Meta-Control: Automatic Model-based Control Synthesis for Heterogeneous Robot Skills

Tianhao Wei[*,1], **Liqian Ma**[*,1,2], **Rui Chen**[1], **Weiye Zhao**[1], **Changliu Liu**[1]
[*] Equal contribution, [1] Carnegie Mellon University, [2] Tsinghua University

Abstract: The requirements for real-world manipulation tasks are diverse and often conflicting; some tasks necessitate force constraints or collision avoidance, while others demand high-frequency feedback. Satisfying these varied requirements with a fixed state-action representation and control strategy is challenging, impeding the development of a universal robotic foundation model. In this work, we propose *Meta-Control*, the first LLM-enabled automatic control synthesis approach that creates customized state representations and control strategies tailored to specific tasks. *Meta-Control* leverages a generic hierarchical control framework to address a wide range of heterogeneous tasks. Our core insight is the decomposition of the state space into an abstract task space and a concrete tracking space. By harnessing LLM's extensive common sense and control knowledge, the LLM designs these spaces, including states, dynamic models, and controllers, using pre-defined but abstract templates. *Meta-Control* stands out for its fully model-based nature, allowing for rigorous analysis, efficient parameter tuning, and reliable execution. It not only utilizes decades of control expertise encapsulated within LLMs to facilitate heterogeneous control but also ensures formal guarantees such as safety and stability. Our method is validated both in real-world scenarios and simulations across diverse tasks with conflicting requirements, such as collision avoidance versus convergence and compliance versus high precision. Videos and additional results are at meta-control-paper.github.io

Keywords: Embodied agent, Model-based Control, LLM, Manipulation

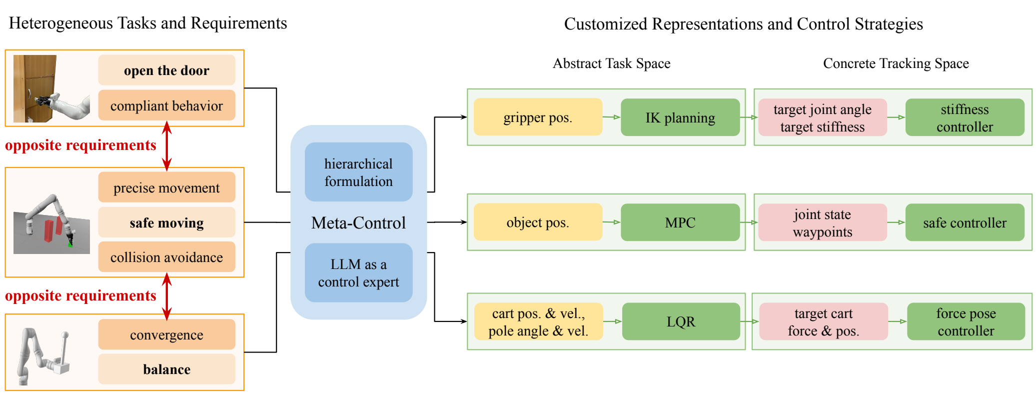

Figure 1: Open world robot tasks have inherently different and even opposite requirements that can be difficult to satisfy holistically with fixed state-action representations and fixed control strategies. To address this problem, *Meta-Control* leverages hierarchical formulation and LLMs, enabling automatic control synthesis by automatically customizing the most appropriate state action representations, dynamic models, and controllers.

1 Introduction

Recent advances in robot learning have enabled embodied agents to interact intelligently with the environments via methods such as predefined action primitives (i.e. skills) [1, 2, 3], parameterized policies [4, 5, 6], and reinforcement learning (RL) [7, 8, 9].

However, existing methods face a common challenge: real-world tasks exhibit inherently diverse and often conflicting requirements that are difficult to satisfy with a fixed control strategy. For instance, as illustrated in fig. 1, a pick-and-place task requires precise movements to ensure collision avoidance, which requires safe position control. Conversely, position control is unsuitable for an open-door task due to the difficulty of planning a trajectory that perfectly aligns with the door's swing path, making compliant control more favorable. Similarly, pole balancing tasks demand controllers that ensure the pole's convergence (a position-attracting goal) that opposes collision avoidance (a position-avoiding goal). On the other hand, although visuomotor methods can potentially address heterogeneous tasks, they often lack reliability and explainability. Models with predefined action pools may cover different task types but require manual construction and have limited applicability, hindering their scalability to diverse tasks. These challenges restrict existing methods from generalizing to various open-world manipulation tasks with varying constraints.

To overcome these limitations, we propose *Meta-Control*, a novel method that automatically customizes robot skills for diverse challenges of open-world tasks. Our core insight is that, although it is difficult to overcome arbitrary challenges with a fixed combination of representation and control strategy, each challenge can be effectively tackled with a specialized combination. If agents could automatically select suitable representation and control strategies, they would be able to handle diverse open-world tasks. For example, the agent would select a compliant controller for opening a door, and a hybrid position-force controller for board wiping tasks

Motivated by these observations, we formulate robot skill design as a control system synthesis problem and leverage LLMs to design the most appropriate representation and control strategy for a given task. We propose a hierarchical design scheme that eases the synthesis while maintaining high generalizability. The hierarchy involves a high-level task space and a low-level tracking space. The task space is an abstracted, intuitive space for accomplishing the task (e.g., Cartesian space or gripper pose space) while the tracking space usually represents the robot state space where low-level constraints can be specified and task-level commands are followed. The LLM determines both the task and tracking spaces, along with the corresponding dynamic models and controllers for each space. In general, the task controller focuses on high-level objectives, while the tracking controller emphasizes low-level control with constraint satisfaction. This hierarchical design significantly enhances the capability of synthesized control systems compared to motion primitive methods.

Meta-Control offers several benefits: 1. It enables the synthesis of challenging heterogeneous robotic skills for unseen tasks, allowing each task to be accomplished with the most suitable representation and control system tailored to task-specific requirements. 2. Unlike previous work that primarily utilizes spatial priors from LLMs (e.g., object localization), *Meta-Control* leverages the internalized control knowledge of LLMs which encompasses decades of modeling and optimization efforts in the form of control system design. 3. The synthesized control system is fully model-based which allows rigorous analysis, efficient parameter tuning, and formal guarantees (e.g., safety and stability), leading to reliable and trustworthy execution compared to end-to-end methods.

2 Related work

Recent advancements in designing embodied agents for diverse tasks can be broadly classified into four categories: skill libraries, parameterized policies, end-to-end methods, and representation learning. *Meta-Control* integrates aspects of these approaches. A *Meta-Control* synthesized control system can be viewed as an end-to-end parameterized policy composed hierarchically of unit primitives. The representation is chosen by LLM. The parameters are inferred by LLM from trial and error, akin to RL with an LLM provided reward. By integrating the strengths of these approaches, *Meta-Control* sets a new standard for synthesizing robot skills, offering a flexible, robust, and explainable solution for a wide range of real-world tasks.

Skill library allows diverse control strategies and composite execution through predefined libraries of motion primitives with high-level APIs. These libraries can be dynamically called or combined by the LLM to accomplish tasks [1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] Other works build skill libraries through behavior cloning, reinforcement learning, or bootstrapping [5, 20, 10, 21, 22, 23, 24, 25, 26, 19, 27, 28, 29]. However, these methods require manual construction and are often limited to specific task types (e.g., pick and place). In contrast, *Meta-Control* can synthesize new, heterogeneous skills on the fly using predefined dynamic models and controller templates.

Parameterized policy differs from the skill library methods in using a fixed strategy, such as MPC or RL with varying parameters to compose policies for different tasks. The parameters may include dynamic models [30, 31, 32, 33, 34, 35], constraints [36, 4, 37], or costs / objectives / rewards [38, 39, 40, 41, 7, 8]. These parameters can be inferred by LLMs [4, 41, 7, 8], or learned from data [30, 38, 39]. Another typical case is a hierarchical policy in which the parameters are high-level commands for an instruction following controller [42, 29]. A notable limitation is that the chosen parameterized policy inherently constrains the method's capability. For example, an MPC that generates end-effector actions in Cartesian space cannot produce force-compliant control in joint space. RL-based methods that learn parameters from rewards are unsuitable for online skill synthesis. In comparison, *Meta-Control* can dynamically generate parameterized policies and is highly flexible.

End-to-End models directly map perceptions to robot actions. [6, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52] leverage visual language models (VLMs) to translate instructions and observations to robot arm actions. Besides explicitly mapping observation and instruction to action, it's also admissible to first learn energy function [53], pixel-wise or voxel-wise affordance [54, 55, 56], and then extract actions from that energy function or affordance map. Recent works utilize diffusion models to learn from demonstration [57, 58, 20], enabling multimodal action distribution. End-to-end methods often output Cartesian actions (position, orientation, velocity, etc.), requiring additional motion planning to generate joint-level movement, and therefore inherently limit the capability. Furthermore, the training data domain limits the generalizability. *Meta-Control* overcomes these limitations by integrating LLM with model-based control strategies, ensuring robustness and explainability.

Representation learning maps high-dimensional observations into low-dimensional representations that are more semantically meaningful and easier to control with [59, 60, 61]. It can be combined with various control strategies [62]. Efforts to develop generic representations for different tasks include task-specific representation [63, 64], visual representation [65, 61, 66, 67, 68, 69, 70, 71], and multi-modal representations [72, 73, 55, 67, 74, 75]. However, learned representations often require fixed raw inputs (e.g., camera input) and lack explainability. In contrast, *Meta-Control* selects a subset of available observations for representation, enhancing explainability and adaptability to different inputs.

3 Method

In this work, we focus on synthesizing robot skills, defined as unit actions (e.g., grab the eraser, erase the marks) from robot tasks instructed via free-form language $\mathcal{L}$ (e.g., clean the whiteboard) [4]. We assume that the decomposition from task to skills: $\mathcal{L} \rightarrow \ell_1, \ell_2, \dots, \ell_n$ is given by a task-level planner, which can be LLM-based or search-based. Our focus is to synthesize a control system to accomplish each of the skills described by ℓ_i.

Control system synthesis is a fundamental aspect of control engineering that involves designing and implementing control systems to manage the behavior of dynamic systems. The goal of control system synthesis is to ensure that a system operates as desired, achieving specified performance criteria such as stability, accuracy, and efficiency. The process typically involves two key steps: system modeling and controller synthesis. Directly designing the control system for open-world skills is very difficult because the design space is infinite. Existing work usually simplifies the

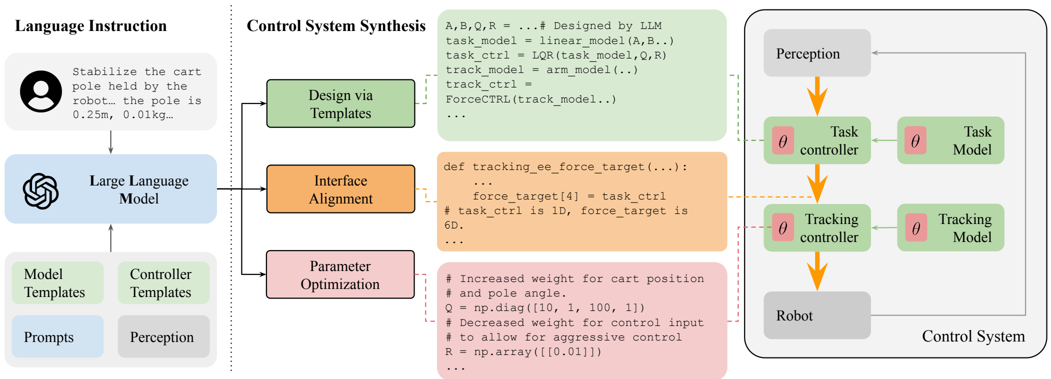

Figure 2: Overview of *Meta-Control*. The skill description is given by the user. Other prompts are automatically generated based on the system setup and code base. The composer generates h, f, v, u based on the prompts, then evaluate the synthesized controllers and revise them based on execution results and optional human feedback.

process in different ways as we have discussed in section 2. We need a proper design scheme that is easy to follow and generalize to a wide range of skills. To address these challenges, we first present a hierarchical control formulation for skill synthesis and then introduce the three key procedures for control system synthesis with LLM: design via templates, interface alignment, and parameter optimization.

3.1 Generic Hierarchical Control Framework

Our key insight is that a bilevel hierarchical control framework can represent various skills. Given a skill description ℓ_i, we can synthesize a high-level task controller in an intuitive abstract space and a low-level tracking controller that tracks the high-level control in the robot state space. We formulate the problem in three spaces: state space, measurement space, and task space.

