Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

DLLM: Developing Legged Locomotion

Author:
Lee Wai Chun Alvis

Supervisor:
Dr. Antoine Cully

Second Marker:
Dr. Edward Johns

June 15, 2020

Abstract

Numerous studies have been proposed to develop locomotion controllers for legged robots. However, in most studies, the legged robot learns to walk in a flat environment without any obstacles. The inability to walk on uneven terrains removes the fundamental purpose of using legged robots. This study proposed a hierarchical behavioral-based control system that can drive the robot on uneven terrains. The controller relies on three methods: (1) Quality Diversity algorithms for generating a collection of walking controllers that can drive the hexapod in different directions. (2) Monte Carlo Tree Search as a path planning algorithm to select walking controllers that drive the hexapod towards a goal. (3) Bayesian Optimization for adjusting the swing heights of each leg movement, such that the hexapod can step over obstacles in an energy-efficient manner. Results show that the proposed controller successfully drives the hexapod on uneven terrains to reach different goals. Overall, the proposed controller is a step towards developing hexapods that fulfill their purpose.

Acknowledgements

I would like to express my very great appreciation to my supervisor Antoine Cully for his valuable and constructive suggestions throughout the project. His willingness to give his time so generously is much appreciated. His enthusiastic encouragement and overall insights in this field have made this an inspiring experience for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advancement in technology, robots are deployed to solve tasks in outdoor scenarios such as agriculture, mining, and rescue missions. These environments require the robot to negotiate obstacles and navigate through uneven terrain. Legged robots have proven to be the solution that can adapt to such conditions. In recent years, many studies focused on developing controllers for legged locomotion as it is the most fundamental skill for a robot to solve any task.

Many methods have been used to develop controllers for legged robots, one of which is by using evolutionary algorithms. An evolutionary algorithm searches for an optimal solution through a process inspired by natural evolution and has successfully developed locomotion controllers for quadrupeds [ZBL04] and hexapods [FKM70]. However, these controllers can only walk forward at a constant speed [HTYF05]. This limitation renders the robot useless in the real world. Instead, new methods are proposed to develop controllers with a wide range of locomotion behaviors.

Novelty search [LS11a] is a diversity driven evolutionary algorithm that has been used to generate diverse locomotion behaviors for legged robots. It does not find the set of controllers with the optimal performance but the set of controllers with different behaviors. In some cases, a controller from the latter might outperform the controllers in the former. For example, in the original novelty search paper, novelty search developed a controller for a biped robot that is better than the ones developed using evolutionary algorithms. However, in most cases, evolutionary algorithms produce controllers with higher quality.

Quality Diversity (QD) algorithms combine the notion of evolutionary algorithms and novelty search to generate solutions that are both diverse and high-performing [CD17]. The most commonly used QD algorithms are Novelty Search with Local Competition (NSLC) [LS11b] and MAP-Elites [MC15]. These algorithms were applied to legged robots to generate repertoires of locomotion controllers that allow robots to move in different directions [CM13b]. The ability to automatically create a repertoire of controllers opens opportunities for new forms of robot control, particularly in behavior-based control.

Behavior-based control contains a collection of primitive controllers and a higher-level controller. Each primitive controller is responsible for one simple action, while the higher-level controller controls the robot by selecting primitive controllers from the collection. In most studies [LLC+19][CVM18][DGOC18], the higher-level controller is a path planning algorithm that steers the robot towards a goal. The collection of primitive controllers used to be manually defined by the user, but with the advent of QD algorithms, they can be generated automatically.

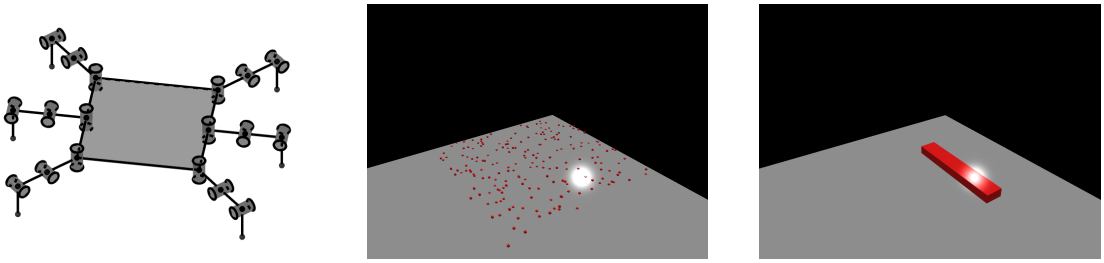
Instead of only using one higher-level controller, Cully et al [CD18] have shown that it is possible to generate a collection of higher-level controllers from the primitive controllers, creating a hierarchal behavioral-based control. Each higher-level controller combines multiple primitive controllers to perform a more sophisticated action. In the hierarchical behavioral-based control, the controllers at the lowest level are the primitive actions. Each collection higher up the hierarchy can perform increasingly complex actions by leveraging those from the previous layer. This approach was used to train a robotic arm to draw different digits.

One problem with behavioral-based control is that it fails to work if a primitive controller does not behave as expected, which could happen if the robot is damaged or affected by external factors in the environment. Cully et al [CCTM15] solve the problem by combining behavioral-based control with Bayesian optimization. Instead of assuming the higher-level controller knows the expected behavior of each primitive controller, it uses Bayesian optimization to learn the actual behavior of

the primitive controllers. The approach enabled a hexapod to accommodate mechanical damages by learning the new behavior of the primitive controllers.

Although there are lots of studies on controlling legged robots, most approaches focused on developing locomotion of legged robots on flat terrains. While it is exciting to see legged robots travel in different directions and adapt mechanical damages, the primal aim is to develop legged robots that can walk on uneven landscapes. Legged robots lose their purpose if they can only walk on flatlands, as it is more convenient to use wheeled robots instead. Rather than developing legged controllers to solve more complicated tasks, this paper returns to the fundamental problem: developing legged locomotors to walk through uneven terrains.

1.1 Objectives



(a) Hexapod (b) Obstacles (c) Walls

(a) The hexapod robot used in this study has 18 degrees of freedom, three for each leg.

(b) Uneven terrain in this study refers to an area covered with small obstacles of irregular heights. The hexapod can overcome these obstacles by raising its leg above a certain height.

(c) Walls in this study refers to large objects that the hexapod cannot climb over.

Figure 1.1: Problem Formulation

This paper aims to develop a control system for a hexapod, such that it can travel on uneven terrains to reach different destinations. Figure 1.1a shows the hexapod robot used in this study. The hexapod has 18 degrees of freedom, where each degree of freedom is actuated by position-controlled servos. The control system controls the hexapod by sending 18 values specifying the angular positions of each servo. In this study, the hexapod moves in an environment containing *walls* and *obstacles*. Obstacles are irregular objects with different heights spread across the environment, creating uneven terrain. The hexapod can overcome these obstacles by lifting its leg above a certain height. On the other hand, walls are tall objects in the environment the hexapod cannot climb over. The hexapod can only walk around the wall if the wall is blocking its path.

The control system should find the shortest path from the starting position to the destination. It should also adjust the swing heights of the hexapod while driving the hexapod towards the goal, such that it can overcome the obstacles along the path. The control system in this study contains three parts.

1. A hierarchical behavioral-based control that moves the hexapod in different directions. It generates a collection of primitive controllers using the novelty search algorithm, then uses these primitive controllers to evolve a collection of walking controllers.

2. A planning algorithm that selects walking controllers to steer hexapod towards the goal. It uses the Monte Carlo tree search to decide which walking controller to pick.

3. An adaptation mechanism that allows the hexapod to adjust its swing height to overcome obstacles in the environment. It uses Bayesian optimization to approximate the best swing height that has the least collisions with the obstacles.

Chapter 2

Background

2.1 Optimization Algorithms

2.1.1 Evolutionary Algorithms (EA)

In artificial intelligence, evolutionary algorithms (EA) [FOW66] are optimization algorithms inspired by biological evolution. The implementation of EA varies considerably by application, but the underlying idea behind all these techniques appeal to the metaphor of natural selection. Given that a population is exposed to environmental pressure, the theory of natural selection suggests that individuals unfit for their environment will be eliminated, while those with more desirable traits will survive and reproduce prolifically.

EAs find optimal solutions by evolving a population of *individuals*. Given a quality function to be maximized, an EA creates a random set of individuals and applies the quality function as a fitness measure to each individual. Based on this fitness, better individuals are used to seed the next generation while unfit individuals are replaced by better offspring. This evolutionary process continues to increase the average fitness of the population in every iteration until it has reached some threshold performance or has reached the computational limit.

In the parlance of EA, a solution is referred to as an individual encoded by a *genotype*. The performance of an individual is calculated using a fitness function, which is used to guide the heuristic search to the optimal solution. In each generation, the selection operator chooses individuals with higher fitness to produce new offspring by applying the recombination and mutation operators. The recombination operator takes two or more individuals and combines portions of each genotype to form a child. The mutation operator then modifies the child's genotype in some random way. In doing so, this allows the population to explore more genetic combinations.

Evolutionary robotics is a field of research that employs EA to generate control systems for autonomous robots. One of the main challenges of robot design is to consider multiple aspects simultaneously, such as morphology, sensors, motor systems, and control architecture, etc – all of which play an important role in determining the robot's behavior. Instead of designing each part separately and integrating them during the end process, evolutionary robotics aims to create adaptive robots by eliminating traditional boundaries between the subfields of robotics and uses EA to automate the design process. This novel perspective on robot design has proven successful in evolving controllers for quadrupedal [ZBL04] and hexapod robots [FKM70], as well as flying robots [MDM06].

One common problem found in EA is premature convergence, where the population converges too early into a suboptimal solution. Since these local optimal solutions are only somewhat optimized, the algorithm may not be able to generate offspring that outperform their parents even if the child is closer to the global optimal. As a result, the population becomes trapped in the local optima of the search space. Running the EA several times with a different initial configuration can mitigate the impact of premature convergence [MMP07]. Furthermore, hyperparameters such as crossover rate [ES08] and mutation rate [BA11] can be tailored to achieve the best performance.

2.1.2 Novelty Search (NS)

Algorithm 1 Novelty Search

```
procedure NOVELTY-SEARCH
    P ← ∅                                          ▷ P is the current population
    X ← ∅                                          ▷ X is the solution archive
    for i = 1 → G do                               ▷ Initialize population
        x ← randomSolution()
        P(i) = x
        addToArchive(x, X)
    while not termination criterion do
        P' ← ∅
        for i = 1 → G do
            x ← selectNovel(P, X)                  ▷ Select parents from population
            x ← mutate(x)                          ▷ Produce offspring via mutation
            P'(i) = x
            addToArchive(x, X)
        P ← P'                                     ▷ Update population
    return P
procedure ADD-TO-ARCHIVE(x, X)                     ▷ Adds a solution to the archive
    b ← descriptor(x)                              ▷ Compute behavior descriptor
    s ← novelty(x)                                 ▷ Compute novelty score
    if s > ϵ then
        X(b) = x
```

Novelty Search (NS) [LS11a] is a diversity-driven EA that rewards individuals based on behavioral novelty instead of performance. This technique is inspired by an alternative view of natural evolution as a diversifier rather than an optimizer. In contrast to the tendency of optimization algorithms to converge into a single optimal solution, real-life nature tends to discover a variety of solutions to meet various survival challenges. Unlike traditional EA that rewards solutions based on its performance towards an objective, NS disregards the goal and rewards individuals with unique behaviors.

NS algorithm inherits a similar structure to the traditional EA but with a different objective function. Instead of creating a gradient towards the prescribed objective, NS generates constant pressure to produce something new. NS consist of the current population as well as an archive of past individuals that keeps track of the unique behaviors throughout the evolution. The novelty of an individual is measured by how much its behavior differs from those in the archive. In every iteration, individuals with a novelty score above a certain threshold are added to the archive.

Contrary to intuition, searching for solutions without an objective can outperform objective-based approaches in some domains. Studies of NS have shown that deceptive problems can be solved more reliably using NS than with traditional EA. A deceptive problem is one that increasing the fitness of the solution might leads further away from the objective. Using the example of a maze, the agent cannot directly follow the path that leads it closer to the final goal because it might end up being trapped in a dead-end. Using an objective function to drive the search in deceptive problems may misdirect the search towards suboptimal solutions because the stepping stones that lead to a solution do not resemble the solution itself. NS is more suitable for deceptive problems as it does not explicitly search for the goal, thus cannot be deceived in the same manner. Similar studies [MD09] confirmed that abandoning the objective and searching for diverge solutions can sidestep deception to find the optimal solution. This counterintuitive result of non-objective search has sparked research interest in the field of diversity-driven EA. For example, Surprise Search [GLY19] replaces the Novelty score in NS with a Surprise score, which assesses an individual by how much its behavior deviates from the expected behaviors based on trends in recent generations.

NS has been applied to evolutionary robotics to create robots with a diverse skillset. It is mainly used in neuroevolution to evolve artificial neural network controllers. For example, this approach is used in EvoRBC-II to generate a set of primitive actions for a robot. EvoRBC-II [GOC18] then combined these actions using a high-level controller and was able to solve a broad range of tasks.

2.2 Quality Diversity (QD) Optimization

A new search paradigm is emerging within evolutionary computation that aims to maintain a balance between divergence and convergence when searching for solutions. Quality Diversity (QD) algorithms, combine the notion of EA and NS to generate a collection of solutions that are both diverse and high-performing [CD17]. QD algorithms differ from EA in the sense that they return a repertoire of optimal solutions located in different regions of the behavior space. They are also different from NS in the sense that they look for the highest performing individual within each behavioral niche. QD algorithms are a more comprehensively analogical to natural evolution, where it discovers a vast assortment of species and each has to face competition from within their niche. For example, real-life nature has evolved different methods for locomotion, including walking, crawling, swimming, flying, etc. While these methods are fundamentally different, they can all be referred to as an effective solution to movement.

There are optimization algorithms designed to return multiple optimized solutions. For instance, Multi-Modal Optimization (MMO) [YG10] involves finding all the local optima of a function. An important aspect that differentiates QD algorithms from methods like MMO is that it prioritizes diversity over quality, while others mainly focus on quality. Given a behavior space defined by the user, the goal of QD algorithms is to find the best performing solution within each niche even if it is low-performing. In contrast, Multi-Modal Optimization will only return the optimal solution from high-performing regions. It is worth mentioning that QD algorithms promote behavioral diversity instead of genetic diversity. Different genotypes do not directly translate to different behaviors, especially when there is an indirect encoding from genotypes to phenotypes. In some cases, different genotypes may result in solutions with the same behavior.

The two most common QD algorithms are Novelty Search with Local Competition (NSLC) [LS11b] and Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) [MC15]. NSLC is the original QD algorithm that hybridizes NS with performance competition between similar individuals. NSLC uses a multi-objective algorithm, NSGA-II [DPAM02], to simultaneously optimize the novelty and the performance of an individual to other organisms in the same niche. Combining NS with local competition allows NSLC to generate a diverse collection of high-performing virtual creature morphologies and walking strategies in a single run. MAP-Elites is another QD algorithm inspired by NSLC and the Multi-Objective Landscape Exploration Algorithm. It discretizes the behavior space into a grid and stores the best performing individual in each cell. A unifying framework [CD17] has been proposed to formulate all QD algorithms into one algorithm that is composed of a container and a selection operator. Under this framework, NSLC and MAP-Elites is the same algorithm but with different combinations of operators. Unifying these algorithms into a single modular framework allows a direct comparison between different QD algorithms and adds flexibility in designing new QD algorithms in the future.

