Synthesizing Interpretable Control Policies through Large Language Model Guided Search

Carlo Bosio and Mark W. Mueller

***Abstract*— The combination of Large Language Models (LLMs), systematic evaluation, and evolutionary algorithms has enabled breakthroughs in combinatorial optimization and scientific discovery. We propose to extend this powerful combination to the control of dynamical systems, generating interpretable control policies capable of complex behaviors. With our novel method, we represent control policies as programs in standard languages like Python. We evaluate candidate controllers in simulation and evolve them using a pre-trained LLM. Unlike conventional learning-based control techniques, which rely on black box neural networks to encode control policies, our approach enhances transparency and interpretability. We still take advantage of the power of large AI models, but leverage it at the policy design phase, ensuring that all system components remain interpretable and easily verifiable at runtime. Additionally, the use of standard programming languages makes it straightforward for humans to finetune or adapt the controllers based on their expertise and intuition. We illustrate our method through its application to the synthesis of an interpretable control policy for the *pendulum swing-up* and the *ball in cup* tasks. We make the code available at https://github.com/muellerlab/synthesizing_interpretable_control_policies.git**

I. INTRODUCTION

Control systems and artificial intelligence (AI) are two fields with immense practical impact, yet their integration often faces significant challenges. While control theory offers reliable methods to stabilize and steer complex systems, recent advances in machine learning (ML) have dramatically improved our ability to leverage large-scale data. However, the use of black box AI models, particularly neural networks, is not always suitable for critical control applications where transparency and verifiability are essential. Our work introduces a solution to this problem. Inspired by recent breakthroughs in Combinatorial Optimization [1], we propose representing control policies as programs written in standard languages like Python, and evolving them using a pre-trained LLM and a simulation framework for evaluation. Our approach still leverages the power of large AI models, but shifts the abstraction layer, moving the black box component from the runtime execution to the policy design phase (an outline of our method is shown in Fig. 1). The output of our control synthesis framework is a fully interpretable program representing a control policy for the system and task of interest. The key advantage of our approach is the use of code as the policy representation. Programming languages, being our primary tools for instructing machines, are inherently interpretable. A control engineer or system operator can read, understand, and even modify the policy directly, without needing to decipher complex neural network architectures or weight matrices.

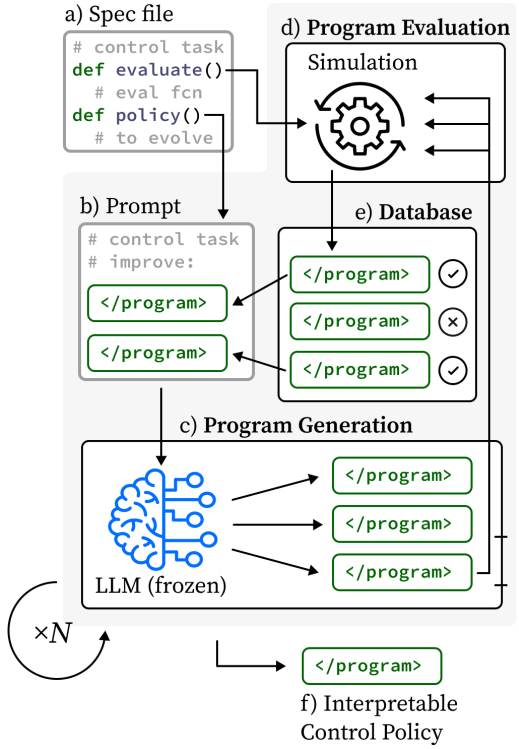

Fig. 1. Schematic of the algorithmic infrastructure for the synthesis of interpretable control policies. The input to the algorithm is a specification file a) containing a task description, the implementation of an evluation function to score programs, and some starter code for the control policy to evolve. A prompt b) is constructed pasting the current best programs (the starter code at the beginning). The prompt is fed to a *Program Generation* block c) containing a pre-trained LLM, which produces more programs. The control policies contained in the LLM outputs are fed to the *Program Evaluation* block d), which scores them based on their performance in simulation. The programs leading to poor performance are discarded, while the higher scoring ones are stored in a *Database* e), from which they are sampled to be included in following prompts and improved.

In the following sections we frame in more detail this research by highlighting relevant previous work. In Section II we describe our methodology and infrastructure. Then, in Section III we show an application of the proposed method and in Section IV we provide a brief discussion and conclude by highlighting possible future directions.