State space: $\dot{x} = f(x, u)$ **Measurement space:** $y = g(x, u)$ **Task space:** $\dot{z} = h(z, v)$

where x is the system state, u is the state space control input, y is the output or measurement of the system, z is the state of the task space, and v is the task space control input. The task space contains intuitive and high-level states, such as the gripper poses for robot arms and the center-of-mass for quadrupeds. y has to be measurable or extractable from perception. f, g, h are dynamic models of the corresponding system. The task space can sometimes be omitted when it coincides with the state space, such as when a robot arm skill directly specifies joint goals. We denote the task space controller by $\pi_v(y)$, and the tracking space controller by $\pi_u(y, v)$. Essentially, we decompose skill synthesis into two sub-problems:

$$\begin{array}{c} \min_{\pi(y)} J(x(t), u(t)) \\ s.t.\ c(x) \leq 0, \end{array} \implies \begin{array}{c} \min_{\pi_v(y)} J_z(z(t), v(t)) \\ s.t.\ c_z(z) \leq 0, \end{array} \quad \& \quad \begin{array}{c} \min_{\pi_u(y,v)} J_x(x(t), u(t)) \\ s.t.\ c_x(x) \leq 0 \end{array}$$

The first sub-problem solves the task control input v under task-space constraint c_z. The second sub-problem tracks v by solving the state control input u under state-space constraint c_x. Our goal is to use LLM to 1) implicitly design the proper task state z; 2) explicitly design the dynamic models $h(z, v)$ and $f(x, u)$; 3) implicitly design the objective J_z and J_x and the constraints $c_z(z)$ and $c_x(x)$. 4) explicitly design the task and tracking controller $\pi_v(y)$, $\pi_u(y, v)$. 5) Adjust the parameters for h, f, $\pi_v(y)$, and $\pi_u(y, v)$ to achieve the desired performance.

3.2 Control System Synthesis by LLM

Designing z, h, f, π_v, and π_u directly is challenging due to the infinite possible spaces and dynamic models, the need for accurate and deep understanding of the robotic system and data flow, and the dependencies between different modules. To address these challenges, we define model and

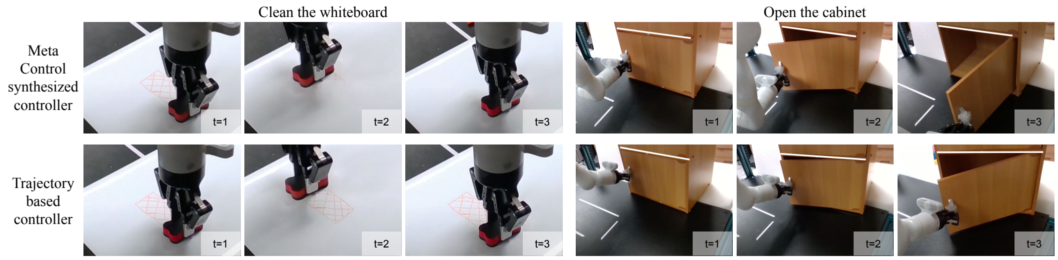

Figure 3: Comparison of Meta-control and trajectory planning based method on real robot for wiping board and opening door. The trajectory based method fails to erase the mark because it neglects force requirement. Furthermore, opening a door with a trajectory-based method leads to displacement of the cabinet because the planned trajectory does not precisely align with the door's movement. That may damage the door if the cabinet is fixed. In contrast, *Meta-Control* addressed these challenges with properly customized control systems.

controller templates to constrain the form of dynamic models and controllers. Then, an LLM synthesizes the control system following three stages: design via templates, interface alignment, and parameter optimization (see fig. 2).

Design via Templates The LLM designs the control system by selecting the appropriate models and controllers from the template library provided in the prompt. Templates are predefined object classes that need to be instantiated with concrete arguments. For example, we offer a dynamic model template called `LinearModel` which can be instantiated by passing in four matrices A, B, C, D describing a linear system, and a controller template `LQRController` requires matrices Q, R and vectors x_0, u_0. The LLM chooses whether a template is for the task space or the tracking space. Templates include functionality descriptions, argument requirements, and interface format. The design also relies on available measurements and expected control signals, which vary across tasks.

Interface Alignment The LLM connects the instantiated modules to each other and the robot by matching the outputs and inputs. It extracts the input of the task controller from available measurements y, the input of the tracking controller from the task controller's output v, and the robot control from the tracking controller's output u. For example, in the balance cart pole skill, the task controller `LQRController` takes a 4D vector as input, requiring the LLM to extract these states from more than 20 available measurements. Then, the `LQRController` computes the desired force on the cart, which is a scalar, while the downstream tracking `PoseForceHybridController` requires two 6D vectors as input for desired pose and force. The LLM must prepare the tracking controller input by padding the task controller output.

Parameter Optimization The LLM infers unknown parameters for the dynamic models and controllers via trial and error. For example, the LQR controller requires setting the Q and R matrices, which are difficult to infer directly. The LLM identifies which parameters need tuning, selects performance metrics from measurements, and tunes the parameters over multiple runs to improve performance metrics. For example, in the balance cart-pole skill, LLM identifies Q and R as tuning parameters and selects cart position and pole angle as metrics. This process is efficient due to the well-studied parameter-performance relationships in common dynamicsand controllers as the internalized knowledge of LLMs.

4 Experiment

The experiment is designed to manifest the following features: 1) *Meta-Control* enables synthesizing skills for heterogeneous requirements; 2) *Meta-Control* exploits control knowledge from LLM; 3) The *Meta-Control* pipeline is critical to the success; 4) *Meta-Control* works on a wide range of open world tasks; 5) *Meta-Control* enables model-based analysis, providing formal guarantees on various properties. 6) *Meta-Control* transfers to real robot and different embodiments easily.

We implement our pipeline with Drake[76], a framework designed for model-based control. For the hardware experiment, we used a Kinova Gen3 robot arm. The language model is GPT 4.0.

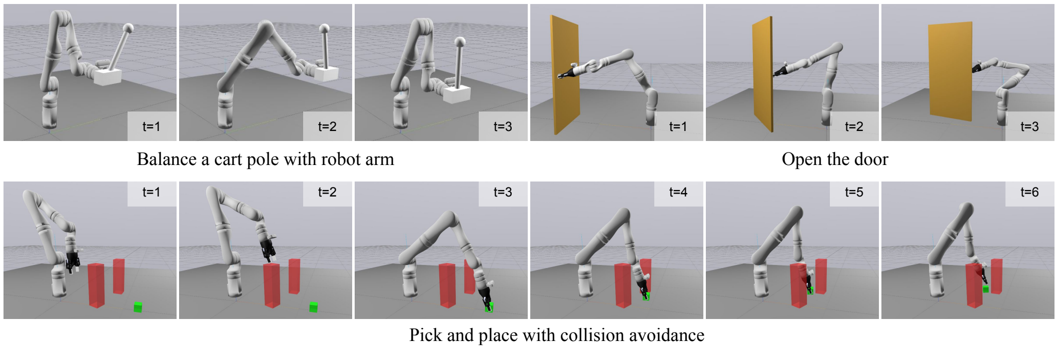

Figure 4: Three manipulation tasks that have inherently different challenges and requirements. The balance task requires an accurate, high-frequency, feedback controller. The open door task requires properly handling articulated objects, the executed trajectory has to perfectly match the swing path. The safe pick and place task requires guaranteeing collision avoidance for the whole robot arm.

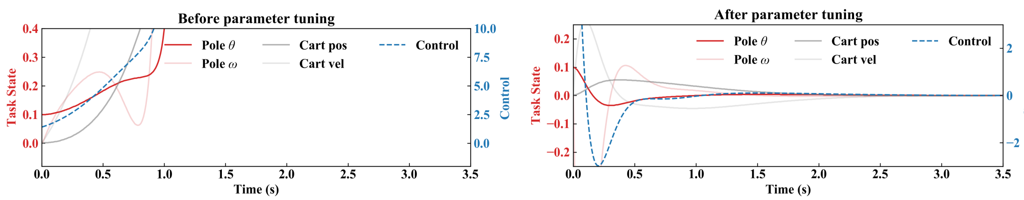

Figure 5: *Meta-Control* can automatically identify hyper-parameters that require tuning and tune them to accomplish challenging tasks. The figure shows the trajectory of the arm-held cart-pole system before and after tuning the synthesized controller. The hyper-parameters Q =diag$(10, 1, 100, 1)$, $R = 0.01$ are chosen and tuned by the LLM with only 2 rounds of trial-and-error.

4.1 Meta control enables synthesizing challenging skills

As shown in fig. 3 and fig. 4, *Meta-Control* successfully synthesizes controllers for various challenging tasks with inherently different requirements, both in simulation and real-world. The task challenges and synthesized control systems are summarized in table 1. A more detailed description can be found in appendix B. fig. 5 shows that *Meta-Control* can identify core metrics and critical parameters that affect performance, then efficiently and effectively tune the parameters based on the metric through trial and error. The complete process of skill synthesis can be found at appendix C.

Task	Challenge	*Meta-Control* designed system
Open the door	The robot trajectory must *perfectly align* with the door's swing path. position control can easily lead to damage or failure.	KinematicTrajectoryMPC + CartesianStiffnessController, allowing imperfect trajectory planning and tracking with compliant behavior to avoid damage or failure.
Wipe the whiteboard	Two different objectives: tracking position and maintaining force	CartesianTrajectoryController + PoseForceController, allowing position tracking while maintaining a desired force on the whiteboard.
Balance the cart pole	The pole is *non-actuated*. The system is sensitive, requiring high-frequency feedback and convergence guarantee.	LQRController + PoseForceController. The LQR controller gives the force to be applied on the cart, and the hybrid position/force controller tracks the desired force on the y-axis while maintaining a neutral pose on the x-axis and the z-axis.
Collision-free pick and place	*whole-body* collision free in *continuous* time during whole task.	KinematicTrajectoryMPC + SafeController, allowing discrete-time planning and continuous-time whole-body collision-free tracking.

Table 1: Experiment tasks, challenges and *Meta-Control* synthesized controllers..

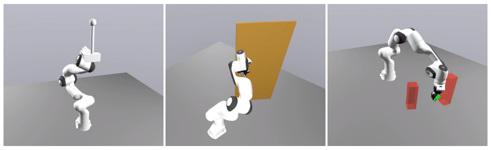

Figure 6: *Meta-Control* generalizes to different embodiments because the synthesized controller is fully model-based. A controller synthesized on Kinova can transfer to Franka Pranda simply by replacing the robot dynamic model.

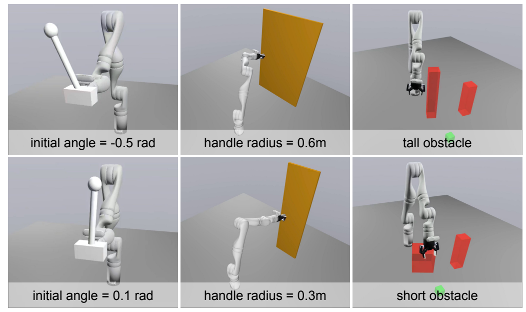

Figure 7: *Meta-Control* synthesized control systems are robust to attribute/state changes because of the model-based design.

Method		API	API + Formulation	API + Template	*Meta-Control*
Balance	design	30%	90%	60%	100%
	implementation	0%	30%	20%	90%
	execution	0%	0%	0%	70%
Open door	design	40%	50%	60%	100%
	implementation	10%	20%	10%	100%
	execution	0%	0%	0%	80%
Safe Pick&place	design	0%	0%	40%	90%
	implementation	0%	0%	0%	90%
	execution	0%	0%	0%	90%

Table 2: Ablation study of Meta Control on three tasks and three steps.

4.2 Meta control exploits dynamics priors

We show that *Meta-Control* exploits the LLM's prior knowledge of dynamics with the balance cart pole task. The synthesized control system is described in appendix B. Specifically, LLM designs the task space dynamics $h(z, v)$ with a linear approximation in the form of $\dot{z} = Az + Bv$ around the upright position of the pole, where A and B are directly given by the LLM:

$$h(z,v) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{m_{\text{pole}} g}{m_{\text{cart}}} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g(m_{\text{cart}}+m_{\text{pole}})}{l_{\text{pole}} m_{\text{cart}}} & 0 \end{pmatrix} z + \begin{pmatrix} 0 \\ \frac{1}{m_{\text{cart}}} \\ 0 \\ -\frac{1}{l_{\text{pole}} m_{\text{cart}}} \end{pmatrix} v.$$

This example demonstrates that *Meta-Control* exploits the dynamics priors from LLM, in contrast to previous work that focuses more on spatial relationship priors from LLM~[1, 4, 5]. Exploiting dynamics priors enables *Meta-Control* to synthesize high-performance controllers rigorously.

4.3 Ablation study of the Meta Control pipeline

To demonstrate the necessity of the hierarchy formulation and the simplification using templates, we performed an ablation study to show the success rate of control system synthesis under different conditions. We test the success rate in 3 stages: design, implementation, and execution. We say that a design is successful if the LLM-designed control system has the potential to perform the skill judged by an expert. We say that the implementation is successful if the system can run without errors. We say that the execution is successful if the control system performs the skill as required. We repeat each task 10 times to compute the success rate. The randomness is caused by the LLM with a default temperature 1.0. As shown in table 2, we can see that the success rates of all steps in the baseline are lower than *Meta-Control*. Although LLM gives reasonable architectures to finish the task in the Balance and Open Door task, the LLM fails to provide correct code and parameters to realize the control system due to the complexity of the system and the huge design space. With all the modules, we achieve the highest success rate for all tasks. The Safe Pick&Place task is

Balance		Open Door		Safe Pick&Place	
Pole Mass 0.01∼0.5 kg	10/10	Handle Height 0.3∼0.75 m	10/10	Obstacle Position 0.01∼0.3 m (y-axis)	10/10
Cart Mass 0.05∼0.5 kg	10/10	Handle Radius 0.3∼0.7 m	10/10	Obstacle Size 0.1∼0.45 m (height)	10/10
Initial Angle -0.5∼0.5 rad	10/10	Door Mass 1∼30 kg	10/10	Place Position -0.3∼0.3 m (x-axis)	10/10

Table 3: The synthesized controllers easily generalize to scenarios with different object states/attributes. The left column lists the specific parameters for each scenario. The right column indicates the success rate (out of 10 trials) for each set of parameters.

especially difficult for the baseline because the baseline methods, even though prompted to avoid collision, were unable to successfully design a controller that can avoid collision continuously.

