

Robot Graph Grammars: Towards Custom Robots for Every Task

by

Allan Zhao

B.S. Electrical Engineering and Computer Science, UC Berkeley, 2017

M.S. Electrical Engineering and Computer Science, UC Berkeley, 2018

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

© 2024 Allan Zhao. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by:	Allan Zhao Department of Electrical Engineering and Computer Science January 26, 2024
Certified by:	Wojciech Matusik Professor of Electrical Engineering and Computer Science Thesis Supervisor
Certified by:	Daniela Rus Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Robot Graph Grammars: Towards Custom Robots for Every Task

by

Allan Zhao

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

As robots find broader applications outside factory floors, they face an increasing number of challenges. For example, they must accommodate rugged terrain, limited battery capacity, and complex dynamics. Existing robots are largely designed by hand to meet a given set of specifications. While highly capable, these manually-designed robots tend to leave performance on the table. These difficulties have motivated research into automatic robot design tools. Early tools were often limited in the range of robot topologies they can explore, however. Current graph-based robot representations can expand the space of possible designs, but it is not always clear how the resulting designs can be fabricated.

To enable efficient design exploration and ensure fabricability, we propose graph grammars as a universal robot design representation. Graph grammars use rewriting rules to incrementally add complexity or select among distinct design alternatives. Because only fabricable components and connections are expressed in the grammar, the generated robot topologies are valid by construction. Through recursion and branching, graph grammars can also generate a large variety of possible designs. To tackle this expansive search space, we propose a specialized learning-based search algorithm called Graph Heuristic Search (GHS). GHS focuses limited simulation resources on the most promising designs. We compare GHS to random search and Monte-Carlo tree search baselines, showing that GHS finds higher-performing designs in less wall-clock time. We combine graph grammars and GHS with other techniques such as differentiable simulation to efficiently optimize multiple types of mobile robots. In doing so, we show that graph grammars are a principled yet general design representation for robot co-design. Their efficiency and versatility brings us one step closer to the dream of generating custom robots for every task.

Thesis supervisor: Wojciech Matusik

Title: Professor of Electrical Engineering and Computer Science

Thesis supervisor: Daniela Rus

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express the deepest appreciation to my advisors Wojciech Matusik and Daniela Rus, who have given me endless opportunities, direction, and encouragement throughout my Ph.D. Special thanks to Andy Spielberg, who has been a great friend and collaborator since my first year at MIT. In addition, I am grateful to Tao Du, Jie Xu, Josie Hughes, Mina Konaković Luković, and Yuri Ivanov for their patient mentorship and help in becoming a better researcher. I would also like to thank my collaborators Juan Salazar, Wei Wang, Pingchuan Ma, and Peter Chen for their assistance and invaluable feedback.

My path towards a Ph.D. started at UC Berkeley, where I had the pleasure of working with Sam Emaminejad and Ali Javey as an undergraduate researcher and Pieter Abbeel during my master's. I want to thank them for their mentorship and guidance in beginning my research career.

I would also like to acknowledge funding from the Intelligence Advanced Research Projects Activity (IARPA), National Science Foundation (NSF), Defense Advanced Research Projects Agency (DARPA), and Amazon Robotics. Without their financial support, the work in this thesis would not have been possible.

Finally, I would like to thank my parents for their helpful advice and unwavering support.

* * *

This dissertation incorporates work that was previously published in the following papers:

- A. Zhao, J. Xu, M. Konaković Luković, *et al.*, “Robogrammar: Graph grammar for terrain-optimized robot design,” *ACM Transactions on Graphics (TOG)*, vol. 39, no. 6, pp. 1–16, 2020
- A. Zhao, J. Xu, J. Salazar, *et al.*, “Graph grammar-based automatic design for heterogeneous fleets of underwater robots,” in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 3143–3149
- A. Zhao, T. Du, J. Xu, *et al.*, “Automatic co-design of aerial robots using a graph grammar,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2022*, IEEE, 2022

Contents

Title Page	1
Abstract	3
Acknowledgments	5
List of Figures	11
List of Tables	15
1 Introduction	17
1.1 Vision	18
1.2 Overview	20
2 Background and Related Work	22
2.1 Design Representation	23
2.1.1 Shape Representation	23
2.1.2 Control Representation	28
2.2 Simulation	30
2.2.1 Rigid Body Dynamics	31
2.2.2 Articulated Body Dynamics	33
2.3 Optimization	34
2.3.1 Gradient-Based Optimization	34
2.3.2 Graph Heuristic Search	35
2.4 Summary	37
3 Land Robots	38
3.1 Introduction	38
3.2 Related Work	40
3.2.1 Generative Design for Robotics	40
3.2.2 Graph Grammars	43
3.2.3 Grammar-based Procedural Modeling	44
3.2.4 Control Methods and Approaches	44
3.3 System Overview	45
3.4 Robot Grammar	46
3.4.1 Robot Representation	47

3.4.2	Grammar Definitions	47
3.4.3	Production Rules and Robot Components	50
3.5	Simulation & Control	52
3.5.1	Simulation	54
3.5.2	Model Predictive Control	54
3.6	Search & Optimization	56
3.6.1	Graph Heuristic Search	56
3.6.2	Monte Carlo Tree Search	61
3.7	Problem Specification	64
3.7.1	Reward Function	64
3.7.2	Terrains	65
3.8	Implementation	65
3.8.1	Hyperparameters	66
3.8.2	Experiment Setup & Computational Time	67
3.9	Results	67
3.9.1	Terrain Driven Optimization	68
3.9.2	Pareto Analysis	69
3.9.3	Gait Patterns	71
3.9.4	Efficiency of Graph Heuristic Search	72
3.9.5	Convergence of Graph Heuristic Search	73
3.9.6	Design Space Bias	73
3.10	Limitations and Future Work	74
3.11	Conclusion	75
4	Underwater Robots	77
4.1	Introduction	77
4.2	Design Space	79
4.3	Simulation	79
4.3.1	Dynamic Model	79
4.3.2	Trim State and Control	84
4.4	Optimization	84
4.4.1	Task Specification	84
4.4.2	Precomputation of Individual Vehicles	85
4.4.3	Fleet Optimization	87
4.5	Results	90
4.5.1	Single-Vehicle Optimization	90
4.5.2	Fleet Optimization	92
4.5.3	Implementation Details	94
4.6	Conclusion and Future Work	94
5	Aerial Robots	96
5.1	Introduction	96
5.2	System Overview	98
5.3	Shape Design	99
5.3.1	Continuous Parametrization	99

5.3.2	Graph Grammar	99
5.4	Controller Design	101
5.4.1	Dynamic Model	103
5.4.2	LQR Control	105
5.5	Simulation	105
5.6	Optimization	106
5.7	Experiments	108
5.7.1	Implementation Details	108
5.7.2	Task Specification	108
5.7.3	Our Results	110
5.7.4	Discussion	112
5.8	Conclusions and Future Work	116
6	Lessons and Future Work	117
6.1	Graph Grammar Learning	117
6.2	Sim-to-Real	118
6.3	Sensing	120
7	Conclusion	121
	References	123

List of Figures

2.1	An example of a robot graph grammar used to design land-based robots. S, T, B, E, T, H are nonterminal symbols.	27
3.1	The input to our system is a set of base robot components, such as links, joints, and end structures, and at least one terrain, such as stepped terrain or terrain with wall obstacles. <i>RoboGrammar</i> provides a recursive graph grammar to efficiently generate hundreds of thousands of robot structures built with the given components. We then use Graph Heuristic Search coupled with model predictive control (MPC) to facilitate exploration of the large design space, and identify high performing examples for a given terrain. Our approach enables co-optimization of both robot structures and controllers.	45
3.2	An example of a kinematic tree (top) with the corresponding robot graph (bottom). To enforce symmetry in leg pairs, after adding nodes for connectors on both sides of the body, both legs of one pair are defined in one branch of the graph.	48
3.3	A list of structural rules of our robot grammar. Here $S, H, Y, B, T, U, E, J, L \in \mathbf{N}$ are non-terminal symbols. Rule r_1 initializes the body structure, while r_2 can be used to extend the body. Note that each body segment U can have at most one pair of limbs attached to it. Rule r_3 enforces symmetry of the limb pairs, and rule r_4 allows body segments without limbs. Rule r_5 serves for extending the limbs, and r_6 and r_7 for adding back and front limbs. . . .	49
3.4	Component-based rules of the robot grammar. Initial-pose angle θ_i and rotational range angle θ_r are attributes of joints. Adding, removing, or editing any link or joint component is straightforward. Note that we do not use a physical representation for the head, tail, hands, feet, and other types of end components for simplicity. However, the grammar can easily incorporate any new component representing an end structure.	51
3.5	A derivation sequence for a Simple Walker robot generated with our grammar. Derivation begins with the start symbol S , then creates the body and extends it with rules r_1, r_2 respectively. Legs are added on both body segments with rule r_3 applied twice. Both pairs of legs are extended, adding additional sets of joints and links, with r_5 . For the Simple Walker, end structures are not used, hence we remove them with r_{21}, r_{22} , and r_{23} . Finally, terminal components are added for each segment of the robot, following the rules from Figure 3.4. . .	52

3.6	Designs generated with <i>RoboGrammar</i> that appear during the search process for various terrains. Further optimization is necessary to identify the best performing ones for a given task.	53
3.7	Selection of best-performing designs generated with the grammar, and optimized with Graph Heuristic Search and MPC for ridged, flat, frozen lake, and gapped terrain respectively.	68
3.8	Pareto optimal designs for pairs of terrains. Designs offering a variety of trade-offs are discovered using only 20,000 iterations of random search, demonstrating the versatile yet compact search space of our grammar.	70
3.9	Time-lapse showing movement of the high performing robots on upward stepped terrain and terrain with wall obstacles. Each front leg of the first robot consists of 3 elbow joints and a final knee joint forming a hook-like ending to facilitate climbing high steps. The second robot has a long body with many rotational joints and front legs designed for turning stability.	71
3.10	Cumulative maximum reward versus iteration for Graph Heuristic Search, Monte Carlo tree search, and random search on four different terrains. Each solid line is the mean of three different seeds, with the error band representing the range. Graph Heuristic Search consistently outperforms the baselines. . .	72
3.11	Training loss, prediction error, and cumulative maximum reward versus iteration for Graph Heuristic Search on multiple grammars. Robots are optimized for flat terrain. Prediction error is the absolute difference between predicted reward and evaluated reward, and is averaged over 100 iterations. Each solid line is the mean of at least three different seeds, with the error band representing the range. Graph Heuristic Search consistently converges in all three criteria.	74
4.1	The rules comprising our AUV grammar. Symbols are either in uppercase (representing abstract subgraphs) or in lowercase (representing concrete components).	80
4.2	Sequential growth of a propeller-driven AUV from applying 7 of our grammar rules.	81
4.3	The best designs found by our single-vehicle optimization pipeline for each combination of mass limit and depth rating. The survey rate (in m^2/s) is indicated next to each design.	90
4.4	The best fleet found by our fleet optimization algorithm using the precomputed design library, along with the number of sorties performed by each vehicle of a given type. The fleet consists of 4 different designs, with the number of instances of each design indicated.	92

5.1	An overview of our computational UAV design pipeline. (a): Our pipeline starts with a set of discrete and continuous parameters. (b): The design parameters determine the UAV shape (Sec. 5.3) and controller (Sec. 5.4). (c): We evaluate our design in a differentiable simulator for given tasks (Sec. 5.5). (d): The evaluation is then fed into our optimization algorithm (Sec. 5.6), which runs both discrete search and continuous optimization to update the design parameters.	98
5.2	Illustration of a wing segment immersed in a deformation cage (left) and its new shape after the handles of the cage are moved (right).	100
5.3	Rules of our UAV grammar. S, B, L, C, Y are non-terminal symbols. Rule r_1 initializes the body structure, while $r_2 \sim r_4$ can be used to extend the body. Note that each body segment B can have at most one pair of wings attached to it. Rule r_5 connects the body with the wing. Rules r_6 and r_7 produce clockwise and counterclockwise rotors. Rule r_8 and r_9 extend rotor joints with one or two additional rotor joints. Rule r_{10} extends the body. . .	101
5.4	The derivation sequence for a quadrotor using our graph grammar. This design uses 9 rules in total. We visualize the partial design graphs in between rules and the final quadrotor design after applying all rules. The non-terminal and terminal nodes in the graphs are indicated by circles and squares, respectively.	102
5.5	Optimized UAV shapes discovered by our computational pipeline using random search (top row) and GHS (middle row) for each of the four tasks (left to right). For each task, the bottom row shows the better UAV from the two rows above (measured by their final loss in Table 5.1 and highlighted in green) with its deformation cages reset to their default positions.	111
5.6	The optimization progress plots of our two search strategies: random search and GHS. The horizontal axis is the number of designs sampled by the corresponding discrete search algorithm and optimized by SQP, and the vertical axis is the objective of the best design as the optimization proceeds. We generate the curves from three individual runs for each algorithm on each task.	111
5.7	The five classic designs we use as baselines in Table 5.1. Top: (a) X-quadrotor, (b) Plus-quadrotor, (c) Octacopter. Bottom: (d) Fixed-wing plane. (e) Double fixed-wing plane.	113

List of Tables

3.1	Graph Heuristic Search hyperparameter values	66
3.2	Simulation and MPC hyperparameter values	66
3.3	Average number of derivation steps in designs evaluated by each algorithm. Results are for flat terrain.	74
4.1	Simulation parameters for the surveying mission	85
4.2	Mission times using a single vehicle type	93
5.1	Performance of classic UAVs and the optimized UAVs discovered by our method on each task. Each column reports the loss values on a given task with the best number highlighted in bold (lower is better). For the fixed-wing UAV in our TransitionTask, we report N/A because no trim states are found after trying 10 random seeds in the continuous optimization problem.	113
5.2	Performances of the best UAVs discovered by our method with and without optimization of their deformation cages. For each task, we choose the better of the two designs found by GHS and random search. Each column reports the value of the loss function in a given task with the better case highlighted in bold (lower is better). Allowing optimization of cage parameters results in substantially better performance.	115
5.3	The position tracking error L_{pos} of the optimized UAVs discovered by our method when subjected to simulated wind conditions. The numbers are rescaled so that $L_{\text{pos}} = 1$ is the position tracking error in a clean simula- tion environment (wind speed = 0).	115

Chapter 1

Introduction

The number and complexity of applications for robots is steadily increasing. Many of these use cases, such as surveying and transportation, require mobile robots that can traverse a variety of outdoor environments while conserving battery power. Each type of environment also comes with its own unique design challenges. Land-based robots must contend with rugged terrain, underwater robots must balance their centers of mass and buoyancy, and flying robots must deal with strict weight penalties and high sensitivity to disturbances. At the same time, robot designers try to optimize for multiple different metrics such as speed, range, efficiency, and cost. This combination of complex dynamics and interconnected design goals is difficult to reason about. Despite these challenges, existing mobile robots are largely designed by hand. The resulting robots are highly capable, but are the product of many tedious design and testing cycles. Since the space of possible solutions is simply too large, designers are often forced to accept suboptimal solutions.

The difficulty of manual robot design has other knock-on effects as well. Robot designers are often afraid to break the mold, and rely on tried-and-true solutions. The design of walking robots, for example, has largely converged on 12-degree-of-freedom quadrupeds. While four-legged animals are common, nature exhibits a limitless variety of other forms. These include arthropods, which account for more than eighty percent of all known animal species. It is

certainly possible that one of these other forms are more suited to a particular task. With greater ability to explore other robot designs, more diverse robot morphologies could be put into practice.

Automating robot design therefore has the potential to revolutionize robotics. Computers can consider all aspects of the design process simultaneously, leading to more optimal robots than those designed by humans alone. By reducing the time needed to design a robot, designers can more quickly create robots for new use cases. Finally, reducing the workload on robot designers would allow for an explosion in robot diversity. Instead of being occupied with the low-level details of robot design and sticking to known solutions, designers would now be free to explore more interesting robot morphologies. Creativity and aesthetic preferences could take on a greater role in the robot design process.

Tools for automatic robot design already exist, but are limited to optimizing only one robot type. As a result, designers must learn a new tool for every type of robot they want to design. Some of this domain specificity can be attributed to differences in robot dynamics. Walking robot simulations need to consider collision, for example, but do not need to model aerodynamics to the same degree as for flying robots. Similarly, walking robots will likely require more sophisticated controllers than flying ones. Unifying the remainder of the design process might be possible, however. Creating a general-purpose robot design framework with interchangeable modules could flatten the learning curve for designers and allow advances to be shared across the field. This in turn could drive adoption, bringing us closer to our vision of widespread, automated robot design.

1.1 Vision

For automatic robot design to truly catch on, we need to lower the barrier to entry and minimize friction. As such, the ideal robot design workflow should be general and responsive.

Ideally, designers would only need to learn a single, modular workflow that applies to any

form of robot. Current tools can only design one type of robot, leading to a proliferation of different workflows. Components which are necessarily problem-specific, such as simulation, should instead be modular and easily replaceable. This raises the question of interoperability. Interfaces between different tool subcomponents (such as simulation and optimization) are currently ad hoc, complicating code reuse across different projects. By defining a common robot representation that is applicable across multiple domains, unifying these separate efforts becomes possible.

In order to realize a time savings over manual design, automated tools should also be responsive. Designers should be able to obtain optimized robots within a work day, and ideally even faster. This requirement rules out many traditional optimization approaches such as genetic algorithms. Faster optimization algorithms also enable more interactive exploration, which is key to unlocking designer creativity. Modifying the problem setting and solution preferences during the design process would become practical, allowing designers to examine “what-if” scenarios. To make these possibilities a reality, we need faster optimization algorithms that are specially adapted to automated robot design.

This thesis makes contributions towards both of these goals. By using a technique called graph grammars to design land-based, underwater, and aerial robots, we show that they can serve as a general robot design representation for rigid robots. Additionally, we show how these graph grammars can generate a limitless variety of designs while inherently enforcing validity. Finally, we demonstrate a learning-based search strategy called Graph Heuristic Search that can efficiently search a graph grammar for optimal designs. Graph Heuristic Search finds better-performing designs in significantly less wall clock time compared to baseline methods, improving the responsiveness of our design framework.

This dissertation’s main thesis can therefore be summarized in the following sentence: *Graph grammars combined with a learning-based search strategy enable general and efficient automatic design of rigid robots.*

In support of this thesis, we make several contributions across multiple robotic domains:

- A recursive graph grammar that generates a wide range of inherently fabricable land-based robot forms, which can be built in their simulated configurations.
- A Graph Heuristic Search method for efficiently searching the design space described with the grammar. This method is benchmarked against alternatives including Monte Carlo tree search and random search.
- A demonstration of terrain-driven optimization for land robots using MPC-based stochastic evaluation of each proposed design. We show the variety of innovative robot designs that are obtained across six different terrains. In addition, our approach identifies a number of high-performing robots for a single terrain or combination of terrains.
- A novel graph grammar for underwater robots that captures a diverse space of topologies.
- A formulation of heterogeneous robot fleet design as a modified knapsack problem and an efficient, exact algorithm to solve it.
- A novel flying robot co-design space enabling both discrete and continuous variations in shapes.
- An effective, hybrid search algorithm that co-optimizes the discrete and continuous design parameters, together with a differentiable flying robot simulator we developed.
- An evaluation of our computational design pipeline on dynamic flight tasks to demonstrate its value in both automatically re-discovering canonical flying robot designs and revealing new higher-performance designs.

1.2 Overview

The next chapter provides a more detailed explanation of graph grammars and Graph Heuristic Search. It also breaks down the various subcomponents of an automatic robot design

framework. Finally, it places our methods in the context of previous work. The chapter shows how graph grammars and Graph Heuristic Search emerge as a promising design representation and search strategy, respectively, that work seamlessly with other powerful techniques.

Later on, Chapters 3, 4, and 5 show how graph grammars and Graph Heuristic Search enable efficient and broad design exploration across multiple domains of robotics. Combined with the techniques mentioned earlier, they can find higher-performing designs in less wall-clock time than baseline methods. In this way, they demonstrate their utility in a general, efficient robot design framework.

Finally, Chapters 6 and 7 conclude with a discussion of lessons learned, avenues for future work, and closing remarks.

Chapter 2

Background and Related Work

When creating a robot, designers often decide on its hardware configuration before beginning to write its software. This makes sense for at least two reasons: software is more flexible, and it can only use whichever sensors and actuation are available. This sequential design process is not always ideal. Robots are often limited by compute capabilities, and a different selection or placement of components can greatly simplify the control software. Given how much the performance of a robot’s controller depends on its shape and vice versa, it is natural to wonder whether they should be designed at the same time. This idea of simultaneously optimizing both the hardware and software design of a robot is often formally referred to as *robot co-design*.

Robot co-design is not a new concept and is often done manually. For example, many design decisions can be justified from first principles or through simple analysis. This bottom-up approach is not always possible, however. Robots with complex dynamics, such as walking or flying robots, are much more difficult to reason about. The performance of these robots is intrinsically linked to both their morphologies and the controls used to accomplish their task. This complex interplay between shape and control was first addressed by Karl Sims in a series of seminal works[4], [5]. While they were presented in a computer graphics context, all of the building blocks of computational robot co-design are present: design

representation, simulation, and optimization. Previous work in automated robot design provided analytical solutions to particular robot design subproblems [6]. In contrast, Sims’ work proposed a general, computational approach that could apply to a wide variety of scenarios. The following sections describe advances in computational robot co-design and how they lay a foundation for this thesis.

2.1 Design Representation

All computational co-design systems require some way to represent individual designs. An ideal design representation is both expressive and compact. Expressivity allows exploring the full range of design possibilities without excluding any potentially high-performing designs. Compactness reduces both the memory required to store candidate designs and the computation needed to fully explore the space of possible designs. These considerations are largely antithetical, and finding a design representation that balances the two is essential for an effective co-design framework.

For robots, design representations must encompass both shape and control. The two domains are very different, and many robot co-design frameworks choose to represent them separately. In this scenario, the design representation is a combination of a *shape representation* and a *controller representation*. Shape representation is the enabling factor in many robot co-design frameworks, while controller representation tends to take a supporting role. For this reason, the remainder of the section will mainly focus on robot shape representations.

2.1.1 Shape Representation

In the simplest case, a robot co-design problem might only consider a fixed arrangement of components. Each component has its own set of continuous parameters. By concatenating the parameters of all components, the shape design of the overall robot can be expressed as an m -dimensional vector $\phi \in \mathcal{R}^m$. Even in this limited setting, optimizing shape and

controller parameters together can result in greater task performance compared to optimizing the controller alone [7]–[10]. A similar approach can be applied to character animation [11], where the topology of the robot is selected through an interactive interface instead of being determined by source code. In either case, the co-design system is limited to optimizing only a single, pre-defined topology. Performance of the final optimized design is limited by the user’s ability to anticipate the optimal discrete robot topology.

Soft robots, which may not consist of individual components, are more amenable to a continuous-only representation. One popular approach is to discretize material properties such as mass density and Young’s modulus on a volume grid. While simple, this approach suffers from extremely high dimensionality—in other words, the representation is not compact. Dimensionality reduction techniques such as Gaussian mixtures [12] and neural network-based latent representations [13] can provide practical workarounds. Not all real world applications can make use of soft robots, however, and even soft robots can benefit from component selection. We still need a shape representation that can express discrete design choices in a compact and expressive manner.

These discrete choices take on many forms. They can be as simple as choosing a specific model of motor or the rotation direction of a drone propeller. Other discrete choices are *topological* and much more complex. For example, a robot could have a varying number of legs. The number of continuous design variables would then vary as well, adding another layer of complexity. This example could be extended further by allowing a different number of segments per leg. The resulting recursive space of robot topologies becomes very difficult to express using traditional, fixed-size shape representations.

In this general case, graphs consisting of nodes and edges have become the most popular choice of shape representation. Nodes typically correspond to robot components and edges define mechanical connections between them. Continuous parameters, such as dimensions and relative positions of components, are expressed as attributes of the nodes and edges. This graph-based representation is highly expressive. It can express not only continuous variation

of components, but also different numbers of substructures such as legs and propellers.

Graph-based Mechanism Design

Applying graph theory to mechanism design is a decades-old concept. Graph algorithms can determine whether two mechanisms are equivalent [14] or efficiently enumerate all mechanisms of a given type [15]. Enumeration is possible for small-scale design problems, but quickly becomes infeasible for problems of greater complexity. Later work in the field of graph-based mechanism design remains limited in scale, and mainly extends the approach to other classes of mechanisms [16]. Early computational methods [17] automate the analysis and enumeration of mechanism graphs, but are still not capable of complex mechanism synthesis.

Any further breakthroughs would have to wait for advances in both discrete optimization and computational power. In the early 1990s, genetic algorithms began to demonstrate success in large-scale optimization problems. Optimizing mechanical structures such as trusses with thousands of design variables became practical [18]. At the same time, computing hardware improved significantly. Karl Sims' 1994 work, *Evolving Virtual Creatures*, pushed the limits of what was newly possible [4]. Using a combination of genetic algorithms, supercomputing resources, and graph-based shape representations, Sims [4] demonstrated large-scale mechanism synthesis for the first time. Virtual creatures, represented as graphs, are simulated in a dynamic environment. Graphs representing the best-performing creatures undergo crossover and mutation operations to generate new candidate designs. The process repeats, yielding creatures with arm and leg-like appendages capable of swimming, walking, jumping, and following light sources.

While capable of impressive results, a pure graph-based shape representation is not always suitable. Graph modifications are limited to simple operations such as adding or removing nodes and edges, or copying subgraphs in the case of genetic crossover. These operations may not always result in a valid design. Many mechanical components rely on others to

function, and including just one does not make sense. This is not a concern in the case of virtual creatures, which can be constructed from abstract modules. Graphs containing realistic components, however, should be modified using operations that preserve validity.

As demonstrated by *Evolving Virtual Creatures*, graph-based shape representations can be very expressive. They are not necessarily compact, however, since they may admit invalid designs. An additional layer of structure should be imposed in order to obtain a representation that enables broad exploration while enforcing correctness.

Graph Grammars

Graph grammars offer a potential solution. A graph grammar consists of a set of rewriting rules that replace certain patterns of nodes and edges in a graph with other structures. The overall idea is to make multiple modifications simultaneously so that the correctness of the design is maintained at all times. If the replacement structures contain only valid arrangements of components, the overall design remains valid. Another key aspect of the graph grammar-based approach is the use of *nonterminals*. Nonterminals are special nodes and edges that represent abstract concepts. They serve as placeholders and do not correspond to actual components. Rewriting rules must be applied until a design graph no longer contains any nonterminals, ensuring that the final graph contains only valid components. With suitable use of recursion and multiple choices of rules, graph grammars can generate an unlimited variety of unique and valid robot designs. In this way, graph grammars meet both of our criteria for an ideal shape representation.

Much like graph representations, graph grammars have already proven useful for mechanism design. Schmidt and Cagan [19] generate wheeled carts using a graph grammar with multiple levels of abstraction. Nonterminals in the grammar correspond to concepts such as “create rolling,” which can be implemented in multiple ways depending on the next rule chosen. The final graphs generated by the grammar are also functionally valid. They map to physical components from a Meccano® construction set, demonstrating fabricability. Their

Grammar structural rules

Body structure

$$r_1: \quad (S) \Rightarrow (H) \leftarrow (B) \leftarrow (T)$$

$$r_2: \quad (T) \Rightarrow (Y) \leftarrow (B) \leftarrow (T)$$

Adding appendage to body

$$r_3: \quad (B) \Rightarrow \begin{array}{c} (U) \\ \swarrow \quad \searrow \\ (C) \quad (C) \\ \swarrow \quad \searrow \\ (M) \\ \downarrow \\ (E) \end{array}$$

$$r_4: \quad (B) \Rightarrow (U)$$

Appendages

$$r_5: \quad (E) \Rightarrow (J) \rightarrow (L) \rightarrow (E)$$

$$r_6: \quad (T) \Rightarrow (C) \rightarrow (M) \rightarrow (E)$$

$$r_7: \quad (H) \Rightarrow (E) \leftarrow (M) \leftarrow (C)$$

Legend

S	start symbol
H	head part
Y	body joint
B	body part
T	tail part
U	body link
C	connector
M	mount part
E	limb end
J	limb joint
L	limb link

Figure 2.1: An example of a robot graph grammar used to design land-based robots. S, T, B, E, T, H are nonterminal symbols.

method does not address robotic applications, however, since the carts are passive mechanisms. The design process only optimizes the carts' weights and bed areas and does not consider control.

Similar techniques can be used to optimize passive dynamic brachiating mechanisms [20],

showing that graph grammars are applicable to multiple domains. The objective function prioritizes designs that can swing from point to point in a cyclic motion while minimizing swept area. Rules in the graph grammar are crafted to ensure designs remain symmetrical at all times, ensuring the cyclic boundary condition holds. This serves as another example of how graph grammars can enforce validity. Like in previous work, however, the optimized mechanisms are still passive and uncontrolled.

These existing works clearly demonstrate that graph grammar-based shape representations can generate a diverse variety of designs while respecting validity constraints. Moreover, they are applicable to many different domains of mechanism design. In order to accomplish robot co-design, however, control must also be addressed.

2.1.2 Control Representation

The distinguishing factor between a mechanism and a robot is that the latter reacts to its surroundings. Mechanisms start in some initial configuration, and passively evolve over time according to the laws of motion. On the other hand, robots are active systems. They take in observations from their environment and then perform actions that bring them closer to some desired goal. Those actions in turn modify the environment, resulting in a closed feedback loop. The mapping from observations to actions can be referred to as a *controller*. Like shapes, robot controllers also need a representation.

Different control representations may be suitable depending on our goals. Computational graphs [4], [21], for example, allow for the control representation to closely mirror the shape representation. One significant downside of computational graphs is their discrete nature, preventing efficient gradient-based optimization. Neural networks, which have emerged as the standard controller design representation, are largely the opposite. Infamous for being uninterpretable “black boxes,” they are nonetheless easy to optimize using backpropagation and gradient descent. They are also highly expressive and scalable to problems of varying complexity. Neural networks come with a major drawback, however: their parameter count.

The number of parameters in a neural network controller can easily dwarf the number of shape parameters by orders of magnitude. Even with gradient-based optimization, robot co-design with such a controller can be far slower than with simpler controllers.

Given these concerns with computational efficiency and our focus on shape design, it makes sense to simplify the controller representation. As such, the work in this thesis uses simple, low parameter count controller representations such as trim states and linear quadratic regulators (LQRs).

Trim states are sets of control inputs that keep a vehicle moving at a fixed speed and orientation. In this thesis, they are used for underwater and aerial robots (Chapters 4, 5). We optimize each vehicle for multiple trim states, each representing a common phase of flight. By only optimizing for a few representative motions, we avoid having to train a full controller. We can still be reasonably confident such a controller exists because we check a variety of different trim states.

In the case of aerial robots, which need to be robust to wind gusts, we extend the trim state-based control representation with a linear quadratic regulator. An LQR is a form of feedback controller that uses linearized dynamics and a quadratic loss function. The loss consists of a weighted sum of squared control inputs u and squared errors in state variables x relative to the trim state $\bar{x}$:

$$J = (x - \bar{x})^T Q (x - \bar{x}) + u^T R u. \quad (2.1)$$

The weighting matrices Q and R often require significant tuning in order to achieve stable flight [22]. They also have a significant impact on performance metrics such as power efficiency. For these reasons, we treat the diagonals of Q and R as controller design parameters. The resulting control representation, consisting of trim states and LQR weights, strikes a balance between ease of optimization and ensuring design validity.