A. Large Language Models

Recent developments in Large Language Models (such as [2]) have opened novel research areas focused on their

*The authors are with the High Performance Robotics Laboratory, University of California, Berkeley. Contact: {c.bosio, mwm}@berkeley.edu

applications. One popular application is LLMs for code. In fact, a lot of effort has been put in the development, training, and finetuning of large models for code generation [3]–[5]. With the rise of these models for code, also evaluation and benchmarking of these models have been popular research directions [6], [7]. On top of LLMs for code generation, extensive system-level research has been proposed with the aim of integrating LLMs with additional components to achieve more complex tasks than with single prompt engineering. A popular example is the combination of LLMs with evaluators, i.e. programmatic ways of scoring their outputs. This paradigm has been applied in various contexts, such as automated reasoning [8], code debugging [9], and algorithm design [10], [11]. The integration of this generation-evaluation technique in an evolutionary procedure alleviates LLM hallucination, and in some cases leads to self-improving loops which output high performance programs and new knowledge [1]. It is important to highlight that these results are not achieved thanks to domain-specific knowledge contained in the LLM training dataset. Instead, they are achieved through the combination of the capability of LLMs to generate functional code, and evolutionary optimization techniques. Our work shows how to effectively apply these LLM-based frameworks to control systems design.

B. Learning-Based Control

Control systems have benefited in many ways from learning-based techniques [12]. A popular and well-studied field is adaptive control, which revolves around the identification of discrepancies between a system's dynamics model and its real-world behavior, and update of model parameters to compensate this mismatch [13], [14]. Learning-based methods have found successful application in highly repetitive scenarios, where error patterns are iteratively approximated and incorporated in the controller over multiple executions of the same task. These techniques fall under the iterative learning control category [15]. Another family of methods, typically referred to as imitation learning, consists of approximating a control policy for a given task through a set of expert demonstrations (i.e. trajectories accomplishing the task of interest) [16]. Reinforcement learning (RL) represents a significant shift in control paradigms, offering a way to approximate optimal control policies through interaction with the environment [17]. Recent advances in RL have led to important results in areas where traditional methods fall short, such as locomotion [18], [19] and manipulation [20]. However, one fundamental issue preventing all these techniques to be safely and reliably deployed in real world contexts is their lack of interpretability.

C. Interpretability in Learning-Based Control

In the context of Machine Learning systems, interpretability is defined as the *ability to explain or to present in understandable terms to a human* [21]. This property is particularly crucial in the context of control systems and automation, where guaranteeing safety of operation is in many cases essential. In such safety-critical contexts, being able to inspect and understand which building block of a system led to a failure is of fundamental importance, and is not possible with black box components such as neural networks. Interpretability has been investigated through several different approaches [22]. One of them is the use of interpretable architectures for learning-based control. Some examples are decision trees [23] and fuzzy controllers [24], [25]. These techniques facilitate the tracing of the decision-making process, providing some level of insight into the system's behavior. Another approach that has been proposed to make learned components more interpretable is encouraging sparsity in parameter or weight matrices [26], as sparsity is often used as a proxy for interpretability [27]. We argue that interpretability in learning-based control should encompass not only the ability to read and (partially) understand the system's logic, but also the capacity to modify it with a clear understanding of the consequences. By representing control policies in a standard programming language, we aim to take a step towards the intuitive understanding required for the practical, safe deployment of learning-based control systems.

II. METHODOLOGY

Our work addresses the problem of finding a high-performance control policy for a control task of interest. We first describe the problem fundamentals, following the formalism of [28], and then the algorithmic aspects of our interpretable control synthesis approach.

We are concerned with the study of a discrete-time dynamical system with dynamics in the form

$$x_{t+1} = f(x_t, u_t), \tag{1}$$

where $t \in \mathbb{N}$ is the time index, $x_t \in \mathbb{R}^n$ is the state of the system at time step t, and $u_t \in U \subset \mathbb{R}^m$ is the control input. At each time step t, a stage reward

$$r_t = g(x_t, u_t) \tag{2}$$

is incurred. The general objective is to find a control policy $u_t = h(x_t)$ to maximize the cumulative reward

$$R = \sum_{t=0}^{T} r_t, \tag{3}$$

where T is a specified time horizon. It is well known that this problem is in general very complex, and in many cases only approximate solutions can be found. Our goal is to produce an approximate solution (i.e. a control policy approximately maximizing the cumulative reward) that is interpretable. A typical approach would be to pick a functional representation for the control policy (e.g. linear feedback, neural network) and optimize its parameters to maximize the cumulative reward. To guarantee interpretability, in our setting we represent the function $h(\cdot)$ directly as a program $\texttt{policy}(\cdot)$ in Python. Therefore, our control policy is encoded as

$$u_t = \texttt{policy}(x_t). \tag{4}$$

The goal is then to find a high-performing control program $\texttt{policy}^*(\cdot)$ by approximating the solution to the following

optimization problem:

$$\begin{aligned} \texttt{policy}^*(\cdot) = \max_{\texttt{policy}(\cdot)} \quad & R \\ \text{s.t.} \quad & x_{t+1} = f(x_t, u_t), \forall t \in \{0, ..., T\} \\ & u_t = \texttt{policy}(x_t), \end{aligned} \tag{5}$$

where R is defined in eq. 3. In this formulation, the search does not happen in a parameter space encoding a mathematical structure (such as, for instance, a neural network), but directly in the space of programs.