4.4 Generalization to different attributes/states

Meta-Control synthesized control systems can easily generalize to scenarios of different attributes/states due to the model-based nature. Given a successfully synthesized control system, we test different attributes/states and calculate the success rate. The range of change and the results are shown in table 3. Examples are shown in fig. 7. *Meta-Control* achieved 100% success rate for all scenarios.

4.5 Meta Control enables Model-based Analysis

One merit of our method is that the synthesized controller is fully model-based, allowing a rigorous formal analysis for a variety of properties:

Convergence and Stability: For the balance cart-pole task, we can provide a guarantee of convergence by solving the Riccati equation for the LQR controller. The closed-loop system matrix $A-BK$ has the following four eigenvalues: $-412.29, -9.925, -1.502+1.175j, -1.502-1.175j$. All of them have negative real parts, which means that the system is guaranteed to converge. More rigorous analysis can be conducted by taking the linearization error into consideration.

Constraint satisfiability and Forward Invariance: In the pick-and-place task, an MPC task controller is tracked by a safe controller. The safe controller is realized with a safety index (also known as the barrier function), which guarantees collision avoidance with mathematical proofs∼[77, 78]: it ensures the system state always satisfies $\min\{d_{\text{min}}-d(x), 100\cdot(0.02^2-d(x)^2)-10\cdot\dot{d}(x)\} < 0$, where d_{min} is the allowable minimum distance between the robot and the obstacle, $d(x)$ and $\dot{d}(x)$ are the relative distance and relative velocity from the robot to the obstacle, respectively.

4.6 Transfer to real robot and different embodiments

The control system for opening door is synthesized in simulation and is executed in both simulation and the real world. As shown in fig. 3 and fig. 4, the behavior is consistent and no sim-to-real gap is observed because the synthesized controller is model-based, closed-loop, highly explainable, and math-guaranteed. The controller can also generalize to different embodiments with the same low-level API (e.g. 6 DoF joint torque) given the model of the new embodiments, shown in fig. 6.

4.7 Failure analysis

We analyze the failure cases in table 2 and summarized their reasons. 1. *Mathematical error:* Although LLM can give a mathematical description of the approximate dynamic model for an unseen system, it can make mistakes in math. For example, for the $h(z, v)$ synthesized for the Cart Pole system, it can miss a term in the A matrix, or mess up signs (use $+$ when $-$ is desired). 2. *Failure to follow instructions:* We require the LLM to provide a structured response so that a program can extract the code and plug it into the robotic system. However, sometimes LLM fails to follow the instruction, leading to responses of wrong format. 3. *Incorrect reasoning:* In the open-door task, the LLM infers the target location of the door knob. However, the LLM may infer a wrong target given the environment information. Although these are still challenging for LLMs, we believe that they can be overcome with the rapid development of LLMs in the near future.

5 Limitation and Discussion

In this work, we propose *Meta-Control*, a novel framework for zero-shot model-based control system synthesis using LLM, tailored for heterogeneous robotic tasks. Through both simulations and real-world tests, we demonstrate *Meta-Control*'s potential to expand the autonomous capabilities of robots. Despite compelling results, *Meta-Control* has several limitations. Such as dependency on accurate system state estimation, which may not be available in some open-world tasks, and dependency on predefined dynamic models and control templates, which may restrict adaptability to completely new heterogeneous tasks. The synthesis process also demands substantial computational resources, which could hinder real-time synthesis. Future directions include enabling automatic perception selection, incorporating learning-based templates, and accelerating synthesis speed.

6 Acknowledgement

This work was supported by the National Science Foundation under Grant No. 2144489.

References

[1] 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. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[2] 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.

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

[4] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023.

[5] A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog, D. Ho, J. Ibarz, A. Irpan, E. Jang, R. Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pages 287–318. PMLR, 2023.

[6] A. Padalkar, A. Pooley, A. Jain, A. Bewley, A. Herzog, A. Irpan, A. Khazatsky, A. Rai, A. Singh, A. Brohan, et al. Open x-embodiment: Robotic learning datasets and rt-x models. *arXiv preprint arXiv:2310.08864*, 2023.

[7] W. Yu, N. Gileadi, C. Fu, S. Kirmani, K.-H. Lee, M. G. Arenas, H.-T. L. Chiang, T. Erez, L. Hasenclever, J. Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

[8] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv: Arxiv-2310.12931*, 2023.

[9] T. Xie, S. Zhao, C. H. Wu, Y. Liu, Q. Luo, V. Zhong, Y. Yang, and T. Yu. Text2reward: Automated dense reward function generation for reinforcement learning. *arXiv preprint arXiv:2309.11489*, 2023.

[10] Y. Wang, Z. Xian, F. Chen, T.-H. Wang, Y. Wang, K. Fragkiadaki, Z. Erickson, D. Held, and C. Gan. Robogen: Towards unleashing infinite data for automated robot learning via generative simulation, 2023.

[11] J. Zhang, J. Zhang, K. Pertsch, Z. Liu, X. Ren, M. Chang, S.-H. Sun, and J. J. Lim. Bootstrap your own skills: Learning to solve new tasks with large language model guidance. In *7th Annual Conference on Robot Learning*, 2023. URL https://openreview.net/forum?id=a0mFRgadGO.

[12] 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.

[13] S. Huang, Z. Jiang, H. Dong, Y. Qiao, P. Gao, and H. Li. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model. *arXiv preprint arXiv:2305.11176*, 2023.

[14] 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. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023.

[15] J. Wu, R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser. Tidybot: Personalized robot assistance with large language models. *Autonomous Robots*, 47(8):1087–1102, 2023.

[16] Y. Hu, F. Lin, T. Zhang, L. Yi, and Y. Gao. Look before you leap: Unveiling the power of gpt-4v in robotic vision-language planning. *arXiv preprint arXiv:2311.17842*, 2023.

[17] F. Joublin, A. Ceravola, P. Smirnov, F. Ocker, J. Deigmoeller, A. Belardinelli, C. Wang, S. Hasler, D. Tanneberg, and M. Gienger. Copal: Corrective planning of robot actions with large language models. *arXiv preprint arXiv:2310.07263*, 2023.

[18] J. Gao, B. Sarkar, F. Xia, T. Xiao, J. Wu, B. Ichter, A. Majumdar, and D. Sadigh. Physically grounded vision-language models for robotic manipulation. In *2024 International Conference on Robotics and Automation (ICRA)*. IEEE, 2024.

[19] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg. Text2motion: From natural language instructions to feasible plans. *arXiv preprint arXiv:2303.12153*, 2023.

[20] H. Ha, P. Florence, and S. Song. Scaling up and distilling down: Language-guided robot skill acquisition. In *Proceedings of the 2023 Conference on Robot Learning*, 2023.

[21] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence. Palm-e: An embodied multimodal language model. In *arXiv preprint arXiv:2303.03378*, 2023.

[22] Y. Guo, Y.-J. Wang, L. Zha, Z. Jiang, and J. Chen. Doremi: Grounding language model by detecting and recovering from plan-execution misalignment. *arXiv preprint arXiv:2307.00329*, 2023.

[23] C. Jin, W. Tan, J. Yang, B. Liu, R. Song, L. Wang, and J. Fu. Alphablock: Embodied finetuning for vision-language reasoning in robot manipulation. *arXiv preprint arXiv:2305.18898*, 2023.

[24] W. Huang, F. Xia, D. Shah, D. Driess, A. Zeng, Y. Lu, P. Florence, I. Mordatch, S. Levine, K. Hausman, et al. Grounded decoding: Guiding text generation with grounded models for embodied agents. *Advances in Neural Information Processing Systems*, 36, 2024.

[25] 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.

[26] Z. Wang, S. Cai, G. Chen, A. Liu, X. Ma, and Y. Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023.

[27] J. Luketina, N. Nardelli, G. Farquhar, J. Foerster, J. Andreas, E. Grefenstette, S. Whiteson, and T. Rocktäschel. A survey of reinforcement learning informed by natural language. *arXiv preprint arXiv:1906.03926*, 2019.

[28] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. In *International conference on machine learning*, pages 166–175. PMLR, 2017.

[29] Y. Jiang, S. S. Gu, K. P. Murphy, and C. Finn. Language as an abstraction for hierarchical deep reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[30] I. Lenz, R. A. Knepper, and A. Saxena. Deepmpc: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*, volume 10, page 25. Rome, Italy, 2015.

[31] L. Hewing, K. P. Wabersich, M. Menner, and M. N. Zeilinger. Learning-based model predictive control: Toward safe learning in control. *Annual Review of Control, Robotics, and Autonomous Systems*, 3:269–296, 2020.

[32] M. B. Chang, T. Ullman, A. Torralba, and J. B. Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

[33] P. Battaglia, R. Pascanu, M. Lai, D. Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29, 2016.

[34] A. Nagabandi, K. Konolige, S. Levine, and V. Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pages 1101–1112. PMLR, 2020.

[35] Y. Li, J. Wu, R. Tedrake, J. B. Tenenbaum, and A. Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv preprint arXiv:1810.01566*, 2018.

[36] D. Driess, O. Oguz, J.-S. Ha, and M. Toussaint. Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning. In *2020 IEEE international conference on robotics and automation (ICRA)*, pages 9563–9569. IEEE, 2020.

[37] H. Huang, F. Lin, Y. Hu, S. Wang, and Y. Gao. Copa: General robotic manipulation through spatial constraints of parts with foundation models. *arXiv preprint arXiv:2403.08248*, 2024.

[38] C. Finn, S. Levine, and P. Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. In *International conference on machine learning*, pages 49–58. PMLR, 2016.

[39] J. Fu, K. Luo, and S. Levine. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248*, 2017.

[40] B. Amos, I. Jimenez, J. Sacks, B. Boots, and J. Z. Kolter. Differentiable mpc for end-to-end planning and control. *Advances in neural information processing systems*, 31, 2018.

[41] P. Sharma, B. Sundaralingam, V. Blukis, C. Paxton, T. Hermans, A. Torralba, J. Andreas, and D. Fox. Correcting robot plans with natural language feedback. *arXiv preprint arXiv:2204.05186*, 2022.

[42] Y. Tang, W. Yu, J. Tan, H. Zen, A. Faust, and T. Harada. Saytap: Language to quadrupedal locomotion. 2023. URL https://saytap.github.io. https://saytap.github.io.

[43] 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.

[44] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

[45] A. Bucker, L. Figueredo, S. Haddadin, A. Kapoor, S. Ma, S. Vemprala, and R. Bonatti. Latte: Language trajectory transformer. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7287–7294. IEEE, 2023.

[46] A. Stone, T. Xiao, Y. Lu, K. Gopalakrishnan, K.-H. Lee, Q. Vuong, P. Wohlhart, S. Kirmani, B. Zitkovich, F. Xia, et al. Open-world object manipulation using pre-trained vision-language models. *arXiv preprint arXiv:2303.00905*, 2023.

[47] Octo Model Team, D. Ghosh, H. Walke, K. Pertsch, K. Black, O. Mees, S. Dasari, J. Hejna, C. Xu, J. Luo, T. Kreiman, Y. Tan, D. Sadigh, C. Finn, and S. Levine. Octo: An open-source generalist robot policy. https://octo-models.github.io, 2023.

[48] Y. Jiang, A. Gupta, Z. Zhang, G. Wang, Y. Dou, Y. Chen, L. Fei-Fei, A. Anandkumar, Y. Zhu, and L. Fan. Vima: General robot manipulation with multimodal prompts. In *Fortieth International Conference on Machine Learning*, 2023.

[49] A. Szot, M. Schwarzer, B. Mazoure, H. Agrawal, W. Talbott, K. Metcalf, N. Mackraz, D. Hjelm, and A. Toshev. Large language models as generalizable policies for embodied tasks. *preprint*, 2023.

[50] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

[51] K. Bousmalis, G. Vezzani, D. Rao, C. Devin, A. X. Lee, M. Bauza, T. Davchev, Y. Zhou, A. Gupta, A. Raju, et al. Robocat: A self-improving foundation agent for robotic manipulation. *arXiv preprint arXiv:2306.11706*, 2023.

[52] 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. *IEEE Robotics and Automation Letters*, 2023.

[53] P. Florence, C. Lynch, A. Zeng, O. Ramirez, A. Wahid, L. Downs, A. Wong, J. Lee, I. Mordatch, and J. Tompson. Implicit behavioral cloning. *Conference on Robot Learning (CoRL)*, 2021.