An alternative approach, which we use to design walking robots (Section 3), is to have

no controller at all. Advances in online optimization techniques such as model predictive path integral (MPPI) control [23], [24] make it possible to synthesize robot control inputs in real time without an explicit controller. For robots with complex dynamics that are not amenable to trim states and LQR, this can be an appealing option.

Readers may be wondering where reinforcement learning (RL) fits into this picture. RL is capable of training controllers that can generalize to a wide variety of situations. This generality is often unnecessary in the context of robot co-design, however, and comes at the cost of significant training time. Using simpler, less general controller representations allow us to narrow down the space of possible robot shapes more quickly. Once we identify a suitable robot shape, we can then train a general controller for it using reinforcement learning if desired.

Overall, there is no single answer to the question of controller representation. Different representations may be appropriate depending on the type of robot we are trying to design. Since we are focusing on shape design, we choose to simplify the controller representation wherever possible. This means trim states and LQR in the case of underwater and aerial robots, and online optimization in the case of walking robots.

2.2 Simulation

In order to design a robot that will perform well in the real world, we need to emulate its dynamics as closely as possible. Knowing the dynamics lets us compute performance metrics such as speed and power consumption. We will refer to the techniques used to predict a robot’s performance from its design as *simulation*.

Simulation is therefore an integral part of any robot co-design method. The specific type of simulation needed will vary depending on robot type and desired level of accuracy. Most mobile robots, such as the ones optimized in this thesis, are *rigid robots* composed of rigid links connected by movable joints. These robots can be simulated using *rigid body dynamics*

where each rigid link’s state is described using six degrees of freedom.

2.2.1 Rigid Body Dynamics

In rigid body dynamics, the pose of each robot link can be described in terms of its three-dimensional position and rotation. Like other aspects of robot co-design, however, the specific internal representation matters. While position is almost universally described using Cartesian coordinates, rotation representations vary widely. Different representations are appropriate depending on the context.

Rotation Representations

Euler angles remain a popular rotation representation in aerospace applications. They directly correspond to the control axes of fixed-wing aircraft, making them highly intuitive. They also require only three variables, the minimum possible. While it is possible to use them in a robot co-design framework [25], they suffer from a number of disadvantages. For one, certain orientations correspond to singularities. Even when singularities are avoided, the gradient landscape resulting from a Euler angle parametrization is highly non-smooth. The derivatives consist of multiple nonlinear factors multiplied together. This can interfere with techniques such as linearization-based control and gradient-based optimization. Controlling the Euler angle orientation of vehicles using linear quadratic regulators (LQRs) can be prohibitively difficult [25].

While Euler angles are intuitive and compact, other rotation representations may be more appropriate for robot co-design. The work presented in this thesis is based on quaternions, which offer a reasonable compromise between compactness and practicality. A rotation of angle θ around the axis (x, y, z) can be written as the quaternion

$$(w, i, j, k) = \left(\cos \frac{\theta}{2}, x \sin \frac{\theta}{2}, y \sin \frac{\theta}{2}, z \sin \frac{\theta}{2} \right). \quad (2.2)$$

Although quaternions are not directly interpretable, they offer a number of computational advantages. For example, composing quaternions requires fewer floating point operations compared to rotation matrices or Euler angles. This computational savings becomes apparent in simulations with a large number of time steps. The main advantage of quaternions for robot co-design, however, is in computing derivatives. Unlike Euler angle composition and vector rotation, which produce derivatives with multiple trigonometric factors, quaternion operations result in much simpler derivatives composed of only bilinear terms. These derivatives can be more amenable to gradient-based optimization methods, especially when performing backpropagation through time (BPTT). Quaternions only require one additional variable, making them much more compact than rotation matrices.

Equations of Motion

Using quaternions, we can describe the state of a rigid body as a 13-dimensional vector

$$\mathbf{s} = (\mathbf{x}, \mathbf{q}, \mathbf{v}_B, \omega) \quad (2.3)$$

where $\mathbf{x}$ is the rigid body's Cartesian position relative to the world origin, $\mathbf{q}$ is the orientation expressed as a quaternion, $\mathbf{v}_B$ denotes the rate of change of $\mathbf{x}$ (rotated into the body frame), and ω is the angular velocity in the world frame. This follows the practice outlined in Barry, Florence, and Tedrake [26]. The equations of motion describing the time derivatives of $\mathbf{s}$ can be written as

$$m\ddot{\mathbf{x}} = m\mathbf{g} + \mathbf{f}, \quad (2.4)$$

$$\mathbf{I}\dot{\omega} + \omega \times \mathbf{I}\omega = \tau, \quad (2.5)$$

where $\mathbf{g}$ is gravitational acceleration, m the mass, $\mathbf{I}$ the moment of inertia in the world frame, and $\mathbf{f}$ and τ the net external forces and torques expressed in the world frame.

Some of these quantities depend on design variables. The mass m is the sum of individual

component masses, so a robot using fewer, lighter components will be able to accelerate more for the same applied force. This is one example of how the shape design of a robot can influence its performance. A more complete description of rigid body dynamics in the context of flying robots is given in section 5.4.1.

2.2.2 Articulated Body Dynamics

For simplicity, many aerial and marine robots are modeled as a single rigid body. Propellers, despite being separate rigid components connected by a rotating joint, can only contribute external forces and torques. This simplification is often reasonable because the mass of propellers is negligible compared to the mass of the robot chassis. This assumption does not always hold, however. In the case of walking robots, the interactions between adjacent limb segments have a large impact on the robot’s motion. We need special techniques to simulate the dynamics of these *articulated bodies*.

One approach is to treat each rigid segment (or *link*) as its own rigid body and introduce a constraint for each joint. Each constraint restricts the relative pose between adjacent links, ensuring that it reflects the corresponding joint’s range of motion. This approach, known as the *maximal coordinate* formulation, prioritizes implementation simplicity over performance. Enforcing the constraints can be difficult, and some amount of constraint violation is expected. This can manifest as visible misalignment and separation between links. Additionally, the number of state variables greatly exceeds the true number of degrees of freedom in the system. This leads to greater memory consumption and unnecessary computation.

A more recent approach for simulating articulated bodies is known as the *reduced coordinate* formulation. Featherstone [27] shows how the dynamics of a rigid body can be generalized to articulated bodies with internal revolute and prismatic joints. Only one additional coordinate variable is needed for each joint compared to six for the maximal formulation. Computational complexity is only linear in the number of joints, and it becomes possible to

use unconstrained optimization techniques such as gradient descent. The reduced coordinate formulation also inherently eliminates the problem of joint separation, producing more accurate results. Due to these advantages, we simulate land based robots described in chapter 3 using reduced coordinates.

The implementation details of reduced coordinate dynamics are outside the scope of this thesis. [28] provides an introduction to this topic that assumes basic knowledge of rigid body dynamics.

2.3 Optimization

Design representation and simulation make it possible to predict the performance of a robot from its design parameters. While useful to designers, they fall short of a full robot co-design method. We need optimization in order to close the loop and automate the robot design process. Based on the performance of previously simulated designs, optimization algorithms determine which designs to try next. Ideally, they should reveal optimal robot designs in as few iterations as possible.

No single algorithm is suitable in every case, and the appropriate algorithms depend on the specific design representation and simulation used. In the case of a hybrid discrete and continuous design representation, using multiple nested layers of optimization may make sense. The remainder of this section discusses developments in gradient-based optimization and heuristic-based search algorithms that are combined in this thesis.

2.3.1 Gradient-Based Optimization

Normally, simulation is a one-way computation that goes from a robot design to its performance. Based on the results of multiple simulations, gradient-free optimization methods such as genetic algorithms and reinforcement learning can gradually improve the robot's shape and controller. These methods tend to be very slow and require a large number of

samples to converge. This sample inefficiency limits the complexity of designs that can be explored.

Gradient-based optimizers form another category of optimization methods. As the name suggests, they require our simulation to provide *gradient* information. In the simplest case, this consists of first-order derivatives of the objective function with respect to every design parameter. Adding gradient information results in a *differentiable simulation*. Optimizers no longer need to estimate the gradient themselves, increasing sample efficiency.

Differentiable simulation has already proven useful in system identification and control [29]–[31]. In each case, gradient information significantly reduces the time needed to train a controller. More recent work [9], [32], [33] examines how differentiable simulation can be extended to robot co-design. Gradient-based methods can efficiently optimize shape parameters at the same time as the control parameters, often with minimal additional code.

While gradient-based techniques can accelerate robot co-design, they also suffer from a number of pitfalls. Discontinuities in either the objective function or its gradients (which can result from collision) often result in failure to converge. Gradient-based techniques are also prone to local minima, making initialization important.

Even with workarounds for these issues, gradient-based optimization cannot address every aspect of robot co-design. As mentioned earlier, the shape of a robot is described by both continuous and discrete parameters. Gradient-based techniques can only improve the former. Other methods are still needed to decide on the best discrete topology and component selection for a robot.

2.3.2 Graph Heuristic Search

While efficient methods exist for continuous design optimization, discrete design optimization is generally more difficult. We cannot rely on gradient information, and it is easy to miss high-performing designs. Early work in mechanism design relies on exhaustive enumeration [15], [16] to find the best settings of discrete design variables. While simple and

globally optimal, enumeration is only possible for the smallest design problems. Later work improves scalability through intelligent partial search. Instead of finding a global optimum, these methods only intend to find acceptable solutions within a limited number of iterations.

One class of partial search methods is best-first search using hand-crafted heuristics [34], [35]. Very few iterations are needed to achieve a reasonable result, but the quality of the heuristic has a large impact on efficiency. The heuristic functions also require domain-specific knowledge, and they do not generalize to other types of mechanisms.

Another class of partial search methods includes simulated annealing [36] and genetic algorithms [4]. These *stochastic* search methods incrementally modify designs using a set of operators, keeping better-performing modifications with greater probability. This biases the search towards high-performing regions of the design space without needing hand-written heuristics. One significant disadvantage of these methods is their low sample efficiency. Since they are effectively random walks, a large number of iterations is required to find optimal designs.

More recent approaches based on learned heuristics aim to combine the advantages of both heuristic-based and stochastic search. The heuristic function, rather than being designed by hand, is fitted to data from previous design evaluations. After enough iterations, the heuristic becomes capable of guiding a best-first search towards more promising designs. This concept, which can be considered a form of deep Q-learning, combines the generality of stochastic methods with the efficiency of heuristic-guided methods. Deep Q-learning is capable of surpassing human instinct on control [37] and combinatorial optimization [38] problems. The latter work demonstrates how deep Q-learning can be applied to graph data, but relies on predefined graph embeddings. More recent work trains entire graph neural network (GNN) based heuristics from end-to-end [39], [40], taking advantage of their increased representational power. Our Graph Heuristic Search (GHS) algorithm builds on this line of work and extends it to graph grammar-based shape design. Unlike previous work, GHS accounts for nonterminal symbols and the changing cardinality of graphs as they are

incrementally built by a grammar. Section 3.6.1 explains Graph Heuristic Search in detail.

2.4 Summary

The building blocks of computational robot co-design have advanced significantly over the past few decades, but the overall workflow remains the same. All robot co-design systems need a way to represent designs, simulate them, and optimize their design parameters for better performance. This section outlines the specific methods used in this thesis, namely robot graph grammars, rigid body simulation, differentiable simulation, and Graph Heuristic Search. Some of these methods are novel, and we place them in the context of prior developments. Other methods, which are already suitable for our purposes, are chosen to form complete robot co-design workflows. The next three chapters are previously published work that put these methods into practice.

Chapter 3

Land Robots

3.1 Introduction

The automation and understanding of robot design, and the interplay between the structure and controller of a robot has long been a key research question [12]. This is a particularly challenging research problem as the design space is vast and intractable and there are limited tools for automatically and efficiently exploring it. To enable large scale search and optimization of robots we need new approaches for structuring this design space, tools for searching it efficiently, and simulations for exploring and evaluating many thousands of designs [41]. Importantly, the tools developed must allow for the emergence of imaginative, or even inventive solutions, which diverge from those that manifest from more traditional design approaches [42].

To address this challenge, we introduce a simulation-based system for simultaneously optimizing the physical structures and controllers of robots. The goal of the system is to take a set of user-specified primitive components and generate an optimal robot structure and controller for traversing a given terrain. The primitive components include different joint types, links, and wheels, each with user prescribed attributes such as rotational angles and axes, sizes, and weights. The user can specify the makeup of the primitives to match

what physical components they have available. In the *RoboGrammar* framework each robot is represented by a graph. To efficiently search the design space of robot graphs, we introduce a recursive graph grammar that emphasizes mobility and fabricability. Our grammar is expressive and maps realistic components to fabricable configurations, and can adapt to changing component primitives. Furthermore, we couple the grammar with physical simulation and controller synthesis, allowing us to rapidly create, optimize, and test robot designs. We use model predictive control (MPC) to provide a stochastic approach to controller learning. In order to make the search of a large design space efficient, we introduce a novel Graph Heuristic Search algorithm which generalizes the knowledge of explored designs to predict performance of unexplored branches of the search space. Specifically, our search algorithm takes a learning-based approach inspired by reinforcement learning, iteratively exploring a large space of robot designs for a given task, and learning a heuristic function to gradually steer that search toward optimal designs. Our learning model takes a neural-based approach, exploiting a graph neural network architecture to provide a fast method for approximating the performance metrics of best designs.

In summary, this chapter presents the following key contributions:

- A recursive graph grammar that enables the generation of a wide range of inherently fabricable robot forms, which can be built in their simulated configurations.
- A Graph Heuristic Search method for efficiently searching the design space described with the grammar. This is bench-marked against alternatives including Monte Carlo tree search and random search.
- A demonstration of terrain-driven optimization using MPC based stochastic evaluation of each proposed design. We show the variety of innovative robot designs that are obtained across six different terrains. In addition, our approach identifies a number of high-performing robots for a single terrain or combination of terrains.

The following section reviews related work presenting current state-of-the-art research.

In Section 3.3 we provide an overview of the different systems and algorithms used to enable the grammar-driven exploration and optimization of robots. Section 3.4 focuses on the graph grammar which is developed to express a wide range of different robots. We detail the components of the grammar, and the rules enabling the creation of innovative robot structures. In order to evaluate the performance of robots and find controllers we use model predictive control (MPC) with integral control and low-pass filtering (Section 3.5). In Section 3.6 we present the optimization process, providing details of the Graph Heuristic Search and the Monte Carlo tree search implemented for sake of comparison. The results are presented in Section 3.9, showing best-performing designs generated using *RoboGrammar* for six different terrains, as well as combinations of terrains. We conclude with a discussion of the limitations of our approach and identify avenues for future work.

3.2 Related Work

In this section we review techniques for generative design of robots, and existing state of the art work relating to generative design, graph grammars, and controller optimization.

3.2.1 Generative Design for Robotics

The development of effective generative design tools for robotics, including search and evolutionary design algorithms, is a key research challenge [43]. We review several approaches, detailing the various advantages they offer and the applications for which they are best suited.

Many recent approaches use *deep learning* which provides powerful tools for generative design. [21] develop a modular co-evolution strategy in which a collection of primitive agents learn to dynamically self-assemble into a composite body while also learning to coordinate their behavior to control the body. However, their method results in only simple robots with a few components, and assumes physically implausible reconfigurability. Reinforcement

learning has also been applied in order to continually improve agent design [44]. In their design framework, both the environment and robot are altered to enable the agent to learn more effectively. Deep learning methods are particularly suited to co-optimization of the robot body and controller [10], and have the potential to offer improved performance. Within the domain of soft robots, there has also been an example of end-to-end controller and structure design using deep latent representations [13].

Formal Design Methods are an alternative approach which use a model based approach to design robots and their controllers. Jing, Tosun, Yim, *et al.* [45] demonstrate the formal approach using a large design library and bespoke tools to configure modular robots. The arrangements of modular robots are particularly suited for these methods [46], [47]. Such methods can require significant hand crafting and do not scale or generalize well.

To search a parameterized design space, in our case a graph grammar, efficient search algorithms are required. *Genetic algorithms* (GAs) iteratively improve designs through a process inspired by natural evolution [48]. Applied to real-world systems [49], GAs transfer positive ‘phenotypes’ between generations of robots, and allow the development of large populations of robot designs [50]. Sims [4] evolves creatures optimized for movement in a three-dimensional environment using a GA. Creatures are represented as directed graphs, which may include multi-edges. Each creature’s kinematic tree is determined by tracing unique paths through its directed graph. This representation encodes symmetry and repetition in the graph itself, leading to more biologically plausible forms. It is not clear how the creatures translate into robotic components, however. The effect of terrain on optimal morphology is also not addressed, as all creatures are simulated on a flat surface or in fluid. GAs have been widely used within the robotics community with demonstrable success [51], however, in some cases they do not scale well and can have a high sensitivity to input parameters [52]. They can also be sensitive to parameters such as population size and rate of mutations or crossovers, and there is little evidence demonstrating convergence to global or even local minima.

There are also a number of *hybrid approaches* which combine multiple methods. Wang, Zhou, Fidler, *et al.* [53] propose Neural Graph Evolution, an evolutionary search in graph space, which iteratively evolves graph structures using simple mutation primitives. Wang, Lehman, Clune, *et al.* [54] pairs the generation of environments with the optimization of agents. Their method simultaneously explores many different paths through the space of possible problems and solutions, utilizing neural networks and optimization techniques. However, without a judicious grammar to help constrain the search, their approach has only been shown to scale to robot designs with a few joint and link components. Further, their reliance on a genetic algorithm to search over the design space has the same pitfalls as discussed previously.

In this chapter we explore two candidates for grammar-based generative robot co-optimization. First, *Monte Carlo tree search* (MCTS) is a stochastic search algorithm widely used across a variety of problem domains [55]. It has been applied to game based decision making, notably for playing Go [56], as well as robot optimization and planning [57], [58]. The algorithm identifies the most promising moves, expanding the search tree based on random sampling of the search space. Schadd, Winands, Van Den Herik, *et al.* [59] have shown the ability to extend MCTS to a single-player scenario, which is the case when searching over a grammar. Despite its wide applicability, MCTS suffers from low searching efficiency on complex robot design problems. We therefore use it as a baseline. Our core algorithmic contribution is an novel search optimization strategy, referred to as *Graph Heuristic Search* (GHS). GHS generalizes the knowledge from explored designs to untested designs to improve searching efficiency. By learning an estimator of performance, a heuristic can guide the search, helping to find optimal solutions faster. We apply deep learning, specifically graph neural networks (GNNs), to learn this heuristic. GNNs are particularly suited for learning on topological or structured inputs, such as robot designs [40]. A variety of GNN models exist, but the differentiable-pooling model [60] is of particular interest for grammar based exploration.

3.2.2 Graph Grammars

In this work, we examine *recursive graph grammars* as a method for generating expressive search spaces that are restricted to feasible designs with user-defined components. Formal grammars, sets of production rules for creating valid strings from a language’s alphabet, are widely used for linguistics and natural language processing [61]. This concept can be extended to graph grammars, which define sets of valid graphs rather than linear sequences. For the purposes of generative design, the graphs usually describe spatial configurations of mechanical components. Early examples include graph grammars for epicyclic gear train systems [62] and Meccano®-based machines [19]. In robotics, graph grammars have been applied to modeling self-assembly of robotic systems [63] as well as the structure of robot interactions and control laws [64]. More recently, a graph grammar has been developed to describe the physical structure and also the fabrication process for furniture [65]. This grammar allows complex 3D furniture models to be expressed as manufacturable parts and connectors. Scalability and applicability to physical systems are key advantages of graph grammars [66]. They provide a way of parameterizing the search space to prevent intractability, while allowing many different structures to emerge.

Most analogous to our work is Stöckli and Shea [20], which describes passive dynamic brachiating robots with a graph grammar and evaluates them using dynamic simulation. Their work showed the power of graph grammars in automated robot design, with a wide range of robots emerging from a relatively limited grammar. Their work did not examine the expensive control synthesis aspect (as well as the resulting co-optimization problem) and focused solely on 2D systems with swinging locomotion. In our scenario, simulation and robot evaluation is far more complex, and requires a more scalable search algorithm.

3.2.3 Grammar-based Procedural Modeling

Shape grammars use graphical primitives to generate complex geometric shapes. Initially introduced by Stiny and Gips [67], shape grammars have been used in architectural design [68]–[70]. A simplified version of shape grammars which is more popular in computer graphics applications is the set grammar [71]. Grammars for modeling streets and buildings have been proposed in [72]–[76]. Van Diepen and Shea [77] apply shape grammars to soft robot design, using predetermined actuation patterns instead of full control synthesis.

A majority of high-performing grammars are still manually designed by experts. Several works propose automating grammar creation based on a set of example designs [78]–[80]. Lipp, Wonka, and Wimmer [81] develop a framework for interactive grammar editing. Dang, Lienhard, Ceylan, *et al.* [82] expand shape grammars with probability density functions defined through an interactive design exploration tool to obtain designs with higher preference score. A probabilistic grammar is also used in [83] to parse unseen scenes and assign segmentation, labels, and object hierarchies.

3.2.4 Control Methods and Approaches

In addition to generating the structure of the robot, some method of generating appropriate controls is necessary. To allow exploring many different robot structures across different terrains the control approach must be highly efficient and robust.

Model predictive control (MPC) is a family of control algorithms used widely for robotics and process control [84]. A model of the system is used to make predictions about future behaviour, with online optimization used to find optimal controls to meet a desired output. Within robotics it is widely used to develop trajectory tracking controllers [85], [86]. There are many variants of MPC suitable for different settings. When no derivatives of the dynamics are available, for example in a non-differentiable simulator, sampling based MPC methods such as model predictive path integral control (MPPI) [23] are typically used. Low-pass

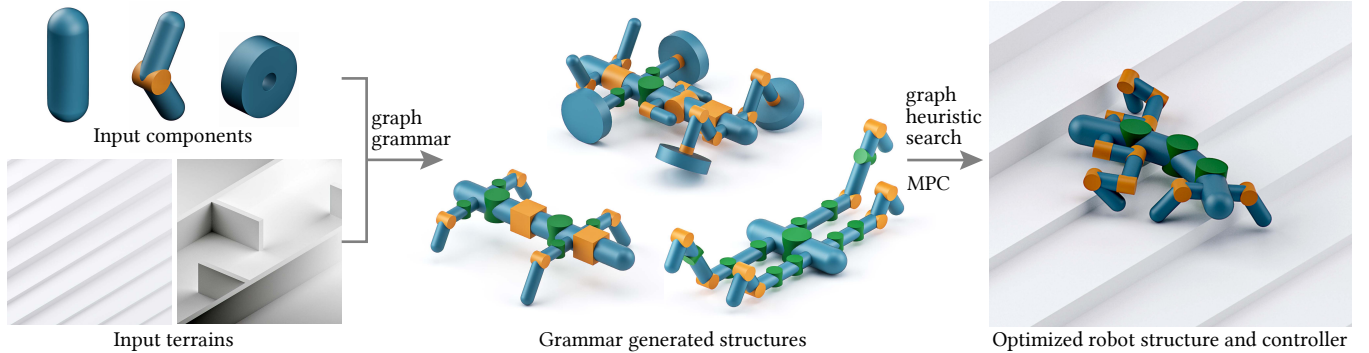


Figure 3.1: The input to our system is a set of base robot components, such as links, joints, and end structures, and at least one terrain, such as stepped terrain or terrain with wall obstacles. *RoboGrammar* provides a recursive graph grammar to efficiently generate hundreds of thousands of robot structures built with the given components. We then use Graph Heuristic Search coupled with model predictive control (MPC) to facilitate exploration of the large design space, and identify high performing examples for a given terrain. Our approach enables co-optimization of both robot structures and controllers.

filtering has been shown to be helpful in other simulation-based applications of MPC [24].

Another increasingly popular control approach is the use of reinforcement learning. Stochastic policy gradient reinforcement learning can enable a 3D biped to walk over rugged terrain [87]. Peng, Berseth, Yin, *et al.* [88] demonstrate how hierarchical deep reinforcement learning can learn dynamic locomotion skills on challenging terrains with a limited amount of prior knowledge. Reinforcement learning is characterized by the need for large amounts of data, which can hamper efforts to wrap control synthesis in an outer design search loop.

3.3 System Overview

RoboGrammar consists of three main components listed below. Figure 3.1 provides a graphical overview.

First, a recursive graph grammar forms the core of our approach to optimizing the structure of robots (Section 3.4). We use a graph representation for robot structure and define the set of components and grammar rules which can be used to assemble robots. Our grammar encodes simple and intuitive rules in order to efficiently generate interesting, feasible robot

designs during the search process.

Secondly, each design generated by the grammar is evaluated using the model predictive control (MPC) algorithm described in Section 3.5. MPC aims to rapidly find a stable, periodic gait. The objective function optimized by MPC rewards high forward speed and maintaining the initial orientation.

The third component of our system is the novel Graph Heuristic Search (GHS) algorithm described in Section 3.6. GHS searches over the design space defined by the grammar to efficiently identify optimal robots and controllers. The algorithm exploits a graph neural network-based heuristic function, whose architecture is analogous to the graph-like structure of rigid robots. The heuristic function is learned as the search progresses using ground-truth data from the MPC-based evaluations.

We describe how users specify problems in our framework in Section 3.7, and provide implementation details in Section 3.8. Different terrains are used to embody objective functions for which the robots are optimized. We demonstrate our system on single and multi-objective optimization problems, and showcase our results in Section 3.9.

3.4 Robot Grammar

Starting with a given set of robot components, our goal is to efficiently explore over the space of robots that can be formed from these components. However, this space can be combinatorially large and primarily composed of nonsensical designs. In order to make the search more tractable, we constrain it with a bio-inspired graph grammar. The structural rules are inspired by arthropods, which make up a majority of known animal species. Note that we only use bio-inspiration as a starting point. We choose to include wheels, which enable even more forms of locomotion at the cost of only one additional rule. Our grammar includes many familiar forms, while allowing novel designs to emerge.

3.4.1 Robot Representation

In order to enable robot design with graph grammars, we represent robots in the form of directed acyclic graphs. Each node of the graph represents a physically realizable component. We consider robot structures to consist of body segments and limbs, with optional head and tail. The structures are composed of rigid links and rigid or articulated joints. Each body segment can have at most one pair of legs attached to it.

As our grammar is based on arthropods, it only describes symmetrical robots. Legs are always added in pairs and each pair has identical leg structure on both sides. Exceptions to this rule are the optional head and tail limbs, which are individually formed and are not necessarily symmetrical. We encode symmetry and repetition by representing each pair of legs with a single branch in the robot graph (see Figure 3.2). This scheme helps simplify rule definitions in the grammar.

After deriving a full robot graph, we convert it to a kinematic tree for efficient simulation. Note that all robots in our grammar, as well as the vast majority of animals and existing robots have kinematics that can be described by a tree. A new robot component is added per graph node, for each unique path connecting the tree root and that node. Finding the root node involves selecting an arbitrary node and following edges backwards until a node with no incoming edges is reached. Each graph node may produce multiple components in the kinematic tree due to our symmetry-enforcing scheme.

3.4.2 Grammar Definitions

We define a recursive graph grammar $\mathcal{G}$ as a tuple

$$\mathcal{G} = (\mathbf{N}, \mathbf{T}, \mathbf{A}, \mathbf{R}, S)$$

where $\mathbf{N}$ and $\mathbf{T}$ are sets of non-terminal and terminal symbols respectively, $\mathbf{A}$ is a set of attributes for certain terminal symbols, $\mathbf{R}$ are production rules and $S \in \mathbf{N}$ is a starting

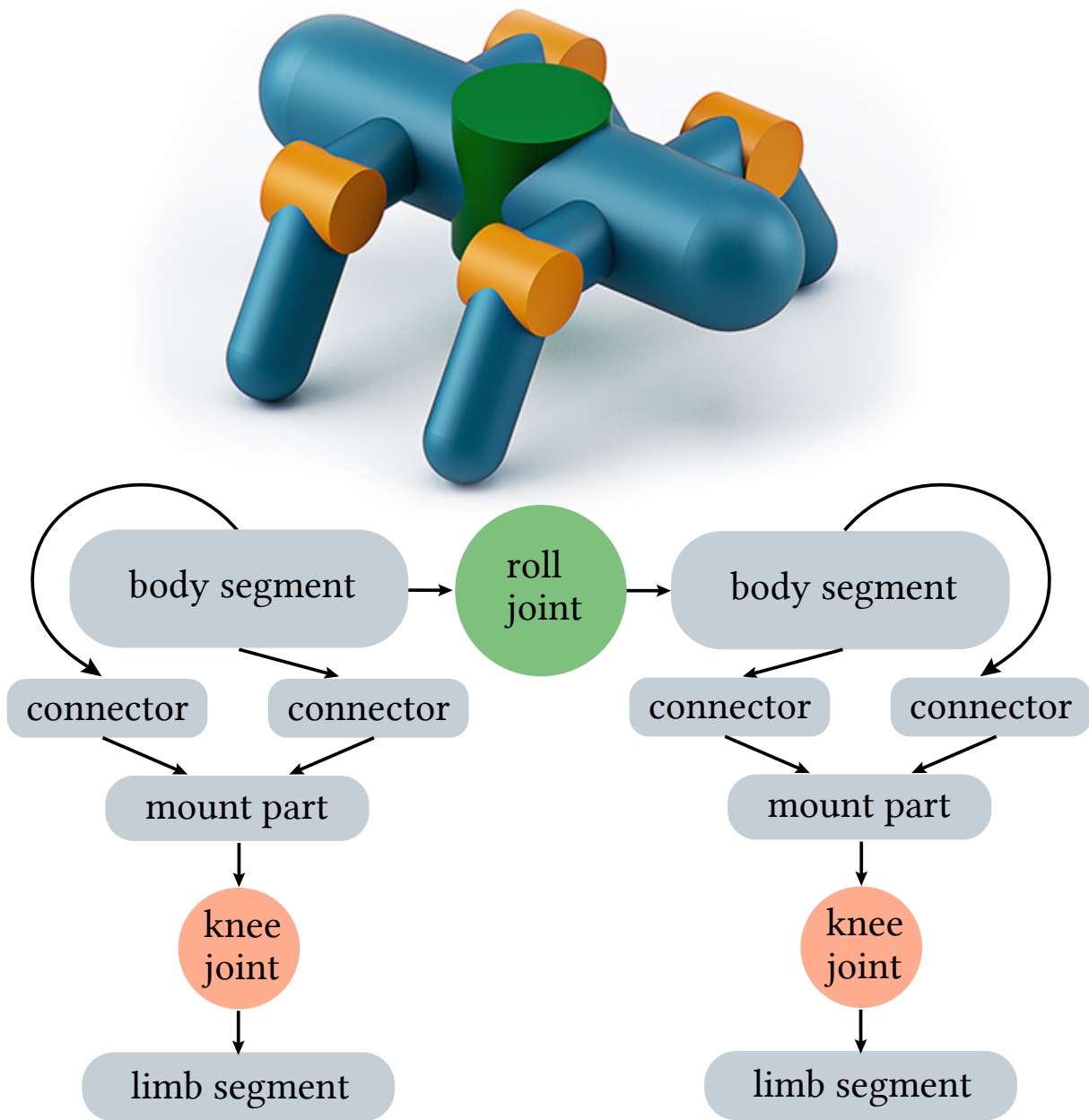


Figure 3.2: An example of a kinematic tree (top) with the corresponding robot graph (bottom). To enforce symmetry in leg pairs, after adding nodes for connectors on both sides of the body, both legs of one pair are defined in one branch of the graph.