This space is infinite dimensional and too complex to search over (for instance, there is no length limit). To work around this complexity, we leverage a Large Language Model finetuned on code to generate candidate programs and explore the space. LLMs for code are trained on a large body of human-written code examples, hence they have a bias towards concise, human-readable code. The use of an LLM shifts the optimization to the space of tokens, i.e. the elements the LLM uses to decompose a string of text (tokens can be individual words, individual characters or a mix of these). Searching in the token space is complex. To start, tokens are discrete, thus there is not a notion of gradient to guide the search, and the number of possible programs of a given length can be combinatorially large. Furthermore, randomly sampled tokens almost surely lead to programs that are syntactically incorrect and do not run. The use of an LLM alleviates these complications.

The main benefit of representing a control policy as a program is the inherent interpretability of programming language, which is by design the way humans instruct machines. In the following, we provide more detail about how to algorithmically deal with this problem, and find good candidate control policies in the form $\texttt{policy}^*(\cdot)$.

Our method is inspired by the work presented in [1]. The overall algorithmic infrastructure is shown in Fig. 1. The input is a specification file, where a description of the task, some starter code for the function to evolve and the implementation of the evaluation function are provided. At each iteration, a *Program Generation* block (containing the LLM) produces candidate control programs for the task of interest. The proposed control policies are then fed to a *Program Evaluation* block, which simulates the underlying system in closed loop. The best performing programs are stored in a *Database* and then fed back into the prompts for the subsequent iterations, where the LLM is instructed to improve upon the previously produced programs. In the following we explain in more detail the functioning of this framework.

A. Specification

The input to our control synthesis framework is a specification file (Fig. 1a). The file is composed by three main parts:

- A description of the control task to solve, together with packages and libraries the LLM-generated code can use;

```python
"""Finds a policy for the control task."""

# import libraries needed
import numpy as np

# evaluation function for the control policy
def evaluate() -> float:
  """Returns the reward."""
  environment = initialize_env()
  observation = environment.get_observation()
  total_reward = 0.0
  for _ in range(1000):
    action = policy(observation)
    reward, observation = environment.step(action)
    total_reward += reward
  return total_reward

# function to evolve
def policy(obs: np.ndarray) -> float:
  """Returns a control action."""
  action = np.random.uniform()
  return action
```

Fig. 2. Example template for a control synthesis specification.

- An initial candidate control program in the form of starter code, which will be evolved across different iterations;
- The implementation of the evaluation function used to score candidate programs.

The different parts of the specification are parsed and used at different stages of the pipeline. The starter code is pasted in the initial prompt (Fig. 1b) and fed to the LLM in the preliminary stages, when better programs have not yet been produced. The evaluation function is used in the *Program Evaluation* block (Fig. 1d) to quantify the performance of candidate programs. An example of a general specification structure for our control synthesis method is shown in Fig. 2.

B. Prompt Construction

The prompt (Fig. 1b) is a crucial component in the pipeline, as it triggers and steers the LLM generation. At each iteration, a prompt is constructed by concatenating two previously generated high-performing programs. During the initial phases of the algorithm execution, only the starter code (from the specification file) is available, and is directly pasted into the prompt. As the evolution progresses and higher-performing programs are generated, these are used in the prompt instead of the starter code. The prompt also contains an instruction to the LLM to improve upon the policies provided. The sampled policies are inserted in the prompt in the form `policy_v0`, `policy_v1`, and the function header of the policy to generate (in the form `policy_vx`) is appended at the end. An example of a general prompt structure is shown in Fig. 3.