[54] A. Zeng, P. Florence, J. Tompson, S. Welker, J. Chien, M. Attarian, T. Armstrong, I. Krasin, D. Duong, V. Sindhwani, and J. Lee. Transporter networks: Rearranging the visual world for robotic manipulation. *Conference on Robot Learning (CoRL)*, 2020.

[55] M. Shridhar, L. Manuelli, and D. Fox. Cliport: What and where pathways for robotic manipulation. In *Proceedings of the 5th Conference on Robot Learning (CoRL)*, 2021.

[56] M. Shridhar, L. Manuelli, and D. Fox. Perceiver-actor: A multi-task transformer for robotic manipulation. In *Proceedings of the 6th Conference on Robot Learning (CoRL)*, 2022.

[57] C. Chi, S. Feng, Y. Du, Z. Xu, E. Cousineau, B. Burchfiel, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Proceedings of Robotics: Science and Systems (RSS)*, 2023.

[58] L. Chen, S. Bahl, and D. Pathak. Playfusion: Skill acquisition via diffusion from language-annotated play. In *Conference on Robot Learning*, pages 2012–2029. PMLR, 2023.

[59] H. van Seijen, S. Whiteson, and L. Kester. Efficient abstraction selection in reinforcement learning. *Computational Intelligence*, 30(4):657–699, 2014.

[60] G. D. Konidaris and A. G. Barto. Efficient skill learning using abstraction selection. In *IJCAI*, volume 9, pages 1107–1112, 2009.

[61] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. Deep spatial autoencoders for visuomotor learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2016.

[62] S. Parisi, A. Rajeswaran, S. Purushwalkam, and A. Gupta. The unsurprising effectiveness of pre-trained vision models for control. In *international conference on machine learning*, pages 17359–17371. PMLR, 2022.

[63] T. Lesort, M. Seurin, X. Li, N. Díaz-Rodríguez, and D. Filliat. Unsupervised state representation learning with robotic priors: a robustness benchmark. *arXiv preprint arXiv:1709.05185*, 2017.

[64] A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction. *arXiv preprint arXiv:2006.10742*, 2020.

[65] S. Lange, M. Riedmiller, and A. Voigtländer. Autonomous reinforcement learning on raw visual input data in a real world application. In *The 2012 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2012.

[66] M. Laskin, A. Srinivas, and P. Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. In *International conference on machine learning*, pages 5639–5650. PMLR, 2020.

[67] A. Khandelwal, L. Weihs, R. Mottaghi, and A. Kembhavi. Simple but effective: Clip embeddings for embodied ai. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14829–14838, 2022.

[68] S. Nair, A. Rajeswaran, V. Kumar, C. Finn, and A. Gupta. R3m: A universal visual representation for robot manipulation. *arXiv preprint arXiv:2203.12601*, 2022.

[69] T. Xiao, I. Radosavovic, T. Darrell, and J. Malik. Masked visual pre-training for motor control. *arXiv preprint arXiv:2203.06173*, 2022.

[70] I. Radosavovic, T. Xiao, S. James, P. Abbeel, J. Malik, and T. Darrell. Real-world robot learning with masked visual pre-training. In *Conference on Robot Learning*, pages 416–426. PMLR, 2023.

[71] Y. J. Ma, S. Sodhani, D. Jayaraman, O. Bastani, V. Kumar, and A. Zhang. Vip: Towards universal visual reward and representation via value-implicit pre-training. *arXiv preprint arXiv:2210.00030*, 2022.

[72] S. Karamcheti, S. Nair, A. S. Chen, T. Kollar, C. Finn, D. Sadigh, and P. Liang. Language-driven representation learning for robotics. *arXiv preprint arXiv:2302.12766*, 2023.

[73] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[74] Y. Cui, S. Niekum, A. Gupta, V. Kumar, and A. Rajeswaran. Can foundation models perform zero-shot task specification for robot manipulation? In *Learning for dynamics and control conference*, pages 893–905. PMLR, 2022.

[75] H. Liu, L. Lee, K. Lee, and P. Abbeel. Instruction-following agents with multimodal transformer. *arXiv preprint arXiv:2210.13431*, 2022.

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

[77] T. Wei and C. Liu. Safe control algorithms using energy functions: A uni ed framework, benchmark, and new directions. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 238–243. IEEE, 2019.

[78] T. Wei, S. Kang, R. Liu, and C. Liu. Zero-shot transferable and persistently feasible safe control for high dimensional systems by consistent abstraction. In *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 8614–8619. IEEE, 2023.

A Joint torques during openning a door

As shown in fig. 8, openning a door with position control can lead to very large joint torques, leading to failure or dangerous behaviors. But with a compliant controller, the joint torques are much smaller and the door is opened successfully without damage.

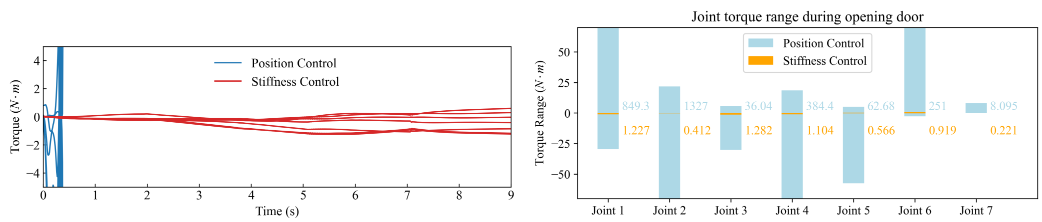

Figure 8: Joint torque range during opening door for the baseline (position control) and the *Meta-Control* synthesized controller (stiffness control). The baseline has a huge torque because the planned trajectory is inaccurate, which leads to damage.

B Task descriptions

Open the door Opening a door is a challenge for robots because a door has a fixed swing path that must be followed exactly. As shown in fig. 3, position control can easily lead to door damage or failure of action. Therefore, it is preferable to open a door with a compliant controller. With multiple rounds of experiments, we found that *Meta-Control* synthesized control system usually involves a `CartesianStiffnessController` acting as the task controller or the tracking controller. Although the trajectory may not be perfectly aligned with the swing path, with the stiffness controller, the robot can still open the door because it complies with the force given by the door.

Wipe the board Wiping a board requires a certain amount of force to be applied on the board while moving the eraser, which involves two different objectives: position tracking and force tracking. As shown in fig. 3, the synthesized controller successfully removes the marks, while control frameworks that only consider spatial relationships are not suitable for this task because of the lack of force constraints. In most trials, *Meta-Control* chooses a `CartesianInterpolationController` as task controller, and a `PoseForceController` as tracking controller. The Cartesian interpolation controller plans the trajectory of the eraser, while the hybrid position/force controller tracks the trajectory while maintaining a desired force on the board to erase.

Balance the cart pole Cart pole is a classic control tasks that have been extensively studied. Attempts were made to synthesize a simple PID controller with LLM to balance a pole with predefined API where the cart can be controlled directly [1]. However, in this experiment, we use a robot arm to hold the cart and ask LLM to balance it by controlling the robot arm. This is a significantly more challenging task because only low-level APIs of the robot arm are given, and the pole is attached to the cart with a *non-actuated free joint*. The LLM has to understand the relationship from the arm to the cart, and from the cart to the pole. In most cases, our method chooses an `LQRController` as the task controller and the `PoseForceController` as the tracking controller. The LQR controller gives the force to be applied on the cart along the pole joint direction (y-axis) to balance the pole, and the hybrid position/force controller tracks the desired force on the y-axis while maintaining a neutral pose on the x-axis and the z-axis. Profile of the pole's angle is shown in fig. 5, which shows that the synthesized controller efficiently balanced the pole.

The synthesized control system is described below:

$$\begin{aligned} y &= [\text{Pole}_\theta, \text{Pole}_\omega, \text{Cart}_y, \text{Cart}_{\dot{y}}] \\ v &= \text{End-effector (EE) force on } y\text{−axis} \\ h(z, v) &= \text{Linearized Cart Pole dynamics} \\ \pi_v(y) &= \text{LQR controller} \end{aligned} \qquad \begin{aligned} x &= \text{Joint states}, \text{EE}^{\text{target}}_{\text{force}}, \text{EE}^{\text{target}}_{\text{pose}} \\ u &= \text{Joint torques} \\ f(x, u) &= \text{Kinova dynamics model} \\ \pi_u(x, y, v) &= \text{Pose Force Controller} \end{aligned}$$

where

$$h(z, v) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{m_{\text{pole}} g}{m_{\text{cart}}} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g(m_{\text{cart}}+m_{\text{pole}})}{l_{\text{pole}} m_{\text{cart}}} & 0 \end{pmatrix} z + \begin{pmatrix} 0 \\ \frac{1}{m_{\text{cart}}} \\ 0 \\ -\frac{1}{l_{\text{pole}} m_{\text{cart}}} \end{pmatrix} v.$$

Collision-free pick and place Pick and place is a very common skill in daily life. In this task, we require the robot arm to reach a goal position while maintaining *whole-body* collision free. The goal position can be either the location of the object or the target location. In most cases, the LLM chooses a `KinematicTrajectoryModelPredictiveController` as the task controller to generate collision-free way-points for reaching the goal, and a `SafeController` as the tracking controller to guarantee collision-free in continuous time. In most cases, the LLM chooses a `KinematicTrajectoryModelPredictiveController` as the task controller to generate collision-free way-points for reaching the goal, and a `SafeController` as the tracking controller to guarantee collision-free in continuous time.

C Full conversation

The full conversation on skill synthesis is attached below. Some long numerical arrays are omitted for the sake of clarity. Controller templates, dynamical model templates, and input port samples are attached after the conversation.