Grammar structural rules

Body structure

$$r_1: \quad (S) \Rightarrow (H) \leftarrow (B) \leftarrow (T)$$

$$r_2: \quad (T) \Rightarrow (Y) \leftarrow (B) \leftarrow (T)$$

Adding appendage to body

$$r_3: \quad (B) \Rightarrow \begin{array}{c} (U) \\ \swarrow \quad \searrow \\ (C) \quad (C) \\ \swarrow \quad \searrow \\ (M) \\ \downarrow \\ (E) \end{array}$$

$$r_4: \quad (B) \Rightarrow (U)$$

Appendages

$$r_5: \quad (E) \Rightarrow (J) \rightarrow (L) \rightarrow (E)$$

$$r_6: \quad (T) \Rightarrow (C) \rightarrow (M) \rightarrow (E)$$

$$r_7: \quad (H) \Rightarrow (E) \leftarrow (M) \leftarrow (C)$$

Legend

S	start symbol
H	head part
Y	body joint
B	body part
T	tail part
U	body link
C	connector
M	mount part
E	limb end
J	limb joint
L	limb link

Figure 3.3: A list of structural rules of our robot grammar. Here $S, H, Y, B, T, U, E, J, L \in \mathbf{N}$ are non-terminal symbols. Rule r_1 initializes the body structure, while r_2 can be used to extend the body. Note that each body segment U can have at most one pair of limbs attached to it. Rule r_3 enforces symmetry of the limb pairs, and rule r_4 allows body segments without limbs. Rule r_5 serves for extending the limbs, and r_6 and r_7 for adding back and front limbs.

symbol. Non-terminal symbols are temporary graph nodes that help us construct different body and leg parts. Terminal symbols are final graph nodes which represent physical robot

components (e.g., links, joints, wheels, etc.). We refer to graphs with only terminal symbols as “complete” robot designs, and all other graphs as “partial” robot designs. In addition, we assign attributes to several terminal symbols. The attributes define the initial state of the robot by determining initial lengths and angles between robot parts. Each production rule from $\mathbf{R}$ has the form:

$$Q \rightarrow W$$

where $Q \in \mathbf{N}$ is a non-terminal symbol, and W is a graph that has at least one node, no matter if it is a non-terminal or a terminal symbol (see Subsection 3.4.3 for more details). A production rule is applied to a current robot graph by detecting an occurrence of Q to replace with W . Since our grammar is recursive, a non-terminal symbol Q can appear again on the right hand-side of the rule as part of W . A recursive grammar allows us to concisely represent a wide variety of structures, including complex body shapes with numerous limbs such as a centipede.

3.4.3 Production Rules and Robot Components

Here we present the production rules used to create robot structures. We divide our robot grammar rules into two main categories: *structural rules* and *component-based rules*. Structural rules serve to construct a physically realistic topology for the robot and define the number of body and limb segments. Every structural rule has at least one non-terminal symbol on the right-hand side. The list of these rules is given in Figure 3.3.

Component-based rules replace non-terminal symbols with terminal symbols representing robot components. Graph nodes with terminal symbols cannot be replaced. Several terminal symbols have assigned attributes. Attribute $\theta_r \in \mathbf{A}$ limits the rotational range of a joint. Attribute $\theta_i \in \mathbf{A}$ determines the initial position of the links connected with the corresponding joint and it can be chosen from a given set of possible values. All the angle values θ_i are defined relative to the position of the previous robot component. A list of the component-

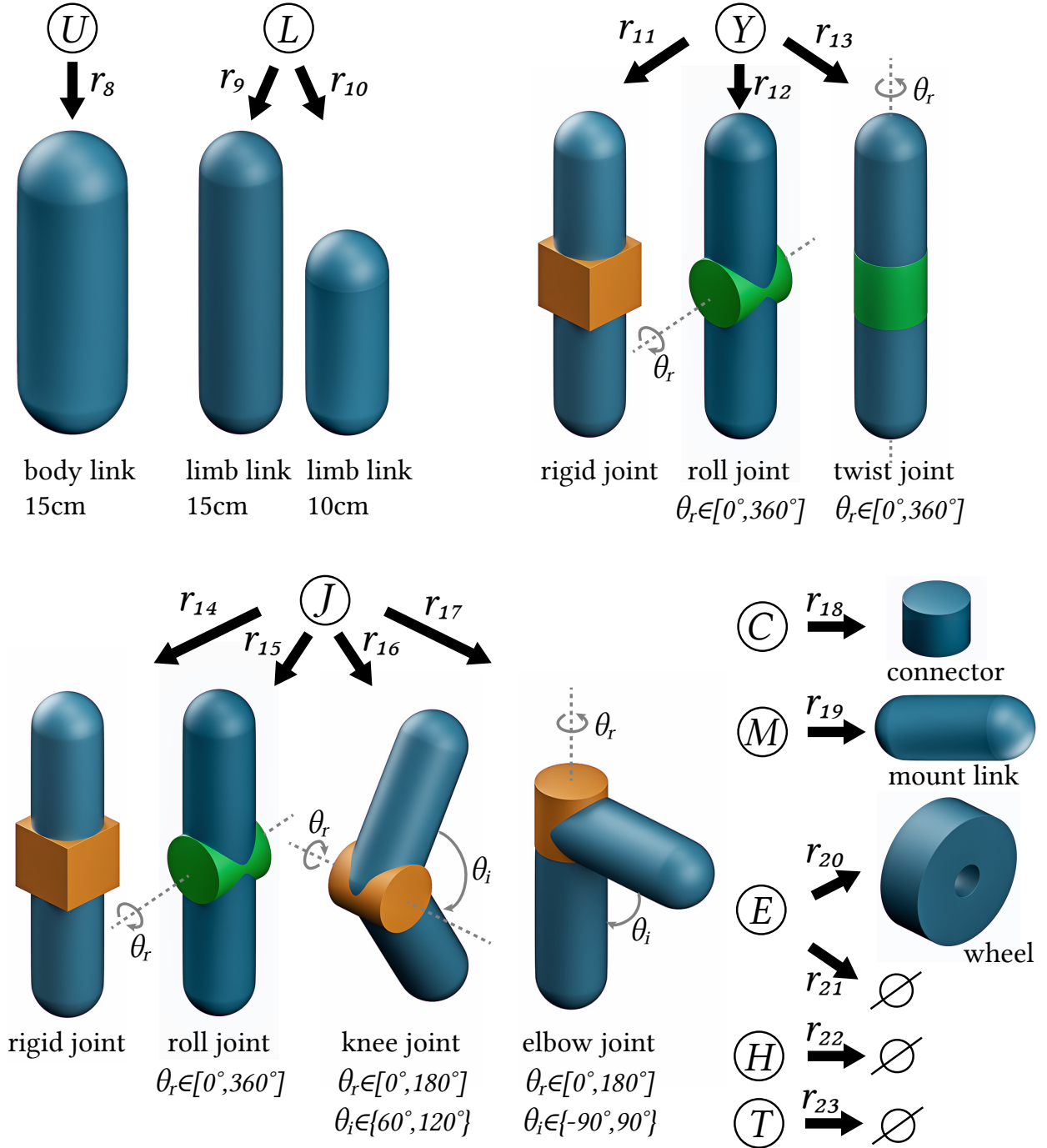


Figure 3.4: Component-based rules of the robot grammar. Initial-pose angle θ_i and rotational range angle θ_r are attributes of joints. Adding, removing, or editing any link or joint component is straightforward. Note that we do not use a physical representation for the head, tail, hands, feet, and other types of end components for simplicity. However, the grammar can easily incorporate any new component representing an end structure.

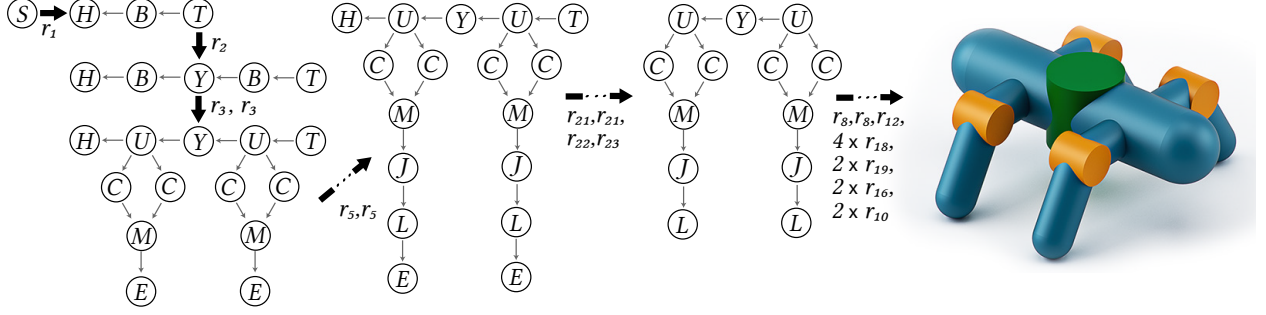


Figure 3.5: A derivation sequence for a Simple Walker robot generated with our grammar. Derivation begins with the start symbol S , then creates the body and extends it with rules r_1, r_2 respectively. Legs are added on both body segments with rule r_3 applied twice. Both pairs of legs are extended, adding additional sets of joints and links, with r_5 . For the Simple Walker, end structures are not used, hence we remove them with r_{21}, r_{22} , and r_{23} . Finally, terminal components are added for each segment of the robot, following the rules from Figure 3.4.

based rules in our framework is presented in Figure 3.4. Note that thanks to our grammar-based approach and the set of rules we propose, the list of components used for robot construction can be easily adapted. There is no requirement that structural rules be applied before component-based rules or vice versa.

Our recursive grammar is designed in a way that allows for a potentially infinite number of legs and body segments. In order to limit the design space, our implementation uses a recursion counter. The recursion counter counts the total number of derivation steps. We set the maximum as 40 in our experiments. Increasing this parameter would allow for creation of more complex designs, while also increasing algorithm run time.

An example robot derived from grammar $\mathcal{G}$, along with the sequence of production rules applied, is shown in Figure 3.5. More interesting examples generated with our grammar are presented in Figure 3.6.

3.5 Simulation & Control

Here, we describe our method for simulating and controlling generated designs. This is necessary both for co-optimizing designs and controllers, as well as evaluating the quality of

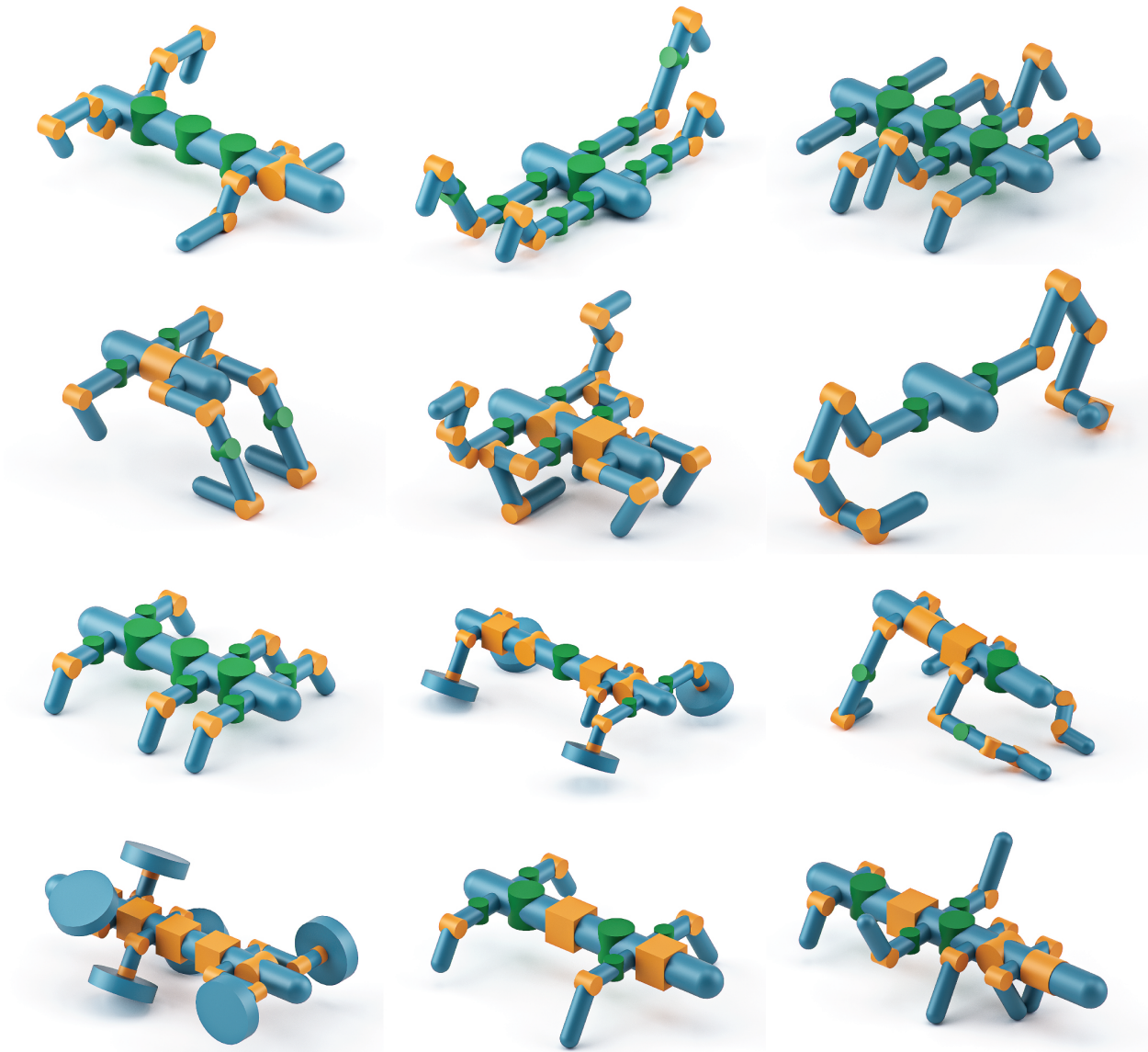


Figure 3.6: Designs generated with *RoboGrammar* that appear during the search process for various terrains. Further optimization is necessary to identify the best performing ones for a given task.

generated designs.

3.5.1 Simulation

Robot performance on a given task is evaluated using rigid body dynamics simulation. The robots are modeled as articulated rigid bodies using an efficient recursive formulation based on Featherstone’s algorithm [27]. This recursive formulation reduces the complexity of the equations of motion, providing scalability and numerical stability. Collisions between the robot and the environment as well as between different rigid components of the robot (self-collision) are fully modeled. Graphs describing robots that self-collide in the initial configuration are discarded. Wheels are velocity-controlled, while all other joints are position-controlled. Our simulation is implemented using the Bullet Physics library [89], which is widely used in robotics.

3.5.2 Model Predictive Control

We generate optimized control inputs for each design using model predictive control, specifically the MPPI algorithm [24]. MPPI was chosen because it does not rely on derivatives and is relatively simple to implement. Our MPC implementation maintains a sliding window U of control inputs H time steps long, representing the “best” control inputs found so far. Each iteration begins by sampling K perturbed sequences of control inputs based on U , and applying each sequence of control inputs to a separate instance of the simulation. A new sequence U is then computed as a weighted sum of the perturbed sequences, with each sequence’s weight depending on the discounted sum of rewards for that simulation. The first control input of U is appended to the optimized control sequence, which is the output of the algorithm. U is shifted forward one time step to prepare for the next iteration, filling with zeroes as necessary. The algorithm repeats until an optimized control sequence of length T (the episode length) is obtained. To take advantage of multiple CPU cores, we run the simulation instances in parallel using a thread pooling library.

Note that each MPC time step corresponds to multiple time steps in simulation, with the ratio being the *control interval*. Control inputs are repeated for the control interval, reducing the number of MPC time steps necessary. This reduces computation time significantly without noticeably degrading robot performance.

An outline of our MPPI implementation is given in Algorithm 1.

Algorithm 1: Model Predictive Control based on MPPI

Inputs: Episode length T , number of samples K , horizon H , initial state s_0 , initial input sequence $U = [u_0, u_1, \dots, u_{H-1}]$, simulation dynamics f .
Output: The optimal control input sequence $[a_0, a_1, \dots, a_T]$.
for $t \leftarrow 0$ **to** $T - 1$ **do**
 for $k \leftarrow 1$ **to** K **do**
 Sample input sequence U_k based on U .
 Apply inputs U_k starting at s_t , yielding return r_k .
 end for
 for $k \leftarrow 1$ **to** K **do**
 $w_k \leftarrow e^{\kappa(r_k - \max_l r_l)}$
 end for
 $U \leftarrow \sum_{k=1}^K U_k \frac{w_k}{\sum_{l=1}^K w_l}$
 $a_t \leftarrow u_0$
 $U \leftarrow [u_1, \dots, u_{H-1}, \mathbf{0}]$
 $s_{t+1} \leftarrow f(s_t, a_t)$
end for

Sampling

Careful design of the sampling distribution is key to achieving high MPC sample efficiency. Input sequences U_k are matrices where each entry is sampled from an independent normal distribution. The parameters of the distribution vary depending on the type of sample: *warm-start* or *history*.

Half of the samples are *warm-start* samples, whose distribution is centered on the input sequence U from the previous iteration shifted forwards by one step. Their standard deviation is σ_1 .

As we expect many high-performing gaits to be periodic, the remaining samples are *history* samples. They are sampled from a normal distribution whose mean is the last M

control inputs repeated until the MPC horizon. M varies per sample, and ranges from $H/2$ to H . Their standard deviation is σ_2 .

Combining the two different types of samples enables MPC to generate lifelike, periodic gaits while reacting to obstacles and changing terrain conditions.

3.6 Search & Optimization

Our grammar forms the front end of the *RoboGrammar* system, constraining the search space of all robotic structures to a tractable and meaningful subset. Next, we describe the search and optimization algorithms that efficiently search for high-performing designs and controllers, exploiting the reduced search space this grammar provides. Here, we present two search algorithms: our novel Graph Heuristic Search, and our variant of Monte Carlo tree search, which serves as a baseline.

3.6.1 Graph Heuristic Search

Our heuristic search algorithm is learning-based, using a learned heuristic function to inform and accelerate the search of the design space. This heuristic takes the form $V(g) : (\mathcal{V}, \mathcal{E}) \rightarrow \mathbb{R}$. The input to the function is a graph representing a partial robot design, where some nodes correspond to non-terminal symbols. The partial design may be expanded into one of many complete designs which have only terminal symbols. The function $V(g)$ aims to outputs the highest achievable performance across all of these complete designs. Our search algorithm is agnostic to the model used for the heuristic, and thus we describe it in general terms. In practice, we take a deep-learning-based approach and use graph neural networks to create our learnable heuristic.

Search Algorithm

Our Graph Heuristic Search algorithm works by interleaving a design phase (in which a candidate robot is sampled, guided by our heuristic function), an evaluation phase (in which the candidate robot is evaluated in simulation), and a learning phase (in which the heuristic function is improved based on the simulated data). These three phases are repeated over N episodes, or until they converge on an optimal design. The algorithm is described in Alg. 2.

Design Phase During the design phase, K possible candidate robot designs are generated and one of them is selected for evaluation. Each design is generated by the following procedure. Starting from a partial robot design s_0 , composed solely of the initial start symbol ($s_0 := S$), production rules of the grammar are iteratively applied to the partial robot design until it contains only terminal symbols. The selection of production rules is inspired by Q -learning [90] and follows an ϵ -greedy approach. Given a partial robot design s_l after l production rules have been applied, the $l + 1^{\text{th}}$ rule a_{l+1} is selected as follows. With probability ϵ , a random rule is applied from the set of possible rules. Otherwise, with probability $(1 - \epsilon)$, the rule that leads to the design with the highest heuristic score is applied, *i.e.* $a_{l+1} \leftarrow \arg \max_a V_\theta(P(s_l, a))$, where $P(\mathcal{S}, \mathcal{A})$ is a function which applies production rule $\mathcal{A}$ to partial design $\mathcal{S}$. Once a candidate design with only terminal symbols is produced, it is added to the list of possible candidate designs. From the final list of K candidates, a random robot is selected with probability ϵ ; with probability $1 - \epsilon$, the design with the highest heuristic score is chosen as the candidate to continue to the evaluation phase.

Two ϵ -greedy selection steps are applied during candidate robot generation. This strategy is necessary to ensure that the space of possible robot designs is sufficiently explored; if one begins with a pure greedy strategy, the algorithm quickly converges to a suboptimal design. This is because we are taking a learning-based approach; our heuristic function is inaccurate at the beginning (and not strictly admissible) and improves in accuracy as the algorithm progresses. Until the heuristic function converges to an accurate estimator mapping robot

Algorithm 2: Graph Heuristic Search

Inputs: Number of iterations N , number of candidate designs M , Adam optimization steps opt_iter and batch size M .

Output: The best design s^* .

Initialize the look up table $\hat{V} \leftarrow \{\}$.

Initialize the graph neural network $V_\theta(s)$ with random parameters θ .

Initialize the best design $s^* \leftarrow \text{None}$, $r^* \leftarrow 0$.

for episode $j \leftarrow 1$ **to** N **do**

▷ **Design Phase:** Generate a candidate design

$\mathcal{P} \leftarrow \{\}$

▷ Initialize possible design candidates

{Sample K designs by ϵ -greedy approach}

for $k \leftarrow 1$ **to** K **do**

$s \leftarrow$ initial design graph

while s has non-terminals **do**

With probability ϵ select a random rule a from available rules

otherwise select $a = \arg \max_a V_\theta(P(s, a))$.

$s \leftarrow P(s, a)$

end while

Add possible candidate s to $\mathcal{P}$.

end for

{Choose one to be the candidate}

With probability ϵ select a random sampled design from $\mathcal{P}$ as the candidate design d ,

otherwise select $d = \arg \max_{d \in \mathcal{P}} V_\theta(d)$ by heuristic function V_θ .

{**Evaluation Phase:** Compute the average reward for the design}

Run MPC to evaluate d and get average reward r .

{Update the best design and $\hat{V}$ }

if $r > r^*$ **then**

$s^* \leftarrow d$

$r^* \leftarrow r$

end if

for Each partial ancestor design d_p of d **do**

Update $\hat{V}(d_p) \leftarrow \max(\hat{V}(d_p), r)$.

end for

{**Learning Phase:** train heuristic value function V_θ }

for $i \leftarrow 1$ **to** opt_iter **do**

Sample a minibatch S of seen designs (partial or complete) of size M .

Update $V_\theta(s)$ one step by Adam with the loss:

$$\sum_{s \in S} \|V_\theta(s) - \hat{V}(s)\|^2$$

end for

end for

design to performance, it is necessary to generate a diverse collection of robot designs from which to learn from (beginning with $\epsilon = 1$). The first ϵ -greedy exploration rule, within a given design generation, helps guarantee a diverse collection of possible candidate designs (with the variance of that set parameterized by ϵ). Since K applications of this process (with large K and small ϵ) makes it likely that at least one robot resembling the pure greedy-strategy will be generated, the second ϵ -greedy exploration guarantees that the same best candidate is not chosen each time. ϵ is decreased with each episode toward 0 as the heuristic’s accuracy increases, according to an exponential decay schedule (as in Q-learning); this is made possible by a fast, accurate learned heuristic function, and shifts the algorithm from exploration to exploitation.

Evaluation Phase After a candidate robot has been decided on, its performance must be evaluated. We simulate the candidate robot with actuation inputs generated by the MPC algorithm described in Section 3.5.2. It is possible for the same design candidate to be proposed multiple times. Because our MPC algorithm is sampling-based and stochastic, different average rewards $\hat{V}$ may be seen for the same design between episodes. We consider the $\hat{V}$ of a design to be the best average reward over all evaluations of the design. If this average reward is the best seen so far, the candidate is stored as the current best design, along with $\hat{V}$. Regardless, the candidate robot design and its corresponding $\hat{V}$ are stored (or updated) in a lookup table, and the $\hat{v}$ label of all of that design’s partial design ancestors are updated to be the maximum of their current value and the candidate robot’s average reward. This is important for the upcoming learning phase, which must learn a heuristic function that is valid for both complete and partial designs.

The number of candidate designs evaluated in each iteration is an algorithm design trade-off. Evaluating more candidates will collect more data, helping to train a more accurate prediction function. It will also significantly increase the computation time, however, since evaluation is the time bottleneck in our algorithm. We therefore choose to evaluate only one

design per iteration.

Learning Phase The heuristic is trained using the data stored in the lookup table. For `opt_iter` epochs, minibatches of $(s_i, \hat{V}_i)$ pairs are sampled, and the loss $L = \frac{1}{2} \|V_\theta(s_i) - \hat{V}\|_2^2$ is minimized using Adam [91].

Heuristic Function Model

To implement our heuristic function, we choose to leverage the expressive nature of graph neural networks. Graph neural networks (GNNs) are neural network architectures which aim to extend the benefits of deep learning to a graphical setting. Unlike other neural network types such as CNNs, which operate on images and data with fixed grid-like topologies, graph neural networks aim to be flexible and operate on structures with arbitrary topologies. The input to a GNN consists of a graph topology (*e.g.* an adjacency matrix), and values associated with nodes (*e.g.* feature vectors). While many GNN models have been proposed in recent years, our architecture is based on the differentiable-pooling model. This model was designed for inference tasks involving graphs with a *hierarchical nature*, by iteratively reducing the graph to a “lower resolution” graph in a manner similar to hierarchical clustering. Please see [60] for more details. This model is well-suited for our scenario, where each robot is itself created through a hierarchical substitution of grammar rules.

The differentiable-pooling GNN extends the GraphSage framework from [92], which in turn is based on graph convolutional networks (GCN) [93]. Analogously to CNNs, GCNs apply a generalized convolution operator that operates on graphs rather than grids. We adopt a similar model as [60]. In this model, two “mean” GraphSage+BatchNormalization layers are applied, followed by the hierarchical clustering DiffPool layer, followed by three layers of graph convolutions. This process is repeated one additional time, followed by a final GraphSage layer, a mean pooling layer, and a final ReLU. The output is a positive real-valued scalar representing predicted robot performance. Each DiffPool layer reduces

the node set’s cardinality by 75%.

An important property of this GNN model is that it is isomorphism-invariant, meaning any two isomorphic graphs will have the same value and gradient without the need for explicit transposition tables. This greatly simplifies the bookkeeping in Graph Heuristic Search.

We convert robot design graphs into inputs for the GNN model as follows. The graph is first converted to a kinematic tree, so that each link has a unique joint connecting it to its parent link. Each link and its parent joint is considered a node in this new graph. Note that this representation differs from the one described in Section 3.4. Next, an m -dimensional feature vector is extracted from each node. If the link associated with the node is a terminal symbol, the feature vector encodes the link’s initial position, orientation, and geometric description. The parent joint’s rotation and servo parameters are included similarly, if present. If either the link or joint are non-terminal symbols, the feature vector one-hot encodes the non-terminal type.

3.6.2 Monte Carlo Tree Search

For comparison, we also implement Monte Carlo tree search (MCTS), specifically the UCT algorithm [94]. An outline of our MCTS implementation is given in Algorithm 3. The state of the algorithm is represented by a directed acyclic graph of nodes representing partial designs and associated statistics. Directed edges between the nodes represent actions, or rule applications. MCTS implementations tend to differ in which statistics they store. Because our problem setting is single-player and nonadversarial, we choose to store the visit count and maximum reward. Like Graph Heuristic Search, each iteration of our MCTS implementation incorporates design, evaluation, and learning phases.

Our design phase combines the selection and expansion steps of the standard MCTS iteration. The selection step begins at the root of the tree, representing the start symbol of the grammar. Edges are followed repeatedly until a node with no children (a leaf node) is

Algorithm 3: Monte Carlo Tree Search

Inputs: Number of iterations N , maximum number of simulation attempts M .
for $i \leftarrow 1$ **to** N **do**
 Select node s using UCT formula.
 Expand node s .
 $j \leftarrow 0$ {Number of attempted playouts}
 repeat
 if $j = M$ **then**
 Block node s . {Do not allow s to be selected again}
 Select new node s using UCT formula.
 Expand new node s .
 $j \leftarrow 0$
 end if
 Starting with graph in s , apply randomly sampled rules until only terminal symbols are left.
 $j \leftarrow j + 1$
 until graph is simulable
 Run MPC (Algorithm 1) to obtain result (average reward).
 Update all nodes from s to the root with the result.
end for

reached. When multiple edges are available, the one with the highest UCT score is chosen:

$$\arg \max_{a \in A(s)} \left(Q_{s,a}(t) + \sqrt{\frac{2 \ln N_s(t)}{N_{s,a}(t)}} \right) \quad (3.1)$$

where $Q_{s,a}(t)$ is the maximum result from all iterations where rule a was applied to graph s , $N_s(t)$ is the visit count of s , and $N_{s,a}(t)$ is the number of iterations where a was applied to s . If the last node reached represents a partial design, an expansion step is performed: a randomly selected rule is applied and the node for the resulting design added to the tree. Without further modifying the tree, rules are then randomly selected and applied until a complete design is obtained. The complete design is evaluated in simulation using MPC, in a way similar to Graph Heuristic Search.

In the learning phase, the statistics of nodes in the search tree that were visited during the design phase are updated. Each visited node's visit count is incremented, and its maximum reward is replaced with the evaluation result if it is higher. These updated statistics will

guide node selection in the next iteration.

In addition to the basic UCT algorithm described above, we implement several common enhancements which promote faster convergence. To address the possibility of designs that are not simulable due to self-collision, we apply rejection sampling. These modifications are described in the following paragraphs.

Transpositions Some designs can be reached through multiple different rule sequences (transpositions). To improve efficiency, only one node is created per unique design. Statistics from all iterations that visited a node, regardless of the exact sequence of rules, are aggregated together. Childs, Brodeur, and Kocsis [95] show that this scheme can improve efficiency compared to the basic UCT algorithm for games with many transpositions. An approximate hashing-based scheme is used to detect isomorphic designs with a low probability of false positives.

UCT-RAVE Certain rules may tend to improve performance no matter when they are applied. The UCT-RAVE algorithm [96] takes this insight into account through the all-moves-as-first (AMAF) heuristic [97]. AMAF estimates the value of an action at a node based on iterations that visited the node, like vanilla UCT. However, AMAF also uses iterations where the action is taken anytime in the future, instead of immediately at the node. UCT-RAVE blends between the vanilla Monte Carlo value of an action and the AMAF value of an action according to a schedule. As the visit count of a node increases, the AMAF value is weighted less heavily.

Rejection Sampling A simulation may fail to produce a valid result for a number of reasons, including self-collision in the starting configuration. In that case, the search algorithm tries another random playout from the same leaf node. Note that simply returning a fixed, low result unnecessarily biases the search towards simpler designs which are less likely to self-collide.

There may be partial designs that no sequence of rule applications can convert into simulable designs. All playouts starting from the corresponding nodes will fail to produce a valid result. We address this potential issue by limiting the number of playout attempts for a single node. Upon reaching the limit (set to 100), the node is *blocked* and can no longer be selected. The iteration restarts at the selection step and a different leaf node is expanded.

3.7 Problem Specification

Here, we describe the remaining (user-specified) components needed to define a co-optimization problem with RoboGrammar.

3.7.1 Reward Function

A single reward function (Equation 3.2) is used to evaluate designs on every terrain, and is computed at every time step of the simulation. For the purposes of design optimization we use the average reward across all time steps.

$$r(t) = \vec{w}_x \cdot \vec{d}_x(t) + \vec{w}_y \cdot \vec{d}_y(t) + \vec{w}_v \cdot \vec{v}(t) \quad (3.2)$$

We consider the base link of the robot, or the forwardmost wide body segment, to be representative of the robot’s motion. $\vec{d}_x$ and $\vec{d}_y$ are unit vectors pointing forward and upward in the base link’s reference frame, respectively, and $\vec{v}$ is the base link’s velocity. All quantities are expressed in world coordinates.