C. Program Generation

A pre-trained LLM is the generation engine of the *Program Generation* block (Fig. 1c), in which candidate control programs are sampled. The LLM is queried with a prompt containing two high-performing control programs generated in previous iterations, and is instructed to improve them.

```
"""Finds a policy for the control task.
   On every iteration, improve policy_v1 over the
   policy_vX methods from previous iterations.
"""
import numpy as np

def policy_v0(obs: np.ndarray) -> float:
  """Returns a control action."""
  action = np.random.uniform()
  return action

def policy_v1(obs: np.ndarray) -> float:
  """Returns a control action."""
  action = 0.0
  return action

def policy_v2(obs: np.ndarray) -> float:
  """ Improved version of 'policy_v1'."""
```

Fig. 3. Example template for a prompt. The LLM generates a body for the provided function signature.

This step encourages the combination of ideas from previous programs, and is equivalent to a crossover in classical evolutionary algorithms. It is important to highlight that no additional training or finetuning is carried out, and the LLM is kept frozen across the execution of our method.

A number of hyperparameters affect the generation performances of an LLM. We found that, for our use case, the randomness of the LLM generation and the amount of repetition in the output sequence are most central. At each token generation step the LLM produces a vector containing a numerical score for every token in the vocabulary. For token i, there is a score x_i. These scores are then normalized to obtain a distribution to sample from. The sampling probabilities are obtained as

$$p_i = \frac{e^{x_i/T}}{\sum_i e^{x_i/T}}, \tag{6}$$

where T is the LLM temperature. A low T makes the token distribution narrow, a larger T makes it more uniform. In our implementation we set $T = 1$. The token generation is also steered by other hyperparameters. One of them is `top_p` (with value within $[0, 1]$), which sets the percentile of the token options to consider. If `top_p` is 1, then the whole vocabulary is considered, otherwise a fraction is discarded and the probabilities p_i are renormalized. We set `top_p` to 0.95. The `repeat_last_n` parameter sets the length (in number of tokens) of the sliding window used to check for repetition. We set `repeat_last_n` to 15 (i.e. the 15 previously generated tokens are not considered for sampling).

D. Program Evaluation

Candidate control programs are parsed from the LLM output and fed to the *Program Evaluation* block (Fig. 1d). The evaluation consists of testing the programs and quantifying their performance using the evaluation function provided in the specification file. The candidate control policy is deployed in a simulation environment, in closed loop with the dynamics of the system of interest. The performance in simulation is quantified by a numerical score (in our case, the return of eq. 3). The score is then associated to the candidate control program to make a program-score pair. Syntactically incorrect programs are discarded, while promising program-score pairs are stored in the *programs database*, from which they are sampled to be added to the subsequent prompts, and evolved. The simulation is run inside a sandbox to prevent issues (syntactic or of other nature) contained the LLM-generated control policy to interrupt the outer optimization routine.

E. Programs Database

The generated high performing programs are stored in the *programs database* (Fig. 1e). At each iteration, two programs are sampled to be integrated into the prompt and fed back to the LLM, which is instructed to provide higher performing variants. To discourage getting stuck in local optima, an island approach is implemented, where different instances of the program search are run in parallel, independently [29]. Therefore, an equivalent number of program populations are stored and evolved. Periodically, the islands containing less promising populations are emptied and re-initialized with the best performing programs from other islands. When constructing a prompt, a two stage sampling procedure happens: first, an island is sampled, then programs contained within the selected island are sampled to be added to the prompt. More detail about the algorithm implementation can be found in [1]. In our case, we use 10 independently evolved islands.

III. SET UP AND CASE STUDIES

In the following we provide more detail about the practical aspects of the implementation of our method. We then introduce the control tasks that we found interesting for its application.

A. Setup

We run our control synthesis framework on a workstation equipped with an NVIDIA RTX 3090 GPU, which has enough memory to fully load the LLM for inference. As a simulation framework for program evaluation we use the open source simulator MuJoCo [30], through the well known DeepMind Control Suite [31] library. As LLM for program generation we used an 8-bit quantized version [32] of *StarCoder2-Instruct* [4], an open source 15 billion parameter model finetuned on code generation from natural language instructions. We applied our method to the *pendulum swing-up* and *ball in cup* tasks included in the DeepMind Control Suite. A supplementary video showing examples from the case studies can be found at `https://youtu.be/7T7yRGya-q8`.