C.1 Design via Templates

```
## System
The GPT is a professor in robot control and a proficient programmer of Python and PyDrake. A robot task
↪  can be accomplished by sequentially executing several skills. The GPT will help the user write code to
↪  compose one of the skills after reading the requirements. The GPT will be instructed step by step. A
↪  skill is accomplished hierarchically by a task controller and a tracking controller. The task
↪  controller is responsible for generating high-level and abstract control, and the tracking controller
↪  is responsible for tracking the high-level control and satisfying constraints.

Here are some optional principles to design the task controller and the tracking controller:
1. Design the task controller for the object of interest or end-effector, and design the tracking
↪  controller for the robot.
2. Design the task controller on simplified or approximated dynamics, such as Cartesian space or an
↪  approximated linear model, and design the tracking controller on the full robot dynamics.
3. If the skill involves multiple objectives, design the task controller to take care of the primary
↪  objective and design the tracking controller to take care of the rest of the objectives and
↪  constraints. The task controller and the tracking controller can both output joint torque.
4. Task control is often the position, velocity, acceleration or force in the Cartesian space. But in some
↪  cases, the task control is best to be joint torque.
5. The final output of the tracking controller must be joint torques.
```

````
The GPT will proceed the following steps: task_controller, tracking_controller, task_model, tracking_model. In the GPT's response, the most important part will be <step_name> followed by a code block enclosed by ```python```. Each code block should correspond to a single step. This explicitly indicates the code is written for a specific step. The GPT can feel free to make some plans to help it think before writing the code blocks. There is no need to explain the code after the code block. The GPT should not import any module. The GPT should not make up arguments, variables, or anything else that is not mentioned.

## User
Please help me to compose a skill to stabilize a cart pole. The cart is attached to the end effector of the robot arm. The pole rotates freely around the x-axis and is unactuated. Please stabilize the pole around the upright position (when the pole angle, denoted by PolePin, is 0) by exerting force on the cart along the y-axis. A secondary objective is to keep the end-effector pose as close as possible to [1.57078469e+00, 0.0, 1.57079455e+00, 5.96598901e-01, 1.34625984e-03, 4.33589236e-01] ([roll, pitch, yaw, x, y, z]). The cart is 0.1 kg, and the pole is 0.25 m, 0.01 kg.

## Assistant
Certainly! I'm good at writing controllers for robots with PyDrake. I will follow your instructions carefully and think carefully.

## User
Let's first take a look of all available measurements, dynamic models, and controllers.

Here are all available inputs to the skill:
{skill_inputs_sample}

Here are all available dynamic model setup functions in yaml format:
{dynamic_model_brief_str}

Here are all available controllers in yaml format:
{controller_brief_str}

====================================================

Based on the available measurements, models, and controllers, please think step by step to choose the models and controllers:
1. Decide the subject of the task controller, is it an object, the end effector, or something else.
2. Decide the desired task control to be applied on the subject, is it Cartesian position, Cartesian force, or something else.
3. Decide the task space model that can model the dynamics of the subject with the task control. When the analytical form of the dynamics model is needed, draw upon your extensive knowledge of control theory and system modeling. Think step by step to make sure the model is correct according to the task description.
4. Decide the task controller that can apply the desired task control and realize the goal of the skill. Make sure the controller can realize the goal with the given available measurements. There are no additional inputs or functions provided to the controller.
5. Decide the tracking space model.
6. Decide the tracking controller to track the output of the task controller and to satisfy possible constraints.

Detail about how can the chosen models and controllers realize the skill goal. Be specific.

Please return the choice with the following format.

<task_model>
```python
task_model = setup_model(arg1, arg2)
```

<tracking_model>
```python
tracking_model = setup_model(arm_type, object_info)
```

<task_controller>
```python
task_controller = ControllerClass(arg1, arg2)
```

<tracking_controller>
```python
tracking_controller = ControllerClass(arg1)
```

Make sure your response complies with the following requirements:
* <step_name> must be included before each code block so the code block can be recognized. step_name must be enclosed in angle brackets <>.
* Choices have been made for all steps. A correct solution exists with the given information.
````

~~~~
* Pass the arguments (args) when instantiating the controller and the model. The arguments needed are stated in the yaml file (args). **Do not** include items in "input_ports" (such as "controller_parameters") as arguments when instantiating the controllers. They are not arguments for instantiating the controller.
* The arguments should be defined before use, except `arm_type` and `object_info`, which can be used directly without definition.
* Do not use any placeholder or assume any variable is defined. All information and available variables needed are provided. If your choice of model and controller requires additional information, rethink it.
* The models and controllers are instantiated correctly as stated in the materials.
* The chosen model matches the required model of the controllers as stated in the materials.

## Assistant
Considering the requirements and the available models and controllers, let's proceed step by step to accomplish the task of stabilizing the cart pole attached to the end effector of the robot arm and achieving the secondary objective of keeping the end-effector's pose as close as possible to the specified position.

1. **Subject of the Task Controller:** Since the primary goal is to stabilize the pole around the upright position by exerting force along the y-axis to the cart, the task subject is "the cart and the pole system". This suggests we require a control method directly influencing the cart to alter the pole's position, making it a system-level control problem involving both the cart and the pole.

2. **Desired Task Control:** The control needed is a force along the y-axis on the cart, which indirectly influences the pole's angle to achieve stabilization.

3. **Task Space Model:** We need a model that accurately represents the dynamics of the cart-pole system for the application of a controlling force. Given the problem's nature, a linear model approximation around the upright position of the pole can be suitable. Therefore, I will choose the `setup_linear_model` because it can approximate the dynamics of the cart-pole system in an affine manner suitable for designing a controller to apply the required force.

4. **Task Controller:** To apply the desired task control (force), an appropriate choice is an `LQRController`. LQR (Linear-Quadratic Regulator) is well-suited for stabilizing the system around the equilibrium point (the upright position), given it requires a model of the system dynamics, which aligns well with our choice of a linear model.

5. **Tracking Space Model:** For the secondary goal, the model needs to incorporate the arm dynamics along with the constraints such as the end-effector's pose. The `setup_arm_with_objects_model` would be a suitable choice because it returns a model considering the robot arm and static objects, allowing control over the arm to maintain the end-effector's position while keeping the primary task of pole stabilization.

6. **Tracking Controller:** Considering we must respect both a desirable end-effector pose and exert control forces to stabilize the pole, a `PoseForceController` is selected for the tracking controller. This controller is capable of handling specific forces and poses, making it perfect for managing the secondary objective of maintaining the end-effector's pose.

Let's proceed with specifying the models and controllers as per our choice:

<task_model>
```python
x0 = np.zeros((4,1))  # Initializing the state for the cart-pole system
u0 = np.zeros((1,1))  # Initial control input (force)
m_cart = 0.1  # mass of the cart
m_pole = 0.01  # mass of the pole
l_pole = 0.25  # length of the pole
g = 9.81  # gravitational acceleration
# A and B matrices are placeholders and must be defined based on the specific dynamics of the cart-pole system
A = np.array([[0, 1, 0, 0],
              [0, 0, m_pole * g / m_cart, 0],
              [0, 0, 0, 1],
              [0, 0, (m_cart + m_pole) * g / (l_pole * m_cart), 0]])
B = np.array([[0], [1/m_cart], [0], [-1/(l_pole*m_cart)]])
C = np.eye(4)
D = np.zeros((4, 1))
task_model = setup_linear_model(x0, u0, A, B, C, D)
```

<tracking_model>
```python
tracking_model = setup_arm_with_objects_model(arm_type, object_info)
```

<task_controller>
```python
Q = np.diag([1, 1, 10, 10])  # Emphasizing the importance of the pole's position and velocity
R = np.array([[0.1]])  # Control effort cost
task_controller = LQRController(task_model, Q, R, x0, u0)
```
~~~~

~~~markdown
<tracking_controller>
```python
force_control_axis_mask = np.array([False, False, False, False, True, False])  # Force control along the y-axis
tracking_controller = PoseForceController(tracking_model, force_control_axis_mask)
```

This sequence of models and controllers fulfills the skill's requirements by utilizing a systematic and hierarchical control structure.

## User
Here is the output of running the skill: <empty>

Human expert supplementary instructions: <empty>

Do you think it is working as expected? (Empty output means the system is running without errors)
If it is, please answer with one word "yes".
If it is not, please identify which step(s) went wrong and explain how to fix it by including which step(s) to modify (<step_name>) and the updated code (enclosed by ```python```). We have proceeded with the following steps:
- task_model
- tracking_model
- task_controller
- tracking_controller

Please do not make up a step.

If the system is not working as expected, you must modify at least one step. You can modify multiple steps. Remember to add "<step_name>" before the corresponding code block. A correct solution exists with the given information. Try your best.

Here is a sample response when it is not working as expected:

I think this error occurred because [for some reason].

<tracking_controller>

updated_code:
```python
tracking_controller = NewTrackingController(arg1, arg2)
```


## Assistant
yes

## User
Please summarize the choice of the task controller and the tracking controller. How would you expect the input output ports of the controllers to be connected to perform the skill? This response will be used as prompts to implement the port connector, so please include all necessary information, including:

* The detailed description of the input ports for the task controller and the tracking controller, including the meaning of each dimension (be careful about the order), and how to get them from the available measurements or outputs.
* A description of any constant parameter you have used that may affect the performance of the control system.
* Everything else you think is necessary to implement the controller by only reading the summary.

## Assistant
To implement the port connector for performing the skill of stabilizing a cart pole attached to the end effector of a robot arm, we selected two controllers: an `LQRController` for the task controller and a `PoseForceController` for the tracking controller. Here is the summary of how these controllers are expected to interact and how their inputs and outputs should be connected:

### Task Controller (LQRController)
- **Input Ports:**
  - `state`: It represents the current state of the cart-pole system. For the `LQRController`, this state vector is expected to have dimensions `[x, x_dot, theta, theta_dot]` where `x` and `x_dot` represent the position and velocity of the cart along the y-axis (as force is applied along the y-axis), and `theta` and `theta_dot` represent the pole's angle (around the x-axis) and angular velocity, respectively. This state can be constructed from the `measured_body_poses`, `measured_body_velocities` for the 'Cart', and 'PolePin' measurements from `measured_joint_states`.

- **Output Ports:**
  - `control`: This port outputs the computed control input based on the LQR control law. For our application, this control input is a single scalar value representing the force to be applied along the y-axis to the cart.

### Tracking Controller (PoseForceController)
- **Input Ports:**
~~~

```
  - `ee_pose_target`: This input requires the desired end-effector pose in the format `[roll, pitch, yaw, x, y, z]`. Since our primary goal is to keep the end-effector's pose as close as possible to a specified target while applying force to stabilize the pole, this target pose is set as a constant parameter `[1.57078469, 0.0, 1.57079455, 0.596598905, 0.00134625984, 0.483592036]`. Note that the force along the y-axis, which acts on the cart to stabilize the pole, is managed by this controller's `ee_force_target`.
  - `ee_force_target`: This input specifies the desired end-effector force in the format `[x-rotation, y-rotation, z-rotation, x-translation, y-translation, z-translation]`. The output from the task controller (LQRController) indicating the force to be applied along the y-axis to the cart will be fed into this port. All other dimensions are set to 0 as we are only interested in exerting force along the y-axis.
  - `arm_position` and `arm_velocity`: These inputs provide the current joint positions and velocities of the robot arm, respectively. They are directly obtained from `measured_arm_position` and `measured_arm_velocity`.

- **Output Ports:**
  - `applied_arm_torque`: This port outputs the computed joint torques for the robot arm, which are derived based on the desired force and pose control objectives.

### Constant Parameters Used
1. For the `LQRController`, we introduced a state cost matrix `Q` with higher weights on the pole's angle and angular velocity to emphasize the importance of stabilizing the pole. A relatively small control cost `R` is chosen to allow the application of necessary control efforts without being overly penalized.

2. In the `PoseForceController`, we used a constant target pose for the end-effector, with force control primarily along the y-axis. The Boolean array `force_control_axis_mask` is configured to apply force control only along the y-axis and pose control for all other dimensions.

### Summary
To connect these controllers effectively:
- The `state` input port of the `LQRController` must be assembled from available measurements, particularly focusing on the cart position and velocity along the y-axis and the pole's angle and angular velocity.
- The output control force from the `LQRController` must be properly directed to the `ee_force_target` input port of the `PoseForceController`, specifically into the dimension controlling force along the y-axis.
- The `ee_pose_target`, `arm_position`, and `arm_velocity` ports of the `PoseForceController` require direct measurements or set targets, with the end-effector pose being a specified constant.
- The final output of `applied_arm_torque` from the `PoseForceController` determines the joint torques to achieve both stabilization of the pole and maintaining the end-effector's desired pose.

This approach integrates both controllers to manage the primary objective of pole stabilization and the secondary objective of maintaining the end-effector pose efficiently.
```

C.2 Interface Alignment

```
## System
The GPT is a proficient programmer of Python and PyDrake and a professor in control. The GPT will help the user to accomplish the code after reading the requirements. The GPT will be instructed step by step so please only complete mentioned tasks. The code implements a robot skill through a hierarchical design. A task controller is chosen to give high-level control, and a tracking controller is given to track the task control. The task controller and the tracking controller will be given. The GPT needs to accomplish a task controller converter and a tracking controller converter. A converter transforms available measurements and ports to the required input ports of the controller.

The GPT will proceed with the following steps: task_callback, and tracking_callback. In the GPT's response, the most important part will be <step_name> followed by a code block enclosed by ```python```. Each code block should correspond to a single step. This explicitly indicates the code is written for a specific step. The GPT can feel free to make some plans to help it think before writing the code blocks. There is no need to explain the code after the code block. The GPT should not import any module. The GPT should not make up arguments, variables, and anything else that is not mentioned.

## User
Please help me to compose a skill to stabilize a cart pole. The cart is attached to the end effector of the robot arm. The pole rotates freely around the x-axis and is unactuated. Please stabilize the pole around the upright position (when the pole angle, denoted by PolePin, is 0) by exerting force on the cart along the y-axis. A secondary objective is to keep the end-effector pose as close as possible to [1.57078469e+00, 0.0, 1.57079455e+00, 5.96598901e-01, 1.34625984e-03, 4.33589236e-01] ([roll, pitch, yaw, x, y, z]). The cart is 0.1 kg, and the pole is 0.25 m, 0.01 kg.

## Assistant
Certainly! I'm good at writing callback functions for ports in PyDrake. I will follow your instructions carefully and think carefully.

## User
```

````
Now please implement callback functions of a `LeafSystem` that acts as a connector which I have already constructed. The input ports correspond to inputs to the skill (observations and controller parameters). The output ports correspond to the input ports of the task controller.
Here is a yaml file describing what the inputs to the skill are:
{skill_input_summary}

Here we print the port name, type, and sample value for all the input ports. You can get the value of the port by `value = self.GetInputPort(port_name).Eval(context)` when implementing the call back functions:
{input_port_sample}

The chosen task model, tracking model, task controller, and tracking controller are
```python

x0 = np.zeros((4,1))  # Initializing the state for the cart-pole system
u0 = np.zeros((1,1))  # Initial control input (force)
m_cart = 0.1  # mass of the cart
m_pole = 0.01  # mass of the pole
l_pole = 0.25  # length of the pole
g = 9.81  # gravitational acceleration
# A and B matrices are placeholders and must be defined based on the specific dynamics of the cart-pole system
A = np.array([[0, 1, 0, 0],
              [0, 0, m_pole * g / m_cart, 0],
              [0, 0, 0, 1],
              [0, 0, (m_cart + m_pole) * g / (l_pole * m_cart), 0]])
B = np.array([[0], [1/m_cart], [0], [-1/(l_pole*m_cart)]])
C = np.eye(4)
D = np.zeros((4, 1))
task_model = setup_linear_model(x0, u0, A, B, C, D)


tracking_model = setup_arm_with_objects_model(arm_type, object_info)