$\vec{w}_x = [-2, 0, 0]^T$, $\vec{w}_y = [0, 2, 0]^T$, and $\vec{w}_v = [2, 0, 0]^T$ are weighting vectors which set the relative importance of each term. They also scale the reward function’s magnitude to the range expected by the design search algorithm.

The first two terms encourage maintaining the initial orientation, and the last term rewards forward progress. The robot starts with its local x -axis pointing in the negative x direction in world coordinates.

3.7.2 Terrains

Terrains, in conjunction with the reward function, define tasks to optimize robot structures for. Each terrain is intended to result in a different set of optimal designs.

Flat terrain A featureless surface with a friction coefficient of 0.9, the flat terrain accommodates the greatest variety of locomotion styles.

Frozen lake terrain A flat surface with a low friction coefficient of 0.05, the frozen lake terrain encourages designs which maximize traction or use the low friction to their advantage.

Ridged terrain Ridges or hurdles spaced an average of one meter apart span the entire width of the ridged terrain, requiring designs to jump or crawl in order to make progress.

Wall terrain Walls which are too high to traverse directly are placed in a slalom-like arrangement. Designs must move around the walls, requiring them to change their direction of motion rapidly.

Gap terrain A series of platforms separated by gaps require designs to tread carefully. As the gaps become progressively wider, designs with the ability to take larger steps are favored.

Upward stepped terrain A series of steps resembling a flight of stairs test the ability of designs to climb. The steps are of varying height, producing different gait variations over time.

3.8 Implementation

Before presenting our results, we briefly describe important details regarding the implementation of *RoboGrammar*, including hyperparameter choices and experimental setup.

Table 3.1: Graph Heuristic Search hyperparameter values

Hyperparameter	Value
Number of iterations (N)	2000
Initial ϵ (ϵ_0)	1.0
Final ϵ (ϵ_1)	0.1
ϵ exponential decaying factor (ϵ_{decay})	0.3
Number of possible candidate robots (K)	16
Optimization steps (opt_iter)	25
Optimization batch size (M)	32
Adam learning rate	1×10^{-4}

Table 3.2: Simulation and MPC hyperparameter values

Hyperparameter	Value
Simulation time step (dt)	1/240 s
Default coefficient of friction	0.9
Joint torque limit	1 Nm
Control interval	16 time steps
Episode length (T)	128 control intervals
Number of samples (K)	64
MPC horizon, default (H)	16 control intervals
MPC horizon, wall terrain (H)	32 control intervals
Sample standard deviation, <i>warm-start</i> (σ_1)	0.25
Sample standard deviation, <i>history</i> (σ_2)	0.05

3.8.1 Hyperparameters

We run all experiments with the same hyperparameters unless otherwise specified. The exploration parameter ϵ in GHS follows an exponential decay schedule: $\epsilon(i) = \epsilon_1 + (\epsilon_0 - \epsilon_1) \exp(-\frac{i/N}{\epsilon_{decay}})$, where i is the current iteration, N is the total number of iterations, $\epsilon_0 = 1$, $\epsilon_1 = 0.1$ and $\epsilon_{decay} = 0.3$. We run GHS for 2,000 iterations ($N = 2000$). In the design phase of each iteration, GHS uses the two ϵ -greedy steps to select one design to be tested by MPC from 16 sampled possible candidate robots. In the learning phase, the Adam optimizer runs for 25 steps with batch size 32 and learning rate 1×10^{-4} .

Hyperparameters of the simulation and MPC are either chosen based on our problem setting or are manually tuned. We use the default time step of 1/240 s for Bullet Physics, which is the recommended value for robotics applications. The default friction coefficient

of 0.9 approximates that of rubber on pavement. Joint torques are limited to one Newton-meter, which is within the capability of affordable hobby servos. The MPC control interval, sample count, and horizon are tuned to produce consistently high rewards while limiting computation time. Computation time increases with control interval, sample count, and horizon. Sample standard deviations are tuned to balance progress on terrains with smooth motion. Higher standard deviations result in faster but more erratic motion. A summary of hyperparameters is provided in Table 3.1 and Table 3.2.

3.8.2 Experiment Setup & Computational Time

We implemented our Graph Heuristic Search algorithm in Python, and the simulation and MPC in C++. Experiments were run on VM instances with either 32 or 64 Intel Cascade Lake vCPUs on Google Cloud Platform. Each iteration of GHS spends less than 1 second per possible candidate robot in the design phase, 40-60 seconds in the MPC evaluation phase, and 6-8 seconds in the learning phase. Since each possible candidate robot is sampled independently, the design phase can be fully parallelized for further speedup. The time bottleneck of the MPC evaluation phase shows the necessity of our Graph Heuristic Search algorithm, which is able to find the best-performing robots while evaluating a significantly fewer number of robot designs. Section 3.9.4 describes this efficiency in detail.

Our two baseline search algorithms, MCTS and random search, are also implemented in Python. Both baselines also spend the vast majority of their computation time on MPC, with their design and learning phases (if applicable) taking a negligible amount of time.

3.9 Results

Here we demonstrate *RoboGrammar* on a collection of different problems, examining how terrain affects which designs and controllers are optimal. Furthermore, we examine how designs translate over multiple terrains, analyzing the Pareto set of synthesized controllers

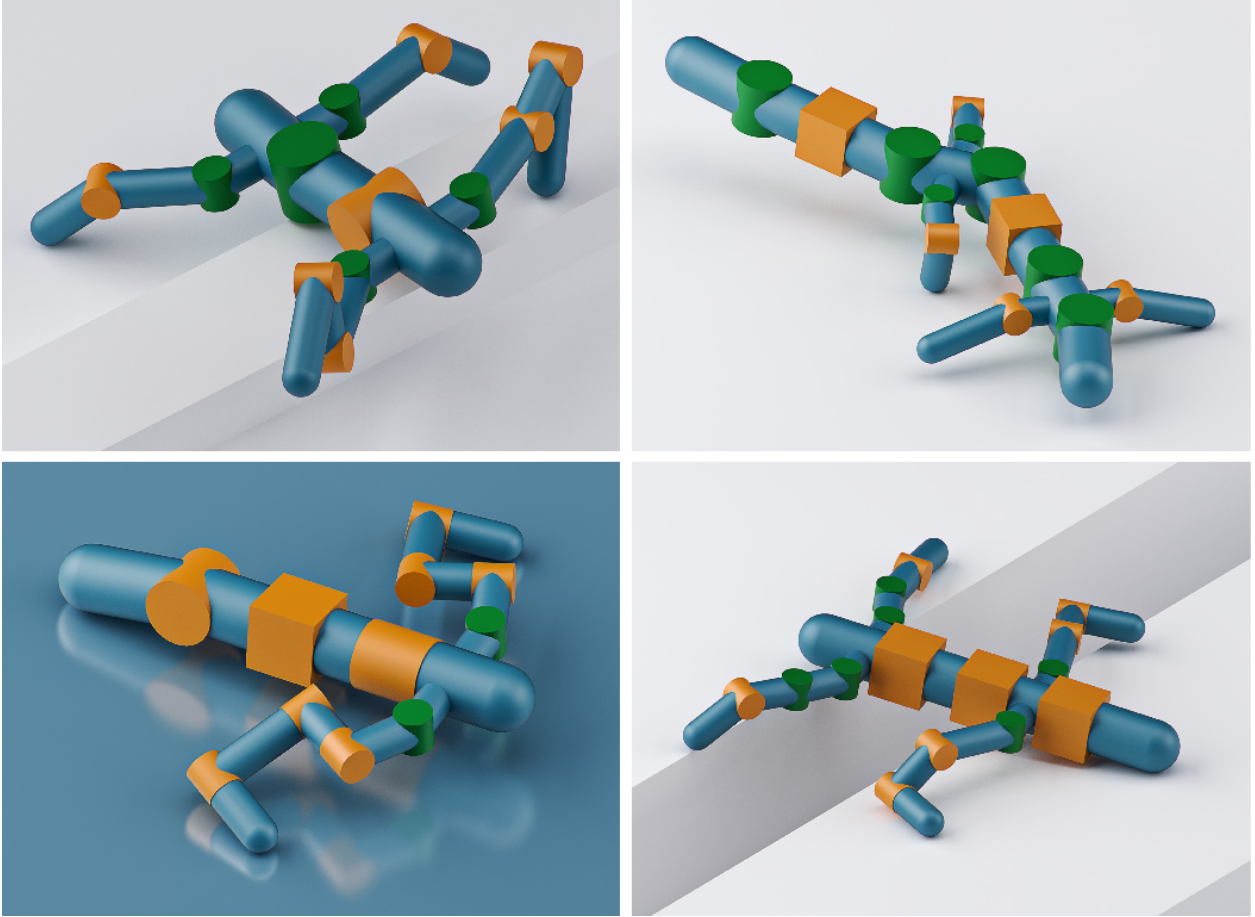


Figure 3.7: Selection of best-performing designs generated with the grammar, and optimized with Graph Heuristic Search and MPC for ridged, flat, frozen lake, and gapped terrain respectively.

over terrain pairs. We qualitatively demonstrate the efficacy of MPC for our problems and quantitatively analyze the efficiency of our Graph Heuristic Search algorithm compared to the baselines.

3.9.1 Terrain Driven Optimization

As an end-to-end demonstration of the *RoboGrammar* pipeline, we run Graph Heuristic Search on several different terrains. Each search run consists of 2,000 iterations. A selection of best-performing designs is shown in Figure 3.7.

Optimal designs for the ridged terrain are characterized by long limbs which are able to

swing upwards and clear obstacles. Although the set of optimal designs consists mainly of quadrupeds, a few tripedal designs emerge. These designs use their body as a third point of contact with the ground.

The flat terrain, despite having no obstacles, still produces specialized designs. One successful strategy is to place short limbs spaced far apart on the body, giving them a full range of motion. Although short limbs would be unable to clear obstacles on most of the other terrains, their low inertia enables quick movement.

The frozen lake is superficially similar to the flat terrain, but its low friction coefficient requires a different strategy. Successful designs can both overcome the low friction and use it to their advantage. The bottom-left design in Figure 3.7 serves as an example. Highly articulated yet compact arms maintain contact with the ground during the stance phase, while the rear body segment slides freely.

The gap terrain tends to produce designs with long limbs, much like the ridged terrain. However, designs for gap terrain tend to have limbs that are optimized for forward reaching instead of climbing. Green joints, which enable limbs to move horizontally, are more prevalent than orange joints, which enable vertical motion.

3.9.2 Pareto Analysis

In addition to single terrain optimization, we also show how our grammar can help in discovering high-performing robots for multiple terrains simultaneously. We choose to use random search for this experiment to avoid biasing the search towards a particular objective. Each random design is evaluated on flat, ridged, and wall terrain. The average reward of designs on two pairs of tasks is plotted in Figure 3.8, with the Pareto optimal designs highlighted. The Pareto sets include a variety of morphologies, each of which offers an optimal trade-off, showing that our grammar is highly expressive. Note that these results were obtained after evaluating only 20,000 random designs. This evaluation count is comparable to evolutionary robot design methods operating on a single objective, showing that our grammar also has a

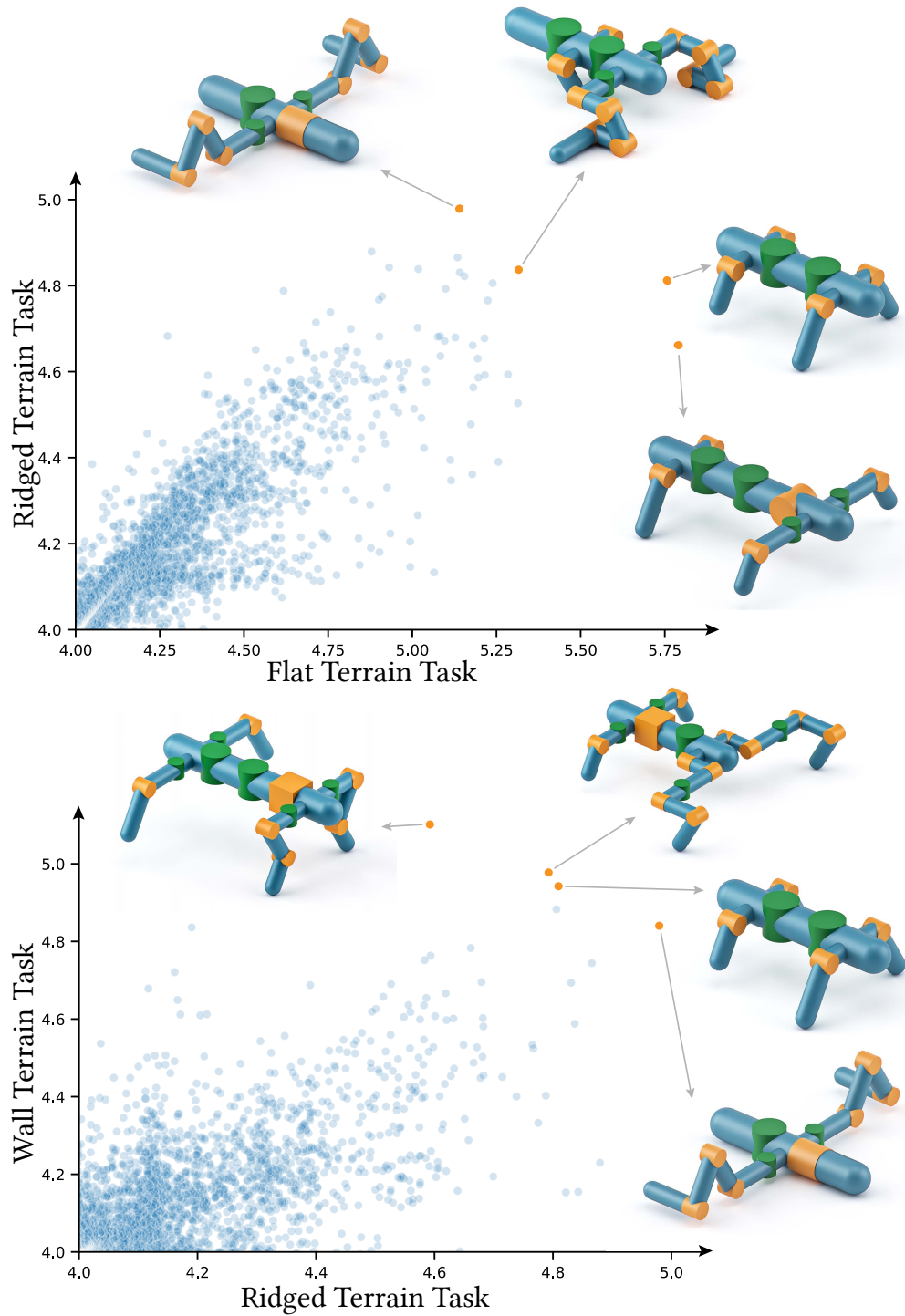


Figure 3.8: Pareto optimal designs for pairs of terrains. Designs offering a variety of trade-offs are discovered using only 20,000 iterations of random search, demonstrating the versatile yet compact search space of our grammar.

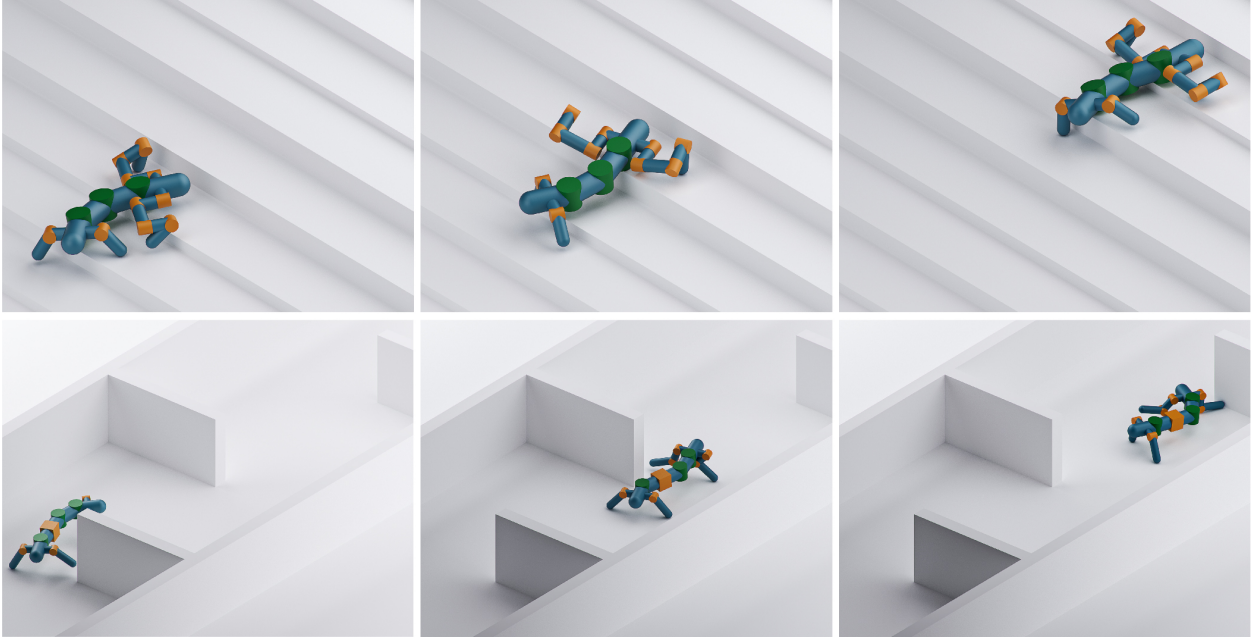


Figure 3.9: Time-lapse showing movement of the high performing robots on upward stepped terrain and terrain with wall obstacles. Each front leg of the first robot consists of 3 elbow joints and a final knee joint forming a hook-like ending to facilitate climbing high steps. The second robot has a long body with many rotational joints and front legs designed for turning stability.

compact search space.

3.9.3 Gait Patterns

In Figure 3.9, we demonstrate that our MPC implementation can generate plausible gaits for challenging terrains. Time-lapses of high performing designs traversing the upward stepped terrain and wall terrain are shown. The robot on the upward stepped terrain initially adopts a cyclic trotting gait. Most steps are low, requiring only one of the front legs and one of the back legs to make contact. Some of the steps are higher, causing MPC to switch to an ad-hoc gait as necessary. The wall terrain-optimized robot also employs a trotting gait, but with exaggerated body movements. Multiple joints in the body allow the robot to curve sharply and change direction rapidly. This can be seen in the last frame of the time-lapse, where the robot turns sharply to the left in order to avoid a wall.

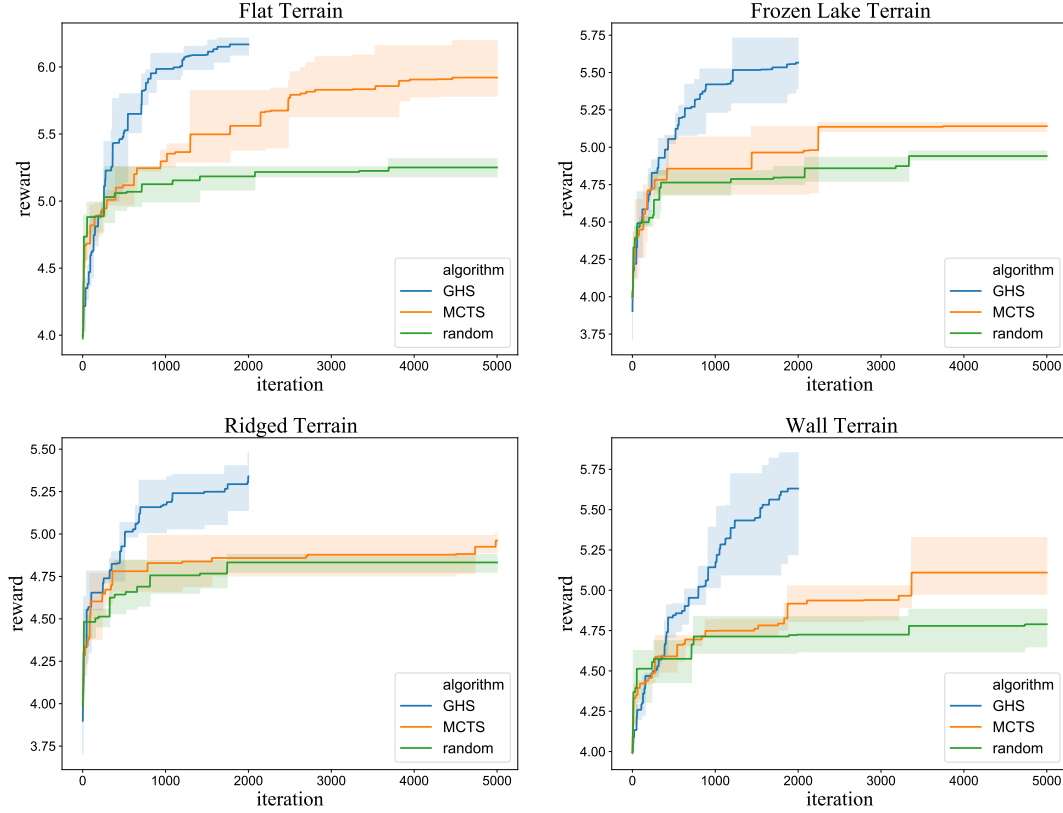


Figure 3.10: Cumulative maximum reward versus iteration for Graph Heuristic Search, Monte Carlo tree search, and random search on four different terrains. Each solid line is the mean of three different seeds, with the error band representing the range. Graph Heuristic Search consistently outperforms the baselines.

3.9.4 Efficiency of Graph Heuristic Search

To show the efficiency of the proposed Graph Heuristic Search algorithm, we compare our algorithm with two baselines on four different terrain tasks (flat terrain, frozen lake terrain, ridged terrain, and wall terrain) described in Section 3.7.2. Specifically, the first baseline is an adapted Monte Carlo tree search (MCTS) algorithm described in Section 3.6.2. The second baseline algorithm is a random search algorithm. In each iteration, one candidate design is selected by applying random rules. Due to the stochasticity of the search algorithms, we run each algorithm on each task at least three times with different random seeds. Note that we run our Graph Heuristic Search algorithm for only 2,000 iterations, as opposed to 5,000 iterations for each baseline algorithm. The results in Figure 3.10 show that Graph

Heuristic Search consistently finds better designs (achieves greater reward) than the baseline algorithms in much fewer iterations. Due to the expensive MPC-based evaluation of designs and the combinatorial nature of our design space, this sample efficiency is key to finding high-performing designs in a reasonable amount of time. The efficiency comes from the ability of the learned heuristic function to generalize knowledge from explored designs to predict the performance of untested designs and effectively prune the search space.

Running 2,000 iterations of Graph Heuristic Search requires approximately 31 hours on a 32-core Google Cloud machine (instance type `n2-highcpu-32`). This computational intensity is comparable to other state-of-the-art robot design methods. For example, Neural Graph Evolution [53] is evaluated using 12 hours on a 64-core machine. Note that we simulate designs with greater numbers of joints on more complex terrains, resulting in more expensive evaluations. Evaluations account for approximately 20 hours out of our total run time.

3.9.5 Convergence of Graph Heuristic Search

To demonstrate that Graph Heuristic Search is agnostic to the specific grammar described (the *standard* grammar), we optimize robots for flat terrain using two modified grammars. The *simple* grammar removes the rules for long limb links and elbow joints (r_9 and r_{17} in Figure 3.4 respectively), whose functionality is provided by other rules. The *asymmetric* grammar increases complexity by allowing opposite limbs to develop independently. Figure 3.11 shows that Graph Heuristic Search consistently converges in training loss, prediction error, and maximum cumulative reward. More complex grammars require a greater number of iterations to achieve the same reward.

3.9.6 Design Space Bias

MCTS and GHS both rely on pruning the search space, so some degree of bias can be expected in the resulting designs. In each given state, MCTS explores every available action once before exploiting a promising branch. We therefore expect the average number of

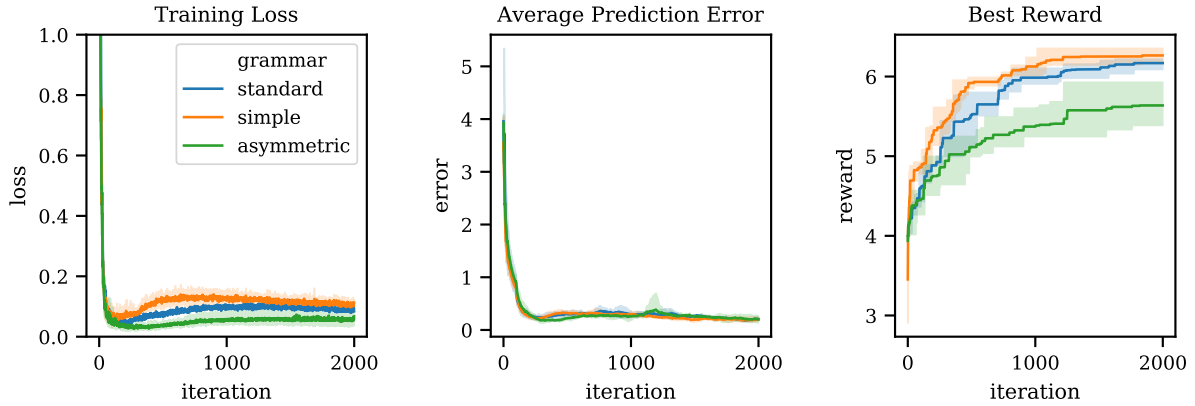


Figure 3.11: Training loss, prediction error, and cumulative maximum reward versus iteration for Graph Heuristic Search on multiple grammars. Robots are optimized for flat terrain. Prediction error is the absolute difference between predicted reward and evaluated reward, and is averaged over 100 iterations. Each solid line is the mean of at least three different seeds, with the error band representing the range. Graph Heuristic Search consistently converges in all three criteria.

derivation steps in designs explored by MCTS to be low. The statistics in Table 3.3 reflect this hypothesis, with designs found by MCTS having fewer derivation steps than GHS on average. When considering all designs explored, random search has the lowest average derivation length. This reflects the fact that random search does not focus its exploration and will visit the same (simple) designs repeatedly.

Table 3.3: Average number of derivation steps in designs evaluated by each algorithm. Results are for flat terrain.

Algorithm	All designs	Best 100 designs
GHS	24.4	25.0
MCTS	24.0	20.9
random	11.4	23.4

3.10 Limitations and Future Work

The example grammar presented in this work only considers bilateral symmetry and joint types that occur in natural systems. To explore designs outside this scope — for example,

sea creatures — the grammar from Section 3.4 could be extended. The grammar has been designed to be easily expandable, and we leave this exploration as an interesting direction for future work.

Model predictive control is able to produce stable gaits across a wide variety of terrains and robot topologies. Dynamic gaits may also exist, but they are unlikely to be found. This is due to the bias introduced by our sampling scheme, as well as the limited torque and high damping of the simulated motors.

Our framework ensures that simulated designs are at least fabricable, or able to be built in their simulated configurations. With the appropriate rules, robot grammars enforce the use of a limited set of components and ensure that the components are connected in a feasible manner. Providing an accurate match between simulated and physical motions is not explicitly addressed in this work, however, and it remains an interesting direction for future research.

Our grammar does not include continuous parameters, because we are focused on optimizing discrete topology. Our framework could easily be extended to handle continuous parameters using attribute grammars, or in a post-processing step after the discrete robot structure is identified.

3.11 Conclusion

Intelligent and efficient generative robot design methods will drive the future of design processes for robotics. In this chapter we presented a novel approach which leverages several different computational tools to provide a new pipeline for efficient and large-scale exploration of robot designs. By developing a graph grammar that allows for a wide variety of robots to be generated, we show that creative solutions can emerge in response to different terrains. We introduce a Graph Heuristic Search algorithm to search the combinatorial search space, and couple it with MPC for evaluation. Unlike many alternative approaches

to generative robot design, this allows us to structure and limit the design space by applying a graph grammar, whilst allowing creative solutions to emerge. Importantly, the emergent designs are observably physically fabricable and there is significant potential for the designs to be translated to real-world scenarios and environments.

Chapter 4

Underwater Robots

4.1 Introduction

Autonomous underwater vehicles (AUVs) have many industrial applications, such as search and rescue, spatiotemporal sampling, sea floor mapping, and offshore oil and gas installations [98], [99]. These applications have driven the design and development of AUVs in a variety of sizes, shapes, thruster configurations, and working depth limits. Optimizing for different mission objectives (such as cost, speed, and endurance) and operational conditions (such as deep sea, seafloor, and extreme weather) translates into significant variation in AUV designs [100]. The resulting design space for AUVs becomes overly expansive and complex for a person to explore without significant expertise. It includes discrete parameters (such as topology and component selection) as well as continuous parameters (such as component locations and dimensions). Engineers designing AUVs for any combination of the conditions mentioned above use methods that tend to rely on their collective knowledge and experience [101]. These manual design approaches offer limited trade-off possibilities.

Prior works on automating the design of AUVs [101]–[106] have limited expressiveness over structures. A key challenge is to find representations that can encode discrete and continuous parameters while supporting effective search strategies. Inspired by previous work

on graph grammars [1], [4], [20], [21], [107] and shape deformation [108]–[110], we present a novel AUV shape representation combining a graph grammar and cage-based deformation. The grammar encodes AUV components and topologies while cage-based deformation allows for continuous shape variation—e.g., changing the length, diameter, or thickness of a hull segment.

We then build upon our single AUV design approach to rapidly design fleets containing multiple vehicle types for tasks subject to heterogeneous constraints—for example, large scale surveying at different depths. Much of the current multi-AUV literature assumes the AUV fleet is homogeneous, or composed of a single vehicle type [111], [112]. Heterogeneous AUV fleets could potentially make full use of individual abilities to satisfy the task requirements with greater efficiency. Prior works on marine exploration using heterogeneous AUV fleets [113]–[115] do not formulate the task as an optimization problem with constraints nor optimize the individual AUV designs along with the multi-AUV task.

In contrast, we begin by optimizing a number of individual AUV designs, each with a different mass limit and depth rating. Each design, consisting of discrete and continuous parameters, is evaluated on a task-specific objective function using a differentiable simulator. Our optimization pipeline uses a combination of discrete graph search and gradient-based continuous optimization to update each AUV’s design parameters to minimize the objective. The result from these previous steps is a diverse library of AUV designs that feature unique pairings of mass and depth rating. Finally, this library is fed into our combinatorial fleet optimization algorithm, which outputs the optimal fleet according to the task specifications.

In the development of our computational fleet design pipeline, we present the following contributions. Our first contribution is a novel graph-grammar for AUVs that captures a diverse space of topologies. Our second contribution is that we formulate heterogeneous fleet design as a modified knapsack problem and propose an efficient, exact algorithm to solve it. We demonstrate our fleet design pipeline on a simulated mission with nonuniform design requirements: surveying a section of seafloor with varying depth.

4.2 Design Space

We describe possible AUV topologies using a custom graph grammar and parameterize the hull shape with deformable cages. A single AUV design is represented as a graph where the nodes are components and the edges are the inter-component connections.

Our AUV grammar is illustrated in Fig. 4.1. Some rules encode components such as underwater thrusters, hull segments, and a parametric battery. Each component has properties based on its real-life counterpart, such as maximum thrust, mass, and energy density. The remaining rules control how nodes in the design graph can expand into subgraphs. Fig. 4.2 illustrates how a design is constructed by applying a sequence of these grammar rules.