The potential to uncover a diverse collection of high-quality solutions in a single run opens up new research directions in the field of evolutionary robotics. MAP-Elites was used to evolve a repertoire of controllers for a hexapod to move in different directions [CCTM15]. The same behavior repertoire enabled a hexapod to accommodate mechanical damages by walking differently [CCTM15]. In yet another QD application, a robot was able to learn how to draw different lines and arcs then combing these actions to write numbers [CD18].

2.2.1 MAP-Elites

In a MAP-Elites algorithm, the user defines a fitness function and an n-dimensional grid as *behavior space*, MAP-Elites will then return the highest performing individual for each cell in the behavior space [MC15]. The number of dimensions and the level of discretization depends on the user's preference. In the original paper [MC15], the authors used MAP-Elites on a two-dimensional behavior space to find the fastest soft robot morphology for each combination of bone sizes and body sizes. A further example is evolving robots that can move to different locations. The behavior space can describe the robot's final x,y-coordinates, and the performance can be interpreted as the robot's energy consumption.

MAP-Elites starts by randomly generating a set of individuals and placing them into corresponding cells in the grid. In each iteration, MAP-Elites randomly selects individuals from the grid to produce offspring via the crossover and mutation operators. Each offspring is then evaluated and mapped to the corresponding cell. The offspring is added to the grid if the cell is empty, or if it outperforms the current occupant, in which case the offspring replaces the current occupant.

Algorithm 2 MAP-Elites

procedure MAP-ELITES
  $\mathcal{P} \leftarrow \emptyset$ ▷ $\mathcal{P}$ stores fitness in each cell
  $\mathcal{X} \leftarrow \emptyset$ ▷ $\mathcal{X}$ stores solution in each cell
  for $i = 1 \rightarrow G$ **do** ▷ Initialize grid with G random solutions
    $x \leftarrow$ randomSolution()
    addToGrid$(x, \mathcal{P}, \mathcal{X})$
  while not termination criterion **do**
    $x \leftarrow$ uniformSelect$(\mathcal{X})$ ▷ Select random solution from grid
    $x \leftarrow$ mutate(x) ▷ Produce offspring via mutation
    addToGrid$(x, \mathcal{P}, \mathcal{X})$
  return $\mathcal{P}$ and $\mathcal{X}$

procedure ADD-TO-GRID$(x, \mathcal{P}, \mathcal{X})$ ▷ Adds a solution to the grid
  $b \leftarrow$ descriptor(x) ▷ Compute behavior descriptor
  $p \leftarrow$ performance(x) ▷ Compute fitness
  if $\mathcal{P}(b) = \emptyset$ or $\mathcal{P}(b) < p$ **then**
    $\mathcal{P}(b) = p$
    $\mathcal{X}(b) = x$

The algorithm continues until a predetermined amount of time expires, or the average fitness of the individuals exceeded a certain threshold.

One benefit of using MAP-Elites is that it can highlight the fitness potential of each area in the behavior space. MAP-Elites searches the behavior space to find the highest performing individual in each region. Plotting the n-dimensional grid can illuminate specific areas with higher performance and display the tradeoffs between performance and the features of interest. Such fitness maps can be useful for visualizing how each feature relates to performance. They can also be used for quickly identifying individuals with similar behavior just by looking at neighboring cells.

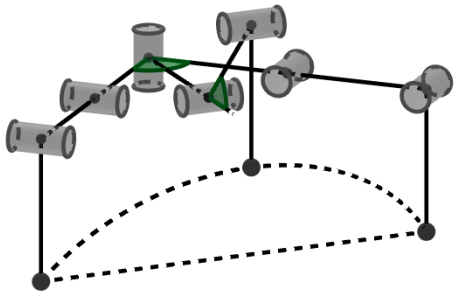
MAP-Elites is a promising QD algorithm that has found success in robotics locomotion. EvoRBC [DGOC18] used MAP-Elites to generate a repertoire of primitive actions for a hexapod, then combined these actions using a high-level controller to navigate the hexapod through a maze. In another application, Cully et al. [CCTM15] used MAP-Elites with Gaussian Process to evolve a hexapod that can adapt physical damages.

2.3 Controllers

2.3.1 Low Level Controllers

The hexapod robot has three joints per leg, giving it 18 degrees of freedom. The first joint of each leg controls its direction, the next one controls its elevation, and the last one controls its extension. Controlling the hexapod is achieved by sending a vector with 18 values to the robot, specifying the position of each joint, changing this vector updates the joints position, and hence the configuration of the legs. A hexapod controller combines a series of leg configurations into a continuous action. More precisely, the controller outputs a leg configuration every time step, each of which is a transition from the previous one, creating smooth walking gaits for the hexapod.

There are two categories of controllers – input-driven controllers and un-driven controllers [CM13b]. Input-driven controllers receive one or more inputs and adjust the outputs to the actuators. Input-driven controllers are more flexible than un-driven controllers because they can perform parameterized actions, such as "move forward 5 meters" or "rotate 20 degrees clockwise". On the other hand, un-driven controllers send the same sequence of commands to the actuators cyclically. Since un-driven controllers do not consider inputs, each controller can only perform a single primitive action like "move forward" or "rotate clockwise". Despite this limitation, the majority of studies about evolving gaits for legged robots focus on un-driven controllers. In these studies, un-driven controllers are characterized by periodic functions [CM13a], artificial neural networks [CSPO11], and central pattern generators [ICRC07].



(a) Leg Cycle

(b) Tripod Gait

(a) The leg cycle consists of a swing phase and a stance phase. The swing phase lifts the leg off the ground and brings it forward, while the stance phase pushes the leg on the floor and brings it backward.

(b) The tripod gait synchronizes the leg cycles of all the legs. The dashed segments represent a leg in swing phase, where as the bolded segments represent a leg in stance phase.

Figure 2.1: Hexapod Control

One of the reasons why un-driven controllers are extensively used in the studies is because its cyclic nature perfectly describes the robot's *leg cycle*. A robot leg cycle consists of a *swing phase* and a stance phase. During the swing phase, the robot lifts the leg off the ground and moves it forward to a new position in a swing motion. During the stance phase, the leg is in contact with the ground while traversing back to the starting position, pushing the robot forward. These two phases alternate perpetually to drive the hexapod, meaning the angular positions of each servo follows a cyclic pattern. Since the signals sent from the un-driven controllers also follow a cyclic pattern, they fit perfectly with the leg cycle.

The robot leg cycle is just one part of legged locomotion. Just as humans walk by moving one leg at a time, the robot has to move the legs in coordination to locomote. One of the most commonly observed walking gait in nature is the tripod gait in insects [Gur17]. The tripod gait has three legs in the stance phase and three legs in the swing phase at all times. The three legs touching the ground support the body and propel it forward. At the same time, the other legs move forward through the air. When the supporting legs finished the stance phase, the other legs would have completed the swing and continue to propel the body forward. Figure 2.1b shows the leg cycle for each leg in a tripod gait. The bolded segments represent a stance phase, whereas dotted segments represent the swing phase.

Un-driven controllers can be evolved using EAs and QD algorithms. In the context of EA, each un-driven controller is an individual that executes one action. The fitness of each individual is obtained by running the controller on the robot and measuring the robot's performance. Hornby et al. [HTY+00] presented one of the first applications to produce controllers for a legged robot using EAs. The study used parameterized periodic functions as un-driven controllers, then used EA to find the best controller that optimizes forward-moving speed. In another example, Cully et al. [CM13a] used a QD algorithm, BR-Evolution, to generate a collection of un-driven controllers that drive a hexapod in different directions.

2.3.2 Hierarchical Controllers

Classical robot control architecture follows a three steps structure. First, the robot senses the environment with its sensors. The robot then builds a model of the world from sensor measurements and uses the model to find the best action before finally executing it in the real world. Kolter [KRN08] used this approach to control the LittleDog robot to walk on uneven terrains. Since the advent of using QD algorithms to generate a repertoire of un-driven controllers, it opens new opportunities for a new form of robot control. Behavioral-based control systems have a hierarchical structure with two layers. The bottom layer consists of a repertoire of un-driven controllers, each responsible for one behavior of the robot. The upper layer controls the robot by selecting the most appropriate controller from the repertoire depending on the context and the current state of the robot. Behavioral-based control systems are more efficient than the classical approach because

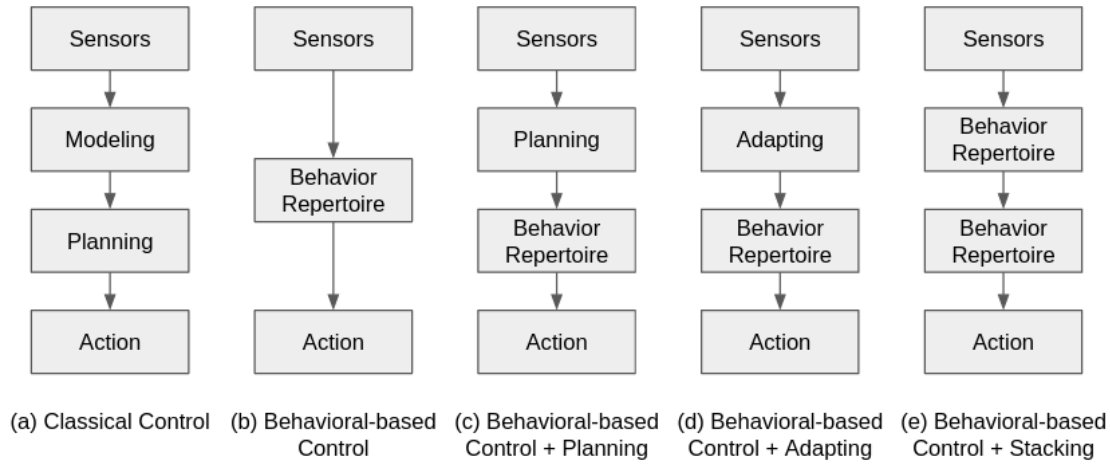


Figure 2.2: Hierarchical Control Architectures

it obviates the modeling process, which can be computationally expensive. Consider the evasive mechanism in animals to escape from predators. The classical control achieves evasion by building a model and calculating the consequences of each action. On the contrary, behavior-based control systems can escape predators by directly choosing the evade action.

Most studies combined behavioral-based controls with path planning algorithms. The behavior repertoire contains controllers to drive the robot in different directions, and the path planning algorithm [CVM18] selects a sequence of locomotion behaviors from the repertoire to steer the robot towards the goal. For instance, the reset-free trial-and-error learning algorithm evolved a collection of primitive actions for a hexapod using MAP-Elites and used A* search to find the shortest path to reach the goal. Other studies used a similar architecture to control a differential-drive robot to solve a double T-maze [DOC15] and a hexapod to reach different goal points [LLC+19].

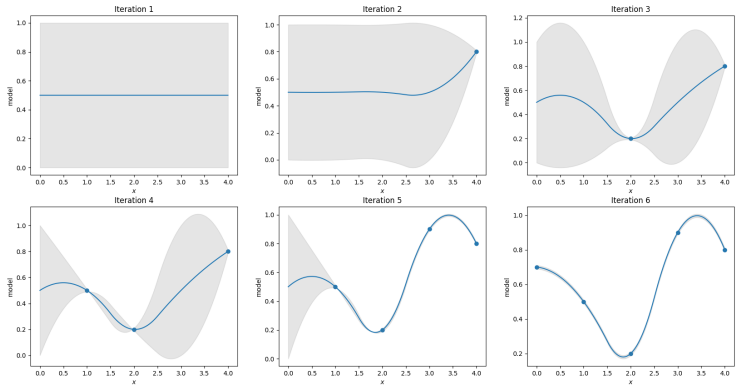
Behavioral-based controls can also be combined with learning algorithms to make the robot fault-tolerant. Each controller in the repertoire is responsible for one behavior of the robot. However, in some cases, running the controller may not perform the expected behavior due to mechanical damages on the robot or changes in the environment. Behavioral-based control fails to work in this case since the controllers in the repertoire do not map to their expected behaviors. For example, running the forward controller might make the robot turn left. Cully et al. [CCTM15] used Bayesian optimization to continuously update the mapping from controllers to their behaviors, allowing a damaged robot to choose the right controllers with the desired behavior.

Behavior repertoires can be stacked to accomplish more sophisticated behaviors. The controllers at the lowest level perform primitive actions. Repertoires higher up the hierarchy can perform increasingly complex actions by leveraging those from the previous layer. Studies have shown evolving controllers hierarchically in a bottom-up fashion gives better performance than evolving the controllers directly for complex tasks [NTL+19]. This approach was used to train a robotic arm to draw digits after learning to draw lines and arcs [CD18]. Furthermore, the hierarchical structure is also reusable. In the previous study [CD18], controllers evolved for the robotic arm were transferred to a humanoid robot by retraining parts of the hierarchy.

2.4 Bayesian Optimization

Bayesian optimization [Moc12] is a technique for finding the global optima for expensive black-box functions. The algorithm optimizes an unknown objective function in two parts. The first part of the algorithm tries to approximate the objective function, while the second part finds the optima in this approximation. Like all other Bayesian methods, Bayesian optimization uses a model to incorporate prior beliefs about the objective function, then updates the model based on observed samples to get a better approximation. Bayesian optimization then uses an acquisition function to direct sampling to areas that are most likely to find the global optima.

A commonly used *model* for Bayesian optimization is Gaussian processes [Ras04]. Gaussian pro-



The figures demonstrates hows Bayesian Optimization updates the model. In each iteration, it samples a point from the unknown objective function then incorporate this new knowledge in the model. Sampling more points reduces the uncertainty and provides a more accurate representation of the objective function.

Figure 2.3: Bayesian Optimization

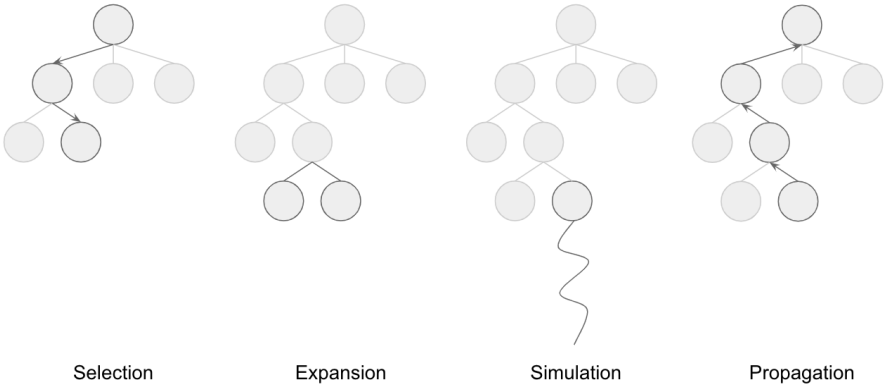
cesses are not limited to regression problems but can also be used to solve classification [KGUD10] and clustering [KL07] tasks. The underlying idea behind approximating a function is to define a probability distribution over all possible functions, then update this distribution as the algorithm samples more data. In the beginning, there is no information about the objective function, so there is an infinite number of possibilities. As the algorithm samples more data, the updated distribution is constrained to those functions that pass through the sampled data points. More specifically, Gaussian processes treat each data point as a random variable from a normal distribution. Given n data points, the Gaussian processes return an n-dimensional vector sampled from an n-variate normal distribution. The ith component of the vector corresponds to the function value of the ith data point. The covariance matrix of the n-variate normal distribution is defined via a kernel function, which takes two data points as inputs and returns a similarity measure between those points. One benefit of Gaussian processes is that the covariance matrix shows the uncertainty associated with each data point. Data points with less information have a higher standard deviation to reflect the lack of knowledge about these points.