Q = np.diag([1, 1, 10, 10])  # Emphasizing the importance of the pole's position and velocity
R = np.array([[0.1]])  # Control effort cost
task_controller = LQRController(task_model, Q, R, x0, u0)


force_control_axis_mask = np.array([False, False, False, False, True, False])  # Force control along the y-axis
tracking_controller = PoseForceController(tracking_model, force_control_axis_mask)

```

Here is the design summary to explain the expected way of how do the controllers work, and how to connect the ports:
{design_summary} (from Design via Templates)

Here is the summary of the input ports of the task controller:
{task_controller_input_summary} (from the controller summary)

The functions you need to implement are:
```python
def task_state(self, context, output: BasicVector)
    pass


```
Here is an example response:

<task_callback>
```python
def place_holder_function(self, context, output: BasicVector):
    abstract_input_port_value = self.GetInputPort(port_name).Eval(context)
    output.SetFromVector(abstract_input_port_value["place_holder_key"])
def place_holder_function2(self, context, output: BasicVector):
    vector_input_port_value = self.GetInputPort(port_name).Eval(context)
    out = np.zeros(6)
    out[3] = vector_input_port_value[0]
    output.SetFromVector(out)
def place_holder_function3(self, context, output: AbstractValue):
    value = self.GetInputPort(port_name).Eval(context)
    output.set_value(value)
```
Please implement all the task callback functions in one code block beginning with <task_callback>.

Here are some tips to implement the callback functions:
* When use a port, make sure the name of the port is correct and the port is available in the current function.
````

````
* Note that some of the ports should be passed through without any modification like control_parameters.
↪  Some of the ports need to be modified.
* Make sure the dimensions of the output match the port. The output value may be only partially available
↪  from the input port. You need to decide the value for the remaining. Be careful of which dimension to
↪  set value, make sure it complies with the description of the port in the controller summary.
* For pose, velocity, and force vectors, rotation always comes first, such as [roll, yaw, pitch, x, y, z]
↪  or [x-rotation, y-rotation, z-rotation, x-translation, y-translation, z-translation].
* If the input port is a dictionary, make sure only use keys that has shown in the printed port value
↪  sample.
* If the output should be an enum type, you can use the enum type mentioned in the controller summary
↪  directly without any import.
* Recall the task requirements given in the beginning. Make sure the call back functions correctly to
↪  solve the task.
* Please avoid using placeholders or simplified examples that require modifications by others. Ensure all
↪  calculations are conclusive. In case of any uncertainties about parameters, utilize the information
↪  provided to derive the most informed inference available.
* You can use `self.num_q` to denote the number of joints


Please implement the call-back functions. Please think step by step then write the code.

## Assistant
<task_callback>
```python
def task_state(self, context, output: BasicVector):
    # Extracting necessary inputs
    measured_body_poses = self.GetInputPort("measured_body_poses").Eval(context)
    measured_body_velocities = self.GetInputPort("measured_body_velocities").Eval(context)
    measured_joint_states = self.GetInputPort("measured_joint_states").Eval(context)

    # Initialize state vector with zeros
    state = np.zeros(4)

    # Position and Velocity of Cart along y-axis from body poses and velocities
    cart_pose = measured_body_poses['Cart'].translation()
    cart_velocity = measured_body_velocities['Cart'].translational()
    state[0] = cart_pose[1]  # y position of Cart
    state[1] = cart_velocity[1]  # y velocity of Cart

    # Pole's angle (around x-axis) and angular velocity from joint states
    pole_angle = measured_joint_states['PolePin']['positions']
    pole_angular_velocity = measured_joint_states['PolePin']['velocities']
    state[2] = pole_angle[0]  # Assuming 'positions' is a numpy array with one element
    state[3] = pole_angular_velocity[0]  # Assuming 'velocities' is a numpy array with one element

    output.SetFromVector(state)
```

## User
Now please implement callback functions of a `LeafSystem` that acts as a connector which I have already
↪  constructed for the tracking controller. The input ports correspond to inputs to the skill
↪  (observations and controller parameters) and task control (the output of the task controller). The
↪  output ports correspond to the input ports of the tracking controller.

Recall the input port of the skill given previously, and the following is a description of the task
↪  control port. You can use all skill input ports and the task control port.
{task_control_port_summary} (from the controller summary)

Here is the summary of the input ports of the tracking controller:
{tracking_controller_input_summary} (from the controller summary)

The functions you need to implement are:
```python
def tracking_ee_pose_target(self, context, output: BasicVector)
    pass

def tracking_ee_force_target(self, context, output: BasicVector)
    pass

def tracking_arm_position(self, context, output: BasicVector)
    pass

def tracking_arm_velocity(self, context, output: BasicVector)
    pass


```
Here is an example response:

<tracking_callback>
```python
````
````

````
def place_holder_function(self, context, output: BasicVector):
    abstract_input_port_value = self.place_holder_port.Eval(context)
    output.SetFromVector(abstract_input_port_value["place_holder_key"])
def place_holder_function2(self, context, output: BasicVector):
    vector_input_port_value = self.place_holder_port.Eval(context)
    out = np.zeros(6)
    out[3] = vector_input_port_value[0]
    output.SetFromVector(out)
def place_holder_function3(self, context, output: AbstractValue):
    value = self.place_holder_port.Eval(context)
    output.set_value(value)
```

Follow the previously given tips for implementing the callback functions. Furthermore:
* Note that output ports of the task controller have been renamed by adding the prefix: 'task_' + port_name.

Now please implement the call-back functions. Please think step by step then write the code.

## Assistant
<tracking_callback>
```python
def tracking_ee_pose_target(self, context, output: BasicVector):
    # Desired end-effector pose is constant as per design summary
    desired_ee_pose = [1.57078469e+00, 0.0, 1.57079455e+00, 5.96598901e-01, 1.34625984e-03, 4.33589236e-01]
    output.SetFromVector(np.array(desired_ee_pose))

def tracking_ee_force_target(self, context, output: BasicVector):
    # Obtaining the control action from task controller, directing it to apply force along y-axis
    task_control = self.GetInputPort("task_control").Eval(context)
    # Only force along the y-axis is controlled, the rest is masked out with zeros.
    ee_force_target = np.array([0.0, 0.0, 0.0, 0.0, task_control[0], 0.0])
    output.SetFromVector(ee_force_target)

def tracking_arm_position(self, context, output: BasicVector):
    # Directly passing the robot arm's measured position
    measured_arm_position = self.GetInputPort("measured_arm_position").Eval(context)
    output.SetFromVector(measured_arm_position)

def tracking_arm_velocity(self, context, output: BasicVector):
    # Directly passing the robot arm's measured velocity
    measured_arm_velocity = self.GetInputPort("measured_arm_velocity").Eval(context)
    output.SetFromVector(measured_arm_velocity)
```

## User
Here is the output of running the skill: <empty>

Human expert supplementary instructions: <empty>

Do you think it is working as expected? (Empty output means the system is running without errors)
If it is, please answer with one word "yes".
If it is not, please identify which step(s) went wrong and explain how to fix it by including which step(s) to modify (<step_name>) and the updated code (enclosed by ```python```). We have proceeded with the following steps:
- task_callback
- tracking_callback

Please do not make up a nonexist step.

If the system is not working as expected, you must modify at least one step. You can modify multiple steps.
For each step, you can only modify the function that you think is wrong. Others will be kept unchanged.
Remember to add "<step_name>" before the corresponding code block. A correct solution exists with the given information. Try your best.

Here is a sample response when it is not working as expected:

I think this error occurred because [some reason].

updated_code:
<tracking_callback>

```python
def tracking_arm_position(self, context, output: BasicVector):
    ...
```


## Assistant
````
````

```
yes

## User
Please summarize the task_controller, tracking_controll, task_callback and the tracking_callback. Explain the data flow. This response will be used as a prompt to choose the performance metric from measurements and tune the controller parameters, so please include all necessary information.

## Assistant
The integration of the task and tracking controllers with their respective callback functions forms a comprehensive control loop for stabilizing a cart-pole system attached to the end effector of a robotic arm. Here's a summary of the components and their data flow:

### Task Controller (`LQRController`)
- **Purpose:** Stabilizes the cart-pole system around the upright position by computing a control force to be applied along the y-axis of the cart.
- **Input Port:**
  - `state`: Accepts a state vector `[x, x_dot, theta, theta_dot]`, representing cart position and velocity along the y-axis, and pole angle and angular velocity, respectively.
- **Output Port:**
  - `control`: Outputs a control action (force) based on the LQR control law.
- **Implemented by Function:** `task_state` in `<task_callback>`.

### Tracking Controller (`PoseForceController`)
- **Purpose:** Aligns the end effector's pose with a target pose while applying the computed force from the task controller to stabilize the pole.
- **Input Ports:**
  - `ee_pose_target`: Takes a constant target pose for the end effector.
  - `ee_force_target`: Receives the control action (force along the y-axis) from the task controller to be applied by the end effector.
  - `arm_position`: Provides current joint positions of the robot arm.
  - `arm_velocity`: Provides current joint velocities of the robot arm.
- **Output Port:**
  - N/A (As these inputs directly dictate the behavior of the end effector and arm based on internal tracking control logic).
- **Implemented by Functions:** `tracking_ee_pose_target`, `tracking_ee_force_target`, `tracking_arm_position`, and `tracking_arm_velocity` in `<tracking_callback>`.

### Data Flow
1. **Observations to Task Controller:**
   - The cart's position and velocity, along with the pole angle and angular rate, are extracted from the `measured_body_poses`, `measured_body_velocities`, and `measured_joint_states`. These values are aggregated into a state vector `[x, x_dot, theta, theta_dot]` by the `task_state` function in the task controller callback, which feeds into the `LQRController`.

2. **Task Controller to Tracking Controller:**
   - The computed control action (force) from the `LQRController` is passed to the `tracking_ee_force_target` function in the tracking controller callback. It sets this force as the end effector's desired force along the y-axis while keeping forces in other dimensions at zero.

3. **Observations to Tracking Controller:**
   - Direct observations of the robot arm's joint positions and velocities (`measured_arm_position` and `measured_arm_velocity`) are passed through `tracking_arm_position` and `tracking_arm_velocity` functions to their respective input ports on the tracking controller.

4. **Constant Parameters:**
   - A constant target pose for the end effector is set within the `tracking_ee_pose_target` function, reflecting the secondary objective of maintaining the end-effector pose close to a predefined pose.

#### Performance Metric Selection and Controller Tuning:
To tune the controller parameters and select an appropriate performance metric, the complete flow of data and objectives should be considered:
- For the **task controller**, the performance metric might include the Euclidean distance between the current pole position/angle and its upright position (zero angle), the root-mean-square error (RMSE) of the pole's angle over time, or the control effort exerted.
- Tuning parameters for the task controller (`LQRController`) include adjusting the `Q` and `R` matrices, which balance between penalizing deviation from the target state and the magnitude of the control effort.

- For the **tracking controller**, the performance metric might focus on the precision of the end effector's pose alignment with its target and the magnitude of applied forces.
- Tuning involves the force control axis mask and potentially adjusting internal pose control parameters within the `PoseForceController` to achieve desired tracking performance and responsiveness.

By carefully monitoring these performance metrics and adjusting controller parameters accordingly, the control schemes can be refined to optimize both the primary objective of stabilizing the pole and the secondary objective of maintaining the desired end-effector pose.
```

C.3 Parameter Optimization

````
## System
The GPT is a proficient programmer of Python and PyDrake and a professor in control. The GPT will help the user to revise the code after reading the requirements. The GPT will be instructed step by step so please only complete mentioned tasks. The code implements a robot skill through a hierarchical design. A task controller is chosen to give high-level control, and a tracking controller is given to track the task control. The task controller and the tracking controller are connected through callback functions. The GPT will help to tune the parameters and check variable assignments of the hierarchical control systems. A time series of the task controller state and control will be given to help you determine if the system is running as expected. Then GPT needs to decide if the time series is desired, and then identify parameters and variables that can affect the time series. In the end, The GPT needs to tune the parameters and variables through multiple rounds of interaction.

The GPT can update any of the following code blocks: <task_model>, <tracking_model>, <task_controller>, <tracking_controller>, <task_callback>, <tracking_callback>. Each time, the GPT needs to pick and rewrite one or more blocks to update the parameters or variable assignments. The parameters and variables can be anything defined in the code. Sometimes, it may require switching the order of dimensions to correct the implementation error.

In the GPT's response, the most important part will be <block_name> followed by a code block enclosed by ```python```. This explicitly indicates the code is written for a specific block. The GPT needs to give the complete updated code block that contains the revision. The GPT can feel free to make some plans to help it think before writing the code blocks. There is no need to explain the code after the code block. The GPT should not import any module. The GPT should not make up arguments, variables, and anything else that is not mentioned.

## User
Please help me to compose a skill to stabilize a cart pole. The cart is attached to the end effector of the robot arm. The pole rotates freely around the x-axis and is unactuated. Please stabilize the pole around the upright position (when the pole angle, denoted by PolePin, is 0) by exerting force on the cart along the y-axis. A secondary objective is to keep the end-effector pose as close as possible to [1.57078469e+00, 0.0, 1.57079455e+00, 5.96598901e-01, 1.34625984e-03, 4.33589236e-01] ([roll, pitch, yaw, x, y, z]). The cart is 0.1 kg, and the pole is 0.25 m, 0.01 kg.

## Assistant
Certainly! I'm good at tuning parameters for control systems in PyDrake. I will follow your instructions carefully and think carefully.

## User
Here is the summary of the control system to explain the expected way of how do the controllers work, and how the ports are connected:
{implementation_summary} (from Interface Alignment)

Here is a yaml file describing what the inputs to the skill are:
{skill_input_summary}

Here we print the port name, type, and sample value for all the input ports:
{input_port_sample}

Here are the detailed implementation of the task_model, tracking_model, task_controller, tracking_controller, task_callback (the callback functions for the task input port), and tracking_callback (callback functions for the tracking input ports).

=======
<task_model>:
```python

x0 = np.zeros((4,1))  # Initializing the state for the cart-pole system
u0 = np.zeros((1,1))  # Initial control input (force)
m_cart = 0.1  # mass of the cart
m_pole = 0.01  # mass of the pole
l_pole = 0.25  # length of the pole
g = 9.81  # gravitational acceleration
# A and B matrices are placeholders and must be defined based on the specific dynamics of the cart-pole system
A = np.array([[0, 1, 0, 0],
              [0, 0, m_pole * g / m_cart, 0],
              [0, 0, 0, 1],
              [0, 0, (m_cart + m_pole) * g / (l_pole * m_cart), 0]])
B = np.array([[0], [1/m_cart], [0], [-1/(l_pole*m_cart)]])
C = np.eye(4)
D = np.zeros((4, 1))
task_model = setup_linear_model(x0, u0, A, B, C, D)

```

=======
<tracking_model>:
````

````
```python

tracking_model = setup_arm_with_objects_model(arm_type, object_info)

```

=======
<task_controller>:
```python