B. Pendulum swing-up

The *pendulum swing-up* task with input constraints is not easily solvable through a classical linear control. The pendulum has to accomplish a number of oscillations to accumulate enough energy, and then swing into the upright configuration. The maximum applicable torque is $1/6^{\text{th}}$ as required to lift it from motionless horizontal. The dynamics

```python
def policy(obs: np.ndarray) -> float:
    """Returns an action between -1 and 1.
    obs size is 3.
    """
    theta = np.arctan2(-obs[1], obs[2])
    theta_dot = obs[2]
    if abs(theta) < 0.5:
        action = 5*theta - 0.9*theta_dot
    else:
        action = np.sign(theta_dot)

    return action
```

Fig. 4. Best performing control program generated for *pendulum swing-up* with our technique. The proposed policy applies positive work when the pendulum is not within a certain angular threshold from the upright position. Otherwise, it switches to a linear controller. The control action is normalized within $[-1, 1]$, therefore, in the initial phases, the control takes one of the two limit values, depending on the sign of the angular velocity.

equations, with the angle θ representing the deviation from the upright position, are

$$\ddot{\theta} - \frac{g}{\ell}\sin\theta + b\,\dot{\theta} = u, \tag{7}$$

where $g = 9.81\,\mathrm{m}\cdot\mathrm{s}^{-2}$ is the gravitational acceleration, $\ell = 0.5\,\mathrm{m}$ is the length of the pendulum (massless) rod, $b = 0.4\,\mathrm{s}^{-1}$ is a normalized damping coefficient, and u is a normalized torque input. When simulated, the dynamics equations are discretized through a semi-implicit Euler method and a sampling time of $15\,\mathrm{ms}$ (details in [30]). At each time step t a reward r_t is computed. The overall score of the candidate control policy is the summation of individual rewards, i.e. R defined in eq. 3. For the *pendulum swing-up* task, the stage reward is defined as

$$r_t = \begin{cases} 1 - \frac{|\theta_t|}{\pi} - 0.1\,|u_t| & \text{if } |\theta| > 0.5 \\ 2 - \frac{|\theta_t|}{\pi} - 0.1\,|u_t| & \text{otherwise.} \end{cases} \tag{8}$$

This reward function led to the best results, and was shaped through trial and error. The goal is to reward configurations which are close to the upright position, and penalize the input. However, another common behavior arising is an indefinite uniform spinning after the first transient phase.

To find the high scoring policies for the *pendulum swing-up* task, our framework generated in the order of 10^4 sample programs per individual run. One run in particular led to the control policy shown in Fig. 4. As it is possible to observe, the policy is highly compact and interpretable. Rewritten in mathematical terms, it corresponds to

$$u_t = \begin{cases} 5\,\theta_t - 0.9\,\dot{\theta}_t & \text{if } |\theta_t| < 0.5 \\ \mathrm{sgn}(\dot{\theta}_t) & \text{otherwise,} \end{cases} \tag{9}$$

where $\mathrm{sgn}(\cdot)$ is the sign function. Unpacking the policy, it applies positive work whenever the pendulum is not within a certain angular threshold from the upright position. Otherwise, it is a linear feedback controller. A user could make changes to this policy, such as tuning the linear control gains, or making the torque input a smoother function of the angular velocity, and keep iterating with the LLM in the loop. We also note that a Lyapunov indirect (local) stability analysis could be trivially carried out on eq. 9.

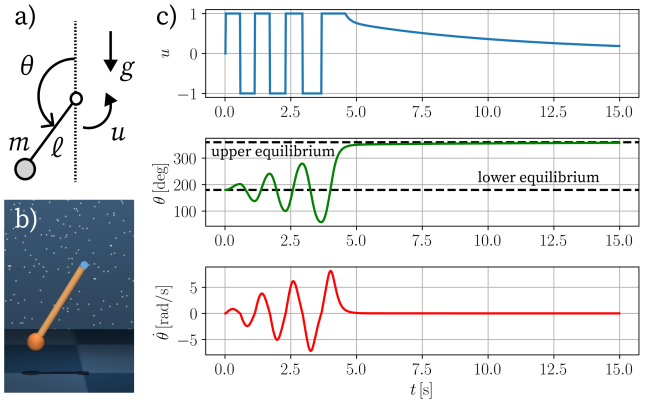

Fig. 5. a) Schematic of the pendulum system and the angle convention. b) Screenshot of a visualization from the simulation environment. c) Example plots of the closed loop evolution for the *swing-up* task. In the top graph, it is possible to observe a bang-bang style control in the first phase, followed by a linear feedback in the second phase.

Plots of the closed-loop dynamics of the *swing-up* task are shown in Fig. 5. It is possible to observe that in the first stage the policy is of bang-bang type, and in a second stage it switches to a linear feedback.