After producing an estimate of the objective function, Bayesian optimization uses an *acquisition function* to find the optimum point. The acquisition function determines which point to sample next from the Gaussian process model while balancing the tradeoff between exploration and exploitation. Exploitation means sampling data from regions of the model that is certain to have high performance. On the other hand, exploration samples locations with lower performance but high uncertainty to discover data points that can potentially be better than the current optimum. The most commonly used acquisition functions include Maximum Probability of Improvement, Expected Improvement, and the Upper Confidence Bound (UCB) [BCdF10]. Previous studies [BCdF10] have shown that using UCB has the best results.

Bayesian optimization has been used as an adaptation technique in hexapod control for damage recovery. Cully et al. [CCTM15] used Bayesian optimization on a hexapod to find the best alternative action that compensates the damage. This technique successfully allows a hexapod to adapt to different leg damages, and even works with a missing leg.

2.5 Monte Carlo Tree Search

Most studies in artificial intelligence use tree search algorithms for planning. These algorithms find the next best action by constructing a search tree [Kai90]. Each node in the tree represents a state, and each edge represents an action the agent could take to move from one state to another. For instance, in the game of chess, the nodes and edges correspond to the board positions and moves, respectively. The search algorithm starts from the current state and traverses the search tree by exploring all the possible actions until it reaches the goal state. The algorithm then returns the best action after considering all the possible outcomes.



The four phases of MCTS:

(1) Selection phase traverse the tree by selecting nodes with the highest UCT value.

(2) Expanding phase adds new child nodes to the tree by trying out new actions.

(3) Simulation phase performs random actions until it reaches a simulation cap.

(4) Backpropagation phase updates the values of the nodes the lead to the final result.

Figure 2.4: Monte Carlo Tree Search

While tree search algorithms work on small problems, such as tic-tac-toe and checkers, it becomes infeasible when there are a large number of states and actions. Monte Carlo Tree Search (MCTS) [CBSS08] combines tree search algorithms with Monte Carlo simulations to focus the search on exploring the most promising actions. This new method of exploration significantly reduces the number of states to traverse hence allowing MCTS to plan much faster than classical tree search algorithms. MCTS has been used in various applications, one of which is combining with deep neural networks to achieve superhuman performance in Go [SHM+16].

Whereas classical tree search algorithms build a full search tree at every step, MCTS expands the search tree incrementally. MCTS finds the next best action by executing four phases. The algorithm starts from the current state and traverses the tree based on a selection function, which returns the nodes with the highest estimated value. When it reaches the end of the tree, it expands the node by adding children nodes. Then it picks one of the newly created nodes and simulates a series of random actions until it reaches a simulation cap. Lastly, it propagates the result of the simulation to all the nodes that lead to the final result, updating the value of each node.

The most common selection function in MCTS is Upper Confidence Bounds Applied to Trees (UCT) [KS06]. Each node maintains the mean rewards received for each action, v, the number of times the node has been visited, n, and the number of times its parent node has been visited, N. The value each node is given by the following formula.

$$UBC = v + C\sqrt{\frac{\ln N}{n}}$$

During the selection phase, MCTS chooses the child with the highest value. The constant C is a hyperparameter that controls the exploration-exploitation tradeoff and is tuned for the specific problem domain.

Most selection functions, including UCT, requires trying every action during the expansion phase, which becomes problematic when the action space is very large. The problem can be solved by progressive widening [CWH+08], which artificially limits the number of actions evaluated by MCTS. Progressive widening does not try out every action during the expansion phase, but only considers one additional action. Furthermore, it can decide when to end the selection phase and expand the current node if the performance of the best action for the current node is estimated sufficiently well. While MCTS encourages promising parts of the tree to grow deeper, progressive widening strategy assures the same parts of the tree to grow wider.

Chapter 3

Implementation

3.1 Controller Architecture

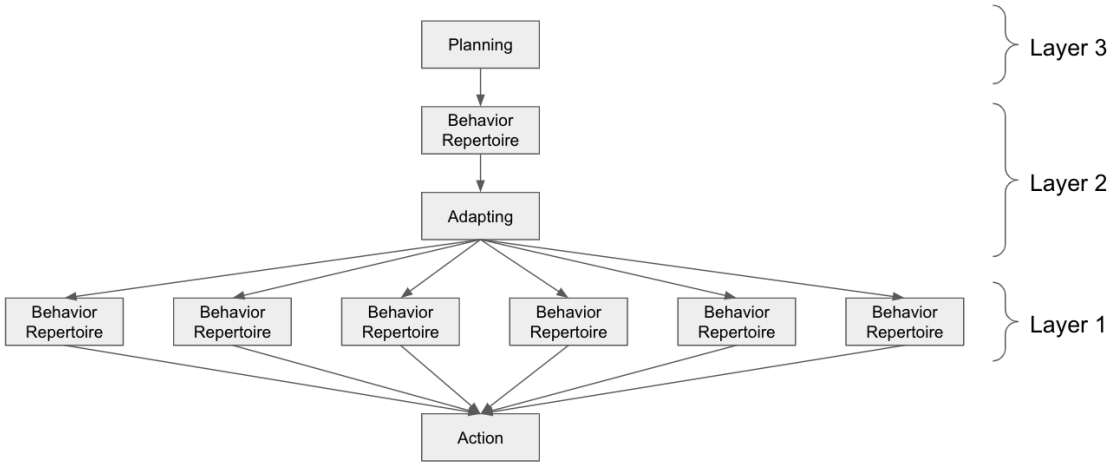


Figure 3.1: Controller Architecture

The objective of the study is to control the hexapod on uneven terrains to reach the goal while avoiding walls. The proposed controller combines hierarchical behavioral-based control, path planning, and adaptation. The idea is to use QD algorithms to generate a collection of walking controllers that can drive the hexapod in different directions. The walking controllers are composed of primitive controllers that control the movement of each leg, which allows the hexapod to adjust the swing height of a walking controller. A path planning algorithm selects the appropriate walking controller to avoid walls and drive the hexapod towards the goal. At the same time, an adaptation algorithm continuously updates the swing height of the hexapod, such that it can overcome the obstacles in its path. Figure 1 shows the overall architecture of the proposed controller. The first layer contains the repertoires of primitive controllers. The second layer is the collection of walking controllers. It also includes an adaptation layer for adjusting the swing heights of the hexapod. Lastly, the third layer is the path planning algorithm to steers the hexapod towards the goal.

The path planning algorithm in layer 3 finds the shortest path to the destination then selects a sequence of walking behaviors to follow the path. The path planning algorithm uses Monte Carlo Tree Search (MCTS) to choose the most suitable walking controllers.

Layer 2 uses the MAP-Elites algorithm to evolve the set of walking controllers. The controllers use primitive actions from the first layer, which controls the movement of each leg. Layer 2 also integrates an adaptation layer for adjusting the swing heights. It uses Bayesian optimization to find the best swing height that allows the hexapod to overcome the obstacles in its path.

Layer 1 consists of six repertoires, each repertoire uses the NS algorithm to discover a diverse set of primitive actions for one leg.

3.2 Primitive Behavior Repertoire

The lowest level of the controller consists of six behavior repertoires, one for each leg of the hexapod. The idea is to breakdown the leg cycle into a sequence of alternating swing and stance phases and considers each of them as a primitive action. Each behavior repertoire contains a set of controllers that controls the swing and stance for one leg. By selecting different controllers, the leg can perform a swing (or a stance) with various widths, heights, and starting positions.

While many other works rely on the MAP-Elites algorithm to generate behavior repertoires [MC15] [DGOC18], the first layer uses the NS algorithm to avoid the need for defining a fitness measure. The main goal of the first layer is to generate a diverse set of controllers for each leg to expand the assortment of walking gaits in the next layer. Another reason for using the NS algorithm instead of MAP-Elites is that swing and stance actions do not have a notion of quality. Using NS can focus the search on evolving unique behaviors without being bias to certain types of controllers. Studies have shown that the NS algorithm can achieve performance comparable to MAP-Elites, in terms of exploration in the behavior space [PSS16].

The original NS algorithm has a current population and an archive that keeps track of all encountered solutions throughout the evolution. The algorithm returns the population after the evolutionary process, which only contains the most novel solutions. On the other hand, the archive holds a more extensive set of behaviors, making it more suitable as the behavior repertoire. Instead of using the population, the first layer adopts the archive of the NS algorithm as the behavior repertoire.

3.2.1 Genotype and Phenotype

The phenotype is an un-driven controller that controls the movement of one leg. Each controller is a set of parameterized periodic functions that define the angular position of the leg joints at time t. More specifically, a controller assigns a separate function to servos 1, 2, and 3, which controls the direction, elevation, and extension of the leg, respectively. The genotype is a set of three parameter values, $\{\alpha_1, \alpha_2, \phi\}$, that specifies the amplitudes and shifts of the periodic functions.

$$\begin{aligned}
\text{servos 1: } & \frac{\pi}{8} \times (\alpha_1 (\cos(2\pi t) - 1) + \phi) \\
\text{servos 2: } & \frac{\pi}{4} \times \alpha_2 \sin(2\pi t) \\
\text{servos 3: } & \frac{\pi}{4} \times -\alpha_2 \sin(2\pi t)
\end{aligned}$$

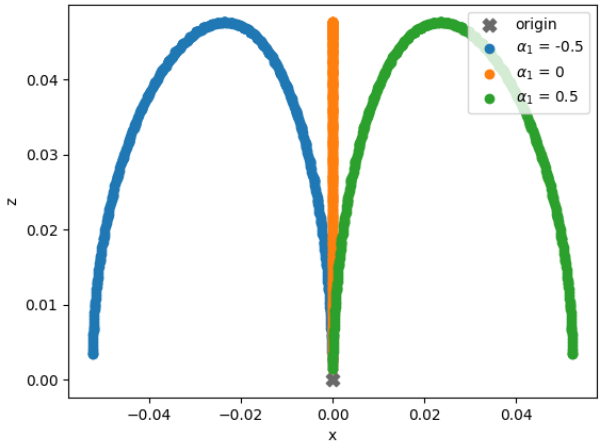
Parameterized periodic functions are one of the simplest methods to implement un-driven controllers. Apart from simplicity, periodic functions do not encounter the black-box problem, which makes it easier to reason and analyze the internal working of these controllers. The controllers have a frequency of 1Hz but are only executed for half a second, such that each controller only completes half of the leg cycle.

The angular positions of servos 1 follow a cosine function that is characterized by the amplitude (α_1) and vertical shift (ϕ). The effect of these parameters is shown in figure 3.2. The amplitude determines the width of the swing (or stance). A positive amplitude swings the leg forward and vice versa. On the other hand, shifting the cosine curve allows the leg to start at various positions.

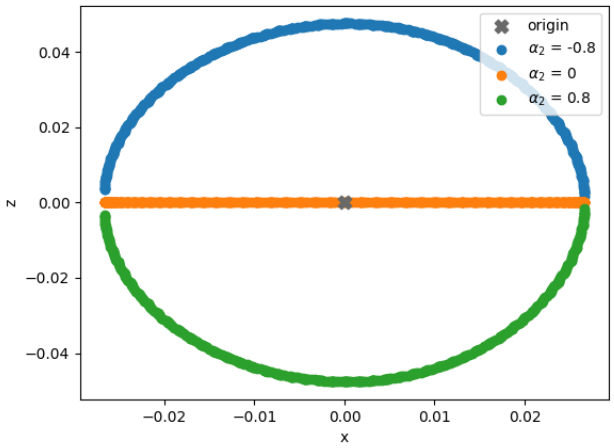
The angular positions of servos 2 follow a sine function. The half cycle of a sine function begins and ends at zero, enforcing the leg to touch the ground at the start and end of each action. Figure 3.2 shows the leg trajectories using different amplitudes for the sine function. A negative amplitude lifts the leg to produce a swing, whereas a positive amplitude pushes the leg towards the ground to form a stance. During the swing phase, the amplitude (α_2) controls the height of the swing. Lastly, to keep the third leg segment perpendicular to the ground, the control signal of servos 3 is the opposite of servos 2.

3.2.2 Behavior Descriptor

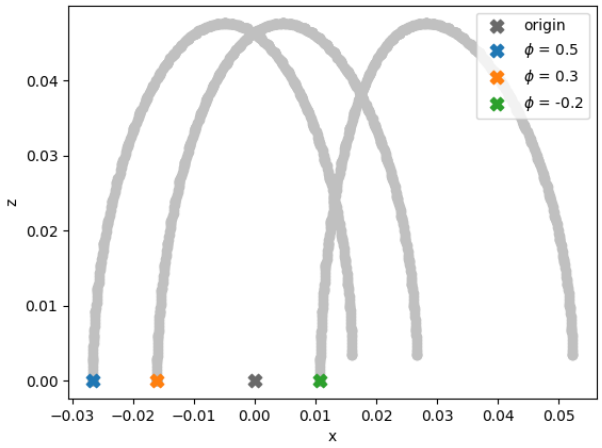
The behavior descriptor not only describes the leg movement of a particular controller but also defines the features that constitute a different behavior. The descriptor should promote behavioral diversity without biasing the search for specific actions. In layer 1, this diversity refers to swing and stance controllers with varying start and end positions. These controllers can be combined



(a) α_1 controls the width of the leg movement



(b) α_2 controls the height of the leg movement



(c) ϕ controls the starting position of the leg movement

Figure 3.2: Primitive Controllers

later to form a leg cycle by choosing two actions, where the start and end positions of one action is the reverse of the other. Moreover, the descriptor should capture different swing heights if a controller were to perform a swing.

Defining a behavior descriptor is one of the most fundamental design decisions that directly influence the outcome of the evolutionary process. An important property of the descriptor is its alignment with the notion of quality. Studies [PSS16] have shown behavior descriptors that align with the goal work best at finding high-quality solutions. However, there is no notion of quality when generation primitive actions in layer 1, and thus no concept of alignment when defining descriptors.

In layer 1, the behavior descriptor, $\{p_s, p_e, h\}$, is a three-dimensional vector that stores the start position, end position, and height of a leg movement. When computing the behavior descriptor, it considers the physical limit of the leg and maps the reachable space to a value between 0 and 1. For example, the positions (p_s and p_e) at 0 and 1 is the furthest the leg can extend backward and forward, respectively. The position at 0.5 is the origin, which is the leg position when the hexapod is in its initial configuration. Similarly, controllers with height (h) 0 or 1 have the largest amplitude. Controllers with a height below 0.5 are stance actions, whereas controllers with a height above 0.5 are swing actions.

3.3 Locomotion Behavior Repertoire

After evolving a repertoire of swing and stance actions for each leg, layer 2 combines these primitive to synthesize complete leg cycles. A solution in layer 2 corresponds to an un-driven controller with a fixed locomotion pattern. These controllers select a swing and a stance action for each leg, then execute the actions alternately in a loop to create different walking gaits. The goal of layer 2 is to create a collection of controllers with different locomotion behaviors such that the robot can move in different directions.