The hull shape is composed of multiple segments, each of which is parameterized using a deformation cage. A cage initially forms an axis-aligned bounding box surrounding its segment, and the positions of its corners are functions of the continuous parameters. As the corners of a cage move, the points on the hull inside the cage are recalculated using trilinear interpolation. By combining these deformation cages with our graph grammar, we can represent AUVs with many different topologies and shapes.

4.3 Simulation

4.3.1 Dynamic Model

We first establish the inertial and body frames to describe the motion of an AUV. We adopt the North-East-Down (NED) frame as the inertial frame. The body frame is defined at the center of mass of an AUV with the three axes pointing forward (x), right (y), and down (z). We define an AUV's state as a 12-dimensional vector $\mathbf{s}$:

$$\mathbf{s} = (\mathbf{x}, \Phi, \mathbf{v}_B, \Omega) \quad (4.1)$$

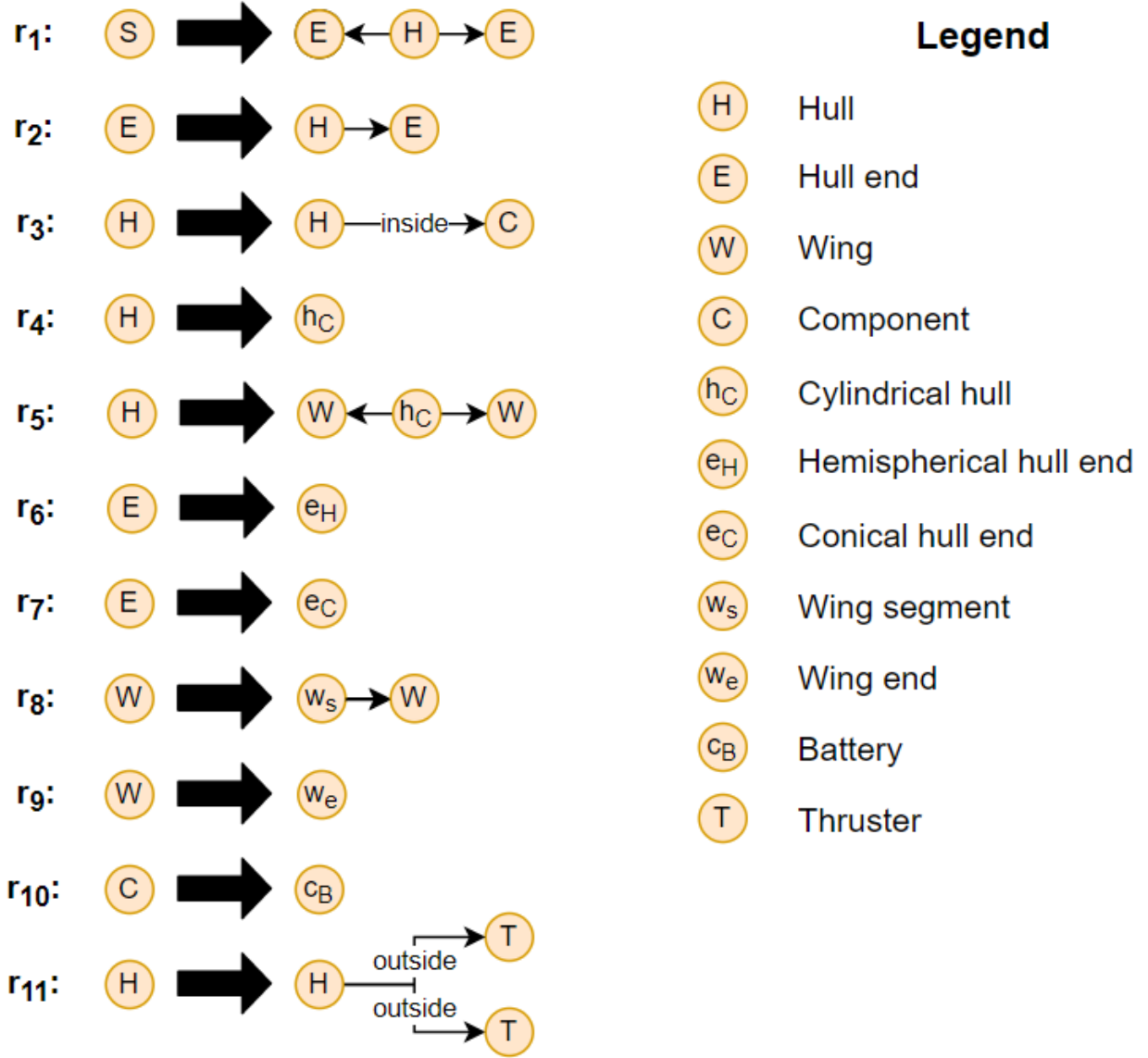


Figure 4.1: The rules comprising our AUV grammar. Symbols are either in uppercase (representing abstract subgraphs) or in lowercase (representing concrete components).

where $\mathbf{x}$ stands for the position of the AUV in the inertial frame, Φ stands for the attitude in the form of Euler angles, $\mathbf{v}_B$ denotes the linear velocity in the body-fixed frame, and Ω is the angular velocity in the inertial frame. The following relation between the defined linear velocities holds:

$$\mathbf{v}_B = \mathbf{T}_I^B \dot{\mathbf{x}} \quad (4.2)$$

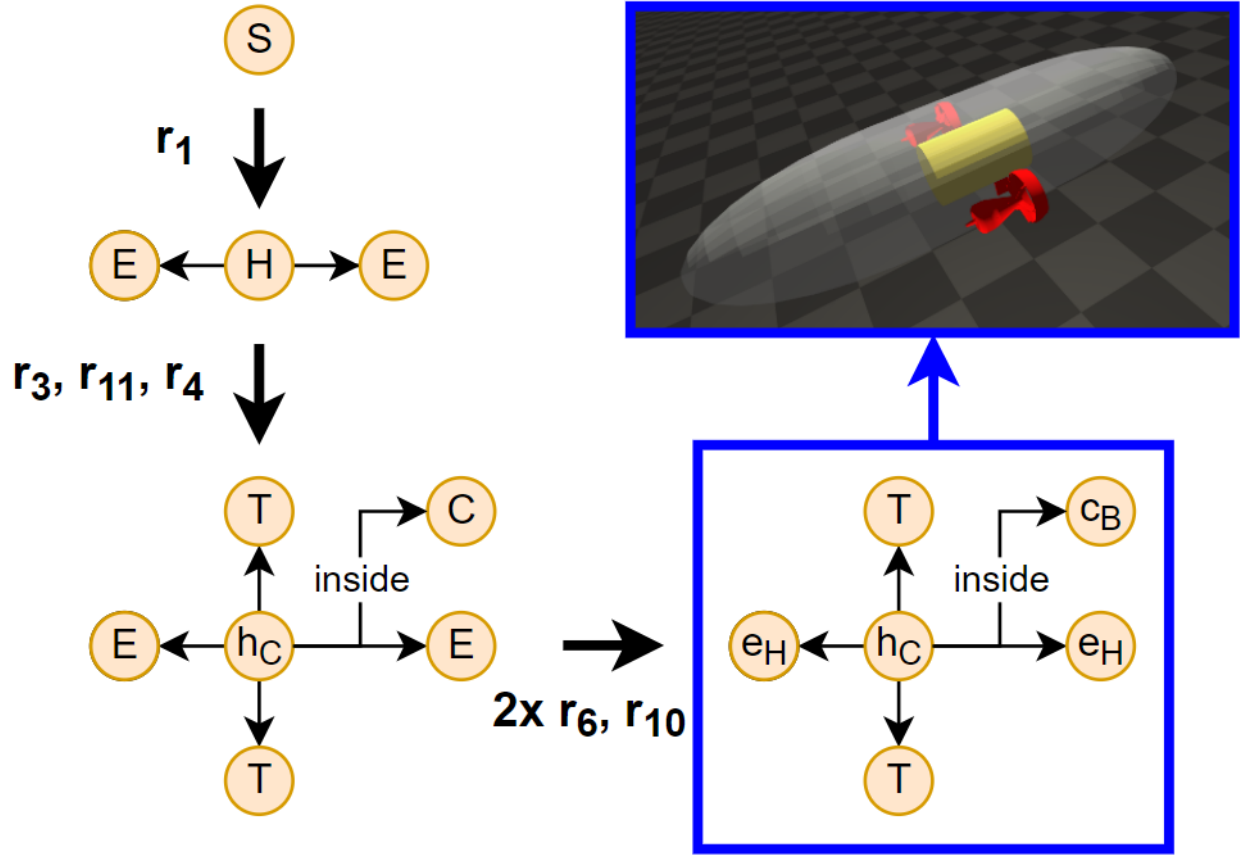


Figure 4.2: Sequential growth of a propeller-driven AUV from applying 7 of our grammar rules.

where $\mathbf{T}_I^B$ is the rotation matrix expressing the transformation from the inertial frame to the body-fixed frame. Moreover, the following relation between the defined angular velocities holds:

$$\Omega = \mathbf{J}(\Phi)\dot{\Phi} \quad (4.3)$$

where $\mathbf{J}(\Phi)$ is a Jacobian matrix. It is useful to collect the kinematic equations in six-dimensional matrix forms. We define the vector $\nu \in \mathbb{R}^6$ as

$$\nu = \begin{pmatrix} \mathbf{v}_B \\ \Omega \end{pmatrix}. \quad (4.4)$$

According to Newton's law and Euler's equations, the overall equations of motion can therefore be written in matrix form as

$$\mathbf{M}_v \dot{\boldsymbol{\nu}} + \mathbf{C}_v(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{D}_v + \mathbf{g}_v(\mathbf{T}_B^I) = \boldsymbol{\tau}_v + \boldsymbol{\tau}_E, \quad (4.5)$$

where $\mathbf{M}_v \in \mathbb{R}^{6 \times 6}$ represents the symmetric positive-definite added mass and inertia matrix; $\mathbf{C}_v(\mathbf{v}) \in \mathbb{R}^{6 \times 6}$ is the skew-symmetric vessel matrix of the Coriolis and centripetal terms; $\boldsymbol{\tau}_E \in \mathbb{R}^{6 \times 1}$ represents environmental disturbances from wind, currents, and waves (which we assume to be zero for simplicity); $\mathbf{D}_v \in \mathbb{R}^6$ is the hydrodynamic damping; $\mathbf{g}_v(\mathbf{T}_B^I)$ represents the forces and moments due to gravity and buoyancy in the body-fixed frame. $\boldsymbol{\tau}_v = [\mathbf{f}^T \ \boldsymbol{\tau}^T]^T \in \mathbb{R}^6$ is the sum of forces and moments exerted by all thrusters. More details of the dynamic model can be found in [98].

For each thruster, we compute its orientation and relative position with respect to the center of mass, which gives us the direction of its thrust and induced torque. The magnitude of the thrust is an optimization variable.

For our hydrodynamic model, we follow the practice from [116], [117] by discretizing the hull geometry into a triangle mesh and computing the lift and drag force on each surface triangle as follows:

$$\mathbf{f}_{\text{drag}} = \frac{1}{2} \rho A C_d(\phi) \|\mathbf{v}_{\text{rel}}\|^2 \mathbf{d} \quad (4.6)$$

$$\mathbf{f}_{\text{lift}} = -\frac{1}{2} \rho A C_l(\phi) \|\mathbf{v}_{\text{rel}}\|^2 \mathbf{n} \quad (4.7)$$

where ρ is the fluid density, A is the triangle's area, $\mathbf{v}_{\text{rel}} = \mathbf{v}_{\text{fluid}} - (\mathbf{v} + \boldsymbol{\Omega} \times \mathbf{x}_{\text{rel}})$ is the fluid velocity relative to the triangle, $\mathbf{d} = \frac{\mathbf{v}_{\text{rel}}}{\|\mathbf{v}_{\text{rel}}\|}$, and $\mathbf{n}$ is the triangle's normal. We ignore the indices of the surface triangles for clarity. $C_d(\phi)$ and $C_l(\phi)$ are the coefficients of drag and lift respectively, where $\phi = \cos^{-1}(\mathbf{n} \cdot \mathbf{v}_{\text{rel}}) - \frac{\pi}{2}$ is the angle of attack. We define $C_d(\phi)$ and $C_l(\phi)$ to resemble typical drag and lift curves, which are symmetric and monotonically

increasing functions respectively:

$$C_d(\phi) = 0.05 \cdot (1 - |\phi|)^3 + 0.05 \cdot 3 \cdot |\phi|(1 - |\phi|)^2 + 1.85 \cdot 3 \cdot |\phi|^2(1 - |\phi|) + 2.05 \cdot |\phi|^3 \quad (4.8)$$

$$C_l(\phi) = -0.8 \cdot (1 - \phi)^3 - 0.5 \cdot 3 \cdot \phi(1 - \phi)^2 + 0.1 \cdot 3 \cdot \phi^2(1 - \phi) + 2.5 \cdot \phi^3, \quad (4.9)$$

The total hydrodynamic damping $\mathbf{D}_v = [\mathbf{f}_H^T \ \boldsymbol{\tau}_H^T]^T$ is computed by adding together the forces on each triangle and their resulting torques:

$$\mathbf{f}_H = \sum (\mathbf{f}_{\text{drag}} + \mathbf{f}_{\text{lift}}), \quad (4.10)$$

$$\boldsymbol{\tau}_H = \sum (\mathbf{x}_{\text{rel}} \times (\mathbf{f}_{\text{drag}} + \mathbf{f}_{\text{lift}})), \quad (4.11)$$

where $\mathbf{x}_{\text{rel}} = \mathbf{x}_{\text{tri}} - CG$ is the position of a triangle relative to the vehicle's center of mass.

Eqn. (4.5) provides us enough information to compute the time derivatives of $\mathbf{s}$. We can now represent the dynamics in a compact form $\mathbf{M}$:

$$\dot{\mathbf{s}} = \mathbf{M}(\mathbf{s}, \mathbf{a}, \mathbf{G}, \theta). \quad (4.12)$$

Here, $\mathbf{a}$ is the action vector consisting of forces exerted by each thruster. The design graph and shape parameters $(\mathbf{G}, \theta)$ determine the configuration of thrusters and the hull geometry. In short, given the current design $(\mathbf{G}, \theta)$, the current state $\mathbf{s}$, and the current action $\mathbf{a}$, the dynamic model $\mathbf{M}$ computes $\dot{\mathbf{s}}$ that evolves the dynamic system.

4.3.2 Trim State and Control

To control an AUV design, we compute a trim state and execute a simple open-loop control strategy. A trim state is a state with zero linear and angular accelerations:

$$(\dot{\bar{\mathbf{x}}}, \dot{\bar{\boldsymbol{\Phi}}}, \mathbf{0}, \mathbf{0}) = \mathbf{M}(\bar{\mathbf{s}}, \bar{\mathbf{a}}, \mathbf{G}, \theta), \quad (4.13)$$

By constantly applying the control action $\bar{\mathbf{a}}$ to the vehicle in the trim state, the vehicle maintains its linear and angular velocity. This results in a straight line motion that allows us to extrapolate the vehicle’s power consumption over a long time horizon and calculate the resulting endurance.

4.4 Optimization

Our pipeline employs multiple optimization techniques to generate an optimal fleet for a given task. In the precomputation stage, we optimize a library of vehicles for multiple mass limits and depth ratings using a combination of discrete search and gradient-based continuous optimization. In the scheduling stage, we then solve a modified knapsack problem using the library of precomputed designs to minimize the mission time under a total mass constraint.

4.4.1 Task Specification

We demonstrate our complete pipeline on a simulated surveying mission, where the vehicles in a fleet work together to scan a region of seafloor with area a_{survey} using sonar. The area being surveyed ranges in depth from 200 to 1000 meters, so at least one of the vehicles must be capable of the full depth. We assume that the survey area is sloped so that there is an equal, infinitesimal fraction of its area at each possible depth.

The goal is to complete the survey using as little time as possible, subject to a total fleet mass constraint m_{total} (which relates to the cost to construct and transport the vehicles).

Table 4.1: Simulation parameters for the surveying mission

Parameter	Value	Description
ρ	1027 kg/m ³	Density of seawater
a_{survey}	4000 km ²	Total survey area
m_{total}	4000 kg	Total fleet mass
d_{transit}	100 km	Transit distance
t_{charge}	24 hr	Charging time
η	0.5	Propulsion efficiency
p_{core}	8 W	Compute and sensor power
p_{sonar}	70 W	Sonar power consumption when active
h_{sonar}	400 m	Sonar scan width
α	0.5	Sonar swath overlap fraction
γ	0.8	Usable fraction of battery capacity

Vehicles start at a shore-based facility, and navigate their way to the survey area located a distance d_{transit} away. The survey does not need to be completed on a single battery charge; vehicles may return to the facility to recharge before resuming the survey. Recharging a vehicle takes a predefined amount of time t_{charge} . The vehicles must return to the shore facility by the end of the mission.

We assume each vehicle travels at one of two possible speeds: v_{transit} when transiting to or from the survey area, and v_{survey} when actively surveying. These speeds may vary between different vehicles in a fleet.

The specific values of the parameters used in our experiments are listed in Table 4.1.

4.4.2 Precomputation of Individual Vehicles

Before running the fleet optimization, we first generate a library of vehicles for multiple combinations of mass limits and depth ratings. The mass limits are chosen so that they roughly form a geometric sequence, while the depth ratings are chosen so that an equal amount of the survey area lies within each depth bin. In our experiments, we select four depth cutoffs $d_1 = 400$, $d_2 = 600$, $d_3 = 800$, $d_4 = 1000$ m resulting in equal areas in each depth bin $\hat{A}_k = 1000$ km². Within each depth bin, we optimize vehicles for mass limits of 200, 300, 500, 1000, 1500, and 2000 kilograms.

For each mass and depth pair, we run our single vehicle design pipeline consisting of a discrete search algorithm paired with gradient-based continuous optimization. The discrete search operates as an outer loop that samples different topologies, and is only concerned with the types of components and how they are connected. Continuous optimization is then responsible for finding locally optimal continuous parameters for each topology. The continuous parameters of a design include not only the positions, orientations, and dimensions of each component θ , but also the steady state speeds in the transit and survey phases $v_{\text{transit}}, v_{\text{survey}}$ and the control inputs $\mathbf{a}$ to the thrusters.

The objective function for continuous optimization is composed of two terms:

- a survey rate term L_{rate} that maximizes the area surveyed by the vehicle per unit time (including transit and charging time between sorties),
- and a penalty term L_{penalty} that helps the optimization converge towards a positive survey time.

Our formulation also includes several constraints:

- linear and angular acceleration are zero in both the transit and survey phases, meaning that the thrust force exactly counteracts the forces from drag and buoyancy,
- the survey time t_{survey} is positive, ensuring that the vehicle has sufficient battery capacity to approach and return from the survey area,
- and the total mass m of the vehicle is no greater than the mass limit m_{max} .

The single-vehicle optimization problem can be formally described as

$$\begin{aligned}
& \min_{\theta, v_{\text{transit}}, v_{\text{survey}}, a} \quad \underbrace{-\frac{a_{\text{sortie}}}{t_{\text{sortie}}}}_{L_{\text{rate}}} + \underbrace{\lambda e^{-t_{\text{survey}}}}_{L_{\text{penalty}}} \\
& \text{s.t.} \quad \dot{v}_{\text{transit}}, \dot{\omega}_{\text{transit}} = \mathbf{0} \\
& \quad \dot{v}_{\text{survey}}, \dot{\omega}_{\text{survey}} = \mathbf{0} \\
& \quad t_{\text{survey}} \geq 0 \\
& \quad m \leq m_{\text{max}}
\end{aligned} \tag{4.14}$$

where $\lambda = 1 \times 10^3$ and $t_{\text{sortie}} = t_{\text{transit}} + t_{\text{survey}} + t_{\text{charge}}$ is the total time per sortie. The objective and constraints, which involve computing the dynamics of the underwater vehicle, are all differentiable and can be optimized by a gradient-based solver.

4.4.3 Fleet Optimization

We now describe our algorithm to find the best fleet (Alg. 4) from the 24 precomputed vehicle candidates in Section 4.4.2, such that the total mass of the fleet is within a limit and the entire region is surveyed in minimal time. Let $\{C_i\} = \{(m^i, d^i, v_{\text{survey}}^i, t_{\text{survey}}^i, t_{\text{transit}}^i)\}$ be the set of vehicles in the fleet, and let N_i be the number of survey sorties scheduled for the i -th vehicle. The fleet optimization problem can be formally described as:

$$\min_{\{(C_i, N_i, \tilde{T}_i)\}} \max_i \tilde{T}_i \tag{4.15}$$

$$\text{s.t.} \quad C_i \in \text{candidate designs} \tag{4.16}$$

$$\sum_i m^i \leq m_{\text{max}} \tag{4.17}$$

$$\tilde{T}_i \geq t_{\text{transit}}^i \times N_i + t_{\text{charge}} \times (N_i - 1) \tag{4.18}$$

$$\tilde{T}_i \leq (t_{\text{transit}}^i + t_{\text{survey}}^i) \times N_i + t_{\text{charge}} \times (N_i - 1) \tag{4.19}$$

Entire survey area is covered

$$N_i \in \mathbb{N},$$

where $\tilde{T}_i$ is the total working time for the i -th vehicle, and (4.18-4.19) are the validity constraints for this surveying time. Specifically, a vehicle has to spend time t_{transit}^i for each sortie, time t_{charge} between two consecutive sorties to recharge, and no more than time t_{survey}^i to scan the area during each sortie.

We perform binary search on the optimal time T^* to complete the survey task, converting the min-max optimization problem in (4.15-4.19) into a feasibility problem with an additional constraint $\tilde{T}_i \leq T^*$. In this way, we find the smallest time limit T^* in which a fleet can complete the mission.

Note that a vehicle with depth rating d^i can survey areas with depth less than or equal to d^i . Given a maximal allowed time T^* , let a^i be the maximal area that can be surveyed by a vehicle i within its depth range. Consequentially, a fleet that can finish the task on time must satisfy:

$$\forall_d \sum_{d_k \geq d} \hat{A}_k \leq \sum_{d^i \geq d} a^i, \quad (4.20)$$

where $\hat{A}_k$ is the total area required to be surveyed at depth d_k .

To compute the maximal survey area a^i for each vehicle, we can formulate the following mixed integer linear programming problem:

$$\begin{aligned} & \max_{N_i, \tilde{T}_i} a^i & (4.21) \\ \text{s.t.} & \\ & a^i = \left(\tilde{T}_i - t_{\text{transit}}^i \times N_i - t_{\text{charge}} \times (N_i - 1) \right) v_{\text{survey}}^i \\ & \tilde{T}_i \geq t_{\text{transit}}^i \times N_i + t_{\text{charge}} \times (N_i - 1) \\ & \tilde{T}_i \leq (t_{\text{transit}}^i + t_{\text{survey}}^i) \times N_i + t_{\text{charge}} \times (N_i - 1) \\ & \tilde{T}_i \leq T^*, \quad N_i \in \mathbb{N} \end{aligned}$$

Note that the analytical solution of (4.21) can be computed using standard algebraic

techniques. Specifically, the maximum is achieved when $N_i = \lfloor \frac{T^* + t_{charge}}{t_{transit}^i + t_{survey}^i + t_{charge}} \rfloor$ or $\lceil \frac{T^* + t_{charge}}{t_{transit}^i + t_{survey}^i + t_{charge}} \rceil$, and $\tilde{T}_i = \min\{T^*, (t_{transit}^i + t_{survey}^i) \times N_i + t_{charge} \times (N_i - 1)\}$.

The feasibility problem can now be regarded as a modified knapsack problem, where each type of candidate vehicle can contribute a^i survey area in its depth range with a weight cost of m^i , and we need to select the correct number of vehicles of each type so that their total surveyed area satisfies (4.20). We convert this modified knapsack problem into a graph search problem and solve it by breadth first search with several pruning techniques.

Specifically, we denote a search node (i, A, M) as achievable if and only if there exists a fleet composed of the first i types of vehicles that can survey an area A within mass budget M . Given an achievable node (i, A, M) , we can consider how many vehicles of type $i+1$ to select. If we select k vehicles of type $i+1$, we can further label the node $(i+1, A + a^{i+1}k, M + m^{i+1}k)$ as achievable. Note that the area constraint (4.20) would be hard to check if we consider the vehicles in arbitrary order, since the whole fleet selection would need to be encoded in our node state instead of just three values (i, A, M) . By sorting the candidate vehicles in descending order of their depths, we can easily prune nodes that violate (4.20) while keeping the node state compact.

Our search algorithm starts with only node $(0, 0, 0)$ being achievable. The feasible solution of the problem exists if and only if there exists an achievable node (n, A, M) where $A \geq \sum_k \hat{A}_k$ and $M \leq m_{\max}$. Since the problem is combinatorial, the search space is extremely large. In addition to the *area constraint pruning* mentioned earlier, we also apply *Pareto pruning* to further speed up the algorithm. Given a set of all achievable nodes after considering the first i types of vehicles, a node $\{(i, A_j, M_j)\}$ can be removed from the set if it is dominated by another achievable node $\{(i, A_k, M_k)\}$; *i.e.*, $A_j \leq A_k$ and $M_j \geq M_k$ and the inequality holds for at least one equation. In other words, we only keep the achievable nodes on the Pareto front of the two metrics A and M after considering each additional vehicle type.

Note that our fleet optimization algorithm provides an exact solution to the problem

described in (4.15-4.19). Only nodes which cannot be a part of the solution are pruned.

4.5 Results

In this section, we provide an end-to-end demonstration of our heterogeneous fleet optimization pipeline, and show that it produces fleets that complete the simulated surveying mission more quickly than fleets composed of a single vehicle type.

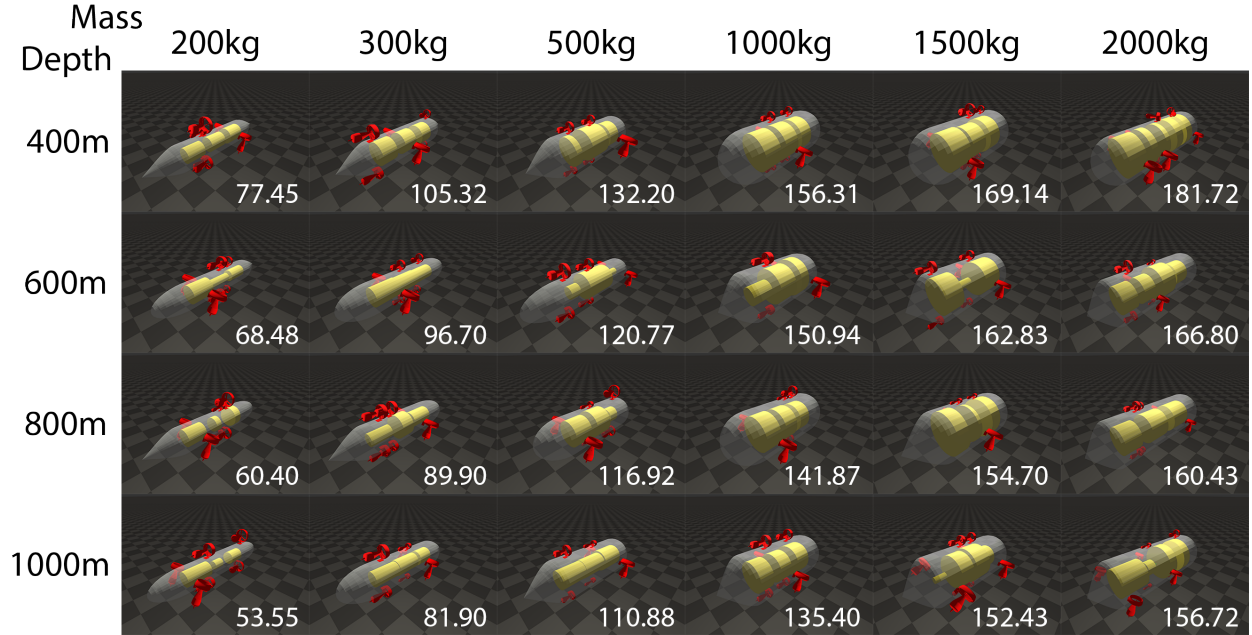


Figure 4.3: The best designs found by our single-vehicle optimization pipeline for each combination of mass limit and depth rating. The survey rate (in m^2/s) is indicated next to each design.

4.5.1 Single-Vehicle Optimization

Figure 4.3 shows the results of the precomputation stage, where we generate a library of designs for multiple mass and depth rating combinations. One of the most noticeable trends is the increasing size and number of battery modules (represented by the yellow cylinders) as the allowed mass increases. The corresponding increase in battery capacity allows higher mass vehicles to spend more time surveying as opposed to recharging, resulting in greater

Algorithm 4: Fleet Optimization

Data: n candidate vehicles, weight limit $m_{\max}$, area to be surveyed at each depth $\hat{A}_k$.

Result: The optimal fleet and the minimum time to complete survey task T .
sort candidate vehicles in descending order of depth.

$T_l \leftarrow 0, T_r \leftarrow$ a large enough number.

```
while  $T_l < T_r - \epsilon$  do                                     /* binary search */
     $T^* \leftarrow (T_l + T_r)/2$ 
    compute  $a^i$  for each vehicle type by solving (4.21).
    // breadth first search
     $success \leftarrow False$ 
     $S = \{(0, 0, 0)\}$                                            /* achievable set */
    for  $i = 1 \rightarrow n$  do
         $S' = \{\}$ 
        for  $(i - 1, A, M) \in S$  do
            for  $k = 0 \rightarrow \lfloor (m_{\max} - M)/m^i \rfloor$  do
                 $S' = S' \cup \{(i, A + a^i k, M + m^i k)\}$ 
            end for
        end for
        remove nodes from  $S'$  by checking area constraint in Eqn. (4.20).
        remove nodes from  $S'$  by Pareto pruning.
        if  $\exists (i, A, M) \in S', A \geq \sum_k \hat{A}_k$  then
             $success \leftarrow True$ 
            save the fleet configuration.
            break
        end if
    end for
    // halve the searching space
    if  $success$  then
         $T_r \leftarrow T^*$ 
    else
         $T_l \leftarrow T^*$ 
    end if
     $T \leftarrow T_l$ 
     $fleet \leftarrow$  last saved fleet configuration.
end while
```

survey rate.

As the required depth rating increases, the thickness and mass of the hull also increases. In order to maintain close to neutral buoyancy, less of the internal volume can be used for batteries. The survey rate reduces accordingly. This trend is noticeable at the higher end of the depth range, as more space is left between the batteries and the hull.

Thruster placement (shown in red) does not appear to follow a clear pattern. One feature stands out however: the upward orientation of some thrusters. Interestingly, the optimization tends to produce negatively buoyant designs that require thrust to maintain a constant depth. Although this may seem counterintuitive (since it results in additional power consumption), the endurance of the vehicles is increased. Reducing the displacement (and therefore mass) of the hull allows allocating more of the mass budget to batteries, resulting in greater endurance compared to a neutrally buoyant design. If recoverability of the vehicles in case of failure is important, an additional constraint enforcing neutral buoyancy (omitted here) could be imposed.

