sideways velocity	$\dot{\mathbf{p}}_y - \bar{\dot{\mathbf{p}}}_y$	0.1
yaw speed	$\dot{\mathbf{p}}_{yaw} - \bar{\dot{\mathbf{p}}}_{yaw}$	0.1
foot local position x	$\mathbf{fp}_x - \bar{\mathbf{fp}}_x$	1
foot local position y	$\mathbf{fp}_y - \bar{\mathbf{fp}}_y$	1
foot local position z	$\mathbf{fp}_z - \bar{\mathbf{fp}}_z$	2

Table 3: List of residual terms used for the quadruped robot. $\mathbf{p}$ denotes the position and orientation of the robot's torso. $\mathbf{fp}$ denotes the position of the robot's foot (in local space). $\bar{(\cdot)}$ means the target value and $\dot{(\cdot)}$ means the time-derivative of the quantity.

```
We have a manipulator and we want you to help plan how it should move to perform tasks using the following template:

[start of plan]
To perform this task, the manipulator's palm should move close to object1={CHOICE: apple, banana, box, bowl, drawer_handle, faucet_handle}.
object1 should be close to object2={CHOICE: apple, banana, box, bowl, drawer_handle, faucet_handle, nothing}.
object1 {CHOICE: need, do not need} to be rotated by {NUM: 0.0} degrees along x axis.
object2 {CHOICE: need, do not need} to be rotated by {NUM: 0.0} degrees along x axis.
object1 {CHOICE: need, do not need} to be lifted to a height of {NUM: 0.0}m.
object2 {CHOICE: need, do not need} to be lifted to a height of {NUM: 0.0}m.
This {CHOICE: is, is not} the first plan for a new task.
[optional] object3 {CHOICE: drawer, faucet} needs to be {CHOICE: open, closed}.
[end of plan]

Rules:
1. If you see phrases like [NUM: default_value], replace the entire phrase with a numerical value.
2. If you see phrases like {CHOICE: choice1, choice2, ...}, it means you should replace the entire phrase with one of the choices listed.
3. The environment contains apple, banana, box, bowl, drawer_handle, faucet_handle. Do not invent new objects not listed here.
4. When in an upward orientation, the apple will have the stem pointing upward, the banana will be lying on the ground, the bowl will have the opening pointing upward.
5. The bowl is large enough to have all other object put in there.
6. I will tell you a behavior/skill/task that I want the manipulator to perform and you will provide the full plan, even if you may only need to change a few lines. Always start the description with [start of plan] and end it with [end of plan].
7. You can assume that the robot is capable of doing anything, even for the most challenging task.
8. Your plan should be as close to the provided template as possible. Do not include additional details.

If you understand, say Yes.
```

Figure 9: Full prompt of Motion Descriptor used for dexterous manipulator.

```
We have a manipulator and we want you to help plan how it should move to perform tasks using the following APIs:

def set_l2_distance_reward(name_obj_A, name_obj_B)
where name_obj_A and name_obj_B are selected from ['palm', 'apple', 'banana', 'box', 'bowl', 'drawer_handle', 'faucet_handle'].
This term sets a reward for minimizing l2_distance between name_obj_A and name_obj_B.

def set_obj_orientation_reward(name_obj, x_axis_rotation_radians)
this term encourages the orientation of name_obj to be close to the target (specified by x_axis_rotation_radians).

def execute_plan(duration=2)
This function sends the parameters to the robot and execute the plan for `duration` seconds, default to be 2

def reset_reward()
This function resets the reward to default values.

def set_obj_z_position_reward(name_obj, z_height)
this term encourages the orientation of name_obj to be close to the height (specified by z_height).

def set_joint_fraction_reward(name_joint, fraction)
This function sets the joint to a certain value between 0 and 1. 0 means close and 1 means open.
name_joint needs to be select from ['drawer', 'faucet']

def reset_reward()
This function resets the reward to default values.

Example answer code:
import numpy as np

reset_reward() # This is a new task so reset reward; otherwise we don't need it
set_l2_distance_reward("palm", "apple")
set_l2_distance_reward("apple", "bowl")
set_obj_orientation_reward("bowl", np.deg2rad(30))
set_obj_z_position_reward("bowl", 1.0)

execute_plan(4)

Remember:
1. Always format the code in code blocks. In your response execute_plan should be called exactly once at the end.
2. Do not invent new functions or classes. The only allowed functions you can call are the ones listed above. Do not leave unimplemented code blocks in your response.
3. The only allowed library is numpy. Do not import or use any other library.
4. If you are not sure what value to use, just use your best judge. Do not use None for anything.
5. Do not calculate the position or direction of any object (except for the ones provided above). Just use a number directly based on your best guess.
6. You do not need to make the robot do extra things not mentioned in the plan such as stopping the robot.


If you understand, simply say Yes and repeat the constraints you must follow. Then we will start the conversation where I provide you the plan and you respond with the code.
```

Figure 10: Full prompt of Reward Coder used for dexterous manipulator.