Layer 2 uses the MAP-Elites algorithm to generate the collection of walking controllers. The MAP-Elites algorithm divides the behavior space into discrete cells and aims to find the best performing controller in each cell. The behavior space is a two-dimensional grid, where each cell represents the final x and y displacement of the hexapod after executing a controller for 5 leg cycles. The quality metric of a controller is the difference between the robot's final orientation and the desired orientation. Studies [MC15] have shown that the MAP-Elites algorithm can produce significantly higher-performing and more diverse solutions than other QD algorithms. One drawback of the MAP-Elites algorithm is it only searches for behaviors in the behavioral space predefined by the user, and thus it cannot exhibit open-ended evolution. However, it is possible to determine the maximum distance the hexapod can travel within 5 leg cycles, therefore viable to define a behavior space that covers all the possible actions.

Controllers in layer 2 select primitives from layer 1, by specifying the behavior descriptor of the primitive. More specifically, the controller defines a behavior descriptor, then searches for the primitive action with this behavior in layer 1. Sometimes, such primitive action may not exist in the repertoire, in which case it looks for the action with the closest behavior.

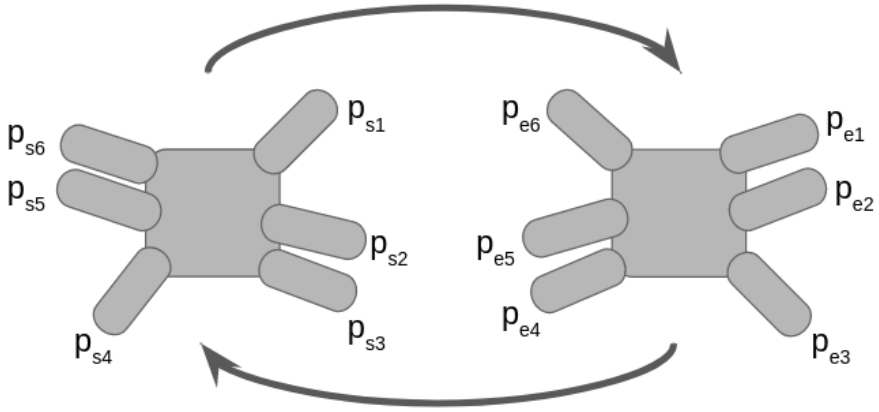
3.3.1 Genotype and Phenotype

The phenotype in layer 2 is a controller that defines the walking behavior of the robot. The locomotive pattern of the controllers conforms to the tripod gait. The tripod gait is the fastest movement gait for hexapod robots [DXZM17], and it is also very stable because three legs are always touching the ground to balance the robot. The controller selects a swing and a stance action for each leg, then executes them alternately for 5 cycles.

The genotype is a set of 12 values that specify the start and end positions of each leg.

$$\{p_{s1}, p_{s2}, p_{s3}, p_{s4}, p_{s5}, p_{s6},\\ p_{e1}, p_{e2}, p_{e3}, p_{e4}, p_{e5}, p_{e6}\}$$

Given the start and end positions of a leg, the controller selects primitives that begin and end at these positions. For example, a genotype that defines the start and end positions of leg 1 to be p_{s1} and p_{e1}, respectively, the corresponding controller selects a swing action from p_{s1} to p_{e1} and a stance action from p_{e1} to p_{s1} in the leg 1 repertoire. Combining these two actions will form a leg



The genotype of a walking controller has 12 values. The first 6 values specify the starting positions of the legs (left), while the last 6 values specify the end positions of the legs (right). The controller alternates between these two configurations for 5 cycles.

Figure 3.3: Walking Controllers

cycle that passes through p_{s1} and p_{e1}. Therefore, a controller chooses a total of 12 primitives from layer 1. To simplify the problem, layer 2 only considers the swing and stance actions with height 0 and 1. In theory, the trajectory of the hexapod depends on the stride width of each leg, and less dependent on the swing height. Therefore, it is reasonable to ignore swing heights when evolving walking controllers but focus more on the start and end positions of the leg cycles.

Since the controllers are restricted to obey a tripod gait, the robot cannot execute arbitrary trajectories. For example, a controller cannot first rotate the robot then travel forward. Instead, the robot can only perform rotations or follow circular trajectories with different curvatures. Fortunately, circular trajectories are adequate for path planning algorithms as most paths can be decomposed into a sequence of arcs.

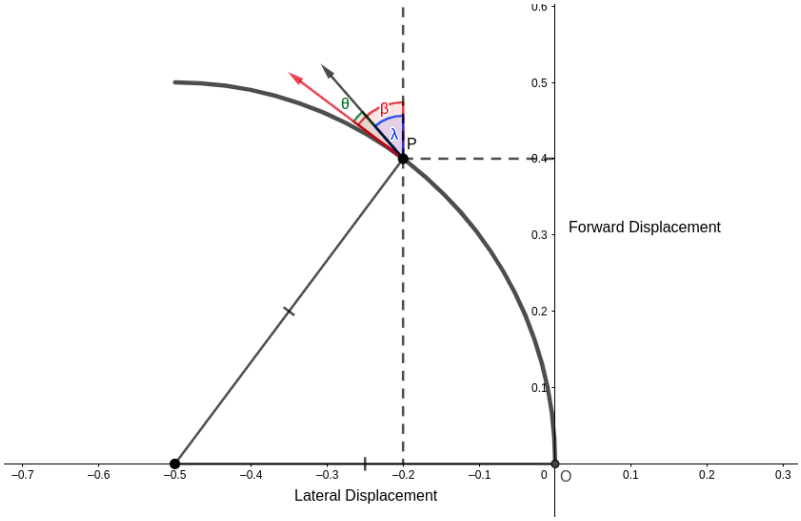
3.3.2 Fitness Function

For the path planning algorithm to combine the walking controllers, it needs to know the final orientation of the previous action to plan the next one. Therefore the MAP-Elites algorithm pays special attention to the arrival orientation of the robot when generating controllers. In particular, the algorithm encourages controllers with arrival orientations that align with their trajectories.

Due to the definition of the controllers, the hexapod can only move in circular paths. These paths can be viewed as arcs of different circles with varying radii and origins, on the lateral axis. The hexapod can also travel in a straight line, considering the circle radius to be infinite. The desired orientation at each point is the tangent of the circle that intersects the robot and that point. The fitness function computes the difference between the arrival orientation, λ, and the desired orientation, β , where controllers with a smaller difference have a higher fitness value.

$$fitness = -|\beta - \lambda| = -|\theta|$$

Using the arrival orientation to compute fitness has several benefits. First of all, controllers with arrival orientation aligned with the trajectory are more likely to have reached the destination traveling the shortest distance. Deviations from the desired angle might indicate the robot performed unnecessary rotations while moving to the location. Therefore, the fitness function can promote efficient walking controllers indirectly. Secondly, the fitness function promotes behavioral continuity, which means nearby controllers in the repertoire display similar behaviors in terms of the arrival orientations. This continuity property is essential to facilitate the path planning algorithm. Since the MAP-Elites algorithm does not guarantee to fill each cell with a solution, the planning algorithm may not be able to find a controller with the desired behavior in the repertoire,



The fitness function calculates the difference between the arrival orientation λ and the desired orientation β. The robot starts from the origin position O facing forward. The arc shows the trajectory of a arbitrary walking controller moving to point P. The red vector shows the desired orientation tangent to the trajectory, and the black vector shows the actual orientation of the hexapod arriving point P.

Figure 3.4: Fitness Function

in which case the planner will search for an alternative controller with the closest behavior. With the continuity property, the planner can easily look for alternatives by exploring the neighboring cells in the repertoire.

3.4 Adaptation

The repertoire in layer 2 contains a collection of walking controllers that can drive the hexapod in different directions. It is possible to increase the versatility of the controllers by varying the swing heights of the legs. Consider the walking gait of a person. A person following the same footsteps will reach the same final position, regardless of how much they lift their leg off the ground in each step. Similarly, using different swing heights does not alter the original behavior of the walking controllers, but adds a dimension of flexibility. This additional adaptability is useful for traversing uneven terrains filled with obstacles of irregular heights. These obstacles will block the leg movement of the hexapod, so the hexapod must lift the leg high enough to step over them. Of course, the hexapod can use the maximum swing height for each step, but this approach is highly inefficient. Instead, the hexapod should try to use the minimum swing height when possible, and only increase the swing height when necessary.

Layer 2 controls the swing height by specifying the height descriptor of the leg controllers. When evolving the walking controller, the MAP-Elites algorithm only considered the leg controllers with height 0 or 1, which correspond to the swing and stance actions with the maximum amplitudes. During adaptation, layer 2 can choose any height values for the swing and stance actions. To simplify the problem, it only considers swing and stance action pairs with the same amplitude. Instead of specifying a separate height value for swing and stance, it uses one value, x, that defines the deviation from largest amplitudes. For example, when x is 0.1, layer 2 finds the swing action with height 0.9 $(1 - x)$ and a stance action with height 0.1 $(0 + x)$.

The adaptation layer of layer 2 uses the Bayesian optimization algorithm for finding the optimal x value. Bayesian optimization is suitable for problems with an unknown objective function. In this case, the objective function describes the energy consumption and collision rate for each swing height. Optimizing this objective function will find the best swing height that balances the trade-off between high energy efficiency and low collision rate. However, this objective function cannot be predefined because the collision rate varies with the environment, and it can only be determined by running the controller.

3.4.1 Gaussian Process

The adaptation layer uses Gaussian processes for the model. Gaussian processes are a common choice for Bayesian optimization [BCdF10], not only because they can model a wide range of functions, but they also provide the uncertainty associated with each prediction. Furthermore, Gaussian processes can incorporate prior knowledge about the shape of the model by choosing different kernel functions. Adding prior knowledge is useful because it can speed up the adaptation process and encourage the hexapod to start with smaller swing heights.

While the collision rate for each swing height is unknown, the model can incorporate prior information about energy consumption. Layer 2 controls the amplitude of the leg movements using x, which is a value between 0 and 0.4. Intuitively, movements with larger amplitudes require more energy because they require more torque at the servos to lift the leg. Therefore, the energy consumption is inversely proportional to x. This prior information is added to the model by modifying the update equation for the mean function. Incorporating prior knowledge encourages the hexapod to start sampling x values near 0.4, which are actions with the smallest swing heights.

$$\mu_t(x) = x$$

Another important feature for Gaussian processes is the kernel function. The kernel function defines the covariance of the Gaussian process and controls the shape of the model. The Gaussian kernel and the Matérn kernel are most commonly used in Gaussian processes [Ras04]. The adaptation layer uses the Gaussian kernel for the model. The Gaussian kernel is infinitely differentiable and therefore creates very smooth models. Stein [Ste99] argues that this strong smoothness assumption is unrealistic for modeling, and suggested the Matérn kernel. However, assuming strong smoothness is reasonable for this problem, because similar swing heights have approximately the same energy consumption and are very likely to have the same collision rate.

$$k(x_1, x_2) = \exp\left(-\frac{|x_1 - x_2|^2}{2l^2}\right)$$

The unknown objective function that the adaptation layer is trying to optimize is defined as

$$\mu_t(x) = x - 0.1 \times collisions(x)$$

where $collisions(x)$ is the number of collisions when using the swing height x. The objective function rewards swing heights with low energy consumption and penalize those with a high collision rate. In every iteration, the model updates its beliefs about the performance of swing height x, after running the controller in the environment. One problem with Bayesian optimization is that it is not designed for dynamic objective functions. Bayesian optimization is traditionally used for finding the optimal value of objective functions that remain the same. However, in this case, the objective function changes with the environment because different parts of the environment can have different collision rates. In other words, the optimal swing height is always changing depending on the hexapod's location. Having an everchanging objective function means that old data points will become unreliable and may have an inaccurate representation of the current landscape. This study made minor modifications to the Gaussian process model such that it emphasizes more on the new data points. More specifically, the model only keeps track of the five most recent data points and discards the older data.

3.4.2 Acquisition Function

Bayesian optimization samples the next swing height by optimizing an acquisition function. The optimal point of the acquisition function corresponds to the swing height that is most likely to have the best performance, after considering the mean and uncertainty of the Gaussian process model. Since the adaptation problem only has one dimension, using grid-based optimization is sufficient for finding the optimal value of the acquisition function. Using grid-based optimization might even outperform other optimizers in small scale problems because it searches exhaustively through the entire domain.

The most commonly used acquisition functions include Maximum Probability of Improvement, Expected Improvement, and the Upper Confidence Bound (UCB) [BCdF10]. Previous studies [BCdF10] have shown that using UCB has the best results.

$$\boldsymbol{x}_{t+1} = \arg\max_{\boldsymbol{x}} \mu_t(\boldsymbol{x}) + k\sigma_t(\boldsymbol{x})$$

UCB finds the next sampling data point by choosing the data point with the highest weighted sum of expected performance and uncertainty. Meaning data points with high performance or high uncertainty have high acquisition function values. The parameter k in the UCB function defines the degree of exploration. With a low k value, the algorithm has a higher tendency to exploit the current best data points. Conversely, higher k values will increase the exploration of the algorithm. This study uses a k value of 0.2 to emphasize exploitation over exploration because the objective function is dynamic. While more exploration can increase the accuracy of the model, this model will become obsolete after a few iterations because the objective function is always changing. Therefore the hexapod should focus more on exploiting the current model to find the best swing height.

3.5 Planning

Layer 3 is the path planning algorithm, which finds the shortest path from the current position to the target position while simultaneously avoiding walls. The planning algorithm selects a sequence of actions, from layer 2, to drive the robot towards the goal. This selection process is equivalent to solving a Markov Decision Process (MDP), which models decision making in discrete, stochastic, sequential environments [Lit15]. The MDP assumes the robot to be in a state at any given time. At each time step t, the robot at state S_t executes an action, from the action space, and move to state S_{t+1}. As a consequence of its action, it receives a reward [SB18]. Solving the MDP involves finding the sequence of actions that maximizes the expected reward.

Path planning can be formulated as an MDP, where the state and action space corresponds to the robot's position and the walking behavior repertoire, respectively. The robot receives a reward if it reaches the goal and a penalty if it collides with a wall. Therefore, optimizing the expected reward means finding a path to the goal without colliding with the walls. Layer 3 solves the MDP using Monte Carlo Tree Search (MCTS) [CBSS08], a sample-based search algorithm for decision processes. While MCTS traditionally samples actions randomly, the planning algorithm guides the MCTS with A* search algorithm to speed up the process. Moreover, the planning algorithm uses model predictive control [Ros03] to handle stochastic actions. Stochastic actions are problematic because the robot will end up in a different position the planner predicted. Model predictive control re-plans the MDP at every step, allowing the robot to correct its course throughout the trajectory.

3.5.1 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is suitable for problems where the best action may not be the most optimal action. For example, in path planning, always choosing the action towards the goal may not be optimal because there are walls in the environment blocking the robot. The MCTS algorithm finds a balance between exploitation and exploration. It exploits the best actions to find the shortest route to the goal, while simultaneously exploring other alternatives in case the shortest route is blocked. Exploration is essential because it helps to discover potentially better paths, but it becomes inefficient when the action space is huge. Therefore, the planning algorithm employs simple progressive widening (SPW) and the A* search algorithm to reduce the amount of exploration.