4.5.2 Fleet Optimization

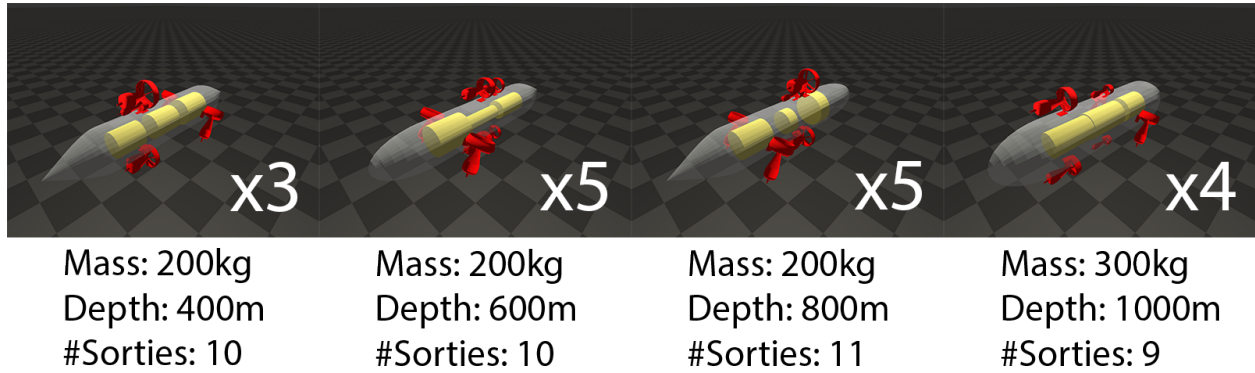


Figure 4.4: The best fleet found by our fleet optimization algorithm using the precomputed design library, along with the number of sorties performed by each vehicle of a given type. The fleet consists of 4 different designs, with the number of instances of each design indicated.

Using the precomputed library of vehicles, we run our fleet optimization algorithm (Alg. 4) to produce an optimal fleet for the mission (Fig. 4.4). We notice that the fleet

Table 4.2: Mission times using a single vehicle type

Number of vehicles	Mass per vehicle (kg)	Mission time (days)
1	4000	187.5
2	2000	97.6
3	1333	72.3
4	1000	59.8
5	800	49.5
6	666	49.0
7	571	46.9
8	500	45.3
10	400	44.0
12	333	49.2
14	285	49.1
16	250	54.4
18	222	57.5
20	200	62.5
22	181	68.3
24	166	76.1

includes a large number of lightweight vehicles as opposed to fewer, more capable vehicles. This makes sense, as overall survey rate of the fleet scales roughly linearly with the number of vehicles. The vehicle type chosen to cover the deepest portion of the survey area is not the lightest possible, however. This may be due to the fact that the hull of the lightest vehicle capable of 1000 m depth weighs 145 kg alone, leaving very little of the mass budget for batteries.

Taking advantage of varying design requirements within the same mission (in our case, the depth rating) enables a better solution composed of multiple vehicle types. The heterogeneous fleet designed by our pipeline completes the mission in only 39 days, as opposed to the best achievable using a single vehicle type under the same constraints: 44 days using 10 identical vehicles of 400 kg each. The results in Table 4.2 suggest that parallelizing across multiple identical vehicles can decrease mission time, but only until a certain point. As the number of vehicles increases (and the mass budget per vehicle decreases), the hull represents a nontrivial fraction of the vehicle mass.

4.5.3 Implementation Details

The majority of our pipeline is implemented in Python using the PyTorch library [118] for autodifferentiation. Performance-critical components of the continuous optimization objective, namely the underwater vehicle dynamics, are implemented in C++ with manually derived gradients.

We choose to use random search as the discrete search algorithm for simplicity, and SLSQP from the NLOpt package [119] for gradient-based continuous optimization. Each iteration of the discrete search, which involves up to 1000 SLSQP iterations, takes an average of 8 seconds per topology on a Google Cloud N2 instance with 24 cores. The continuous optimization is single-threaded, but multiple design searches for different mass and depth combinations can be run in parallel. The entire precomputation stage can be completed in less than 5 hours on the 24-core machine.

With the 24 precomputed designs (Fig. 4.3) as input, our fleet optimization algorithm then produces the same fleet as brute force enumeration in less than a second.

4.6 Conclusion and Future Work

Designing optimal heterogeneous fleets of autonomous underwater vehicles (AUVs) for a given task requires both vehicle-level and fleet-level optimization. Individual vehicles are described by both discrete parameters, such as topology and component selection, as well as continuous parameters such as the placement and sizing of the components.

In this work, we propose a novel shape design space for AUVs that encompasses both discrete parameters (represented in a graph) and continuous parameters associated with the nodes of the graph. We further contribute a complete pipeline for heterogeneous AUV fleet design, and evaluate it on a simulated surveying task covering a range of depths. The resulting solution, consisting of multiple distinct AUV designs, can complete the surveying task in less time than the best fleet composed of a single design.

One possible direction for future work would be a more principled method for selecting the mass and depth bins. We also rely on a simple analytical drag model to estimate the power needed for propulsion. Modeling drag using computational fluid dynamics (CFD) would enable more detailed shape design. Finally, designs in our framework are specified at a high level of abstraction. This allows for mission-level planning, but still leaves many details of fabrication to the human designer.

Chapter 5

Aerial Robots

5.1 Introduction

Improvements in hardware capabilities, perception, and control have led to a rapid uptake of aerial robots across many application domains [120], including remote sensing [121], search and rescue [122], delivery of goods [123] and precision farming [124]. In this work, we focus on designing unmanned aerial vehicles (UAVs), which show significant variation in topology for different applications due to their operational requirements, such as load carrying capabilities, maneuverability, and operational range. For example, quadcopters are preferable [125] for applications requiring hovering, fixed-wing gliders offer high efficiency [126] for long-distance tasks, and hybrid systems, combining both rotor and fixed wings, offer a compromise [127]. The design space for UAVs is therefore expansive, highly varied, and application-specific. It is also particularly challenging to parameterize as it must capture both discrete parameters (which describe the topology) and continuous parameters (which describe, for example, the size of the frame or the coefficients of a controller). Currently, engineers typically perform the design of a UAV manually, utilizing domain-specific knowledge, design instinct, and experience [128].

There have been several computational design tools for automating the UAV design

process [129]–[131]. While these tools make significant advances, they explore only a limited shape space describing a fixed category of UAVs with the same topology, e.g., multirotors with continuously varying sizes [129]. A significant bottleneck in existing methods is the lack of a proper UAV shape representation that allows for both continuous variations and topological changes. Inspired by previous work on graph grammars [1], [4], [20], [21], [107] and shape deformation [108]–[110], we present a novel, hybrid UAV shape representation combining a new graph grammar and cage-based deformation. Our proposed graph grammar consists of a set of carefully chosen rules that encode various UAV topologies, including quadrotors, fixed-wing planes, and hybrid UAVs. Moreover, the cage-based deformation allows for continuous variations of individual UAV components, e.g., changing the thickness of a frame shell or the wingspan of a UAV. Similar to previous work [129], we further equip a UAV shape with an LQR controller whose coefficients are trainable continuous parameters. The shape design space and the LQR controller design space define our co-design space of UAVs’ shapes and controllers with both discrete and continuous parameters.

Unlike existing papers which only optimize continuous parameters in UAV designs [129], [130], our work presents a hybrid design space. To find an optimized UAV design for a given task, we need a novel and effective search algorithm to explore both the discrete topologies and the continuous parameters. We present a hybrid algorithm that combines the benefits of discrete search and gradient-based continuous optimization. Specifically, we run gradient-based continuous optimization algorithms for a fixed UAV topology to tune the continuous shape and control parameters, with the gradient information from a differentiable UAV simulator we developed in this work. With the continuous parameters optimized, we then run discrete search strategies to explore various UAV topologies. We demonstrate the efficacy of our algorithm on a few flight tasks, and our experimental results show that it can automatically re-discover canonical UAV designs typically handcrafted by engineers. In addition, our algorithm reveals novel UAV designs with performances comparable to or better than these canonical designs, highlighting the advantage of our computational UAV

co-design pipeline.

Our work makes the following contributions in developing and implementing this computational UAV design pipeline: First, we present a novel UAV co-design space enabling both discrete and continuous variations in UAV shapes. Second, we provide an effective, hybrid search algorithm that co-optimizes the discrete and continuous design parameters, together with a differentiable UAV simulator we developed. Finally, we evaluate our computational design pipeline on dynamic flight tasks to demonstrate its value in both automatically re-discovering canonical UAV designs and revealing new higher-performance designs.

5.2 System Overview

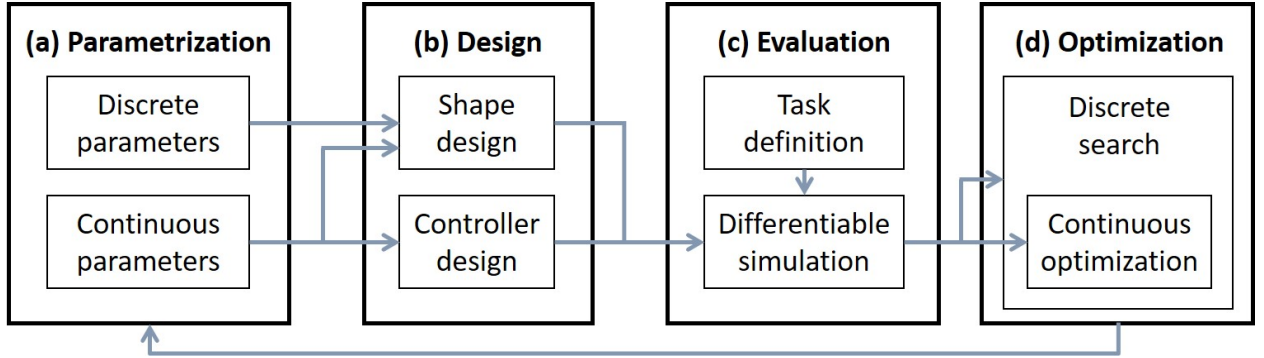


Figure 5.1: An overview of our computational UAV design pipeline. (a): Our pipeline starts with a set of discrete and continuous parameters. (b): The design parameters determine the UAV shape (Sec. 5.3) and controller (Sec. 5.4). (c): We evaluate our design in a differentiable simulator for given tasks (Sec. 5.5). (d): The evaluation is then fed into our optimization algorithm (Sec. 5.6), which runs both discrete search and continuous optimization to update the design parameters.

We present an overview of our computational UAV design pipeline in Fig. 5.1. Our design consists of discrete and continuous parameters (Fig. 5.1 (a)). The discrete parameters refer to the choices of the graph grammar rules, which determine the topology of the UAV shape (Fig. 5.1 (b) top). The continuous parameters consist of the cage variables and the LQR coefficients, which define the shape of individual UAV components and the controller (Fig. 5.1 (b) bottom), respectively. Once we define the design, we evaluate its performance on a given

task (Fig. 5.1 (c) top) using a differentiable simulator (Fig. 5.1 (c) bottom). Finally, we use the evaluation result to guide the discrete search and perform gradient-based optimization (Fig. 5.1 (d)) to update the design parameters (Fig. 5.1 (a)).

5.3 Shape Design

Our shape design space can represent many existing UAV designs of various sizes and topologies by incorporating both continuous and discrete design parameters. Below, we explain them in detail.

5.3.1 Continuous Parametrization

We first define a library of UAV components, e.g., rotors, wings, rods, and batteries. The location of each component in a UAV design is parametrized by continuous parameters, e.g., position and orientation of a motor or installation angle of a wing. Furthermore, we utilize a deformation-cage representation [108]–[110] that enables continuous variations of the geometry of certain components. This is especially useful for expressing various wing designs with different cross sections. More concretely, we immerse a default wing shape in an axis-aligned bounding box and store the relative position of each surface point from the wing (Fig. 5.2, left). By varying the eight corners of the bounding box, we recalculate the position of each surface point by a trilinear interpolation using its stored relative position as the weight (Fig. 5.2, right).

5.3.2 Graph Grammar

The above continuous parametrization allows us to change smoothly the position, orientation, or geometry of an individual UAV component. To assemble these components into a UAV, we now introduce a graph-based grammar that allows us to express various UAV topologies. The grammar, illustrated in Fig. 5.3, includes a starting symbol and other non-terminal and

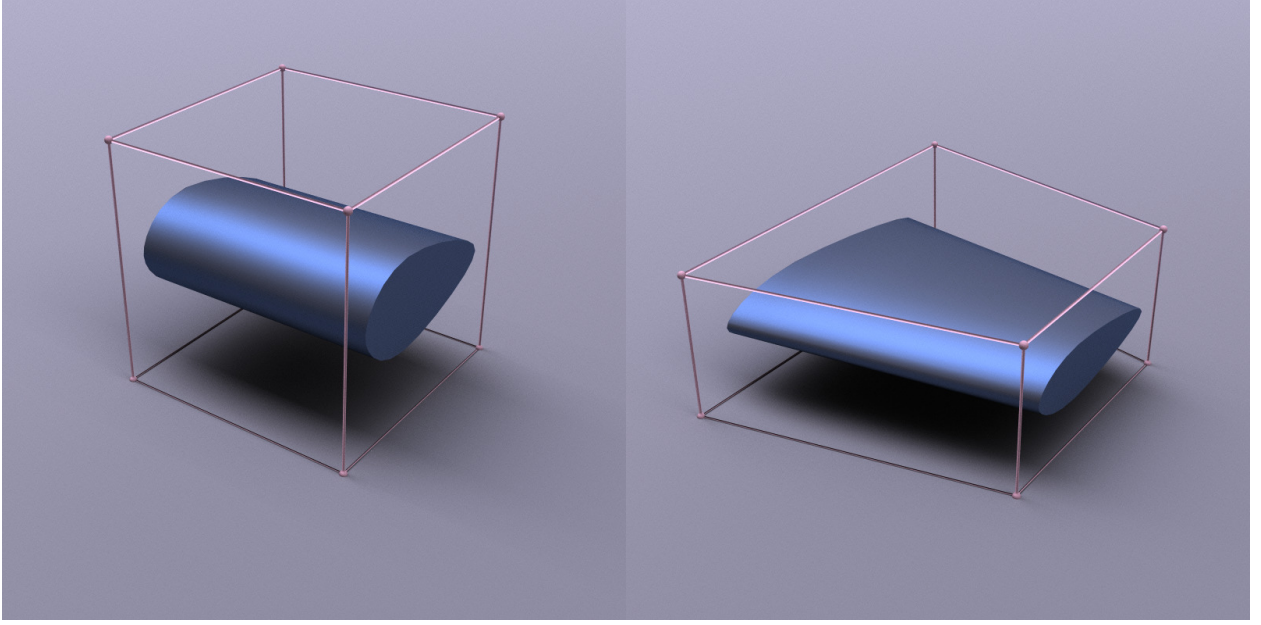


Figure 5.2: Illustration of a wing segment immersed in a deformation cage (left) and its new shape after the handles of the cage are moved (right).

terminal symbols. It also includes a number of rules which define the growth of a graph G from the starting symbol. These rules are of the form of $A \rightarrow B$ where a non-terminal symbol A is replaced by B , which contains terminal or non-terminal symbols enabling the recursive growth. We repeatedly apply rules to grow a graph from the starting symbol and terminate when no more non-terminal symbols exist in the graph. In our UAV designs, each terminal symbol is a component from the UAV component library with a continuous parametrization described before, and the rules define the permissive connections between these components.

As an example, we present the progressive update of a graph that leads to a standard quadrotor design and highlight the final design graph in Fig. 5.4.

To summarize, we represent a UAV shape with a graph G and the union of the continuous parameters in its terminal symbols. We stack these continuous parameters into a column vector θ and represent a UAV shape as a (G, θ) pair. The graph G determines the topology of a UAV, and the vector θ controls the size and shape of each individual component. The (G, θ) parametrization defines a mixed continuous-discrete shape space which grows combinatorially

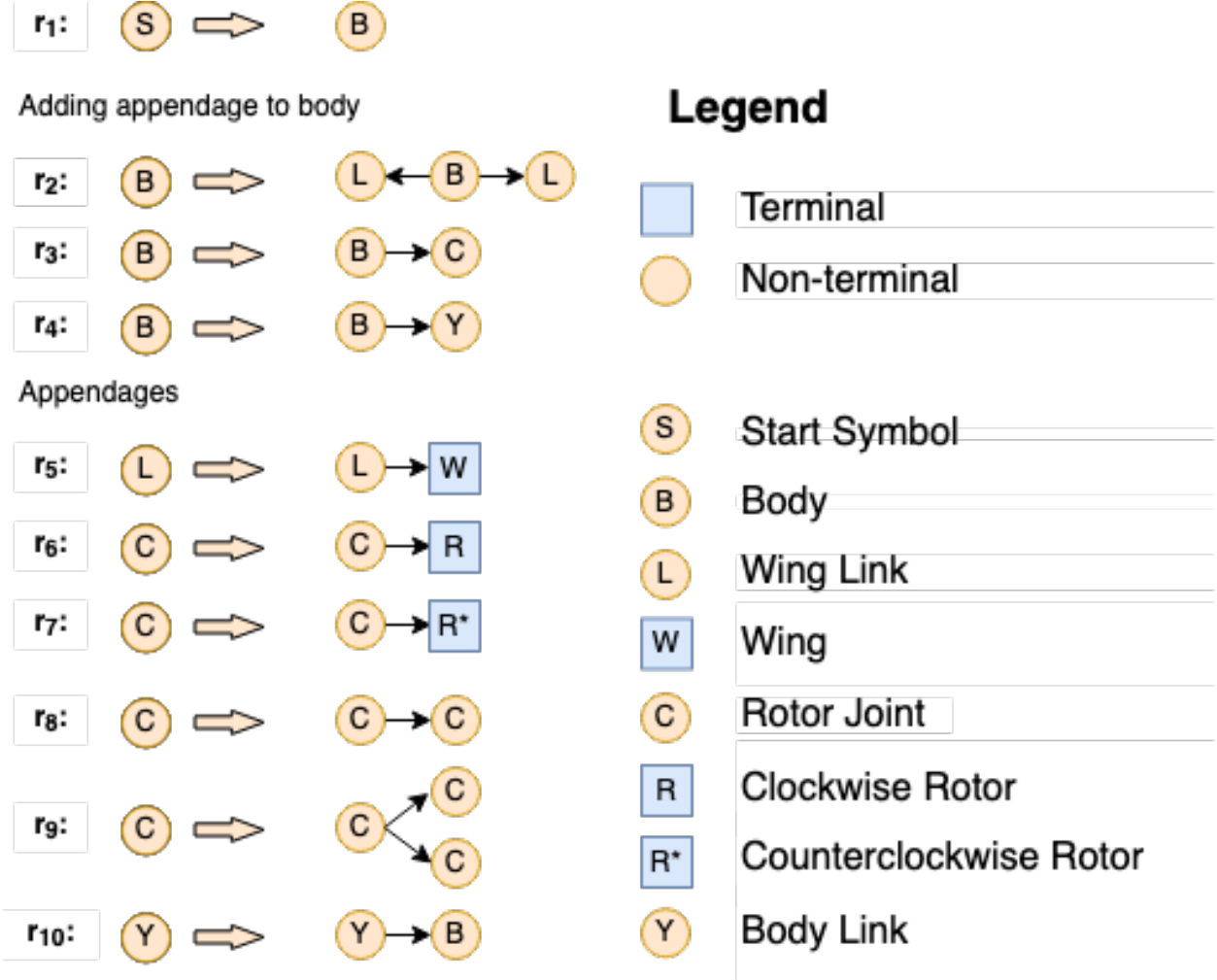


Figure 5.3: Rules of our UAV grammar. S, B, L, C, Y are non-terminal symbols. Rule r_1 initializes the body structure, while $r_2 \sim r_4$ can be used to extend the body. Note that each body segment B can have at most one pair of wings attached to it. Rule r_5 connects the body with the wing. Rules r_6 and r_7 produce clockwise and counterclockwise rotors. Rule r_8 and r_9 extend rotor joints with one or two additional rotor joints. Rule r_{10} extends the body.

with the number of grammar rules.

5.4 Controller Design

For any flight task, a high-performing UAV requires not only a proper shape design but also a suitable controller. In this work, we focus on controlling the dynamic motion of a UAV using the Linear-Quadratic Regulator (LQR), a classical control method that stabilizes a

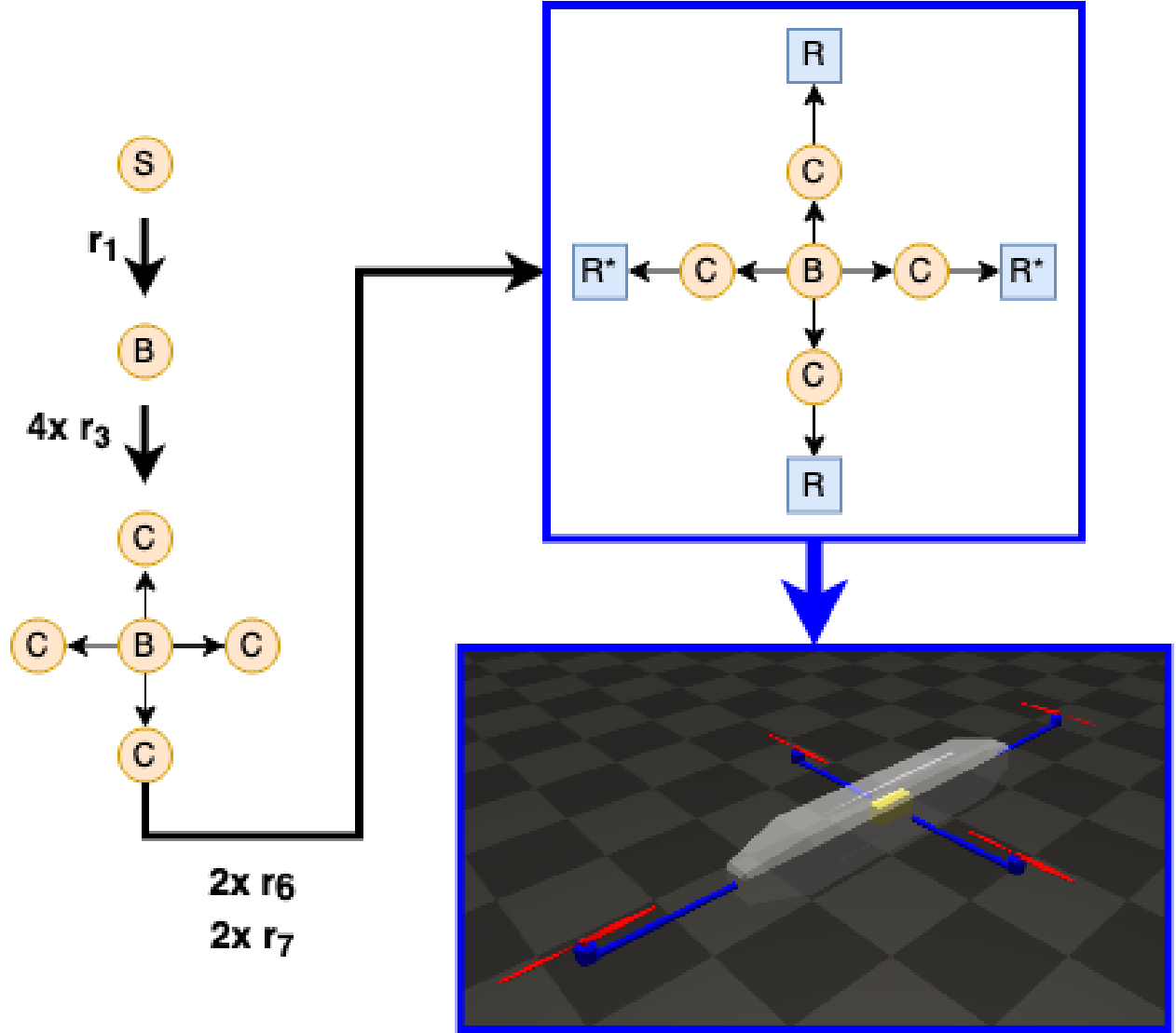


Figure 5.4: The derivation sequence for a quadrotor using our graph grammar. This design uses 9 rules in total. We visualize the partial design graphs in between rules and the final quadrotor design after applying all rules. The non-terminal and terminal nodes in the graphs are indicated by circles and squares, respectively.

UAV around its steady states, also known as trim states. In this section, we first describe the dynamic model of our UAV designs. Next, we explain our method for finding a trim state of a given UAV and associating it with an LQR controller.

5.4.1 Dynamic Model

We first define the world and body frames for describing a UAV's motion. We use the standard North-East-Down (NED) frame system in aerodynamics as the world frame. The body frame defined on a UAV is a 3D coordinate system rigidly attached at its center of mass with the three axes pointing forward (x), right (y), and down (z). We define a UAV's state by its linear and angular positions and velocities [26], which form a 13-dimensional vector $\mathbf{s}$:

$$\mathbf{s} = (\mathbf{x}, \mathbf{q}, \mathbf{v}_B, \omega). \quad (5.1)$$

Here, $\mathbf{x}$ is the world position of the UAV's center of mass, $\mathbf{q}$ is the orientation expressed as a quaternion, $\mathbf{v}_B$ denotes the linear velocity of $\mathbf{x}$ in the body frame, and ω is the angular velocity in the world frame. The time derivative of $\mathbf{s}$ is defined by Newton's second law and Euler's equations stated below:

$$m\ddot{\mathbf{x}} = m\mathbf{g} + \mathbf{f}, \quad (5.2)$$

$$\mathbf{I}\dot{\omega} + \omega \times \mathbf{I}\omega = \tau, \quad (5.3)$$

where $\mathbf{g}$ is gravitational acceleration, m the mass, $\mathbf{I}$ the moment of inertia in the world frame, and $\mathbf{f}$ and τ the net forces and torques exerted from all rotors, wings, and the fuselage in the world frame. Computing m is straightforward: we simply sum the component masses scaled by their change in volume. Computing the moment of inertia $\mathbf{I}$ is more involved as components may be scaled, translated, and rotated. To compute $\mathbf{I}$, we first find each component's moment of inertia in its body frame followed by applying the corresponding transformation. Next, we use the parallel axis theorem to update the moment of inertia after shifting the transformed component to its location in the body frame. Accurate moments of inertia are important, as the orientation of a UAV is usually the state component that is

most sensitive to control inputs.

To compute $\mathbf{f}$ and τ , we take into account each rotor's orientation and relative position with respect to the center of mass to find the direction of its thrust and induced torque. The magnitude of the thrust is given by the controller to be described shortly. For the fuselage and the wings, we sum the lift and drag forces from each triangle in their deformed meshes as described in [116], [117]:

$$\mathbf{f}_{\text{drag}} = \frac{1}{2}\rho AC_d(\phi)\|\mathbf{v}_{\text{rel}}\|^2\mathbf{d} \quad (5.4)$$

$$\mathbf{f}_{\text{lift}} = -\frac{1}{2}\rho AC_l(\phi)\|\mathbf{v}_{\text{rel}}\|^2\mathbf{n} \quad (5.5)$$

where ρ is the air density, A is the triangle's area, C_d and C_l are polynomial functions of the angle of attack ϕ given in [2], $\mathbf{v}_{\text{rel}}$ is the relative air velocity, $\mathbf{d} = \frac{\mathbf{v}_{\text{rel}}}{\|\mathbf{v}_{\text{rel}}\|^2}$, and $\mathbf{n}$ is the triangle's normal. We add the resulting net force and torque to $\mathbf{f}$ and τ , respectively. Inertial and gyroscopic effects of rotor motion are omitted since the rotors comprise a small fraction of the overall mass.

Eqns. (5.2, 5.3) give us enough information to compute the time derivatives of $\mathbf{s}$, which we compactly represent as the dynamic model M , whose definitions are given in previous work [26]:

$$\dot{\mathbf{s}} = M(\mathbf{s}, \mathbf{a}, G, \theta). \quad (5.6)$$

Here, $\mathbf{a}$ is the action vector consisting of the magnitude of thrust sent to each rotor. As stated above, the shape parameters (G, θ) influence the dynamic model, e.g., by determining a rotor's direction or a wing's size. In short, given the current design (G, θ) , the current state $\mathbf{s}$, and the current action $\mathbf{a}$, the dynamic model M computes $\dot{\mathbf{s}}$ that evolves the dynamic system.

5.4.2 LQR Control

After building the dynamic model for a given UAV sampled from our shape space in Sec. 5.3, we associate it with an LQR controller parametrized by its trim state $(\bar{\mathbf{s}}, \bar{\mathbf{a}})$ and its $\mathbf{Q}$ and $\mathbf{R}$ matrices. A trim state is a state with zero linear and angular accelerations:

$$(\dot{\bar{\mathbf{x}}}, \dot{\bar{\mathbf{q}}}, \dot{\bar{\mathbf{v}}}_B, \dot{\bar{\omega}}) = \mathbf{M}(\bar{\mathbf{s}}, \bar{\mathbf{a}}, \mathbf{G}, \theta), \quad (5.7)$$

$$\dot{\bar{\mathbf{v}}}_B, \dot{\bar{\omega}} = \mathbf{0}, \mathbf{0}. \quad (5.8)$$

Once a trim state is found, we linearize the dynamic model $\mathbf{M}$ at the trim state and use the $\mathbf{Q}$ and $\mathbf{R}$ matrices to compute the LQR gain matrix $\mathbf{K}$ from the Continuous Algebraic Riccati Equation (CARE). The action $\mathbf{a}$ is then determined by the control policy $\mathbf{a} = -\mathbf{K}(\mathbf{s} - \bar{\mathbf{s}}) + \bar{\mathbf{a}}$.

Our controller design introduces a number of continuous parameters $(\bar{\mathbf{s}}, \bar{\mathbf{a}}, \mathbf{Q}, \mathbf{R})$. These control parameters and the shape parameters $(\mathbf{G}, \theta)$ are the decision variables we consider in the problem of co-designing a UAV. Additionally, the controller design introduces to the co-design problem the trim-state constraints described in Eqns. (5.7, 5.8).

5.5 Simulation

To evaluate the performance of a given UAV shape and controller, we developed a differentiable UAV simulator based on the dynamic model described in Sec. 5.4.1. Given an initial state $\mathbf{s}_0$ of the UAV design, we compute the action $\mathbf{a}_0$ from its LQR controller and integrate Eqn. (5.6) using the fourth-order Runge Kutta method (RK4). The chosen time step of $\Delta t = 1/60$ s was sufficiently small to avoid instability while maintaining performance. With a loss function defined on the sequence of states at each time step, we backpropagate through each time step to compute its gradients with respect to all the continuous parameters: θ , $\bar{\mathbf{s}}$, $\bar{\mathbf{a}}$, $\mathbf{Q}$, and $\mathbf{R}$. This allows us to optimize them with classical gradient-based numerical optimization methods. Our simulator also provides the option to add random wind forces

at each time step, which helps us evaluate the robustness of UAV designs and controllers.