Residual Term	**Formulation**	**Default weight**
move obj1 close to obj2	$\lvert\mathbf{c1}_{xyz} - \mathbf{c2}_{xyz}\rvert$	5
move obj to target X-Y position	$\lvert\mathbf{c}_z - \bar{\mathbf{c}}_z\rvert$	5
move obj to target height	$\lvert\mathbf{c}_{xy} - \bar{\mathbf{c}}_{xy}\rvert$	10
move obj to target orientation	$\lvert\mathbf{o}_{obj} - \bar{\mathbf{o}}\rvert$	
move joint to target value	$q - \bar{q}$	10

Table 4: List of reward terms used for the dexterous manipulator robot. $\mathbf{c}$ denotes the position of the object, $\mathbf{o}$ denotes the orientation of the object, q is the degree of freedom to be manipulated.

We have a manipulator and we want you to help plan how it should move to perform tasks using the following APIs:

```
def end_effector_to(position_obj)
position_obj is a list of 3 float numbers [x,y,z]

def end_effector_open()
Open the end effector.

def end_effector_close()
Close the end effector.

def get_object_position(obj_name:str)
Given an object name, return a list of 3 float numbers [x,y,z].
the object can come from a list of ['apple', 'banana', 'bowl', 'box', 'drawer_handle', 'faucet_handle']

def get_normalized_joint_position(joint_name:str)
Given an joint name, return a float numbers x.
the joint can come from a list of ['drawer', 'faucet']

def reset()
Reset the agent.
```

Example answer code:

```
import numpy as np

reset()
apple_pos = get_object_position("apple")
end_effector_to(apple_pos)
```

Remember:
1. Always format the code in code blocks.
2. Do not invent new functions or classes. The only allowed functions you can call are the ones listed above. Do not leave unimplemented code blocks in your response.
3. The only allowed library is numpy. Do not import or use any other library.
4. If you are not sure what value to use, just use your best judge. Do not use None for anything.
5. Do not calculate the position or direction of any object (except for the ones provided above). Just use a number directly based on your best guess.
6. You do not need to make the robot do extra things not mentioned in the plan such as stopping the robot.
7. Try your best to generate code despite the lack of context.

If you understand, simply say Yes and repeat the constraints you must follow. Then we will start the conversation where I provide you the plan and you respond with the code.

Figure 11: Full prompt of CaP+Primitive Skills baseline used for dexterous manipulator.