In the MCTS algorithm, each state stores the hexapod's position and also the orientation, which is needed to predict the next state. During the selection phase, MCTS chooses the child with the highest UCT value. At each selection step, it uses SPW to decide whether to continue exploiting known states or move on to the expansion phase. When expanding a node, MCTS first finds the next best state using the A* search algorithm, then sample actions that bring the robot near that state. This procedure increases the efficiency of the planning algorithm. The simulation phase chooses random controllers from the walking behavior repertoire. Since each controller has a behavior descriptor that describes the new position of the robot and the robot's final orientation is approximately aligned with its trajectory, it is possible to predict the next state of the robot after running the controller. The robot receives a reward, $R_{goal} = 1000$, for reaching the goal and a penalty, $R_{collision} = -1$, for colliding with the walls.

Algorithm 3 Simple Progressive Widening

procedure SIMPLE-PROGRESSIVE-WIDENING(s)
  if $n(s)^{\alpha} > children(s).length()$ **then** ▷ The best action is estimated sufficiently well
    $a \leftarrow$ sampleAction() ▷ Explore new action
  else
    $a \leftarrow \arg\max_{a \in chilren(s)} v(s,a) + C\sqrt{\frac{\ln n(s)}{n(s,a)}}$ ▷ Action with highest UCT value
  return a

3.5.2 A* Search

The A* search algorithm finds the shortest path between two nodes in a graph. The planning algorithm uses A* search to guide MCTS to speed up the planning process. Instead of exploring a random action when expanding a node, MCTS first uses A* search to find the optimal path from the current position to the goal, then explore the actions that approximately follow this path. This procedure improves planning efficiency as MCTS does not have to waste resources exploring solutions that deviate from the optimal solution. While there are other pathfinding algorithms, such as Dijkstra, the A* search algorithm has better performance because it considers additional information about the minimal distance to the target. Moreover, the A* search algorithm is optimal and complete, meaning it guarantees to find the shortest path if it exists.

The A* search algorithm only works on graphs. Therefore to use the algorithm, MCTS discretizes the map into a 10×10 grid and considers each cell as a node. Each cell is connected to all of its neighboring cells unless there is an wall blocking the way. A* search algorithm keeps track of a cost for each node and finds the shortest path by constructing a lowest-cost path from the start node to the destination node. The cost of a node, $f(n)$, is given by the sum of the distance traveled so far, $g(n)$, and the estimated distance to the destination, $h(n)$.

The performance of A* search depends on the accuracy of the heuristic function, $h(n)$. An incorrect heuristic function can prevent the algorithm from finding the optimal path. Ideally, the heuristic function should approximate the exact cost of reaching the destination and should never overestimate. The planning algorithm uses Euclidean distance heuristic, which is an accurate estimate of the cost and will never overestimate.

Chapter 4

Evaluation

4.1 Primitive Behavior Repertoire

Layer 1 uses the Novelty Search algorithm to generate a repertoire of primitive behaviors for each leg. The parameters of the algorithm are shown in Table 4.1. The final behavior repertoire is the NS archive after the evolutionary process. These repertoires should capture as much of the behavior space as possible, creating a diverse collection of swing and stance controllers for the next layer to utilize. The repertoires are evaluated based on the diversity and versatility of the evolved controllers. These qualities are measured using the following metrics:

1. The *collection size* is the total number of controllers in the repertoire. Larger collection size is generally better because it means that the next layer has more options to choose from the repertoire. However, this metric alone cannot indicate diversity because it does not consider the distribution of the solutions. The repertoire may contain lots of controllers but are all concentrated it certain regions of the behavior space.

2. The *coverage score* is the percentage of the behavior space covered by the repertoire. It is calculated by first discretizing the behavior space into a grid then counting the number of cells filled with a controller. Having many solutions does not necessarily mean diversity because all the solutions might have similar behaviors. The coverage percentage depicts the distribution of the controllers in the behavior space. A high coverage indicates the controllers are uniformly-distributed over the behavior space, whereas a low coverage means the controllers are highly-concentrated in some regions of the behavior space.

3. The *versatility score* is the percentage of controllers used by the next layer. The objective of the repertoires is not merely to create diverse behaviors, but also useful behaviors that adapt to different purposes. This score indicates the versatility of the evolved controllers and how beneficial they are to the next layer.

Parameter	**Value**
Initial Size	100
Population Size	2000
Generations	2000
Genotype Size	3
Mutation Type	Polynomial
Mutation Rate	3%
Crossover Rate	disabled
Descriptor Size	3

Table 4.1: Parameter Values for Primitive Behavior Generation

4.1.1 Primitive Repertoire

When designing the behavior descriptor for layer 1, it takes into account the physical limitation of a leg movement and maps this range to a value between 0 and 1. Therefore, the behavior space of a primitive repertoire is a unit cube of side one, where every point inside the cube is a valid controller. Figure 4.1a shows the final behavior repertoires for each leg. These repertoires correspond to the archive of the NS algorithm after the evolutionary process. The behavior repertoires have similar-looking results because all the legs have the same mechanical structure.

Figure 4.1b gives a more detailed view of one of the behavior repertoires. The evolved controllers cover the entire behavior space, which proves that the NS algorithm is an efficient technique for exploring different behaviors. Moreover, there are more controllers located near the center of the behavior space because these controllers are more likely to evolve compared to those located at the boundaries of the behavior space. Controllers further away from the center are more difficult to produce as they require extreme values in their genotype.

Figure 4.1b also shows the leg trajectories associated with the controllers. Taking a vertical slice of the repertoire contains all the controllers starting from the same position. The result shows that the controllers within the vertical slice are uniformly-distributed. These controllers can perform leg movements with various heights, end positions, or any combination thereof. Specifying the height within this slice returns a set of controllers that share the same starting position and height, but reaches different end positions. On the other hand, specifying the end position returns all the swing and stance actions that have the same starting and ending position.

4.1.2 Repertoire Diversity

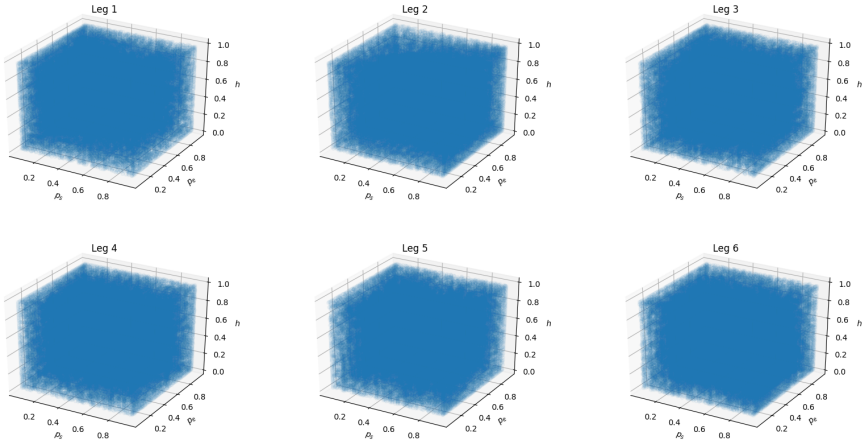
This section studies the diversity of the controllers in the behavior repertoires. More precisely, it examines the size of the archives and their coverage of the behavior space throughout the evolutionary process. Since all the behavior repertoires produced similar results, it is sufficient to analyze one of them in the study. Figure 4.2a shows the evolution of a behavior repertoire, which contains all the primitives that evolved up to a certain generation. It also includes the NS population and a randomly generated collection as a baseline for comparison.

Figure 4.2b shows the change in collection size and coverage of the repertoire throughout evolution. The behavior repertoires contain 70000 controllers on average and cover 88% of the behavior space. The coverage percentage increases rapidly to 81% after the first 200 generations and then level off. On the other hand, the collection size has a steady growth that only starts to slow down after 1800 generations. After 1000 generations, there is no significant increase in the coverage percentage despite the consistent growth in collection size, which suggests that running the NS algorithm for 2000 iterations is sufficient to produce a diverse set of controllers. In comparison, the population of the NS algorithm only contains 2000 controllers and covers 55% of the behavior space. It has a much lower coverage than the archive because it has fewer solutions. The population only contains most novel controllers, whereas the NS archive stores all the controllers throughout the evolution. At the end of the NS algorithm, the size of the archive is 35 times larger than the population, making the NS archive a more suitable choice as the behavior repertoire.

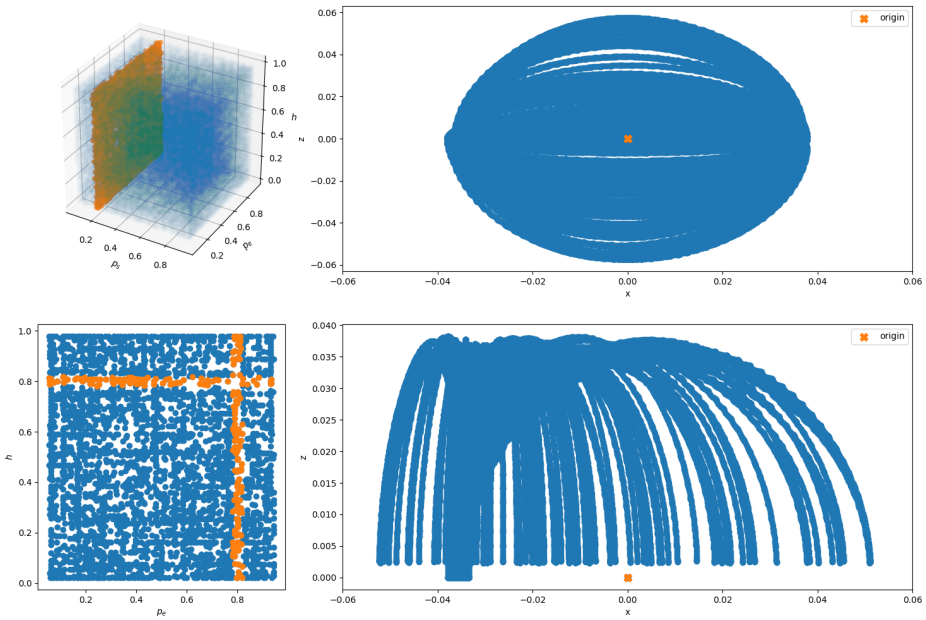
The population of the NS algorithm is compared to a randomly generated collection of the same size, to investigate the exploration capabilities of the NS algorithm. This random collection achieved similar coverage to the population, covering 50% of the behavior space. This result is surprising because random generation does not actively search for different controllers, yet is still able to obtain similar coverage to the NS algorithm. One possible explanation is the simple mapping from the genotype space to the behavior space. There is almost a one-to-one mapping from a genotype to the behavior descriptor. Each parameter of the genotype directly affects one dimension of the behavior space.

$$\{\phi, \alpha_1, \alpha_2\} \rightarrow \{p_s, p_e, h\}$$

This simple mapping diminishes the need for exploring novel controllers since any random genotype can create a different controller in the behavior space. Therefore, randomly generating controllers achieved a comparable result to the novelty search algorithm. Nonetheless, the population of the NS algorithm still obtain a slightly higher coverage, which suggests that actively searching for novel controllers is more or less beneficial for encouraging diversity, even for straightforward problems that seem trivial.



(a) Primitive Repertoires



Top Left: The primitive repertoire of one leg. The vertical slice (orange) highlights all the controllers starting at the same position.

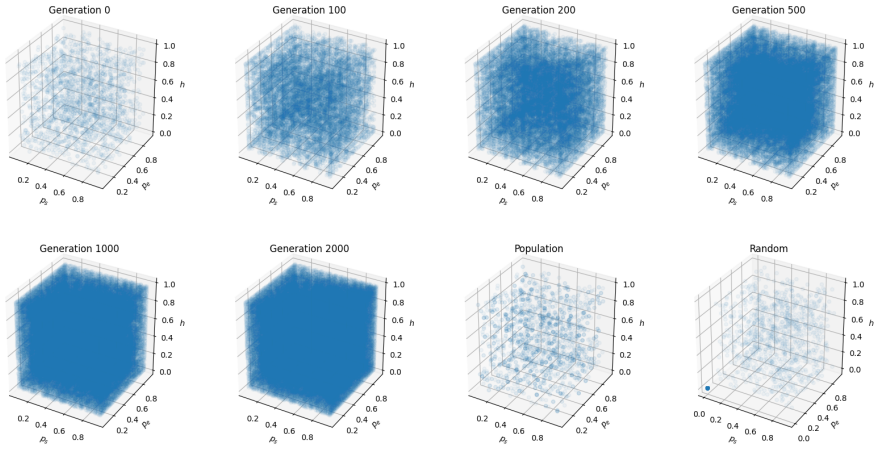
Bottom Left: A vertical slice of the behavior repertoire.

Top Right: All swing and stance actions with the same starting and ending position. These are controllers corresponding to the vertical line (orange) in the slice.

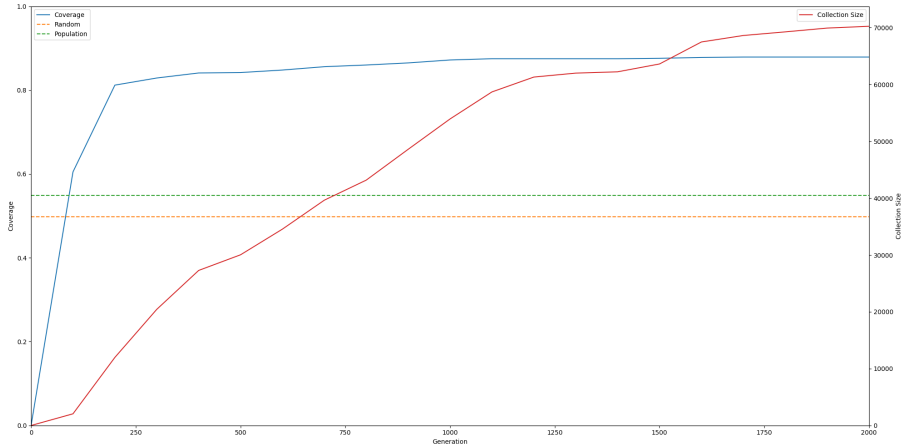
Bottom Right: All actions with the same starting position and swing height. These are controllers corresponding to the horizontal line (orange) in the slice.

(b) Primitive Repertoire

Figure 4.1: Primitive Behavior Repertoire



(a) Repertoire Generation



Blue: The coverage of the NS archive throughout evolution.

Green: The coverage of the final NS population.

Orange: The coverage of the randomly generated repertoire.

Red: The collection size of the NS archive throughout evolution.

(b) Repertoire Coverage

Figure 4.2: Repertoire Evolution

4.1.3 Repertoire Versatility

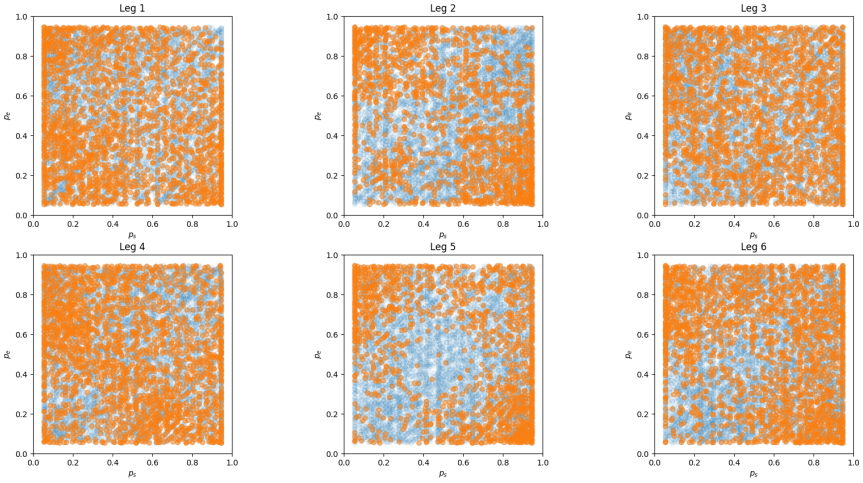


Figure shows the percentage of useful controllers in each primitive repertoire. Controllers selected by layer 2 are shown in orange, while the unused controllers are shown in blue.