C. Ball in Cup

The *ball in cup* system is composed by a two-dimensional double integrator (the cup) and a ball attached to it through a string (which is a unilateral distance constraint between the two entities). The system is planar. A schematic of the system is shown in Fig. 6a. The task is to find a control policy for the cup (independently actuated along the horizontal and vertical direction) to catch the ball. In this scenario, the policy outputs a reference positions for the cup (x_{ref}, z_{ref}), which are then fed to a lower level linear controller. Therefore, this task is higher dimensional than the *pendulum swing-up*, but does not involve any stabilization of the system. For this task, the reward is defined as

$$r_t = \begin{cases} 1 - \frac{|\theta_t|}{\pi} - 0.1\,v_{ball} & \text{if ball outside cup} \\ 1 & \text{otherwise,} \end{cases} \tag{10}$$

where x_{ball}, z_{ball}, x_{cup}, z_{cup} are the x and z coordinates of ball and cup (at time t, omitted for simplicity), $\theta_t = \mathrm{atan2}(x_{ball} - x_{cup}, z_{ball} - z_{cup})$ is an angle measuring how close the ball is to be vertically aligned with the cup, and $v_{ball} = \sqrt{\dot{x}_{ball}^2 + \dot{z}_{ball}^2}$ is the norm of the ball velocity (also at time t). Also in this case, the reward function was shaped through trial and error.

The number of sampled programs is again in the order of 10^4. The best found policy is shown in Fig. 7, where the observation vector `obs` is 8-dimensional, and organized as $(x_{cup}, z_{cup}, x_{ball}, z_{ball}, \dot{x}_{cup}, \dot{z}_{cup}, \dot{x}_{ball}, \dot{z}_{ball})$. At first, the policy is harder to parse compared to the *pendulum swing-up* case. However, all the reasoning steps are clearly outlined and again, being a Python program, it is easy to modify and customize. To support this claim, we manually proceed to simplify the policy. First, the cup is constrained to the box

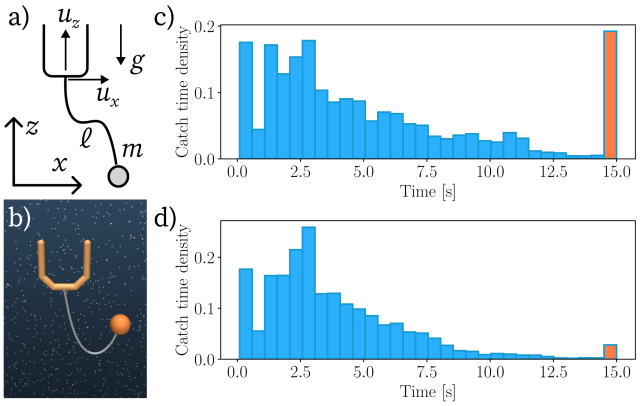

Fig. 6. a) Schematics of the ball and cup system. $\ell = 0.3\,\mathrm{m}$, u_x and u_z are the inputs to the independently actuated linear joints of the cup. b) Screenshot of a visualization from the simulation environment. c) Histogram showing the distribution of ball catching times across 10^4 different episodes using the policy of Fig. 7. At each episode, the cup is initialized at the origin of the task reference frame, and the ball position coordinates are sampled uniformly around the cup. Each episode is terminated when the ball is caught or after 15s. d) Histogram constructed with the same procedure using the user-improved policy. The two experiments use the same random seed. As it is possible to observe, the amount of episodes in which the ball is not caught within $15\,\mathrm{s}$ decreases significantly (orange bin).

$(x, z) \in S = [-0.25, 0.25] \times [-0.25, 0.25]$. Therefore, all the conditional statements checking for values outside this box (i.e. 0.8) are never visited and can be removed. It is possible to also see from the first conditional statements on x_{ref} that the policy commands a narrow motion along x (roughly within $[0.2, 0.25]$). In fact x_{ref} only gets values of either 0.0 if both x_{cup} and z_{cup} are above 0.2, or 1 otherwise (but the motion is constrained to S). Therefore, the policy mostly exploits a vertical stroke to swing the ball. The policy, cleaned of unused logic and with conditional statements grouped more meaningfully, is

$$x_{ref} = \begin{cases} 1 & \text{if } x_{cup} < 0.2 \text{ or } z_{cup} < 0.2 \\ 0 & \text{otherwise,} \end{cases}$$

$$z_{ref} = \begin{cases} 1 & \text{if } z_{ball} < 0.2 \text{ or } \dot{x}_{ball} > 0.5 \text{ or } \dot{z}_{ball} < -0.5 \\ -1 & \text{if } \dot{x}_{cup} > 0.2 \text{ or } \dot{z}_{cup} < -0.2 \\ 0 & \text{otherwise,} \end{cases} \tag{11}$$