Q = np.diag([1, 1, 10, 10])  # Emphasizing the importance of the pole's position and velocity
R = np.array([[0.1]])  # Control effort cost
task_controller = LQRController(task_model, Q, R, x0, u0)

```

=======
<tracking_controller>:
```python

force_control_axis_mask = np.array([False, False, False, False, True, False])  # Force control along the y-axis
tracking_controller = PoseForceController(tracking_model, force_control_axis_mask)

```

=======
<task_callback>:
```python

def task_state(self, context, output: BasicVector):
    # Extracting necessary inputs
    measured_body_poses = self.GetInputPort("measured_body_poses").Eval(context)
    measured_body_velocities = self.GetInputPort("measured_body_velocities").Eval(context)
    measured_joint_states = self.GetInputPort("measured_joint_states").Eval(context)
    # Initialize state vector with zeros
    state = np.zeros(4)
    # Position and Velocity of Cart along the y-axis from body poses and velocities
    cart_pose = measured_body_poses['Cart'].translation()
    cart_velocity = measured_body_velocities['Cart'].translational()
    state[0] = cart_pose[1]  # y position of Cart
    state[1] = cart_velocity[1]  # y velocity of Cart
    # Pole's angle (around the x-axis) and angular velocity from joint states
    pole_angle = measured_joint_states['PolePin']['positions']
    pole_angular_velocity = measured_joint_states['PolePin']['velocities']
    state[2] = pole_angle[0]  # Assuming 'positions' is a numpy array with one element
    state[3] = pole_angular_velocity[0]  # Assuming 'velocities' is a numpy array with one element
    output.SetFromVector(state)

```

=======
<tracking_callback>:
```python

def tracking_ee_pose_target(self, context, output: BasicVector):
    # Desired end-effector pose is constant as per design summary
    desired_ee_pose = [1.57078469e+00, 0.0, 1.57079455e+00, 5.96598901e-01, 1.34625984e-03, 4.33589236e-01]
    output.SetFromVector(np.array(desired_ee_pose))
def tracking_ee_force_target(self, context, output: BasicVector):
    # Obtaining the control action from task controller, directing it to apply force along y-axis
    task_control = self.GetInputPort("task_control").Eval(context)
    # Only force along the y-axis is controlled, the rest is masked out with zeros.
    ee_force_target = np.array([0.0, 0.0, 0.0, 0.0, task_control[0], 0.0])
    output.SetFromVector(ee_force_target)
def tracking_arm_position(self, context, output: BasicVector):
    # Directly passing the robot arm's measured position
    measured_arm_position = self.GetInputPort("measured_arm_position").Eval(context)
    output.SetFromVector(measured_arm_position)
def tracking_arm_velocity(self, context, output: BasicVector):
    # Directly passing the robot arm's measured velocity
    measured_arm_velocity = self.GetInputPort("measured_arm_velocity").Eval(context)
    output.SetFromVector(measured_arm_velocity)

```


Here is time series data of the task space state and task space control, which can be used as a metric to determine if the system is running as expected.
{time series of task state and task control}
````

```
Please first think step by step about the expected time series and determine if the current series is desired.
Then please analyze one by one if any of the following happen.

* Mismatch of order. The order of the state is inconsistent in model definition, controller definition, or callback function. Double-check if the order is consistent. For example, an error can be modeling the task space with [pos_a, vel_a, pos_b, vel_b], but composed the task state by [pos_b, vel_b, pos_a, vel_a] in task callbacks. In this case, the GPT needs to redefine the state, the model, or the callback function.
* Error in definition. Double-check if the assignment is correct for the model, the state, and the callback functions. Make sure the signs are correct, the numbers are desired, there is no missing assignments or careless mistakes. Especially pay attention to the dynamic model. Think step by step about the meaning of each element in the dynamic function.
* Messed up rotation and translation order. In pydrake, for pose, velocity, and force vectors, rotation always comes first, such as [roll, yaw, pitch, x, y, z] or [x-rotation, y-rotation, z-rotation, x-translation, y-translation, z-translation]
* Modeling is inaccurate, that is when the model involves estimated parameters and it turns out the estimation is inaccurate.
* The controller parameter is not optimal to finish the task. For example, the system diverges with the current controller gain.

If the GPT thinks any one of these happens, then identify the parameters and variable assignments that require tuning to correct errors and improve the performance. The identified code blocks that contain these parameters and variables will be updated in later conversations.

## Assistant
Analyzing the provided system setup, time series data for the task state and control, and the goals and structure of the control system, let's break down the evaluation and identification process step by step.

### Analysis of Desired Time Series
The time series of **task_state** shows the cart-pole system's state vector components over time: cart position and velocity along the y-axis, and pole angle and angular velocity. The **task_control** shows the computed control actions (force) to be applied along the y-axis of the cart to stabilize the pole around the upright position.
- The pole angle (`theta`) initially decreases towards 0, indicating movement towards the upright position, which is desired. However, the absolute value of `theta` should ideally approach zero more closely over time, stabilizing around the upright position.
- The **task_control** force values start positive, then decrease, and become negative, suggesting efforts to stabilize the pole by moving the cart back and forth. The force values appear to reduce in magnitude over time, which might indicate a diminishing need for correction as the system stabilizes, a desirable trend.

### Expected and Actual Time Series Comparison
The time series appears to follow expected trends: corrections decrease as the system approaches a stable state. However, for precise stability and optimal control, further refined tuning of controller parameters and system modeling accuracy might be necessary.

### Identification of Potential Issues
1. **Mismatch of Order**: The definitions provided seem consistent in terms of order, especially in the `task_state` callback, where `[x, x_dot, theta, theta_dot]` order matches with how the system model and controller are defined.

2. **Error in Definition**: Initial checks don't reveal obvious errors in sign or incorrect assignments, but deeper evaluation is needed when examining specific code blocks.

3. **Messed Up Rotation and Translation Order**: Given the problem scope, it's primarily about linear motion and rotation around a single axis. The task controller deals with linear motion and doesn't directly manipulate rotations, while the tracking controller's force application is aligned correctly along the y-axis.

4. **Modeling Inaccuracy**: Based on the provided dynamics and control model, the fundamental modeling approach seems aligned with the physical behavior of a cart-pole system. However, specific parameters (masses, lengths) are accurate as provided and need no immediate correction.

5. **Controller Parameter Not Optimal**: Parameters within `Q` and `R` in the <task_controller> and force control axis mask in <tracking_controller> suggest room for optimization. For improved performance, especially in task prioritization (pole stability vs. cart position) and control effort, adjustments to these values may offer better results.

### Preliminary Recommendations for Parameters and Variable Assignments Tuning
To potentially correct errors and improve system performance, adjustments to `Q` and `R` matrix values in **<task_controller>** should be considered to better balance between penalizing the pole's deviation from the upright position and the energy (force) expended for control. The initial emphasis is on the pole's position (`theta`) and velocity (`theta_dot`), hinting at the right direction, but fine-tuning will help achieve a more precise and energy-efficient stabilization.
```

````
Next steps in tuning will be directed towards revising `<task_controller>` parameters with a focus on likely adjusting `Q` for a higher emphasis on reducing the pole's angle deviation and possibly refining `R` to manage control effort more effectively.

## User
Now please decide if the time series is desired.
If it is, please answer with one word "yes".
If not, you must rewrite the code blocks that contain parameters and variables that require tuning to correct errors or to improve performance. Do not fear to make mistakes. We will evaluate the code block, produce a new time series, and iteratively improve the parameters.

The response must start with <step_name> followed by a code block enclosed by ```python```. <step_name> must be included before each code block so the code block can be recognized. step_name must be enclosed in angle brackets <>. <step_name> must be one of <task_model>, <tracking_model>, <task_controller>, <tracking_controller>, <task_callback>, <tracking_callback>. For <task_callback>, <tracking_callback>, you can only rewrite chosen functions of the codeblock. But for other steps, make sure to rewrite the whole code block because we will directly replace the original block with the rewritten one. Do not miss any variable in the code block.  During "rewriting", you are only supposed to modify the existing content of the code block, such as values assigned to parameters and variables. Do not revise functions and variables not presented in the existing code block.

An example output is:

<tracking_callback>
```python
def tracking_func2(self, context, output):
    # updated function body
```

<task_controller>
```python
# updated whole code block
```

## Assistant
The time series is not entirely desired. While it shows that the pole's angle is moving towards stabilization around the upright position, the control effort depicted by task_control is exhibiting a pattern of diminishing force application even when the pole has not reached a stabilized state (angle reaches and crosses 0 but starts to deviate again). The task_control should ideally drive the system more precisely towards stabilization without such a crossover. The adjustments needed seem to be in the tuning of the LQR controller's Q and R matrices, which directly affect the control performance by balancing between state deviations and the cost of control effort.

<task_controller>
```python
Q = np.diag([1, 1, 100, 10])  # Increased emphasis on the pole's angle stabilization
R = np.array([[0.01]])  # Reduced cost for control effort to allow more aggressive control actions
task_controller = LQRController(task_model, Q, R, x0, u0)
```

## User
Here is the time series data of the task space state and task space control for the current code:
{}

Now please decide if the time series is desired.
If it is, please answer with one word "yes".
If not, you must rewrite the code blocks that contain parameters and variables that require tuning to correct errors or to improve performance. Do not fear to make mistakes. We will evaluate the code block, produce a new time series, and iteratively improve the parameters.

The response must start with <step_name> followed by a code block enclosed by ```python```. <step_name> must be included before each code block so the code block can be recognized. step_name must be enclosed in angle brackets <>. <step_name> must be one of <task_model>, <tracking_model>, <task_controller>, <tracking_controller>, <task_callback>, <tracking_callback>. For <task_callback>, <tracking_callback>, you can only rewrite chosen functions of the codeblock. But for other steps, make sure to rewrite the whole code block because we will directly replace the original block with the rewritten one. Do not miss any variable in the code block.  During "rewriting", you are only supposed to modify the existing content of the code block, such as values assigned to parameters and variables. Do not revise functions and variables not presented in the existing code block.

An example output is:

<tracking_callback>
```python
def tracking_func2(self, context, output):
    # updated function body
```

<task_controller>
```python
# updated whole code block
````
````

```

## Assistant
yes

## C.4 Model template summary

```
setup_linear_model:
  description: |
    This function returns an affine dynamic model.
    This model is especially useful when the system is or can be approximated by affine dynamics.
    The dynamic equation is: x_dot = A * x + B * u; y = C * x + D * u.