5.6 Optimization

Given a specific flight task, we describe the problem of finding an optimal UAV design as the following numerical optimization problem:

$$\min_{\mathbf{G}, \theta, \bar{\mathbf{s}}, \bar{\mathbf{a}}, \mathbf{Q}, \mathbf{R}} L(\{\mathbf{s}_i\}, \{\mathbf{a}_i\}, \bar{\mathbf{s}}, \bar{\mathbf{a}}), \quad (5.9)$$

$$s.t. \quad \mathbf{s}_0 = \bar{\mathbf{s}}, \quad (5.10)$$

$$\mathbf{a}_i = \bar{\mathbf{a}} - \mathbf{K}(\mathbf{s}_i - \bar{\mathbf{s}}), \quad (5.11)$$

$$\mathbf{s}_{i+1} = \mathbf{s}_i + \Delta t \mathbf{M}(\mathbf{s}_i, \mathbf{a}_i, \mathbf{G}, \theta), \quad (5.12)$$

$$\mathbf{K} = \text{CARE}\left(\left.\frac{\partial \mathbf{M}}{\partial \mathbf{s}}\right|_{\bar{\mathbf{s}}}, \left.\frac{\partial \mathbf{M}}{\partial \mathbf{a}}\right|_{\bar{\mathbf{a}}}, \mathbf{Q}, \mathbf{R}\right), \quad (5.13)$$

$$\text{Trim conditions from Eqns. (5.7, 5.8)}, \quad (5.14)$$

$$\text{Bound constraints on } \theta, \bar{\mathbf{s}}, \bar{\mathbf{a}}, \mathbf{Q}, \mathbf{R}. \quad (5.15)$$

We now explain the above definition in detail. The objective of the flight task is described as a loss function L defined on the sequences of states $\{\mathbf{s}_i\}$ and actions $\{\mathbf{a}_i\}$ where $i = 0, 1, \dots$ indicates the indices of time steps. We use the trim state $\bar{\mathbf{s}}$ as the initial state $\mathbf{s}_0$ (Eqn. (5.10)). At the i -th time step, we call the simulator with the current state $\mathbf{s}_i$ and compute the action $\mathbf{a}_i$ from the LQR controller (Eqns. (5.11) and (5.13)). $\mathbf{a}_i$ is clipped to respect the same bounds as $\bar{\mathbf{a}}$. It is worth mentioning that the trim state $\bar{\mathbf{s}}$ in Eqn. (5.11) may be updated from time to time depending on the task definition. For example, in the task of waypoint navigation, $\bar{\mathbf{s}}$ is occasionally updated to the next waypoint on the flight trajectory. The new state $\mathbf{s}_{i+1}$ at the next time step is then computed from $\mathbf{s}_i$, $\mathbf{a}_i$, and the dynamic model $\mathbf{M}$ (Eqn. (5.12)). Note that we use forward Euler time integration to write Eqn. (5.12) for simplicity, and our implementation uses a more sophisticated RK4 integrator as mentioned in Sec. 5.5. Finally, we include trim state constraints and bound constraints on

the continuous parameters (Eqn. (5.15)).

The above optimization problem is challenging to solve because it mixes both discrete and continuous parameters. We propose a hierarchical algorithm to separate the discrete and continuous parameters in the optimization problem. At a high level, our algorithm repeatedly samples new graphs G from the graph grammar, evaluates the loss of each G , and returns the UAV design with the lowest loss. For each sampled graph G , we fix the graph parameter in the optimization problem and reduce it to a purely continuous optimization problem. We then run Sequential Quadratic Programming (SQP), a gradient-based numerical optimization method, to optimize all continuous parameters until SQP reaches a local minimum or exhausts the computation budget. This numerical optimization step associates a given G with the best loss found for that graph, allowing us to employ a discrete search strategy on top of that to improve the choice of G . We provide two discrete search strategies in our algorithm: a random and greedy search which randomly samples the graph G and only stores the G with the lowest loss, and a graph heuristic search (GHS) strategy [1], [107] which trains a graph neural network to predict the loss of each G to guide the search process. We summarize the whole optimization algorithm in Alg. 5.

Algorithm 5: Optimization algorithm

Inputs: a loss function L , a maximum number of graph proposals N .
Output: UAV shape and controller design parameters: $G, \theta, \bar{s}, \bar{a}, \mathbf{Q}, \mathbf{R}$.
 $L_{\min} \leftarrow +\infty$
for $i \leftarrow 1$ **to** N **do**
 {Run discrete search to propose a graph}
 Sample a graph G_i from the grammar
 {Run continuous optimization}
 $\theta_i, \bar{s}_i, \bar{a}_i, \mathbf{Q}_i, \mathbf{R}_i \leftarrow \text{SQP}(G_i)$
 {Update the best loss for G }
 $L_i \leftarrow L(G_i, \theta_i, \bar{s}_i, \bar{a}_i, \mathbf{Q}_i, \mathbf{R}_i)$
 if $L_i < L_{\min}$ **then**
 $L_{\min} \leftarrow L_i$
 $G, \theta, \bar{s}, \bar{a}, \mathbf{Q}, \mathbf{R} \leftarrow G_i, \theta_i, \bar{s}_i, \bar{a}_i, \mathbf{Q}_i, \mathbf{R}_i$
 end if
end for

5.7 Experiments

5.7.1 Implementation Details

The differentiable simulator is written in C++ with a Python wrapper, and the remainder of our pipeline is implemented in Python. Specifically, we use PyTorch Geometric [132] to implement the graph neural networks and the SLSQP solver from NLOpt [119] to optimize the continuous design parameters. We run all of our experiments on Google Cloud N2 instances with 80 cores and 64G or 80G of memory. Our discrete search is parallelized across all CPU cores.

5.7.2 Task Specification

We consider four flight tasks requiring a UAV to perform dynamic motions: StraightLineTask, TurningTask, TakeoffTask, and TransitionTask. Each task is described by its loss function defined on a one-second-long flight, which we state in detail below:

StraightLineTask This task requires following a horizontal and straight trajectory at a predefined speed while maximizing energy efficiency. 1 kg is added to the UAV’s center of mass to represent a payload requirement, which we found necessary to encourage the use of wings. Specifically, we consider the following objective:

$$L = \underbrace{I_b(\{\mathbf{s}_i\}, \{\mathbf{a}_i\})}_{L_{\text{energy}}} + \gamma \underbrace{\max_i \|\mathbf{x}_i - \mathbf{x}_i^{\text{ref}}\|_2}_{L_{\text{pos}}}. \quad (5.16)$$

The loss function consists of two terms. The first term L_{energy} estimates the battery current I_b , where V_k is the effective, PWM voltage applied to motor k , n_k is the motor RPM, K_v is the motor velocity constant, R_w is the motor winding resistance, and $e = 0.95$ is ESC

efficiency:

$$I_b = \sum_k \frac{V_k - n_k/K_v}{R_w e} \quad (5.17)$$

Minimizing L_{energy} encourages energy-efficient flight. The second term in the loss function penalizes the discrepancy between the desired and actual location of the UAV at each time step. In this task, the reference position $\mathbf{x}_i^{\text{ref}} = (i\Delta t \dot{\mathbf{x}}(0), 0, 0)$, i.e., the UAV is expected to fly along the x -axis at a constant speed equal to the trim speed. Putting them together, L prefers a UAV design that can fly steadily along the x -axis while maximizing its energy efficiency. The coefficient γ represents our preferences over the two objectives, and we use $\gamma = 10$ in all tasks.

TurningTask Our TurningTask has an objective similar to the StraightLineTask above but with two differences: the payload requirement is removed, and the reference position $\mathbf{x}_i^{\text{ref}}$ is now evenly distributed on an arc whose radius is 5 meters. This task prefers a UAV design that can closely track the arc without consuming much battery power.

TakeoffTask This task focuses on optimizing the takeoff behavior of a UAV. Its objective is identical to the TurningTask except that the reference point $\mathbf{x}_i^{\text{ref}}$ is now evenly distributed on a two-meter-long vertical trajectory. Additionally, we force the initial velocity to be $\mathbf{0}$ so that the UAV must accelerate from a standstill and catch up to the reference points above.

TransitionTask Our last task is a concatenation of takeoff and horizontal flight tasks. The objective function is defined the same as in the TurningTask except that the reference point $\mathbf{x}_i^{\text{ref}}$ is evenly distributed along a two-meter-long vertical trajectory followed by a two-meter-long horizontal trajectory. We design this task to encourage a UAV design that can combine the benefits from quadrotors and fixed-wing planes: quadrotors are suitable for vertical takeoff and precise tracking but consume more battery power, while fixed-wing planes may

lower L_{energy} but lead to a much larger L_{pos} due to reduced maneuverability.

5.7.3 Our Results

We now present the UAV designs optimized by our computational pipeline. For each task, we run our pipeline described in Alg. 5 with both random search and GHS and visualize the optimized UAV shape designs they find in Fig. 5.5. We refer interested readers to our supplemental material for the flight videos of these UAV designs in simulation. Furthermore, we report their final losses for each task in Table 5.1 and the optimization progress of both random search and GHS in Fig. 5.6. For both random search and GHS, we run 4000 iterations in Alg. 5, i.e., $N = 4000$. For all four tasks, the overall search algorithm finishes within 15 to 20 hours.

Our first observation from Fig. 5.5 is that our pipeline discovers high-performing UAV designs with topologies that are quite different from any traditional designs. The novelty and variety of these UAV topologies highlight the benefits of incorporating our new graph grammar to express and explore the discrete shape design space. In particular, our graph grammar induces a hybrid shape space that flexibly mixes rotors and wings in a single design, as can be seen in the optimized design for our StraightLineTask. It is also interesting that the optimized designs often exhibit a topology that matches an engineer’s instinct. For example, wings tend to be favored in the StraightLineTask due to the emphasis on horizontal flight. In the TakeoffTask and TransitionTask however, the high-performing designs use multiple rotors without wings, as wings hinder vertical takeoff.

A second observation is that although we run our algorithm for up to 4000 iterations, we notice that both random search and GHS search strategies reduce the loss drastically in the first few hundreds of iterations, after which the loss plateaus. This observation reveals specific properties of the design space induced by our graph grammar: it is not so difficult to find a UAV design with a near-optimal performance for these tasks when searching in this design space. However, a lot more effort is needed to push the performance limit to its

extreme. Furthermore, users of our pipeline can use this observation as an empirical rule for choosing the number of iterations N when running Alg. 5.

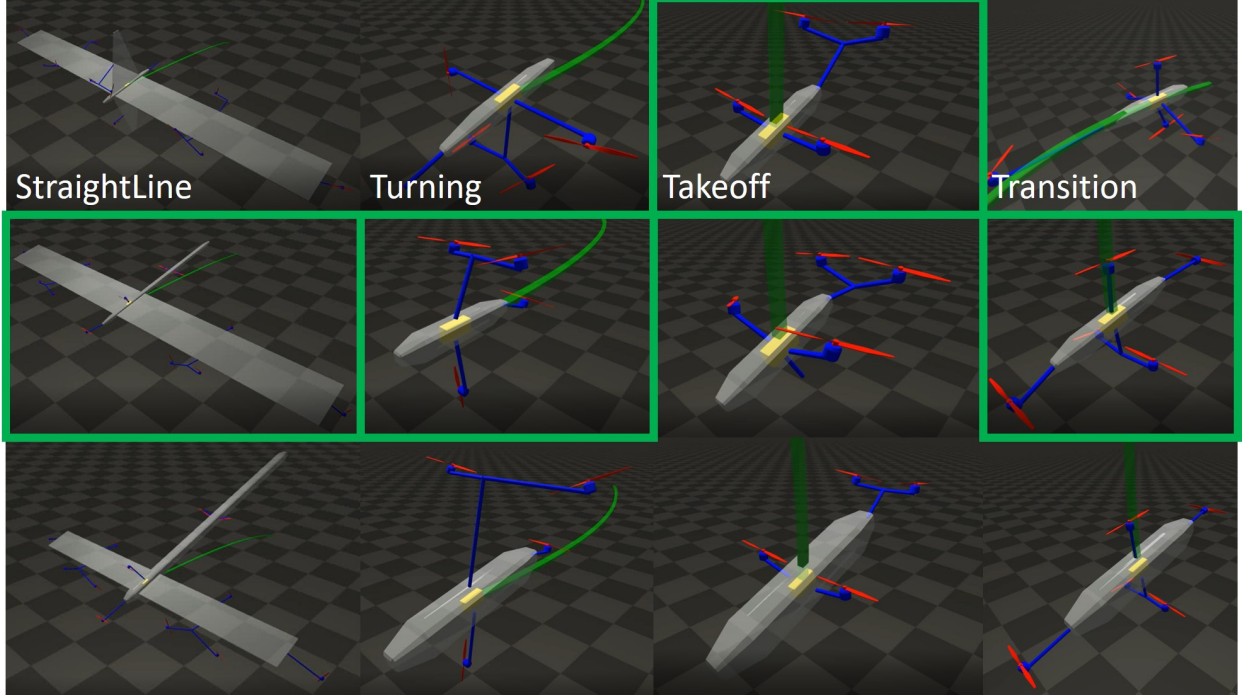


Figure 5.5: Optimized UAV shapes discovered by our computational pipeline using random search (top row) and GHS (middle row) for each of the four tasks (left to right). For each task, the bottom row shows the better UAV from the two rows above (measured by their final loss in Table 5.1 and highlighted in green) with its deformation cages reset to their default positions.

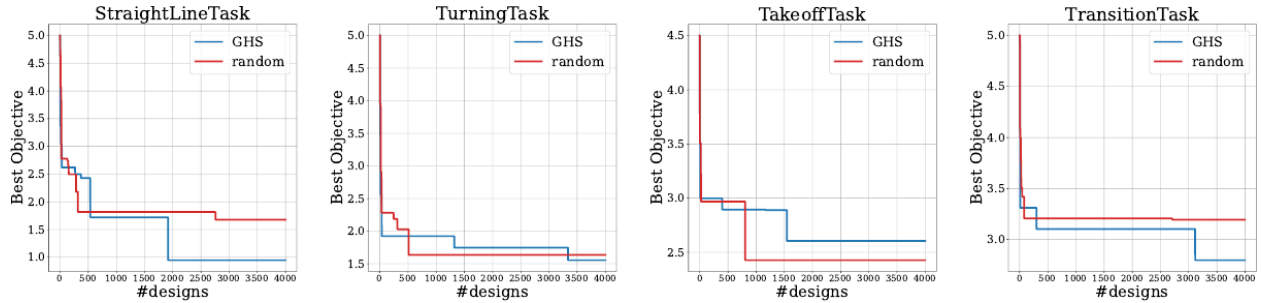


Figure 5.6: The optimization progress plots of our two search strategies: random search and GHS. The horizontal axis is the number of designs sampled by the corresponding discrete search algorithm and optimized by SQP, and the vertical axis is the objective of the best design as the optimization proceeds. We generate the curves from three individual runs for each algorithm on each task.

5.7.4 Discussion

Classic designs To show the advantage of searching over topologies using our computational UAV design pipeline, we compare the performances of the best designs discovered by our pipeline with a number of classic UAV designs (Fig. 5.7) that we handcraft using our grammar rules. For each classic design, only the continuous parameters are optimized while the topology remains fixed. We report their performance on each task in Table 5.1. The performances of these classic designs can serve as a sanity check for the validity of our task definitions. For example, in our TakeoffTask, we notice that the two quadrotors have similar losses (2.48 and 2.30) while the octacopter has a much higher loss (3.47). This is consistent with the definition of the loss function: as copters are suitable for precise position tracking, the loss largely comes from its power consumption, and an octacopter should consume much more battery power than quadrotors because it has more rotors (leading to greater mass). Similarly, in our StraightLineTask, both fixed-wing planes perform better than the rotor-based designs because we expect fixed-wing planes to be more efficient at carrying heavy payloads.

From the table, we can conclude that both random search and GHS manage to reveal UAV designs that outperform the classic UAVs in three tasks, as indicated by their lower losses, and achieve comparable performance on TakeoffTask. Only upward thrust is required to solve the TakeoffTask, which may favor simpler designs like the quadrotors. The traditional, handcrafted designs appear to be suboptimal for the other three standard flight tasks, suggesting that our automatic, computational UAV design method is justified in exploring non-traditional UAV topologies. In fact, as can be seen in Fig. 5.5, most of the high-performing UAV designs discovered by our method have quite unusual topologies that are not covered by any of the classic designs in Fig. 5.7. These novel topologies would be difficult to find using previous UAV design methods without our graph grammar.

Table 5.1: Performance of classic UAVs and the optimized UAVs discovered by our method on each task. Each column reports the loss values on a given task with the best number highlighted in bold (lower is better). For the fixed-wing UAV in our TransitionTask, we report N/A because no trim states are found after trying 10 random seeds in the continuous optimization problem.

UAVs	StraightLine	Turning	Takeoff	Transition
X-quadrotor	4.26	1.98	2.48	3.82
Plus-quadrotor	4.23	8.87	2.30	3.63
Octacopter	4.16	2.90	3.47	4.86
Fixed-wing	3.98	2.28	7.18	N/A
Double fixed-wing	3.78	2.47	5.46	5.01
Ours (random)	1.68	1.63	2.43	3.20
Ours (GHS)	0.95	1.55	2.60	2.80

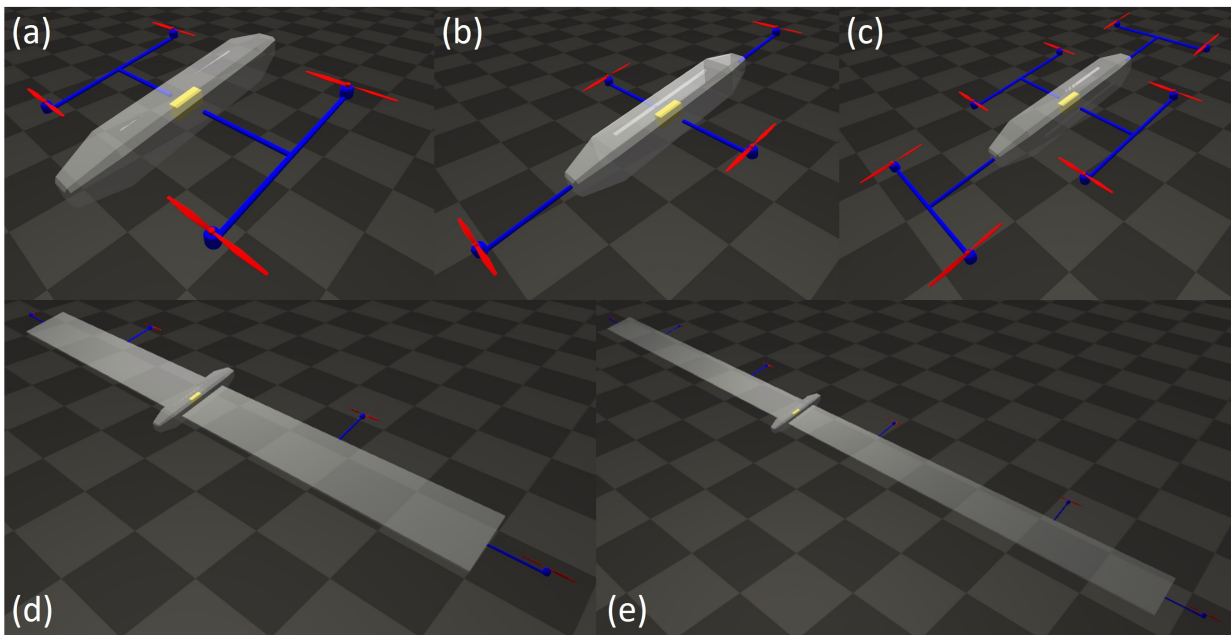


Figure 5.7: The five classic designs we use as baselines in Table 5.1. Top: (a) X-quadrotor, (b) Plus-quadrotor, (c) Octacopter. Bottom: (d) Fixed-wing plane. (e) Double fixed-wing plane.

Discrete search To compare the performances of the random strategy and our GHS strategy on discrete topology search, we test both strategies on all four tasks with three individual runs. Each individual run searches for 4000 UAV designs. We report the objective loss of the best designs discovered by each algorithm in Table 5.1 and plot the optimization

progress curves in Fig. 5.6. From Table 5.1, we can see GHS is able to find better designs than random search within the same design budget in three out of four tasks. This might be due to the additional learning phase in each GHS iteration that progressively trains a prediction network (i.e., heuristic function) from the searched designs [1]. In this way, the learned prediction network can guide the search towards the high-performing designs in our large, combinatorial search space. Fig. 5.6 shows the advantage of GHS more clearly. On all four tasks, the random strategy nearly converges to its best performance in a very early stage. In contrast, GHS keeps improving the best design as it learns a more accurate prediction function as the algorithm proceeds.

It is worth mentioning that random search also performs considerably well in our tasks because most high-performing designs (as shown in Fig. 5.5) have relatively simple structures which can be easily sampled by a random strategy. We still keep GHS as an alternative option for discrete search since as demonstrated in previous work [1], [107], GHS shows a huge performance gain over random search when the optimal designs tend to have more complicated structures.

Deformation cages To see the benefits of incorporating deformation cages in our design space, we restrict the deformation cages in the optimized UAVs found by our pipeline to their default positions during the search process (Fig. 5.5, bottom). We report the performances of the optimized UAVs with and without deformation cage optimization in Table 5.2.

It is apparent that optimizing the deformation cages allows for greater performance compared to the case where cage parameter optimization is not allowed. The improvement is especially profound in the StraightLine task, which is understandable because the best design found leverages a pair of large wings whose shapes are entirely controlled by deformation cages. This observation confirms the benefit of incorporating deformation cages in the shape design space.

Table 5.2: Performances of the best UAVs discovered by our method with and without optimization of their deformation cages. For each task, we choose the better of the two designs found by GHS and random search. Each column reports the value of the loss function in a given task with the better case highlighted in bold (lower is better). Allowing optimization of cage parameters results in substantially better performance.

UAVs	StraightLine	Turning	Takeoff	Transition
Ours	0.95	1.55	2.43	2.80
Ours (no cages)	3.03	2.22	3.26	3.23

Controller robustness Finally, to validate the robustness of our controller design, we simulate the best design found for each task with random wind gusts of a fixed speed. The direction of the wind is sampled uniformly from the unit sphere, and changes every 0.25 seconds. We report the maximum position tracking error L_{pos} for each best-performing design in this noisy simulation environment (Table 5.3). We normalize the position tracking error by dividing it by L_{pos} in the clean environment. In other words, $L_{\text{pos}} = 1$ indicates the UAV has comparable performance in the noisy and clean environments. The copter-like designs for the TurningTask, TakeoffTask, and TransitionTask are relatively insensitive to wind gusts due to their minimal surface area. Their rotor configuration and low mass allows them to counteract disturbances quickly. In comparison, the winged design optimized for the StraightLineTask (and its cargo carrying requirement) is sent off course much more easily. All controllers manage to recover however, and allow only a finite amount of deviation.

Table 5.3: The position tracking error L_{pos} of the optimized UAVs discovered by our method when subjected to simulated wind conditions. The numbers are rescaled so that $L_{\text{pos}} = 1$ is the position tracking error in a clean simulation environment (wind speed = 0).

Wind speed (m/s)	StraightLine	Turning	Takeoff	Transition
0.5	2.40	1.04	1.00	1.15
1.0	2.93	1.07	1.01	1.28
2.0	3.86	1.21	1.02	1.65

5.8 Conclusions and Future Work

Designing UAVs is challenging as the design space is large and difficult to parameterize. We present a computational UAV design pipeline that tackles this problem with a novel graph grammar and a mixed continuous-discrete optimization strategy. Our pipeline manages to discover unusual UAV designs that substantially outperform traditional UAV designs in three out of four evaluation tasks, while matching their performance in the fourth. We verify the robustness of the UAV shapes and LQR controllers co-designed by our pipeline in noisy simulation environments.

While the approach presented in this work produces high-performing aerial robot designs, there are several limitations that we leave as future research directions. First and foremost, our pipeline evaluates all UAV designs in simulation only and does not consider the details of fabrication. Although previous works show that some topologies discovered by our pipeline are indeed plausible to build [129], [130], we can consider how to compile resulting designs to fabrication plans that allow efficient manufacturing. A second limitation in our pipeline is the restriction of the design space to only rigid body structures, while many state-of-the-art aerial robots use articulated structures [133] or even soft or bio-inspired materials and components [134]. Expanding the shape design space to incorporate such features could help us find even higher-performing designs. A third limitation is the assumption that all aerodynamic components such as rotors and wings are independent of each other. Although neglecting interactions between these components can still result in successful sim-to-real transfer [130], a more detailed treatment could produce designs that are even more likely to perform well when fabricated. Finally, the flight tasks we consider in this work have a short time horizon. In future work, we plan to consider tasks with longer time horizons requiring multiple distinct motions or more dynamic maneuvers.

Chapter 6

Lessons and Future Work

This thesis builds on previous efforts in robot co-design by addressing multiple, previously disparate robot design problems using a common representation: graph grammars. At the same time, its work is hardly complete. While graph grammars are expressive and capable of enforcing validity, they still require some degree of manual tuning. Additionally, the generated robots remain entirely within the realm of simulation. Finally, the problems of sensor selection and placement are not examined. The remainder of this chapter expands upon these limitations and suggests potential directions for further research. It also shares some lessons learned from preliminary explorations.

6.1 Graph Grammar Learning

While many aspects of our robot co-design workflow are automated, designing the robot graph grammar itself is still a manual process. Ideally, we would only need to design the grammar once per set of available components. This process should be a one-time investment akin to writing simulation or optimization code. In practice, designing graph grammars is an art which requires experience and multiple iterations. Even if two graph grammars generate the exact same set of robot designs, the specific rules included will strongly influence optimization efficiency. As an extreme example, a grammar with a rule that directly generates

the optimal robot will result in far faster convergence than one which merely encodes the allowed components and connections. The former grammar injects excessive bias into the search (and requires prior knowledge), while the latter grammar results in a slow search. Some compromise between the two is required. Knowing the right amount of prior knowledge to include in the grammar requires domain expertise, limiting the possible user base for our system.

Recent developments in graph grammar learning [135], [136] promise to automate this process as well. Starting from a dataset of known valid examples, they infer graph grammars using bottom-up approaches. The resulting grammars preserve validity while maximizing diversity under sampling. A similar approach can potentially be used to generate a robot graph grammar from examples. Designers would be able to simultaneously input available components, define assembly constraints, and inject prior knowledge in an intuitive manner.

6.2 Sim-to-Real

The purpose of computational robot co-design is to create robots that perform well in the real world. Simulation is a necessary part of the process, however, and many factors can prevent simulated robots from functioning in reality. Some factors, like fabrication error and computational constraints, are difficult to avoid. Others, such as incorrect physical properties and component specifications, are more easily prevented. The body of research concerned with transferring simulated robots to the real world is known as *sim-to-real*.

While the work in this thesis attempts to use only realistic components and simulation techniques, it does not deploy robots in the real world. This is left as an avenue of future work. Nonetheless, we have made some initial findings in this area that are worth sharing.

For one, sim-to-real should be considered from the very beginning. We mistakenly assumed it could be done as a post-processing step, with only minor changes in physical properties needed to achieve a match between simulation and reality. The plan was to quickly

demonstrate that our simulated walking robots can walk in the real world. In reality, major discrepancies emerged. One of the most impactful physical properties, sliding friction, was left at the default value (0.9) provided by the dynamics library. No material at hand could provide this friction coefficient on our surfaces of interest (carpet, concrete). Reducing the friction coefficient in simulation produced qualitatively different results. Since this project was intended as a direct follow-on work, changing the friction coefficient was ruled out. This left us with modifying the floor material. We found that polylactic acid (PLA) plastic and a sheet of unidentified rubber provided roughly the same friction coefficient as in simulation. With this mismatch resolved, we turned our attention to controls. Model predictive control, as used in our simulation, requires full state knowledge. This is usually obtained using motion capture, which would significantly increase the logistical complexity of our project. Open loop control, where we simply replay the same control inputs found in simulation, failed in unpredictable ways. The robots would often shuffle their legs in place or even move in the opposite direction compared to simulation. At this point, we decided that the problem was well beyond the scope of a simple follow-on work and pursued it no further.

If we were to address sim-to-real again, we would probably take a very different approach. We would design and fabricate at least one robot by hand before even starting to write code. Having a complete robot example, even if it is suboptimal, validates the physical properties used in simulation. We would additionally write a working controller by hand and execute it on the physical robot. The resulting behavior would reveal shortcomings in the hardware platform, such as the influence of cables or insufficient friction.

Besides system identification error and fabrication shortcomings, another potential source of sim-to-real error is simulation fidelity. Many of the models used in simulation are only approximations of the true dynamics. This discrepancy is most obvious in the case of aerodynamics and hydrodynamics. In Chapters 4 and 5, we must resort to highly simplified flat-plate models in order to achieve reasonable computation time. Even if higher-fidelity computational fluid dynamics (CFD) models are used, there is no guarantee that all of the

dynamics can be captured. CFD also requires much greater development effort. A different class of methods known as *physics informed neural networks* (PINNs) [137] aims to sidestep these issues by learning dynamics from data. Given a suitable source of time-series data, such as from a CFD simulation, PINNs learn the underlying partial differential equation (PDE) while taking advantage of the problem structure. Alternative approaches include augmenting a simpler model with a learned residual [138], [139] and learning specific sub-modules within a traditional simulation [140]. All of these techniques have the potential to increase simulation fidelity while reducing the necessary computation and development time. As such, they represent potential directions for improving sim-to-real transfer in robot co-design.

6.3 Sensing

So far, we assume that our robots know their current states at all times. Many of the methods we use to control our simulated robots, such as model predictive control and linear quadratic regulators, are based on this assumption. Real world robots must rely on sensors combined with *state estimation* to obtain an approximation of their state. This is often sufficient in practice, and we assume that robots have sensors placed in the appropriate locations for this purpose.

Sensor selection and placement can still be worth exploring, however. Sensors come with both financial costs and weight penalties. Therefore, it might make sense to determine the minimum number of sensors needed to make a robot controllable. The locations of sensors also matter, since incorrect placement can defeat their purpose. Optimizing sensor placement automates yet another aspect of the robot design process, and can lead to better task performance in rigid robot [141] and soft robot [142] co-design. Graph grammars are certainly capable of encoding sensor alternatives and placement restrictions, leaving open another possible avenue of future work.

Chapter 7

Conclusion

Emerging robot applications such as surveying and transportation require mobile robots that can move through complex environments. These applications pose many challenges such as terrain, disturbances, and limited battery capacity. The mobile robots of today are typically designed by hand from intuition and first principles. While they can achieve impressive results, these manually-designed robots require extensive tuning that leaves untapped performance.

Automatic robot design tools can potentially accelerate the design process and find better-performing robot designs. However, current implementations are often limited in the range of robot topologies they can explore. They also tend to rely on hand-designed heuristic functions to search the space of robot topologies, limiting them to only a single type of robot. A more general approach is needed.

This thesis aims to demonstrate how graph grammars and learning-based search algorithms can yield a unified yet flexible approach to robot co-design. Section [2.1.1](#) explains the concept of a graph grammar, and how they can be extended to design fabricable, biomimetic robots. Previous work in graph grammar-based robot design was restricted to passive systems or assemblies of identical modules. The work in this thesis builds on these previous efforts by using individual realistic components and taking control into account.

Graph grammars form the backbone but not the entirety of our approach. The remainder of Chapter 2 introduces the other two components of a robot co-design system: simulation and optimization. Simulation is largely domain-specific and will vary based on the type of robot being designed. In our work, this takes the form of either articulated body dynamics or differentiable rigid body dynamics. The latter offers computational advantages, and we prefer it wherever possible. Optimization consists of gradient-based algorithms such as SLSQP for continuous parameters, and Graph Heuristic Search (GHS) for searching the design space defined by our graph grammar. Although GHS is based on deep Q-learning, we employ graph grammar-specific adaptations.

Chapters 3, 4, and 5 show how graph grammars and GHS enable effective co-design for three different types of mobile robots. Although the specific techniques used vary depending on the domain, graph grammars remain the underlying design representation. In each case, they generate a wide variety of robot designs that respect the validity constraints of each domain. Graph grammars provide other benefits as well. They generate kinematic trees that can be directly simulated, and they admit efficient optimization methods. These advantages suggest that graph grammars are an ideal design representation for robot co-design.

Finally, Chapter 6 examines some of the limitations of the current work and shares some of the lessons learned in attempting to bring simulated robots into reality. It also suggests promising directions for further research. This thesis makes a case for graph grammars as a universal robot shape representation and Graph Heuristic Search as a way of searching over them. However, it is only one of many steps towards the dream of generating custom robots for every task.