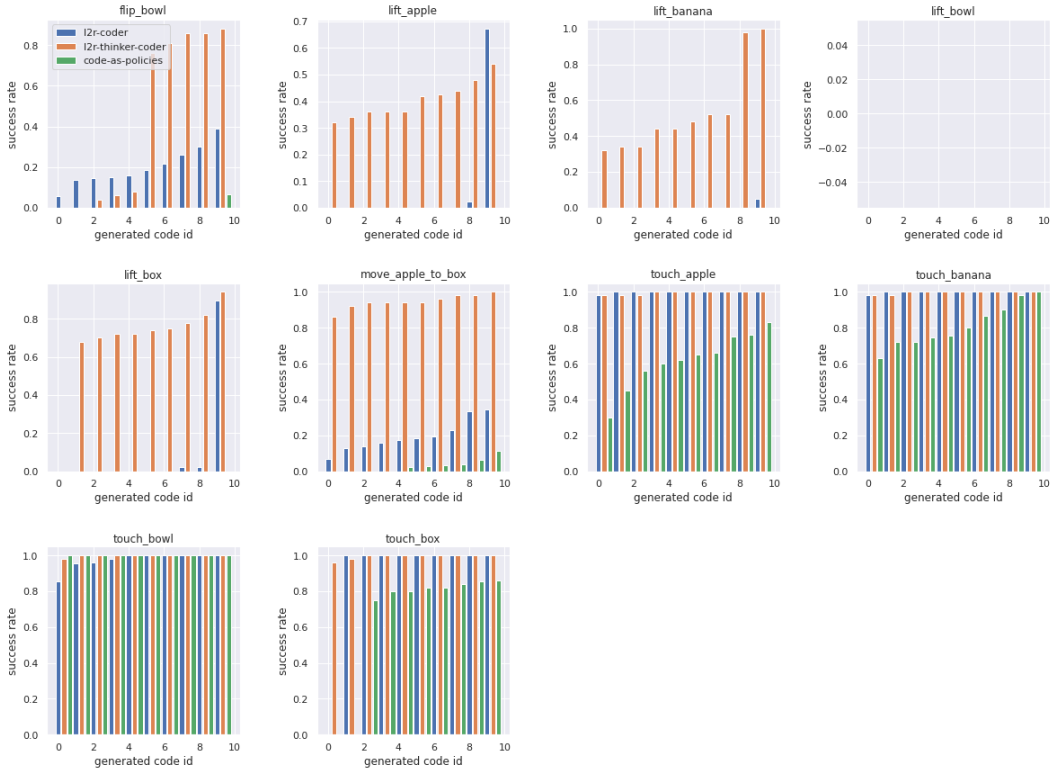


Figure 12: .

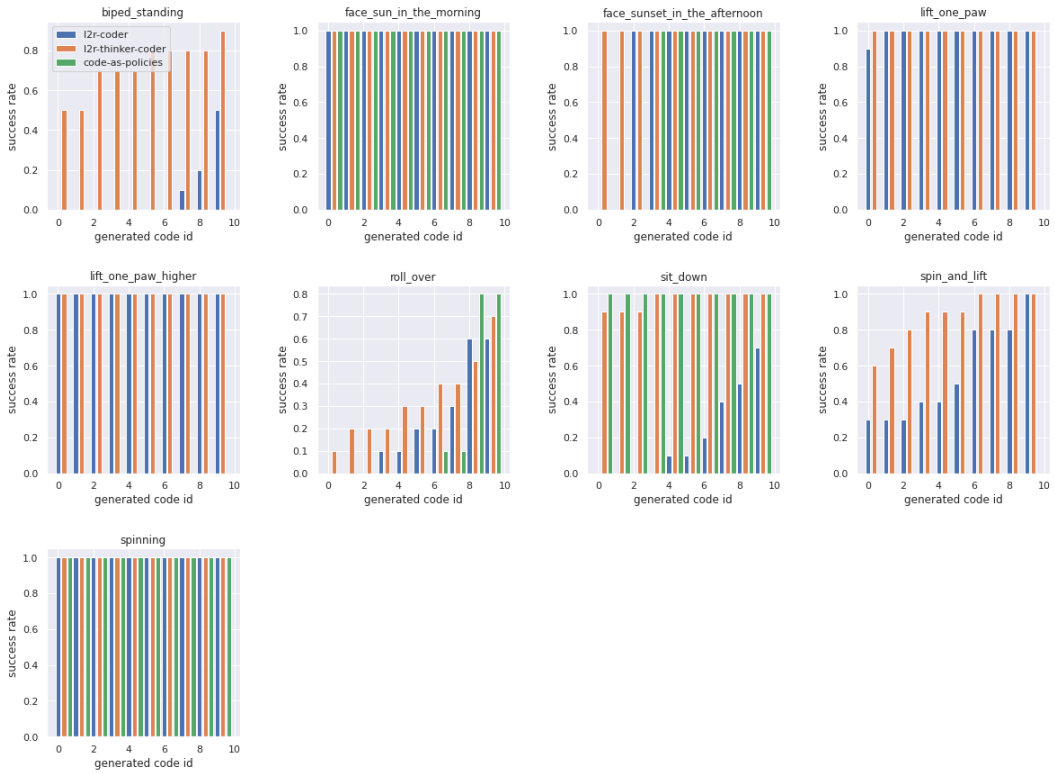

biped_standing
face_sun_in_the_morning
face_sunset_in_the_afternoon
lift_one_paw
lift_one_paw_higher
roll_over
sit_down
spin_and_lift
spinning
l2r-coder
l2r-thinker-coder
code-as-policies
success rate
generated code id

Figure 13: .