Additionally, we can leverage some intuition to even improve this policy. In fact, by visually inspecting the behavior of the system (a video can be found at https://youtu.be/7T7yRGya-q8), we noticed that a common failure is that the ball is not caught because it hits the sides of the cup. An intuitive modification would just be to insert a conditional statement that encourages the cup to lower slightly if the ball is swinging at a height (along z) which is greater than the cup's, to encourage catching. This modification can be easily achieved through, for example, the following statement:

$$\text{if } z_{ball} - z_{cup} > 0.1, \text{ then } z_{ref} \to z_{ref} - 0.1, \tag{12}$$

which is equivalent to adding the following two lines of code as the last two lines of the policy in Fig. 7:

```
def policy(obs: np.ndarray) -> np.ndarray:
    """Returns two actions between -1 and 1.
    obs size is 8."""
    # x_cup, z_cup, x_ball, z_ball = obs[0:4]
    # vx_cup, vz_cup, vx_ball, vz_ball = obs[4:8]

    action = np.zeros((2,)) # x_ref, z_ref
    if obs[0] < 0.2:
      action[0] = 1
    elif obs[0] > 0.8:
      action[0] = -1
    if obs[1] > 0.8:
      action[0] = -1
    elif obs[1] < 0.2:
      action[0] = 1

    if obs[2] > 0.8:
      action[1] = 1
    elif obs[3] < -0.2:
      action[1] = 1
    elif obs[4] > 0.2:
      action[1] = -1
    elif obs[5] < -0.2:
      action[1] = -1

    if obs[6] > 0.5:
      action[1] = 1
    elif obs[7] < -0.5:
      action[1] = 1

    return action
```

Fig. 7. Example policy found for *ball in cup*. The commented parts were added after, to aid the reading of raw code. The output is in the form `action` $= [x_{ref}, z_{ref}]$ providing a reference position, the deviation from which is then consumed by an underlying low-level PD controller.

```
    if obs[3] - obs[1] > 0.1:
        action[1] = action[1] - 0.1
```

We evaluate the original policy of Fig. 7 and the user-improved policy (i.e. the raw policy with the additional condition of eq. 12 at the end) by comparing their ball catching performances across 10^4 different episodes. In particular, histograms showing the distribution of catching times for the raw policy and the user-improved one are shown in Figs. 6c-d, respectively. In each episode, the cup always starts at the origin of the task reference frame, and the ball position is uniformly sampled at random. The episodes are terminated either when the ball is caught, or when 15 seconds have passed. The two experiments were conducted a sufficient number of times to reach with acceptable approximation the convergence of histograms to the underlying distributions. The last bin of the histograms, in orange in Figs. 6c-d, represents the number of episodes that terminated due to maximum time reached (and thus the ball was not caught). This shows that the catching rate improves significantly with the simple intuitive modification made.

IV. DISCUSSION AND CONCLUSIONS

We presented a novel approach to synthesize interpretable control policies. The interpretability is inherently guaranteed by the representation of control policies as programs in standard programming language, and casting the problem as a program synthesis problem, whose solution is achieved with the code generation capability of a Large Language

Model. The policy representation through code allows a user not only to read and understand the control system's logic, but also to have intuitive understanding of the effects of a manual modification to the program. This unlocks properties such as explainability, modularity, verifiability, and paves the way to joint iterative synthesis approaches where a user can actively steer the automatic search and collaborate with an LLM in-the-loop to get different policies based on the design requirements.

The key reason for why this is possible is the shared language and formalism, i.e. Python in our case, between the user and the control system, as opposed to the use of black box models. Interpretability, however, comes with an increase in compute cost related to the absence of gradients to guide the optimization routine. In fact, as an example, a single threaded implementation on the workstation available to the authors takes in the order of 10 hours of wall-clock time to output programs able to successfully execute the proposed tasks.

Another aspect worth discussing is the role of randomness in the LLM generation process. At every generation step, a token is sampled from a distribution over (almost) all possible tokens in the underlying vocabulary. This offers opportunities for further investigations on the robustness of our method, starting for example with a careful tuning of hyperparameters.

Picking the right reward function, as in many reinforcement learning settings, is also crucial for success. Future works can focus on how to make the algorithm more computationally efficient, potentially incorporating gradient-based optimization in-the-loop (such as in [33], for example). This would allow to take advantage of the LLM only as generator of a program skeleton, and unload it from the task of producing the right numerical quantities by tuning them with continuous optimization techniques. Compute availability is also an important aspect of the proposed framework. Implementing a distributed approach and running multiple LLM samplers in parallel allows to speed up the process and achieve high-performing programs in reduced time, as shown in [1].