References

- [1] A. Zhao, J. Xu, M. Konaković Luković, J. Hughes, A. Spielberg, D. Rus, and W. Matusik, “Robogrammar: Graph grammar for terrain-optimized robot design,” *ACM Transactions on Graphics (TOG)*, vol. 39, no. 6, pp. 1–16, 2020.
- [2] A. Zhao, J. Xu, J. Salazar, W. Wang, P. Ma, D. Rus, and W. Matusik, “Graph grammar-based automatic design for heterogeneous fleets of underwater robots,” in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 3143–3149.
- [3] A. Zhao, T. Du, J. Xu, J. Hughes, J. Salazar, P. Ma, W. Wang, D. Rus, and W. Matusik, “Automatic co-design of aerial robots using a graph grammar,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2022*, IEEE, 2022.
- [4] K. Sims, “Evolving virtual creatures,” in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, 1994, pp. 15–22.
- [5] K. Sims, “Evolving 3d morphology and behavior by competition,” *Artificial life*, vol. 1, no. 4, pp. 353–372, 1994.
- [6] Y. Tsai and A. Soni, “The effect of link parameter on the working space of general 3r robot arms,” *Mechanism and Machine Theory*, vol. 19, no. 1, pp. 9–16, 1984.
- [7] J.-H. Park and H. Asada, “Concurrent design optimization of mechanical structure and control for high speed robots,” 1994.

- [8] K. M. Digumarti, C. Gehring, S. Coros, J. Hwangbo, and R. Siegwart, “Concurrent optimization of mechanical design and locomotion control of a legged robot,” in *Mobile Service Robotics*, World Scientific, 2014, pp. 315–323.
- [9] A. Spielberg, B. Araki, C. Sung, R. Tedrake, and D. Rus, “Functional co-optimization of articulated robots,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 5035–5042.
- [10] C. Schaff, D. Yunis, A. Chakrabarti, and M. R. Walter, “Jointly learning to construct and control agents using deep reinforcement learning,” in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 9798–9805.
- [11] S. Coros, B. Thomaszewski, G. Noris, S. Sueda, M. Forberg, R. W. Sumner, W. Matusik, and B. Bickel, “Computational design of mechanical characters,” *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, pp. 1–12, 2013.
- [12] J. Hiller and H. Lipson, “Automatic design and manufacture of soft robots,” *IEEE Transactions on Robotics*, vol. 28, no. 2, pp. 457–466, 2011.
- [13] A. Spielberg, A. Zhao, Y. Hu, T. Du, W. Matusik, and D. Rus, “Learning-in-the-loop optimization: End-to-end control and co-design of soft robots through learned deep latent representations,” in *Advances in Neural Information Processing Systems*, 2019, pp. 8282–8292.
- [14] L. Dobrjanskyj and F. Freudenstein, “Some applications of graph theory to the structural analysis of mechanisms,” 1967.
- [15] F. Crossley, “The permutations of kinematic chains of eight member or less from the graph-theoretic viewpoint,” *Developments in theoretical and applied mechanisms*, vol. 2, pp. 467–486, 1965.

- [16] F. Freudenstein and E. Maki, “The creation of mechanisms according to kinematic structure and function,” *Environment and Planning B: Planning and Design*, vol. 6, no. 4, pp. 375–391, 1979.
- [17] T. Mruthyunjaya, “A computerized methodology for structural synthesis of kinematic chains: Part 1—formulation,” *Mechanism and machine theory*, vol. 19, no. 6, pp. 487–495, 1984.
- [18] S. Rajeev and C. Krishnamoorthy, “Discrete optimization of structures using genetic algorithms,” *Journal of structural engineering*, vol. 118, no. 5, pp. 1233–1250, 1992.
- [19] L. C. Schmidt and J. Cagan, “Ggreada: A graph grammar-based machine design algorithm,” *Research in Engineering Design*, vol. 9, no. 4, pp. 195–213, 1997.
- [20] F. R. Stöckli and K. Shea, “A simulation-driven graph grammar method for the automated synthesis of passive dynamic brachiating robots,” in *ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, American Society of Mechanical Engineers Digital Collection, 2015.
- [21] D. Pathak, C. Lu, T. Darrell, P. Isola, and A. A. Efros, “Learning to control self-assembling morphologies: A study of generalization via modularity,” *arXiv preprint arXiv:1902.05546*, 2019.
- [22] S. Bouabdallah, A. Noth, and R. Siegwart, “Pid vs lq control techniques applied to an indoor micro quadrotor,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, IEEE, vol. 3, 2004, pp. 2451–2456.
- [23] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, “Aggressive driving with model predictive path integral control,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2016, pp. 1433–1440.

- [24] K. Lowrey, A. Rajeswaran, S. Kakade, E. Todorov, and I. Mordatch, “Plan online, learn offline: Efficient learning and exploration via model-based control,” *arXiv preprint arXiv:1811.01848*, 2018.
- [25] T. Du, A. Schulz, B. Zhu, B. Bickel, and W. Matusik, “Computational multicopter design,” *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, pp. 1–10, 2016.
- [26] A. J. Barry, P. R. Florence, and R. Tedrake, “High-speed autonomous obstacle avoidance with pushbroom stereo,” *Journal of Field Robotics*, vol. 35, no. 1, pp. 52–68, 2018.
- [27] R. Featherstone, “The calculation of robot dynamics using articulated-body inertias,” *The International Journal of Robotics Research*, vol. 2, no. 1, pp. 13–30, 1983.
- [28] C. K. Liu and S. Jain, “A quick tutorial on multibody dynamics,” *Online tutorial*, June, p. 7, 2012.
- [29] Y. Hu, J. Liu, A. Spielberg, J. B. Tenenbaum, W. T. Freeman, J. Wu, D. Rus, and W. Matusik, “Chainqueen: A real-time differentiable physical simulator for soft robotics,” in *2019 International conference on robotics and automation (ICRA)*, IEEE, 2019, pp. 6265–6271.
- [30] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, “End-to-end differentiable physics for learning and control,” *Advances in neural information processing systems*, vol. 31, 2018.
- [31] K. M. Jatavallabhula, M. Macklin, F. Golemo, V. Voleti, L. Petrini, M. Weiss, B. Considine, J. Parent-Lévesque, K. Xie, K. Erleben, *et al.*, “Gradsim: Differentiable simulation for system identification and visuomotor control,” *arXiv preprint arXiv:2104.02646*, 2021.
- [32] K. Wampller and Z. Popović, “Optimal gait and form for animal locomotion,” in *ACM Transactions on Graphics (TOG)*, ACM, vol. 28, 2009, p. 60.

- [33] S. Ha, S. Coros, A. Alspach, J. Kim, and K. Yamane, “Joint optimization of robot design and motion parameters using the implicit function theorem,” in *Robotics: Science and systems*, vol. 8, 2017.
- [34] T. W. Lee and F. Freudenstein, “Heuristic combinatorial optimization in the kinematic design of mechanisms—part 1: Theory,” 1976.
- [35] C. Li, S. Tan, and K. Chan, “A qualitative and heuristic approach to the conceptual design of mechanisms,” *Engineering applications of artificial intelligence*, vol. 9, no. 1, pp. 17–32, 1996.
- [36] P. Jain and A. Agogino, “Optimal design of mechanisms using simulated annealing: Theory and applications,” in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, American Society of Mechanical Engineers, vol. 26584, 1988, pp. 233–240.
- [37] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.
- [38] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, “Learning combinatorial optimization algorithms over graphs,” *Advances in neural information processing systems*, vol. 30, 2017.
- [39] Z. Li, Q. Chen, and V. Koltun, “Combinatorial optimization with graph convolutional networks and guided tree search,” *Advances in neural information processing systems*, vol. 31, 2018.
- [40] G. Lederman, M. N. Rabe, E. A. Lee, and S. A. Seshia, “Learning heuristics for automated reasoning through deep reinforcement learning,” *arXiv preprint arXiv:1807.08058*, 2018.

- [41] J. Pugh and A. Martinoli, “Inspiring and modeling multi-robot search with particle swarm optimization,” in *2007 IEEE Swarm Intelligence Symposium*, IEEE, 2007, pp. 332–339.
- [42] J. B. Pollack, G. S. Hornby, H. Lipson, and P. Funes, “Computer creativity in the automatic design of robots,” *Leonardo*, vol. 36, no. 2, pp. 115–121, 2003.
- [43] J. McCormack, A. Dorin, T. Innocent, *et al.*, “Generative design: A paradigm for design research,” *Proceedings of Futureground, Design Research Society, Melbourne*, 2004.
- [44] D. Ha, “Reinforcement learning for improving agent design,” *arXiv preprint arXiv:1810.03779*, 2018.
- [45] G. Jing, T. Tosun, M. Yim, and H. Kress-Gazit, “Accomplishing high-level tasks with modular robots,” *Autonomous Robots*, vol. 42, no. 7, pp. 1337–1354, 2018.
- [46] Z. Bi and W.-J. Zhang, “Concurrent optimal design of modular robotic configuration,” *Journal of Robotic systems*, vol. 18, no. 2, pp. 77–87, 2001.
- [47] I.-M. Chen and J. W. Burdick, “Determining task optimal modular robot assembly configurations,” in *proceedings of 1995 IEEE International Conference on Robotics and Automation*, IEEE, vol. 1, 1995, pp. 132–137.
- [48] J. C. Bongard, “Evolutionary robotics,” *Communications of the ACM*, vol. 56, no. 8, pp. 74–83, 2013.
- [49] L. Brodbeck, S. Hauser, and F. Iida, “Morphological evolution of physical robots through model-free phenotype development,” *PloS one*, vol. 10, no. 6, e0128444, 2015.
- [50] R. A. Watson, S. G. Ficici, and J. B. Pollack, “Embodied evolution: Distributing an evolutionary algorithm in a population of robots,” *Robotics and Autonomous Systems*, vol. 39, no. 1, pp. 1–18, 2002.

- [51] J. R. Koza, “Survey of genetic algorithms and genetic programming,” in *Wescon conference record*, Western Periodicals Company, 1995, pp. 589–594.
- [52] S. Sivanandam and S. Deepa, “Genetic algorithms,” in *Introduction to genetic algorithms*, Springer, 2008, pp. 15–37.
- [53] T. Wang, Y. Zhou, S. Fidler, and J. Ba, “Neural graph evolution: Towards efficient automatic robot design,” *arXiv preprint arXiv:1906.05370*, 2019.
- [54] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, “Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions,” *arXiv preprint arXiv:1901.01753*, 2019.
- [55] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.
- [56] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud, “The grand challenge of computer go: Monte carlo tree search and extensions,” *Communications of the ACM*, vol. 55, no. 3, pp. 106–113, 2012.
- [57] R. Munos *et al.*, “From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning,” *Foundations and Trends® in Machine Learning*, vol. 7, no. 1, pp. 1–129, 2014.
- [58] Q. V. Nguyen, F. Colas, E. Vincent, and F. Charpillet, “Long-term robot motion planning for active sound source localization with monte carlo tree search,” in *2017 Hands-free Speech Communications and Microphone Arrays (HSCMA)*, IEEE, 2017, pp. 61–65.
- [59] M. P. Schadd, M. H. Winands, H. J. Van Den Herik, G. M.-B. Chaslot, and J. W. Uiterwijk, “Single-player monte-carlo tree search,” in *International Conference on Computers and Games*, Springer, 2008, pp. 1–12.

- [60] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, “Hierarchical graph representation learning with differentiable pooling,” in *Advances in neural information processing systems*, 2018, pp. 4800–4810.
- [61] N. Chomsky, “Three models for the description of language,” *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, Sep. 1956, ISSN: 2168-2712. DOI: [10.1109/TIT.1956.1056813](https://doi.org/10.1109/TIT.1956.1056813).
- [62] L. C. Schmidt, H. Shetty, and S. C. Chase, “A graph grammar approach for structure synthesis of mechanisms,” in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, American Society of Mechanical Engineers, vol. 80333, 1998, V003T03A034.
- [63] E. Klavins, R. Ghrist, and D. Lipsky, “Graph grammars for self assembling robotic systems,” in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA ’04. 2004*, IEEE, vol. 5, 2004, pp. 5293–5300.
- [64] B. Smith, A. Howard, J.-M. McNew, J. Wang, and M. Egerstedt, “Multi-robot deployment and coordination with embedded graph grammars,” *Autonomous Robots*, vol. 26, no. 1, pp. 79–98, 2009.
- [65] M. Lau, A. Ohgawara, J. Mitani, and T. Igarashi, “Converting 3d furniture models to fabricatable parts and connectors,” in *ACM Transactions on Graphics (TOG)*, ACM, vol. 30, 2011, p. 85.
- [66] G. S. Hornby and J. B. Pollack, “The advantages of generative grammatical encodings for physical design,” in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, IEEE, vol. 1, 2001, pp. 600–607.
- [67] G. Stiny and J. Gips, “‘shape grammars and the generative specification of painting and sculpture’,” vol. 71, Jan. 1971, pp. 1460–1465.
- [68] G. Stiny and W. J. Mitchell, “The palladian grammar,” 1978.

- [69] F. Downing and U. Flemming, “The bungalows of buffalo,” *Environment and Planning B: Planning and Design*, vol. 8, no. 3, pp. 269–293, 1981.
- [70] J. P. Duarte, “Towards the mass customization of housing: The grammar of siza’s houses at malagueira,” *Environment and Planning B: Planning and Design*, vol. 32, no. 3, pp. 347–380, 2005. DOI: [10.1068/b31124](https://doi.org/10.1068/b31124). [Online]. Available: <https://doi.org/10.1068/b31124>.
- [71] G. Stiny, “Spatial relations and grammars,” *Environment and Planning B: Planning and Design*, vol. 9, no. 1, pp. 113–114, 1982. DOI: [10.1068/b090113](https://doi.org/10.1068/b090113). [Online]. Available: <https://doi.org/10.1068/b090113>.
- [72] Y. I. H. Parish and P. Müller, “Procedural modeling of cities,” in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’01, New York, NY, USA: Association for Computing Machinery, 2001, pp. 301–308, ISBN: 158113374X. DOI: [10.1145/383259.383292](https://doi.org/10.1145/383259.383292). [Online]. Available: <https://doi.org/10.1145/383259.383292>.
- [73] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, “Instant architecture,” *ACM Trans. Graph.*, vol. 22, no. 3, pp. 669–677, Jul. 2003, ISSN: 0730-0301. DOI: [10.1145/882262.882324](https://doi.org/10.1145/882262.882324). [Online]. Available: <https://doi.org/10.1145/882262.882324>.
- [74] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, “Procedural modeling of buildings,” *ACM Trans. Graph.*, vol. 25, no. 3, pp. 614–623, Jul. 2006, ISSN: 0730-0301. DOI: [10.1145/1141911.1141931](https://doi.org/10.1145/1141911.1141931). [Online]. Available: <https://doi.org/10.1145/1141911.1141931>.
- [75] L. Krecklau, D. Pavic, and L. Kobbelt, “Generalized use of non-terminal symbols for procedural modeling,” *Computer Graphics Forum*, vol. 29, no. 8, pp. 2291–2303, 2010. DOI: [10.1111/j.1467-8659.2010.01714.x](https://doi.org/10.1111/j.1467-8659.2010.01714.x). eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01714.x>.

[Online]. Available:

<https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01714.x>.

- [76] D. Jesus, A. Coelho, and A. A. Sousa, “Layered shape grammars for procedural modelling of buildings,” *Vis. Comput.*, vol. 32, no. 6–8, pp. 933–943, Jun. 2016, ISSN: 0178-2789. DOI: [10.1007/s00371-016-1254-8](https://doi.org/10.1007/s00371-016-1254-8). [Online]. Available: <https://doi.org/10.1007/s00371-016-1254-8>.
- [77] M. Van Diepen and K. Shea, “A spatial grammar method for the computational design synthesis of virtual soft locomotion robots,” *Journal of Mechanical Design*, vol. 141, no. 10, 2019.
- [78] M. Bokeloh, M. Wand, and H.-P. Seidel, “A connection between partial symmetry and inverse procedural modeling,” *ACM Trans. Graph.*, vol. 29, no. 4, Jul. 2010, ISSN: 0730-0301. DOI: [10.1145/1778765.1778841](https://doi.org/10.1145/1778765.1778841). [Online]. Available: <https://doi.org/10.1145/1778765.1778841>.
- [79] O. Št’ava, B. Beneš, R. Měch, D. G. Aliaga, and P. Krištof, “Inverse procedural modeling by automatic generation of l-systems,” in *Computer Graphics Forum*, Wiley Online Library, vol. 29, 2010, pp. 665–674.
- [80] F. Wu, D.-M. Yan, W. Dong, X. Zhang, and P. Wonka, “Inverse procedural modeling of facade layouts,” *ACM Transactions on Graphics*, vol. 33, Aug. 2014. DOI: [10.1145/2601097.2601162](https://doi.org/10.1145/2601097.2601162).
- [81] M. Lipp, P. Wonka, and M. Wimmer, “Interactive visual editing of grammars for procedural architecture,” in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH ’08, Los Angeles, California: Association for Computing Machinery, 2008, ISBN: 9781450301121. DOI: [10.1145/1399504.1360701](https://doi.org/10.1145/1399504.1360701). [Online]. Available: <https://doi.org/10.1145/1399504.1360701>.
- [82] M. Dang, S. Lienhard, D. Ceylan, B. Neubert, P. Wonka, and M. Pauly, “Interactive design of probability density functions for shape grammars,” *ACM Trans. Graph.*,

- vol. 34, no. 6, Oct. 2015, ISSN: 0730-0301. DOI: [10.1145/2816795.2818069](https://doi.org/10.1145/2816795.2818069). [Online]. Available: <https://doi.org/10.1145/2816795.2818069>.
- [83] T. Liu, S. Chaudhuri, V. G. Kim, Q. Huang, N. J. Mitra, and T. Funkhouser, “Creating consistent scene graphs using a probabilistic grammar,” *ACM Trans. Graph.*, vol. 33, no. 6, Nov. 2014, ISSN: 0730-0301. DOI: [10.1145/2661229.2661243](https://doi.org/10.1145/2661229.2661243). [Online]. Available: <https://doi.org/10.1145/2661229.2661243>.
 - [84] C. E. Garcia, D. M. Prett, and M. Morari, “Model predictive control: Theory and practice—a survey,” *Automatica*, vol. 25, no. 3, pp. 335–348, 1989.
 - [85] G. Klančar and I. Škrjanc, “Tracking-error model-based predictive control for mobile robots in real time,” *Robotics and autonomous systems*, vol. 55, no. 6, pp. 460–469, 2007.
 - [86] F. Kuhne, W. F. Lages, and J. G. da Silva Jr, “Model predictive control of a mobile robot using linearization,” in *Proceedings of mechatronics and robotics*, Citeseer, 2004, pp. 525–530.
 - [87] R. Tedrake, T. W. Zhang, and H. S. Seung, “Stochastic policy gradient reinforcement learning on a simple 3d biped,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, IEEE, vol. 3, 2004, pp. 2849–2854.
 - [88] X. B. Peng, G. Berseth, K. Yin, and M. Van De Panne, “Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning,” *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–13, 2017.
 - [89] E. Coumans, “Bullet physics simulation,” in *ACM SIGGRAPH 2015 Courses*, ACM, 2015, p. 7.
 - [90] P. Norvig and S. Russell, *Artificial Intelligence: A modern approach*, 3rd. Prentice Hall, 2002, ch. 21.

- [91] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1412.6980>.
- [92] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Advances in neural information processing systems*, 2017, pp. 1024–1034.
- [93] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.
- [94] L. Kocsis and C. Szepesvári, “Bandit based monte-carlo planning,” in *European conference on machine learning*, Springer, 2006, pp. 282–293.
- [95] B. E. Childs, J. H. Brodeur, and L. Kocsis, “Transpositions and move groups in monte carlo tree search,” in *2008 IEEE Symposium On Computational Intelligence and Games*, IEEE, 2008, pp. 389–395.
- [96] S. Gelly and D. Silver, “Monte-carlo tree search and rapid action value estimation in computer go,” *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.
- [97] D. P. Helmbold and A. Parker-Wood, “All-moves-as-first heuristics in monte-carlo go.,” in *IC-AI*, 2009, pp. 605–610.
- [98] G. Antonelli, T. I. Fossen, and D. R. Yoerger, “Modeling and control of underwater robots,” in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Cham: Springer International Publishing, 2016, pp. 1285–1306, ISBN: 978-3-319-32552-1. DOI: [10.1007/978-3-319-32552-1_51](https://doi.org/10.1007/978-3-319-32552-1_51). [Online]. Available: https://doi.org/10.1007/978-3-319-32552-1_51.
- [99] K. L. Vasudev, “Review of autonomous underwater vehicles,” in *Autonomous Vehicles*, G. Dekoulis, Ed., Rijeka: IntechOpen, 2020, ch. 2. DOI: [10.5772/intechopen.81217](https://doi.org/10.5772/intechopen.81217). [Online]. Available: <https://doi.org/10.5772/intechopen.81217>.

- [100] K. Alam, T. Ray, and S. G. Anavatti, "A brief taxonomy of autonomous underwater vehicle design literature," English, *Ocean Engineering*, vol. 88, no. Complete, pp. 627–630, 2014. DOI: [10.1016/j.oceaneng.2014.04.027](https://doi.org/10.1016/j.oceaneng.2014.04.027).
- [101] M. Martz and W. L. Neu, "Multi-objective optimization of an autonomous underwater vehicle," in *OCEANS 2008*, 2008, pp. 1–9. DOI: [10.1109/OCEANS.2008.5151960](https://doi.org/10.1109/OCEANS.2008.5151960).
- [102] C. Brown and R. P. Clark, "Using a novel vehicle conceptual design utility to evaluate a long-range, large payload uuv," in *OCEANS 2010 MTS/IEEE SEATTLE*, 2010, pp. 1–10. DOI: [10.1109/OCEANS.2010.5664540](https://doi.org/10.1109/OCEANS.2010.5664540).
- [103] K. Alam, T. Ray, and S. G. Anavatti, "Design optimization of an unmanned underwater vehicle using low- and high-fidelity models," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 11, pp. 2794–2808, 2017. DOI: [10.1109/TSMC.2015.2390592](https://doi.org/10.1109/TSMC.2015.2390592).
- [104] K. Alam, T. Ray, and S. G. Anavatti, "An evolutionary approach for the design of autonomous underwater vehicles," in *AI 2012: Advances in Artificial Intelligence*, M. Thielscher and D. Zhang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 279–290, ISBN: 978-3-642-35101-3.
- [105] K. Alam, T. Ray, and S. G. Anavatti, "Design of a toy submarine using underwater vehicle design optimization framework," in *2011 IEEE Symposium on Computational Intelligence in Vehicles and Transportation Systems (CIVTS) Proceedings*, 2011, pp. 23–29. DOI: [10.1109/CIVTS.2011.5949527](https://doi.org/10.1109/CIVTS.2011.5949527).
- [106] A. Alvarez, V. Bertram, and L. Gualdesi, "Hull hydrodynamic optimization of autonomous underwater vehicles operating at snorkeling depth," *Ocean Engineering*, vol. 36, no. 1, pp. 105–112, 2009, Autonomous Underwater Vehicles, ISSN: 0029-8018. DOI: <https://doi.org/10.1016/j.oceaneng.2008.08.006>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0029801808001765>.

- [107] J. Xu, A. Speilberg, A. Zhao, D. Rus, and W. Matusik, “Multi-objective graph heuristic search for terrestrial robot design,” in *2021 International conference on robotics and automation (ICRA)*, IEEE, 2021.
- [108] J. Xu, T. Chen, L. Zlokapa, M. Foshey, W. Matusik, S. Sueda, and P. Agrawal, “An end-to-end differentiable framework for contact-aware robot design,” in *Proceedings of Robotics: Science and Systems*, Virtual, Jul. 2021. DOI: [10.15607/RSS.2021.XVII.008](https://doi.org/10.15607/RSS.2021.XVII.008).
- [109] A. Jacobson, I. Baran, J. Popović, and O. Sorkine, “Bounded biharmonic weights for real-time deformation,” *ACM Trans. Graph.*, vol. 30, no. 4, Jul. 2011, ISSN: 0730-0301. DOI: [10.1145/2010324.1964973](https://doi.org/10.1145/2010324.1964973). [Online]. Available: <https://doi.org/10.1145/2010324.1964973>.
- [110] T. Ju, S. Schaefer, and J. Warren, “Mean value coordinates for closed triangular meshes,” *ACM Trans. Graph.*, vol. 24, no. 3, pp. 561–566, Jul. 2005, ISSN: 0730-0301. DOI: [10.1145/1073204.1073229](https://doi.org/10.1145/1073204.1073229). [Online]. Available: <https://doi.org/10.1145/1073204.1073229>.
- [111] Z. Zhou, J. Liu, and J. Yu, “A survey of underwater multi-robot systems,” *IEEE/CAA Journal of Automatica Sinica*, vol. 8, p. 18, 2021.
- [112] J. Yu, C. Wang, and G. Xie, “Coordination of multiple robotic fish with applications to underwater robot competition,” *IEEE Transactions on Industrial Electronics*, vol. 63, no. 2, pp. 1280–1288, 2016. DOI: [10.1109/TIE.2015.2425359](https://doi.org/10.1109/TIE.2015.2425359).
- [113] R. Thenius, D. Moser, J. C. Varughese, *et al.*, “Subcultron - cultural development as a tool in underwater robotics,” in *Artificial Life and Intelligent Agents*, P. R. Lewis, C. J. Headleand, S. Battle, and P. D. Ritsos, Eds., Cham: Springer International Publishing, 2018, pp. 27–41, ISBN: 978-3-319-90418-4.

- [114] T. Schmickl, R. Thenius, C. Moslinger, *et al.*, “Cocoro – the self-aware underwater swarm,” in *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*, 2011, pp. 120–126. DOI: [10.1109/SASOW.2011.11](https://doi.org/10.1109/SASOW.2011.11).
- [115] D. Sutanty, P. Levi, C. Möslinger, and M. Read, “Collective-adaptive lévy flight for underwater multi-robot exploration,” in *2013 IEEE International Conference on Mechatronics and Automation*, 2013, pp. 456–462. DOI: [10.1109/ICMA.2013.6617961](https://doi.org/10.1109/ICMA.2013.6617961).
- [116] S. Min, J. Won, S. Lee, J. Park, and J. Lee, “Softcon: Simulation and control of soft-bodied animals with biomimetic actuators,” *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–12, 2019.
- [117] P. Ma, T. Du, J. Z. Zhang, K. Wu, A. Spielberg, R. K. Katzschmann, and W. Matusik, “Diffaqua: A differentiable computational design pipeline for soft underwater swimmers with shape interpolation,” *ACM Transactions on Graphics (TOG)*, vol. 40, no. 4, p. 132, 2021.
- [118] A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.
- [119] S. G. Johnson, *The nlopt nonlinear-optimization package*, 2014.
- [120] H. Shakhathreh, A. H. Sawalmeh, A. Al-Fuqaha, Z. Dou, E. Almaita, I. Khalil, N. S. Othman, A. Khreishah, and M. Guizani, “Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges,” *Ieee Access*, vol. 7, pp. 48 572–48 634, 2019.
- [121] G. Pajares, “Overview and current status of remote sensing applications based on unmanned aerial vehicles (uavs),” *Photogrammetric Engineering & Remote Sensing*, vol. 81, no. 4, pp. 281–330, 2015.

- [122] P. Doherty and P. Rudol, “A uav search and rescue scenario with human body detection and geolocalization,” in *Australasian Joint Conference on Artificial Intelligence*, Springer, 2007, pp. 1–13.
- [123] K. Kuru, D. Ansell, W. Khan, and H. Yetgin, “Analysis and optimization of unmanned aerial vehicle swarms in logistics: An intelligent delivery platform,” *Ieee Access*, vol. 7, pp. 15 804–15 831, 2019.
- [124] D. C. Tsouros, S. Bibi, and P. G. Sarigiannidis, “A review on uav-based applications for precision agriculture,” *Information*, vol. 10, no. 11, p. 349, 2019.
- [125] J. Kim, M.-S. Kang, and S. Park, “Accurate modeling and robust hovering control for a quad-rotor vtol aircraft,” in *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, Springer, 2009, pp. 9–26.
- [126] B. Uragun, “Energy efficiency for unmanned aerial vehicles,” in *2011 10th International Conference on Machine Learning and Applications and Workshops*, IEEE, vol. 2, 2011, pp. 316–320.
- [127] U. Ozdemir, Y. O. Aktas, A. Vuruskan, Y. Dereli, A. F. Tarhan, K. Demirbag, A. Erdem, G. D. Kalaycioglu, I. Ozkol, and G. Inalhan, “Design of a commercial hybrid vtol uav system,” *Journal of Intelligent & Robotic Systems*, vol. 74, no. 1, pp. 371–393, 2014.
- [128] A. Sobester and A. Keane, “Multidisciplinary design optimization of uav airframes,” in *47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference 14th AIAA/ASME/AHS Adaptive Structures Conference 7th*, 2006, p. 1612.
- [129] T. Du, A. Schulz, B. Zhu, B. Bickel, and W. Matusik, “Computational multicopter design,” 2016.

- [130] J. Xu, T. Du, M. Foshey, B. Li, B. Zhu, A. Schulz, and W. Matusik, “Learning to fly: Computational controller design for hybrid uavs with reinforcement learning,” *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.
- [131] N. Umetani, Y. Koyama, R. Schmidt, and T. Igarashi, “Pteromys: Interactive design and optimization of free-formed free-flight model airplanes,” *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 1–10, 2014.
- [132] M. Fey and J. E. Lenssen, “Fast graph representation learning with pytorch geometric,” *arXiv preprint arXiv:1903.02428*, 2019.
- [133] J. C. Gomez and E. Garcia, “Morphing unmanned aerial vehicles,” *Smart Materials and Structures*, vol. 20, no. 10, p. 103 001, 2011.
- [134] E. Ajanic, M. Feroskhan, S. Mintchev, F. Noca, and D. Floreano, “Bioinspired wing a and tail morphing extends drone flight capabilities,” *Sci. Robot.*, vol. 5, eabc2897, 2020.
- [135] S. Aguinaga, D. Chiang, and T. Weninger, “Learning hyperedge replacement grammars for graph generation,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 3, pp. 625–638, 2018.
- [136] M. Guo, V. Thost, B. Li, P. Das, J. Chen, and W. Matusik, “Data-efficient graph grammar learning for molecular generation,” *arXiv preprint arXiv:2203.08031*, 2022.
- [137] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *Journal of Computational physics*, vol. 378, pp. 686–707, 2019.
- [138] A. Ajay, J. Wu, N. Fazeli, M. Bauza, L. P. Kaelbling, J. B. Tenenbaum, and A. Rodriguez, “Augmenting physical simulators with stochastic neural networks: Case study of planar pushing and bouncing,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2018, pp. 3066–3073.

- [139] A. Kloss, S. Schaal, and J. Bohg, “Combining learned and analytical models for predicting action effects from sensory data,” *The International Journal of Robotics Research*, vol. 41, no. 8, pp. 778–797, 2022.
- [140] 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, eaau5872, 2019.
- [141] D. Cliff, P. Husbands, and I. Harvey, “Explorations in evolutionary robotics,” *Adaptive behavior*, vol. 2, no. 1, pp. 73–110, 1993.
- [142] A. Spielberg, A. Amini, L. Chin, W. Matusik, and D. Rus, “Co-learning of task and sensor placement for soft robotics,” *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1208–1215, 2021.