    Example usage:
    """
    x0 = np.zeros((4,1))
    u0 = np.zeros((1,1))
    m = 0.01
    l = 0.1
    g = 0.81
    A = np.array([[0,1,0,0], [0,0,m*g,0],[0,0,0,1],[0,0,0,1]])
    B = np.array([[0],[1/m],[0],[-1/m]])
    C = np.eye(4)
    D = np.zeros((4, 1))
    model = setup_linear_model(x0, u0, A, B, C, D)
    """
  args:
    - name: x0
      type: numpy.ndarray
      size: (n_x, 1)
      description: Initial state
    - name: u0
      type: numpy.ndarray
      size: (n_u, 1)
      description: Initial input
    - name: A
      type: numpy.ndarray
      size: (n_x, n_x)
      description: State matrix
    - name: B
      type: numpy.ndarray
      size: (n_x, n_u)
      description: Input matrix
    - name: C
      type: numpy.ndarray
      size: (n_y, n_x)
      description: Output matrix
    - name: D
      type: numpy.ndarray
      size: (n_y, n_u)
      description: Feedforward matrix

setup_arm_model:
  description: |
    This function returns a model of the robot arm.
    This model is especially useful when the controller only needs to consider the robot arm.
    Because this function will be called in skill's init function, arm type is directly available
    as arm_type.
    Usage:
    """
    model = setup_arm_model(arm_type)
    """
  args:
    - name: arm_type
      type: str
      description: The robot arm type. It should be consistent as the skill's.

setup_arm_with_objects_model:
  description: |
    This function returns a model of the robot arm and static objects in the scene.
    This model is especially useful when the controller needs to consider the interaction between
    the robot and objects, such as grasping and collision avoidance.
    It can be used for controlling in the cartesian space or in the joint space.
    Because this function will be called in skill's init function, arm type and object_info
    are directly available as arm_type and object_info.
    Usage:
    """
```
```

```
    model = setup_arm_with_objects_model(arm_type, object_info)
    """
  args:
    - name: arm_type
      type: str
      description: The robot arm type. It should be consistent as the skill's.
    - name: object_info
      type: dict
      description: The dictionary of all objects in the scene.  It should be consistent as the skill's.

setup_placeholder_model:
  description: |
    This function returns None.
    This model is especially useful when the controller does not need any dynamic model.
    Example usage:
    """
    model = setup_placeholder_model()
    """
```

C.5 Controller template summary

```
LQRController:
  description: |
    This class implements a Linear Quadratic Regulator (LQR) controller for a given affine system.
    The controller computes control inputs based on the state deviations from a given equilibrium.
  args:
    - name: model
      type: Diagram
      description: A Diagram containing the AffineSystem for which the LQR controller is designed.
    - name: Q
      type: numpy.ndarray
      description: State cost matrix.
    - name: R
      type: numpy.ndarray
      description: Control cost matrix.
    - name: x0
      type: numpy.ndarray
      description: Equilibrium state around which the controller is designed.
    - name: u0
      type: numpy.ndarray
      description: Equilibrium control input.
  input_ports:
    - name: state
      type: BasicVector
      size: "len(x0)"
      description: Represents the current state of the system.
  output_ports:
    - name: control
      type: BasicVector
      size: "len(u0)"
      description: Represents the computed control input based on the LQR control law.

IdentityController:
  description: |
    This class implements an identity controller that directly outputs the arm torque it receives.
    It is the best choice when the task controller is already enough to accomplish the task and we
    only need a placeholder for the tracking controller.
  args:
    - name: model
      type: Diagram
      description: A Diagram containing the MultiBodyPlant model of the robotic system.
  input_ports:
    - name: applied_arm_torque
      type: BasicVector
      size: "plant.num_actuators()"
      description: applied_arm_torque computed by the task controller.
  output_ports:
    - name: applied_arm_torque
      type: BasicVector
      size: "plant.num_actuators()"
      description: directly output the input applied_arm_torque.


PoseForceController:
  description: |
    This class implements a controller that combines pose and force control for a robotic arm.
    The controller computes torques based on the desired end-effector pose and force.
```

```yaml
    It is best suitable for the situation when we have a desired force on some dimensions and a desired
↪  position or rotation on other dimensions. Note that the controller can not track force and pose at the
↪  same time for a dimension.
  args:
    - name: model
      type: Diagram
      description: A Diagram containing the MultiBodyPlant model of the robotic system.
    - name: force_control_axis_mask
      type: numpy.ndarray
      description: Boolean array of length 6 to specify which axes are controlled by force. The order is
↪  [x-rotation, y-rotation, z-rotation, x-translation, y-translation, z-translation]. For each dimension,
↪  True represents force control, False represents pose control.
  input_ports:
    - name: ee_pose_target
      type: BasicVector
      size: 6
      description: Desired end-effector pose (roll, pitch, yaw, x, y, z).
    - name: ee_force_target
      type: BasicVector
      size: 6
      description: Desired end-effector force. (x-rotation, y-rotation, z-rotation, x-translation,
↪  y-translation, z-translation).
    - name: arm_position
      type: BasicVector
      size: "plant.num_positions(arm)"
      description: Current joint positions of the arm.
    - name: arm_velocity
      type: BasicVector
      size: "plant.num_velocities(arm)"
      description: Current joint velocities of the arm.
  output_ports:
    - name: applied_arm_torque
      type: BasicVector
      size: "plant.num_actuators()"
      description: Computed joint torques for the robot arm.

CartesianStiffnessController:
  description: A cartesian stiffness controller using impedance control to determine control torques.
↪  Makes the robot behave as if a spring-damper system is attached to the end-effector. The stiffness can
↪  be adjusted dynamically by providing input to the controller_parameters port. The advantage of this
↪  controller is it can provide compliant behavior with adjustable stiffness.
  args:
    - name: model
      type: Diagram
      description: A Diagram containing the robot model (MultiBodyPlant) for computing dynamics
  input_ports:
    - name: ee_target
      type: BasicVector
      size: 6
      description: Desired end effector target (pose or twist). [roll, pitch, yaw, x, y, z]
    - name: ee_target_type
      type: AbstractValue
      description: Type of the end effector target. This type is enum. The must be EndEffectorTarget.kPose
↪  or EndEffectorTarget.kTwist. Do not use `str`, use enum directly.
    - name: arm_position
      type: BasicVector
      size: "(number of positions of the plant's arm)"
      description: Current joint position of the robot arm.
    - name: arm_velocity
      type: BasicVector
      size: "(number of velocities of the plant's arm)"
      description: Current joint velocity of the robot arm.
    - name: controller_parameters
      type: AbstractValue
      example_value:
        cartesian_stiffness: "[0.1]*3 + [200]*3"
      description: The controller can be parameterized through an abstract input port
↪  "controller_parameters" to modify its behavior during runtime, allowing changes to cartesian_stiffness
↪  and "cartesian_damping" matrices. [roll, pitch, yaw, x, y, z]
  output_ports:
    - name: applied_arm_torque
      type: BasicVector
      size: "(number of actuators in the plant)"
      description: Control torques applied to the joints of the robot arm.

SafeController:
  description: A controller that projects nominal control torque to a safe control torque using a control
↪  barrier function. Currently, the safe constraint is collision avoidance. The advantage of this
↪  controller is to safeguard a nominal control input port to realize reactive collision avoidance.
  args:
    - name: model
```

```yaml
      type: Diagram
      description: Diagram containing the multibody plant and scene graph, as well as the information of obstacles for collision avoidance.
    - name: meshcat
      type: Meshcat
      description: This arg is optional. For visualization purposes.
  input_ports:
    - name: arm_target
      type: BasicVector
      size: "num_q"
      description: Desired joint target for the arm.
    - name: arm_target_type
      type: AbstractValue
      description: Type of the target. This type is enum. The value must be JointTarget.kPosition, JointTarget.kVelocity, or JointTarget.kTorque. Do not use `str`, use enum directly.
    - name: arm_position
      type: BasicVector
      size: "num_q"
      description: Current joint position of the arm.
    - name: arm_velocity
      type: BasicVector
      size: "num_q"
      description: Current joint velocity of the arm.
  output_ports:
    - name: applied_arm_torque
      type: BasicVector
      size: "num_q"
      description: Computed joint torque for the arm.


CartesianTrajectoryController:
  description: Constructs a trajectory in cartesian space and publishes waypoints by interpolating the trajectory. Depending on the polynomial order, it can produce first-order holds or cubic splines for the trajectory.
  args:
    - name: model
      type: Diagram
      description: model is not useful in this controller, but should be passed in for consistency
    - name: polynomial_order
      default: 1
      description: Order of the polynomial for trajectory optimization. Supported values are 1 (for First Order Hold) and 3 (for Cubic Spline).
    - name: meshcat
      type: Meshcat
      description: A visualization tool. (optional)
  input_ports:
    - name: controller_parameters
      type: AbstractValue
      example_value:
        new_trajectory: true
        way_points:
          times: "[list of times]"
          points: "[list of pose in RPY_XYZ format]"
      description: Controller parameters that can indicate the need for a new trajectory and provide waypoints.
    - name: measured_ee_pose
      type: BasicVector
      size: 6
      description: Measured pose of the end effector in RPY_XYZ format.
    - name: measured_ee_twist
      type: BasicVector
      size: 6
      description: Measured twist of the end effector. RPY_XYZ
  output_ports:
    - name: ee_pose_nom
      type: BasicVector
      size: 6
      description: Nominal end-effector pose based on the last solved trajectory spline evaluated at the current time. RPY_XYZ


KinematicTrajectoryModelPredictiveController:
  description: An MPC controller using kinematic trajectory optimization to calculate a collision-free trajectory in joint space. The objective of this controller is to reach a goal in cartesian space with the end effector. Please note that the output is in joint space.
  args:
    - name: model
      type: Diagram
      description: Diagram containing the multibody plant and scene graph, as well as the information of obstacles for collision avoidance
    - name: resolve_period
```

```yaml
      type: float
      default: 3.0
      description: Trajectory optimization resolution period.
    - name: num_steps
      type: int
      default: 20
      description: Number of optimization steps.
    - name: meshcat
      type: Meshcat Optional
      description: For visualization
  input_ports:
    - name: controller_parameters
      type: AbstractValue
      example_value:
        goal_pose: RigidTransform
      description: Controller parameters including desired goal pose for the end effector.
    - name: measured_arm_position
      type: BasicVector
      size: "(number of positions of the robot arm)"
      description: Current joint position of the robot arm.
    - name: measured_arm_velocity
      type: BasicVector
      size: "(number of velocities of the robot arm)"
      description: Current joint velocity of the robot arm.
  output_ports:
    - name: q_nom
      type: BasicVector
      size: "(number of actuators in the plant)"
      description: Nominal joint positions of the arm.
```

C.6 Skill input summary

```yaml
- port_name: controller_parameters
  description: Controller parameters that can be adjusted dynamically during running.
  type: AbstractValue
  data_structure: dict

- port_name: measured_arm_position
  description: Observation of the robot arm's joint position.
  type: BasicVector
  dimensions: num_q

- port_name: measured_arm_velocity
  description: Observation of the robot arm's joint velocity.
  type: BasicVector
  dimensions: num_q

- port_name: measured_ee_pose
  description: Observation of the end effector's pose. The first three elements correspond to rotation.
↪  The last three elements correspond to translation.
  type: BasicVector
  dimensions: 6

- port_name: measured_ee_twist
  description: Observation of the end effector's twist. The first three elements correspond to rotation.
↪  The last three elements correspond to translation.
  type: BasicVector
  dimensions: 6

- port_name: measured_joint_states
  description: Observation of the joint states (including joint states of articulated objects).
  type: AbstractValue
  data_structure: dict

- port_name: measured_body_poses
  description: Observation of the poses of the rigid bodies in the scene.
  type: AbstractValue
  data_structure: dict

- port_name: measured_body_velocities
  description: Observation of the velocities of the rigid bodies in the scene.
  type: AbstractValue
  data_structure: dict
```

C.7 Input port sample

```
============================================
TrackingControllerConverter input port contents:
controller_parameters :  <class 'dict'> ,  {'gripper_command': 'release'}
measured_arm_position :  <class 'numpy.ndarray'> ,  [ 0.          0.26179939  3.14159265 -2.26918531  0.
↪  0.9599
  1.57079633]
measured_arm_velocity :  <class 'numpy.ndarray'> ,  [0. 0. 0. 0. 0. 0. 0.]
measured_ee_pose :  <class 'numpy.ndarray'> ,  [ 1.57108469e+00 -3.92246873e-06  1.57079455e+00
↪  5.96598901e-01
  1.34625984e-03  4.33589236e-01]
measured_ee_twist :  <class 'numpy.ndarray'> ,  [0. 0. 0. 0. 0. 0.]
measured_joint_states :  <class 'dict'> ,  {'Actuator1': {'positions': array([0.]), 'velocities':
↪  array([0.])}, 'Actuator2': {'positions': array([0.26179939]), 'velocities': array([0.])}, 'Actuator3':
↪  {'positions': array([3.14159265]), 'velocities': array([0.])}, 'Actuator4': {'positions':
↪  array([-2.26918531]), 'velocities': array([0.])}, 'Actuator5': {'positions': array([0.]),
↪  'velocities': array([0.])}, 'Actuator6': {'positions': array([0.9599]), 'velocities': array([0.])},
↪  'Actuator7': {'positions': array([1.57079633]), 'velocities': array([0.])}, 'EndEffector':
↪  {'positions': array([], dtype=float64), 'velocities': array([], dtype=float64)},
↪  'world_welds_to_base_link': {'positions': array([], dtype=float64), 'velocities': array([],
↪  dtype=float64)}, 'PolePin': {'positions': array([0.1]), 'velocities': array([0.])},
↪  'end_effector_link_welds_to_Cart': {'positions': array([], dtype=float64), 'velocities': array([],
↪  dtype=float64)}, '$world_obstacle_box_2': {'positions': array([ 0.99875026,  0.        ,  0.        ,
↪  -0.04997917,  0.4        ,
        0.3        ,  0.101      ]), 'velocities': array([0., 0., 0., 0., 0., 0.])}, '$world_goal_box':
↪  {'positions': array([1.   , 0.   , 0.   , 0.   , 0.6  , 0.   , 0.026]), 'velocities': array([0., 0.,
↪  0., 0., 0., 0.])}}
measured_body_poses :  <class 'dict'> ,  {'Cart': RigidTransform(
  R=RotationMatrix([
    [1.771814082172029e-06, 0.9999999584203277, -0.00028836817309205l8],
    [0.9999999999907371, -1.7706828933479899e-06, 3.922979501902304e-06],
    [3.9224687301950226e-06, -0.00028836818003996167, -0.9999999584142024],
  ]),
  p=[0.48659890557829427, 0.001346454619425365, 0.4336209565978073],
)}
measured_body_velocities :  <class 'dict'> ,  {'Cart': SpatialVelocity(
  w=[0.0, 0.0, 0.0],
  v=[0.0, 0.0, 0.0],
)}

task_control :  <class 'numpy.ndarray'> ,  [0.]
============================================
```