Figure 4.3: Repertoire Versatility

The versatility metric tells the proportion of useful controllers in the repertoire. The primitive repertoire may contain a diverse variety of controllers, but layer 2 might only select a few of them if that is all it needs to evolve walking controllers. Ideally, the repertoire should contain a high percentage of useful controllers, where each controller specializes in some locomotion pattern. The MAP-Elites algorithm only uses controllers with height 0 and 1 when evolving the locomotion repertoire, so the versatility score will only consider these controllers when computing the percentage.

Figure 4.3 shows the controllers used by layer 2 in each primitive repertoire. It shows all the available controllers with a height 0 or 1 and the controllers that were chosen by layer 2. The average versatility score is 33%. At first glance, this low versatility score might indicate that layer 1 failed to produce a wide range of useful behaviors. However, this indication is invalid considering the distribution of the selected controllers. In most repertoires, the selected controllers are uniformly-distributed in the behavior space. Meaning different regions of the repertoire contains primitives that perform relatively well in some walking controllers, which suggests that the evolved primitives are indeed diverse and competent for various tasks. The reason for having a lower versatility score is because the primitive repertoires contain a lot of controllers. These repertoires not only contain a diverse set of controllers but also many controllers with similar behaviors. Although this implies the NS algorithm wasted lots of resources evolving primitives, there is no harm to having a larger collection of primitives.

The middle legs of the hexapod have a much lower versatility score than the rest of the legs. The repertoires of leg 2 and leg 5 only contain 28% and 26% of useful controllers, respectively. Unlike other leg repertoires, the selected controllers for the middle legs are concentrated at the two corners of the behavior space. These controllers have endpoints at positions 0 and 1, which corresponds to the controllers with the largest stride width. One hypothesis to explain this uneven distribution is that the middle legs use the same controllers for rotation and moving forward. The middle legs use large stride width to move forward because it gives the maximum speed, and they also use large stride width for rotation because it gives the highest curvature. As a result, most of the controllers with small stride widths are left unused. In comparison, the legs at the front and back use different controllers for rotation and moving forward because they are tilted at an angle. These legs use large stride width for rotation and smaller stride width for moving forward, so their repertoires show a more even distribution of selected controllers.

4.2 Locomotion Behavior Repertoire

Layer 2 uses the MAP-Elites algorithm to generate a collection of locomotion controllers. The parameters of the algorithm are shown in Table 4.2. These controllers allow the robot to travel in different directions. To be able to combine these controllers, the MAP-Elites algorithm encourages controllers with an arrival orientation that aligns with its trajectory. The locomotion repertoire is evaluated based on the diversity and quality of the evolved controllers, which are defined by the following metrics.

1. The *collection size* is the total number of controllers in the repertoire. When using the MAP-Elites algorithm, the collection size directly translates to diversity because the grid container forces the solutions to be uniformly-distributed. Each cell in the MAP-Elites algorithm can only contain one solution. Therefore, a large collection fills up more cells in the container, meaning the solutions cover a higher percentage of the behavior space, thus higher diversity.

2. The *average fitness* is the mean fitness of all the controllers in the repertoire. This metric shows the average quality of evolved controllers. The quality of a controller is the difference between its arrival orientation and the desired orientation. The best controller has a 0 fitness score, which occurs when the arrival orientation is the same as the desired orientation.

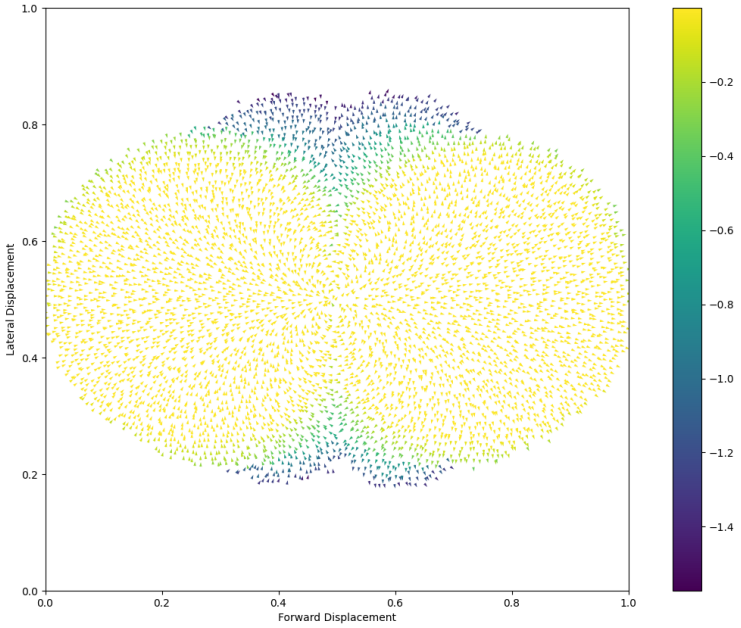
Parameter	**Value**
Initial Size	100
Generations	2000
Genotype Size	12
Mutation Type	Polynomial
Mutation Rate	3%
Crossover Rate	disabled
Descriptor Size	2
Grid Size	100×100

Table 4.2: Parameter Values for Locomotion Behavior Generation

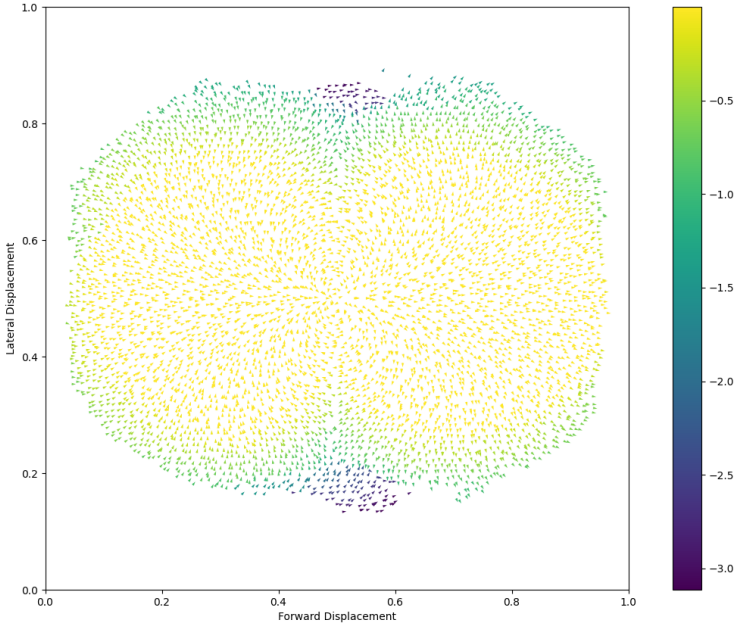
4.2.1 Locomotion Repertoire

Figure 4.4a shows the final repertoire generated by the MAP-Elites algorithm. Each point describes the final position and the arrival orientation of a controller. The hexapod can reach a large number of locations moving forward and backward, but less on the lateral sides. This distribution of the controllers in the behavior space stems from two limitations in the design choice. Firstly, fixing the third leg segment perpendicular to the ground prevents any lateral movements of the hexapod. For the hexapod to move sideways, it must be able to extend the last segment of its leg outwards. When evolving leg controllers, the end of the leg is always pointing downwards to keep to hexapod balanced. This design choice removes a degree of freedom from the leg, hence limits the number of actions the hexapod can perform. Secondly, the restriction of tripod gait forces the hexapod to follow a circular trajectory. This limitation prevents the hexapod from executing arbitrary paths. For instance, it cannot first rotate itself and then travel forward, which would allow the hexapod to reach further positions on its lateral sides.

The controllers in the repertoire have consistently small orientation errors except for those directly on the side of the hexapod. The result shows that controllers with a lateral distance further away from the starting position have lower quality. These locations have significantly larger orientation errors because they are difficult to access. Reaching these positions requires a controller to produce a very high curvature, which only happens when evolving genotypes with extreme parameters. Since it is rare to create such controllers, it makes it more challenging to find new controllers with the same endpoint but better quality. Furthermore, it might be infeasible to produce controllers with the desired orientation using the swing and stance actions from layer 1. In such cases, the current arrival orientation is the best a controller could achieve.



(a) Hierarchical Control



(b) Single Layer Control

Each point describes the final position and arrival orientation of a walking controller. Lighter color indicates higher fitness.

(a) The final repertoire using hierarchical behavioral-based control, which is the one used in this study.

(b) The final repertoire evolving walking controller directly using a single layer.

Figure 4.4: Locomotion Repertoire

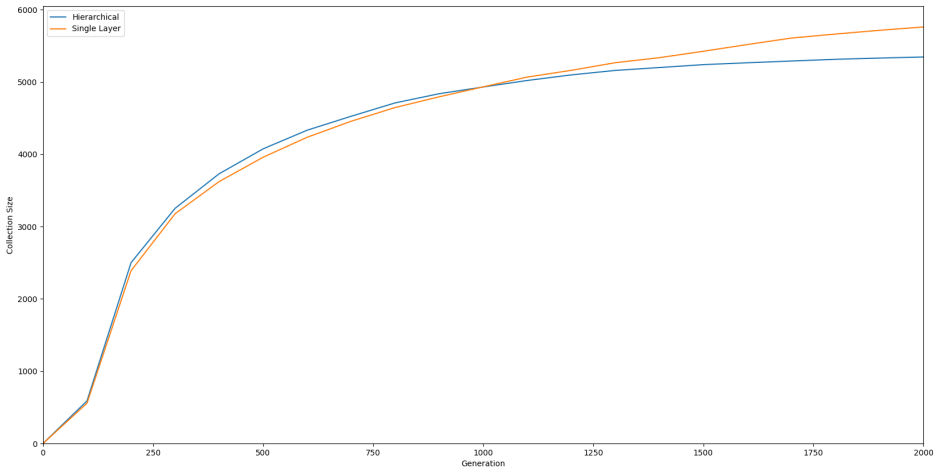
4.2.2 Hierarchical Repertoire

This section evaluates the benefits of using a hierarchical behavioral-based control in terms of coverage and quality of the evolved controllers. To establish a baseline, the study evolves another collection of walking controllers directly without using a hierarchy. The single-layer control uses the MAP-Elites algorithm for evolving controllers. Each controller is a set of parameterized periodic function that controls the angular positions of the servos, as described in section 3.2.1. The MAP-Elites algorithm evolves the 18 parameters, 3 for each leg. The controllers in the single-layer control have the same capabilities as the ones in the hierarchical control. The controllers follow the same periodic functions and have the same degree of freedom, but controllers in the single-layer control do not have to conform to the tripod gait. Restricting the walking behaviors to follow the tripod gait is a design choice that is supposed to generate better controllers by reducing the search space. It is interesting to find out if forcing such a design choice into the evolutionary process will affect the final repertoire.

Figure 4.5b shows the average fitness of the controllers in every generation. The result shows the hierarchical control creates better controllers compared to the traditional single-layer control. The final behavior repertoire of hierarchical control has an average fitness of -0.13, whereas the average fitness of using the single-layer control is -0.26. Multiple features of the hierarchical control contribute to this improvement in quality. First of all, using a hierarchical control reduces the size of the genotype. The hierarchical control reduces the genotype size from 18 to 12 by delegating some of the complexity to the layers below. Reducing the size of the genotype reduces the search space, and hence the complexity of the problem. Secondly, using a hierarchical structure breaks down actions into smaller actions, thus increases the amount of exploration. When evolving controllers using the single-layer control, the MAP-elites algorithm simultaneously explores different leg movements and leg coordinations. In the hierarchical control, layer 1 already searched all the possible leg movements, allowing layer 2 to focus on exploring different leg coordinations to create the best controller. Lastly, the user can integrate design choices in each layer to guide the search. Adding design choices to single-layer control is more difficult because every aspects of the robot is intercorrelated. In contrast, the hierarchical control breaks up everything into different parts, making it easier to target certain aspects of the controller. The design choices incorporate user knowledge on the problem, allowing the MAP-Elites algorithm to find better solutions.

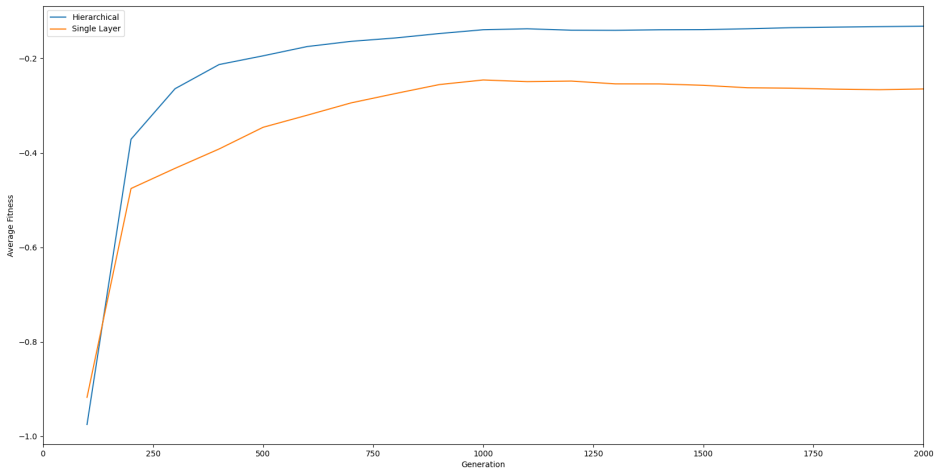
The single-layer structure generated slightly more controllers than the hierarchical structure. The single-layer structure generated 5762 controllers, whereas the hierarchical structure only generated 5347 controllers. The collection of controllers generated by the single-layer structure is shown in figure 4.4b. Same as the hierarchical structure, these controllers cover more of the forward and backward positions and less on the lateral sides. However, the single-layer structure was able to create controllers that can reach further positions at the sides. Since the controllers in the single-layer structure and the hierarchical structure have the same capabilities, the only explanation is that tripod gait reduces the reachable space of the hexapod. Indeed, restricting the controllers to use tripod gaits removes some degree of freedom, and prevents the hexapod from reaching certain positions. Note that the difference in coverage is merely due to the tripod gait constraint, and has nothing to do with the hierarchical structure. Figure 4.5a shows the number of controllers in the repertoire throughout the evolutionary process. The figure shows the collection size for both structures increased at the same rate for the first 1000 generations, which proves that both of them have the same exploration capabilities. The collection size of the hierarchical structure only slows down because it has finished exploring all the reachable space. Otherwise, the collection size of the hierarchical structure will continue to increase without the tripod gait constraint.

One advantage of using a hierarchical structure is it allows the user to intervene with the evolution process at various points. This study has shown there are both advantages and disadvantages for meddling with the evolution process. In some cases, it guides the search to produce better results. In other cases, it may cause unforeseeable side effects that hinder the quality of the final results. In this study, adding the tripod gait constraint may have increased the quality of the controllers at the cost of reducing the reachable space of the hexapod. However, the improvement in quality is far more significant, and so using the tripod gait is the right design choice.