It is also worth to mention that the amount of domain-specific information provided in the specification can significantly impact the performances (in terms of runtime and sample efficiency) of the control synthesis procedure. LLMs, in fact, are in general sensitive to prompt variations. As a simple example, prompting an LLM with a generic plain instruction like "generate a function", compared to instead providing more details about the task of interest, can make a significant difference. Even if both approaches, down the line, could lead to high-scoring programs, it is typically beneficial to provide context information for sample efficiency.

To conclude, we believe that code provides a compact, extremely expressive, and fully interpretable representation for a control policy. Thanks to these properties, we claim that our approach can reduce the gap between learning-based control systems and verifiable, reliable real world applications.

ACKNOWLEDGMENT

This work was supported by the Hong Kong Center for Logistics Robotics (HKCLR), and the Powley fund of the Mechanical Engineering department of UC Berkeley.

REFERENCES

[1] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi *et al.*, "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, no. 7995, pp. 468–475, 2024.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[3] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[4] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[5] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

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

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

[8] E. Zelikman, Y. Wu, J. Mu, and N. Goodman, "Star: Bootstrapping reasoning with reasoning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 476–15 488, 2022.

[9] P. Haluptzok, M. Bowers, and A. T. Kalai, "Language models can teach themselves to program better," *arXiv preprint arXiv:2207.14502*, 2022.

[10] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," in *Handbook of Evolutionary Machine Learning*. Springer, 2023, pp. 331–366.

[11] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang, "An example of evolutionary computation+ large language model beating human: Design of efficient guided local search," *arXiv preprint arXiv:2401.02051*, 2024.

[12] Z.-S. Hou and Z. Wang, "From model-based control to data-driven control: Survey, classification and perspective," *Information Sciences*, vol. 235, pp. 3–35, 2013.

[13] N. Hovakimyan and C. Cao, *$\mathscr{L}1$ adaptive control theory: Guaranteed robustness with fast adaptation*. SIAM, 2010.

[14] S. Sastry and M. Bodson, *Adaptive control: stability, convergence and robustness*. Courier Corporation, 2011.

[15] D. A. Bristow, M. Tharayil, and A. G. Alleyne, "A survey of iterative learning control," *IEEE control systems magazine*, vol. 26, no. 3, pp. 96–114, 2006.

[16] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017.

[17] R. S. Sutton, "Reinforcement learning: An introduction," *A Bradford Book*, 2018.

[18] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, p. eaau5872, 2019.

[19] I. Radosavovic, T. Xiao, B. Zhang, T. Darrell, J. Malik, and K. Sreenath, "Real-world humanoid locomotion with reinforcement learning," *Science Robotics*, vol. 9, no. 89, p. eadi9579, 2024.

[20] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke *et al.*, "Scalable deep reinforcement learning for vision-based robotic manipulation," in *Conference on robot learning*. PMLR, 2018, pp. 651–673.

[21] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[22] C. Glanois, P. Weng, M. Zimmer, D. Li, T. Yang, J. Hao, and W. Liu, "A survey on interpretable reinforcement learning," *Machine Learning*, pp. 1–44, 2024.

[23] R. Paleja, L. Chen, Y. Niu, A. Silva, Z. Li, S. Zhang, C. Ritchie, S. Choi, K. C. Chang, H. E. Tseng *et al.*, "Interpretable reinforcement learning for robotics and continuous control," *arXiv preprint arXiv:2311.10041*, 2023.

[24] D. Hein, S. Udluft, and T. A. Runkler, "Generating interpretable fuzzy controllers using particle swarm optimization and genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2018, pp. 1268–1275.

[25] D. Hein, S. Limmer, and T. A. Runkler, "Interpretable control by reinforcement learning," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 8082–8089, 2020.

[26] H. K. Chu and M. Hayashibe, "Discovering interpretable dynamics by sparsity promotion on energy and the lagrangian," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 2154–2160, 2020.

[27] C. Rudin, C. Chen, Z. Chen, H. Huang, L. Semenova, and C. Zhong, "Interpretable machine learning: Fundamental principles and 10 grand challenges," *Statistic Surveys*, vol. 16, pp. 1–85, 2022.

[28] D. Bertsekas, *Reinforcement learning and optimal control*. Athena Scientific, 2019, vol. 1.

[29] E. Cantú-Paz *et al.*, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, no. 2, pp. 141–171, 1998.

[30] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[31] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq *et al.*, "Deepmind control suite," *arXiv preprint arXiv:1801.00690*, 2018.

[32] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[33] P. Ma, T.-H. Wang, M. Guo, Z. Sun, J. B. Tenenbaum, D. Rus, C. Gan, and W. Matusik, "Llm and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery," *arXiv preprint arXiv:2405.09783*, 2024.