The coverage of the controllers throughout evolution.

(a) Coverage



The average fitness of the controllers throughout evolution.

(b) Average Fitness

Figure 4.5: Repertoire Evolution

4.3 Path Planning

Layer 3 is the path planning algorithm. It selects walking controllers from the locomotion repertoire to drive the robot towards the goal. The planning algorithm uses Monte Carlo Tree Search (MCTS) together with the A* search algorithm for deciding the next walking controller to execute. A good planning algorithm should find the shortest path to the destination, then chooses the set of controllers that follow this path. If there are walls in the environment, the planning algorithm should also find ways to avoid them. This section evaluates the path planning algorithm based on two tasks. The first task requires the hexapod to reach different positions, and the second task requires the hexapod to reach the goal while avoiding walls.

The first task shows how walking controllers can be combined to reach different positions on the map. It also examines the ability of the planner to find the shortest path and choosing the right controllers to walk the path. In an environment without walls, this shortest path is a straight line from the starting position to the destination, and selecting the right controllers should bring the hexapod closer to the goal at every step.

The second task investigates the performance of the planner to avoid walls. The planner should find the shortest path to the goal while keeping the hexapod at a safe distance from the walls. This task also evaluates how well the planner performs in deceptive problems, in which the straight line distance to the goal is blocked by walls.

4.3.1 Reaching Goals

This section analyzes how the path planner combines walking controllers to reach different positions. In this experiment, the hexapod has to reach eight different goals, starting from the center of the map. These goals are located at the corners and the midpoints of each side, surrounding the hexapod. The hexapod must travel in all four cardinal directions, as well as the four intermediate directions, to reach all the goals. Therefore, being able to reach these positions proves that the hexapod can travel in any direction and can move anywhere on the map.

Figure 4.6 shows that the hexapod can achieve all the goals. Each dot represents the position of the hexapod after executing a controller. The dots also show the direction in which the hexapod is facing. The dots form a straight line from the center to the goals, which proves that the planner can find the shortest paths to the targets. Furthermore, the orientation of the hexapod shows that the planner selected the most efficient controllers. For example, the planner used forward-moving controllers to reach the goals in front of the hexapod, and backward-moving controllers for the goals behind it.

The goals of the experiment are purposely chosen, such that the planner has to combine multiple controllers to reach these targets. The results show that each trajectory is composed of arcs, forming an S-Shape motion towards the goal. This pattern is particularly noticeable when hexapod is moving towards the goals at the sides, where the hexapod travels in a series of U-turns. These trajectories demonstrate the flexibility of the walking controllers. Although walking in arcs are inefficient because they travel a longer distance, they can be combined with to reach any position on the map.

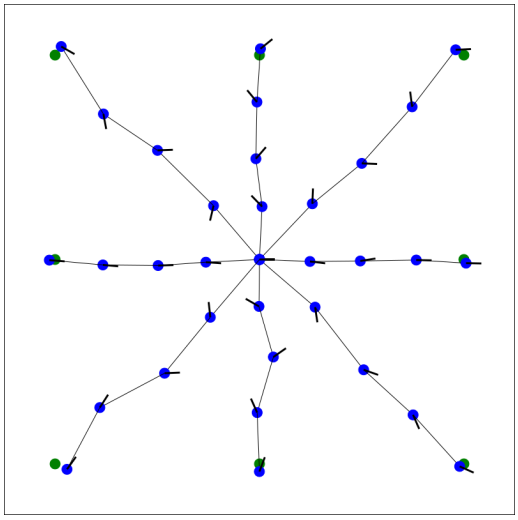
4.3.2 Obstacle Avoidance

Bringing the hexapod to the goal is only one part of the path planning algorithm, the algorithm should also find the shortest path in an environment full of walls. In the second experiment, the hexapod has to go around a wall to reach the destination. This problem is deceptive because the shortest path to the goal gets blocked by the wall. The hexapod must temporarily move away from the goal to reach it. The planning algorithm mainly uses MCTS for controlling exploration and exploitation. The A* search algorithm only assists the MCTS by providing an approximation of the optimal path. This section first evaluates the performance of MCTS on its own, then analyzes the benefits of complementing it with the A* search algorithm.

Figure 4.7a shows the path of the hexapod using only MCTS. For comparison, it also shows the optimal route computed by the A* search algorithm. Each dot represents the position of the hexapod, along with its orientation. Notice the dots only plot the center of the hexapod, but does not represent the actual radius of the hexapod. The planner managed to find a path to reach the goal, which proves that MCTS alone is sufficient to solve the deceptive problem. However, this path is slightly off from the optimal solution. At the beginning of this path, the hexapod deviated

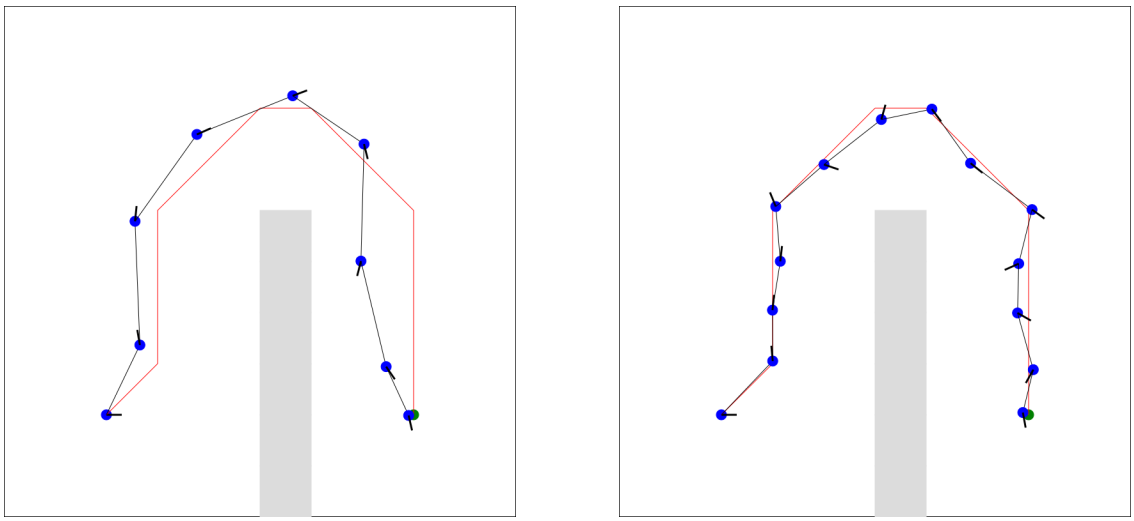
from the optimal solution and traveled a further distance. After the hexapod went around the wall, it collided with the wall before finally reaching the goal. These defects mean that even though MCTS alone successfully found a path to the goal, the algorithm can still be improved.

Figure 4.7b shows the path of the hexapod with the assistance of the A* search algorithm. It shows that using the A* search algorithm to guide MCTS produces a better result that is closer to the optimal path. Apart from having better and safer results, using the A* search algorithm benefits the planning algorithm in multiple ways. First of all, the A* search algorithm speeds up the planning process. Because the A* search algorithm already calculated the optimal route, MCTS does not have to waste time exploring different solutions. Instead, MCTS can use the computed path as reference and sample actions that follow this path. Using A* search reduced the number of iterations of the MCTS from 10000 to 1000. Secondly, using A* search with MCTS creates more waypoints along the path. As shown in the figure, MCTS alone used eight controllers, whereas MCTS with A* search used thirteen. MCTS with A* search used more controllers because MCTS tries to follow every step of the A* search algorithm. The A* search algorithm works by discretizing the map into a grid then consider each cell as a node. Since the distance between each cell is smaller than the maximum distance a controller can travel, following every step of the A* search algorithm will use more controllers. Using more controllers means switching the controllers more often, which is beneficial for adaptation because it allows the hexapod to adjust its swing height more frequently.



The trajectories of the hexapod reaching eight different goals starting the center of the map. Each dot represent the position of the hexapod after executing a controller, the pointer on the dot shows the hexapod's orientation.

Figure 4.6: Different Goals



(a) MCTS

(b) MCTS with A* Search

The robot starts from the bottom left and has to reach the goal at the bottom right by going around the wall at the center. The red line shows the optimal path computed using A* search algorithm.

(a) The trajectory of the robot using only MCTS.

(b) The trajectory of the robot using A* search to guide MCTS.

Figure 4.7: Obstacle Avoidance

4.4 Adaptation

Layer 2 uses Bayesian optimization for controlling the swing heights of the walking controller. It uses a model to estimate the performance of different swing heights. In each iteration, it chooses the best swing height based on the model, then updates the model after executing the controller. This section evaluates the adaption layer based on its ability to choose the right swing height and its adaptability in a changing environment.

The purpose of adaptation is so the hexapod can step traverse uneven terrains using minimal energy. Choosing the right swing height means the hexapod uses a swing height that minimizes both energy consumption and collision rate. This problem alone is difficult because the two objectives contradict each other. While using a smaller swing height uses less energy, it increases the chances of collisions. Having frequent collisions will slow down the hexapod, or even prevents the hexapod from moving forward. Ideally, the hexapod should use the minimum swing height whenever possible and only increase the swing height when it collides with obstacles in the environment.

Adaptability is about how quickly the hexapod can react to changes in the environment. In other words, it tests how long it takes for the hexapod to find the right swing height. In theory, the hexapod will eventually find the right swing height if it stays in the same environment long enough to build an accurate model. In homogenous environments, knowing the hexapod will ultimately converge to the best swing height is good enough, because the hexapod can then use the same height in every part of the environment. However, in most cases, the best swing height varies in different areas of the environment because the obstacles are not uniformly distributed. Therefore the hexapod has to adapt its swing height when transitioning from one area to another.

4.4.1 Choosing Heights

This section evaluates the ability of the hexapod to find the best swing height in a homogenous environment. In this experiment, the hexapod has to reach the goal traversing through uneven terrain. Since this study focuses on choosing the best swing height, it uses an environment without walls and is entirely covered with obstacles. The experiment evaluates the performance of the adaptation by measuring the time it takes for the hexapod to reach the goal and its energy consumption. As a comparison, the study includes the performance of the hexapod using only maximum and minimum swing heights.

1. *Total time* measures how long it took the hexapod to reach the goal, which is an indication of the collision rate. Frequent collisions obstruct the movement of the hexapod, and in some cases, change the course of the hexapod, bringing it further away from the goal. As a result, the hexapod spends more time reaching the goal.

2. *Total energy* measures the energy consumption of the hexapod. Since this energy is directly related to swing height, which is inversely related to x, this study approximates the energy consumption as $1 - x$. The largest swing height ($x = 0$) consumes 1 unit of energy, whereas the smallest swing height ($x = 0.4$) only uses 0.6 units of energy.

Figure 4.8b shows the distance of the hexapod from the goal over time. The hexapod with maximum swing height traveled the fastest, whereas the hexapod with minimum swing height spent twice as much time to reach the goal. When the hexapod used the maximum swing height, the distance from the goal declined consistently because the hexapod was able to avoid most obstacles along the path. In contrast, when the hexapod used the minimum swing height, the distance from the goal decreased irregularly. This irregularity indicates that the hexapod was impeded by the obstacles on its path. The hexapod with adaptation had similar speed as using maximum swing height. This result shows that the adaptation layer successfully found a swing height with a collision rate that is comparable to using the maximum swing height, which is the lowest collision rate the hexapod could achieve.

Figure 4.8c shows the total energy used by the hexapod over time. The hexapod using minimum swing height had the highest total energy. Although using minimum swing height consumes less energy in each step, the hexapod wasted a lot of energy struggling to bypass the obstacles on its way. Also, the obstacles changed the course of the hexapod, which made it took a longer path. As a result, the hexapod using minimum swing height ends up using as much energy as the one using maximum swing height. The result shows that the hexapod with adaptation had the lowest total energy. Its average swing height consumes 0.78 units of energy in each step, which is closer

to the energy consumption rate of the minimum swing height. The figure also shows that the total energy increased at a constant rate, which means the hexapod is choosing controllers with similar heights in each iteration. This consistency proves that the hexapod is choosing swing heights systematically instead of randomly trying out different heights.

These results confirm that the adaptation mechanism can find the best swing height that reduces energy consumption and collision rate. On the one hand, the hexapod achieved a similar collision rate as using the maximum swing height. On the other hand, it uses a lower energy consumption rate that is closer to using the minimum swing height.

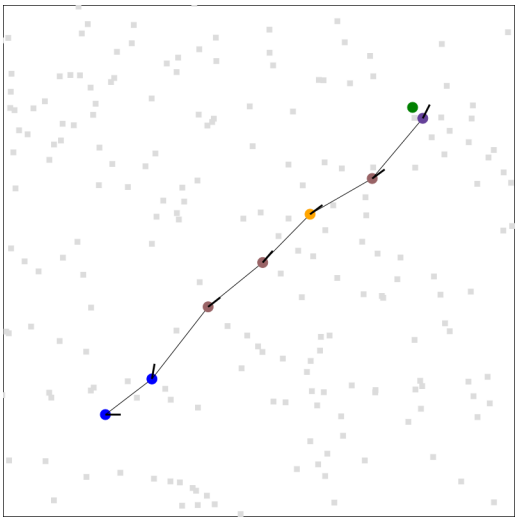
4.4.2 Adaptation in Changing Environments

Choosing the best swing height is only one part of the adaptation. The other part is to adjust the swing height as the hexapod moves across different landscapes. This section investigates the adaptability of the hexapod in changing environments. The environment used in this study is similar to the one in 4.3.2, except part of the map becomes an uneven terrain. The only way to reach the goal requires the hexapod to transition from flat landscape to uneven terrain then back to the flat landscape. This experiment also reveals how the adaptation mechanism collaborates with the rest of the controller.

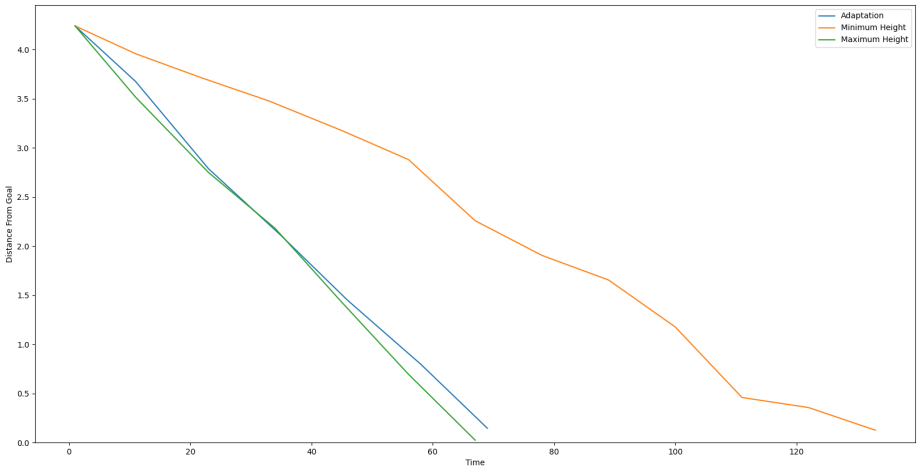
Figure 4.9a shows the trajectory of the hexapod. The trajectory resembles the optimal path computed by the A* search algorithm, which shows that adaptation does not affect the performance of the planning algorithm. More importantly, it proves that altering the swing height does not change the behavior of the walking controllers. Furthermore, the trajectory demonstrates the benefits of using model predictive control. It shows that the hexapod deviated from the original optimal path when traversing the uneven terrain, yet still managed to accommodate these deviations using model predictive control. For example, the hexapod was blocked when it first entered the uneven terrain. As a result, the hexapod stayed in the same position, which is not what the planner predicted. Model predictive control allows the hexapod to replan its path at every step, adjusting its course when the hexapod deviates from the expected behavior.

Figure 4.9a also uses color to represent the swing height of the chosen controllers. Blue and orange denote the minimum and maximum swing height, respectively. All other swing heights are interpolations of the two colors. The result shows that the hexapod uses the minimum swing height on flatland and maximum swing height in the uneven terrain, which demonstrates that the hexapod can find the most suitable swing height according to the landscape. The hexapod changed its swing height instantly when transitioning between different landscapes, which shows that the hexapod can quickly detect and react to changes in the environment. Furthermore, the hexapod did not immediately switch to the maximum swing height when it detected collision, but instead gradually increased the swing height in each iteration. This progressive increment shows that while the hexapod tries to reduce collision, it also minimizes energy consumption.

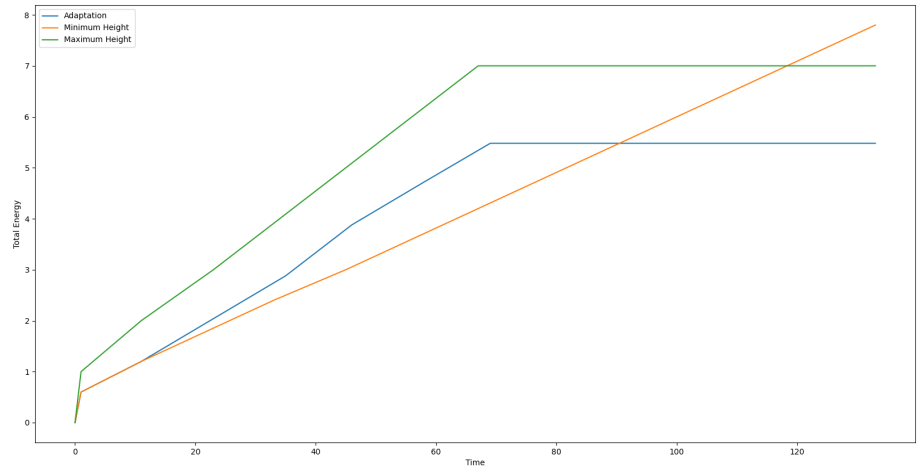
Figure 4.9b shows the Gaussian process model in each iteration, which approximates the performance of the swing heights. In the beginning, this model only incorporates prior knowledge about energy consumption, hence encouraging the hexapod to start with the minimum swing height. During the first few iterations, the hexapod chooses the minimum swing height because it gives the best performance on flat land. Iteration 6 shows how the model changes when the hexapod enters a new environment. The model decreased the estimated performance of the minimum swing height because the hexapod collided with obstacles, consequently the hexapod increases its swing height in the next iteration. Iteration 11 shows how discarding old data allows the model to reselect swing heights that previously had a bad performance. Only keeping track of the five most recent data points allows the minimum swing height to reclaim the highest estimated performance in the model, which means the hexapod can regularly test out smaller swing heights, and switch to using it whenever possible.



(a) Hexapod Path

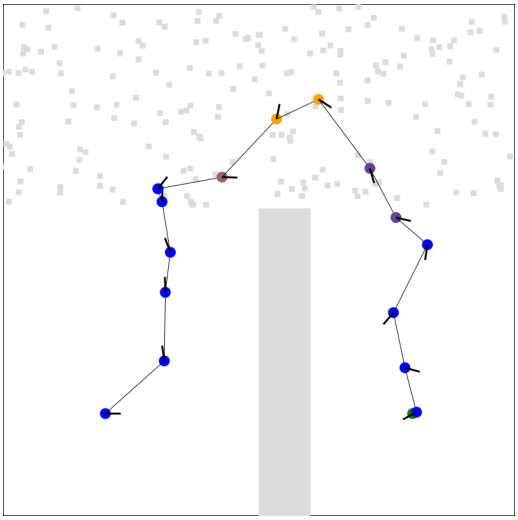


(b) Distance To Goal

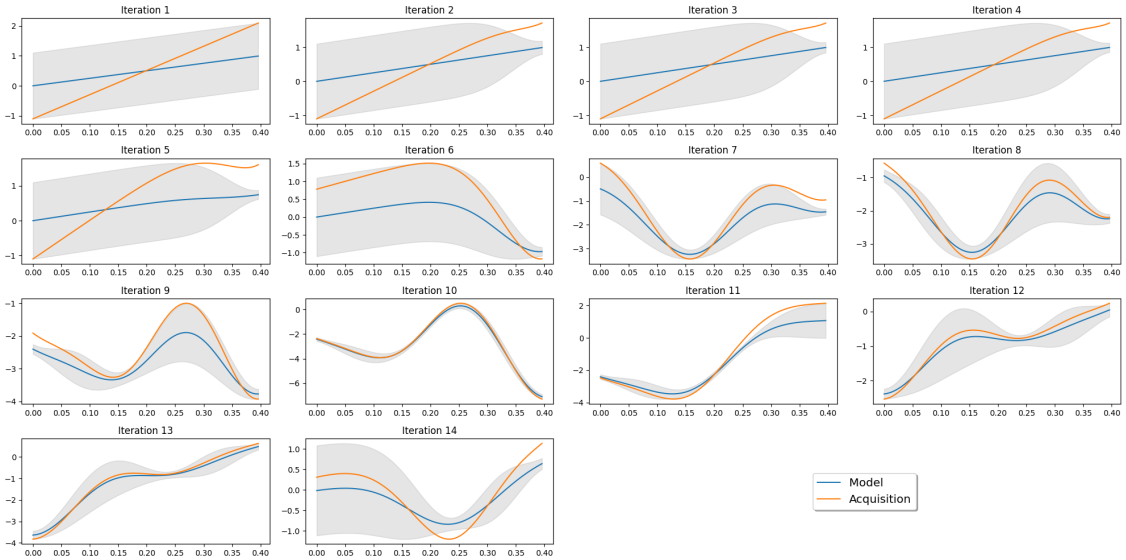


(c) Energy Consumption

Figure 4.8: Adaptation in Homogeneous Environment



(a) Hexpaod Path



(b) Bayesian Optimization

(a) The trajectory of the hexapod. Each dot shows the final position and orientation of the robot after executing a controller. The color represents the swing height of the controller. Orange is the maximum swing height and blue is the minimum swing height. All other swing heights are interpolations of the two colors. The top part of the map is the uneven terrain covered with obstacles.

(b) The Gaussian process model at each iteration. The blue line shows the current model along with the uncertainty. The orange line shows the acquisition function.

Figure 4.9: Adaptation in Changing Environment

Chapter 5

Conclusion

This study aims to create a hierarchical behavioral-based control system to control a hexapod in uneven terrains. The proposed control system consists of three parts. The first part is for the hexapod to learn a set of walking controllers automatically using QD algorithms. The walking controllers are composed of smaller primitive actions that control the swing and stance of each leg. The second part develops a path planning algorithm to use these walking controllers to drive the hexapod. Finally, the third part uses Bayesian optimization to adjusts the swing heights of the hexapod, such that it can walk over the obstacles. Overall, the results show that the proposed controller can successfully control the hexapod to walk on uneven terrains.

This study illustrates the benefits of using a hierarchical behavioral-based control. It shows that using a hierarchical structure can significantly improve the quality of the evolved controllers. More importantly, decomposing walking controllers into primitive actions increases the adaptability of the controllers. In this study, decomposing the walking controllers into a sequence of swing and stance actions allows the hexapod to adjust its swing height, thus giving it the ability to traverse uneven terrains.

Most studies on developing legged controllers focus on driving the hexapod in different directions and focus less on the swing height. This study shows that adjusting the swing heights will not affect the behavior of the walking controllers, and can improve energy efficiency. Bayesian optimization can find the optimal swing height that balances the tradeoff between energy consumption and collision rate. This optimization even works in environments with different landscapes, where the optimal swing height is always changing.

This study is a step towards developing hexapod that can reach arbitrary goals in unknown environments. However, there are still many avenues for improvement. This study assumed the hexapod moves in an environment with only obstacles and walls, and the hexapod does not have to distinguish the two objects. The planning algorithm deals with walls, while the adaptation algorithm handles the obstacles. In the real situation, the hexapod will have to differentiate the two to decide whether it can step over the object. Additionally, behavior repertoires may have to include new controllers to cover more settings. For example, to include actions for climbing staircases or walking on different surfaces.

In future work, the proposed controller can be tested on a physical robot. Cully et al. [CM13b] used the transferability approach with a QD algorithm to evolve a set of controllers for a physical hexapod. For this study, it would be interesting to see if the transferability approach works on simple primitives such as the swing and stance action of a leg. Furthermore, the study can investigate the benefits of using a hierarchical controller to cross the reality gap. In particular, it would be interesting to know if the whole controller can cross the reality gap by only transferring the first layer to the physical robot.

Bibliography

[BA11] Z. Brain and M. Addicoat. Optimization of a genetic algorithm for searching molecular conformer space. *The Journal of chemical physics*, 135:174106, 11 2011.

[BCdF10] E. Brochu, V. M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[CBSS08] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. 01 2008.

[CCTM15] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[CD17] A. Cully and Y. Demiris. Quality and diversity optimization: A unifying modular framework. 2017.

[CD18] A. Cully and Y. Demiris. Hierarchical behavioral repertoires with unsupervised descriptors. *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO'18*, 2018.

[CM13a] A. Cully and J.-B. Mouret. Behavioral repertoire learning in robotics, 2013.

[CM13b] A. Cully and J.-B. Mouret. Evolving a behavioral repertoire for a walking robot, 2013.

[CSPO11] J. Clune, K. O. Stanley, R. T. Pennock, and C. Ofria. On the performance of indirect encoding across the continuum of regularity. *IEEE Transactions on Evolutionary Computation*, 15(3):346–367, 2011.

[CVM18] K. Chatzilygeroudis, V. Vassiliades, and J.-B. Mouret. Reset-free trial-and-error learning for robot damage recovery. *Robotics and Autonomous Systems*, 100:236–250, Feb 2018.

[CWH+08] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04:343–357, 11 2008.

[DGOC18] M. Duarte, J. Gomes, S.M. Oliveira, and A.L. Christensen. Evolution of repertoire-based control for robots with complex locomotor systems. *IEEE Transactions on Evolutionary Computation*, 22(2):314–328, 2018.

[DOC15] M. Duarte, S. Oliveira, and A. Christensen. Evolution of hybrid robotic controllers for complex tasks. *Journal of Intelligent Robotic Systems*, 78:463–484, 2015.

[DPAM02] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[DXZM17] H. Deng, G. Xin, G. Zhong, and M. Mistry. Gait and trajectory rolling planning and control of hexapod robots for disaster rescue applications. *Robotics and Autonomous Systems*, 95:13–24, 06 2017.

[ES08] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2008.

[FKM70] D. Filliat, J. Kodjabachian, and J.-A. Meyer. Incremental evolution of neural controllers for navigation in a 6-legged robot. 1970.

[FOW66] Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. Artificial intelligence through simulated evolution. 1966.

[GLY19] D. Gravina, A. Liapis, and G.N. Yannakakis. Quality diversity through surprise. *IEEE Transactions on Evolutionary Computation*, 23(4):603–616, 2019.

[GOC18] J. Gomes, S. Oliveira, and A. Christensen. An approach to evolve and exploit repertoires of general robot behaviours. *Swarm and Evolutionary Computation*, 2018.

[Gur17] C. S. Gurel. Hexapod modelling, path planning, and control. 05 2017.

[HTY+00] G. S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita. Evolving robust gaits with aibo. 3:3040–3045 vol.3, 2000.

[HTYF05] G. S. Hornby, S. Takamura, T. Yamamoto, and M. Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410, 2005.

[ICRC07] A. J. Ijspeert, A. Crespi, D. Ryczko, and J.-M. Cabelguen. From swimming to walking with a salamander robot driven by a spinal cord model. *Science*, 315(5817):1416–1420, 2007.

[Kai90] H. Kaindl. Tree searching algorithms. In T. Anthony Marsland and Jonathan Schaeffer, editors, *Computers, Chess, and Cognition*, pages 133–158. Springer New York, 1990.

[KGUD10] A. Kapoor, K. Grauman, R. Urtasun, and T. Darrell. Gaussian processes for object categorization. *International Journal of Computer Vision*, 88(2):169–188, 2010.

[KL07] H. Kim and J. Lee. Clustering based on gaussian processes. *Neural Computation*, 19:3088–3107, 2007.

[KRN08] J. Z. Kolter, M. P. Rodgers, and A. Y. Ng. A control architecture for quadruped locomotion over rough terrain. pages 811–818, 2008.

[KS06] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293. Springer Berlin Heidelberg, 2006.

[Lit15] M. L. Littman. Markov decision processes. In James D. Wright, editor, *International Encyclopedia of the Social Behavioral Sciences (Second Edition)*, pages 573 – 575. Elsevier, second edition edition, 2015.

[LLC+19] T. Li, N. Lambert, R. Calandra, F. Meier, and A. Rai. Learning generalizable locomotion skills with hierarchical reinforcement learning. 2019.

[LS11a] J. Lehman and K. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–233, 2011.

[LS11b] J. Lehman and K. Stanley. Evolving a diversity of creatures through novelty search and local competition. *Genetic and Evolutionary Computation Conference, GECCO'11*, 2011.

[MC15] J.-B. Mouret and J. Clune. Illuminating search spaces by mapping elites. 2015.

[MD09] J.-B. Mouret and S. Doncieux. Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity. 2009.

[MDM06] J.-B. Mouret, S. Doncieux, and J.-A. Meyer. Incremental evolution of target-following neuro-controllers for flapping-wing animats. 2006.

[MMP07] H. Maaranen, K. Miettinen, and A. Penttinen. On initial populations of a genetic algorithm for continuous optimization problems. *J. of Global Optimization*, 37(3):405–436, 2007.

[Moc12] J. Mockus. *Bayesian Approach to Global Optimization: Theory and Applications*. Mathematics and its Applications. Springer Netherlands, 2012.

[NTL+19] O. Nachum, H. Tang, X. Lu, S. Gu, H. Lee, and S. Levine. Why does hierarchy (sometimes) work so well in reinforcement learning? 2019.

[PSS16] J. Pugh, L. Soros, and K. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3, 07 2016.

[Ras04] C. E. Rasmussen. *Gaussian Processes in Machine Learning*, pages 63–71. Springer Berlin Heidelberg, 2004.

[Ros03] J. Rossiter. *Model-based Predictive Control-a Practical Approach*. 01 2003.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[SHM+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[Ste99] M.L. Stein. *Interpolation of Spatial Data*. Springer My Copy UK, 1999.

[YG10] X. Yu and M. Gen. *Introduction to Evolutionary Algorithms*. Springer London, 2010.

[ZBL04] V. Zykov, J. Bongard, and H. Lipson. Evolving dynamic gaits on a physical robot. 2004.