Mechanical Design and Control of Flexible-Legged Jumping Robots

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Masters of Science

Andrew Albright

Spring 2022

Mechanical Design and Control of Flexible-Legged Jumping Robots

Andrew Albright

APPROVED:

Joshua E. Vaughan, Chair Assistant Professor of Mechanical Engineering	Anthony S. Maida Associate Professor of Computer Science
Alan A. Barhorst Department Head of Mechanical Engineering	Mary Farmer-Kaiser Dean of the Graduate School

To all the poor souls using Word, one day you will see the light that is LaTeX.

"Before we work on artificial intelligence why don't we do something about natural stupidity?"

— Steve Polyak

Acknowledgments

I would like to firstly thank my advisor Dr. Joshua Vaughan for his leadership, both academic and personal, during my time here at the university. His support has been invaluable in helping me to understand the underlying ideas behind this material. Additionally, his constant drive to produce high quality material has kept me motivated in my attempts to do the same. I would not be here without him.

Additionally, I would like to thank my committee members, Dr. Alan Barhorst, Dr. Anthony Maida and Dr. Brian Post for their input in regards to this work. Along with my lab members during my time in the C.R.A.W.L.A.B., Gerald Eaglin, Adam Smith, Y (Eve) Dang, Brennan Moeller and Darcy LaFont. Their conversations and advice have greatly assisted me in my efforts to complete this work.

Lastly, I would like to thank the Louisiana Crawfish Board for their financial support in the form of a grant from the University of Louisiana at Lafayette. This grant has allowed me to work knowing my fiscal security was in tack.

Table of Contents

Dedication . . . iv

Epigraph . . . v

Acknowledgments . . . vi

List of Tables . . . ix

List of Figures . . . x

I Introduction and System Description . . . 1
- **1.1 Improving Performance with Flexible Components** . . . 2
- **1.2 Controlling Flexible Systems** . . . 3
- **1.3 Concurrent Design** . . . 3
- **1.4 Reinforcement Learning** . . . 4
- **1.5 Twin Delayed Deep Deterministic Policy Gradient** . . . 5
- **1.6 Contributions** . . . 9

II Learning Efficient Jumping Strategies for the Monopode System . . . 11
- **2.1 Monopode Jumping System** . . . 12
- **2.2 Training Environment** . . . 13
- **2.3 Efficient Control Strategies** . . . 14
- **2.4 Input Complexity** . . . 16
- **2.5 Deploying TD3** . . . 17
- **2.6 Average Performance of Network Controller** . . . 18
  - **2.6.1 Input Commands** . . . 18
  - **2.6.2 Jumping Height Performance** . . . 19
  - **2.6.3 Height Reached vs. Power Used** . . . 21
- **2.7 Optimal Performance of Network Controller** . . . 21
  - **2.7.1 Input Commands** . . . 21
  - **2.7.2 Jumping Height Performance** . . . 23
- **2.8 Conclusion** . . . 24

III Using Input Shaping to Validate RL Controller . . . 25
- **3.1 Input Shaping Controller Input** . . . 25
- **3.2 RL Controller Input** . . . 25
- **3.3 Training a Robust Controller** . . . 26
- **3.4 Conclusion** . . . 27

IV Mechanical Design of a the Monopode Jumping System . . . 28
- **4.1 Controller Input** . . . 28
- **4.2 Environment Definition** . . . 29
- **4.3 Rewards for Learning Designs** . . . 30

4.4 Ability to Learn a Design . . . 30
4.5 Deploying TD3 . . . 32
4.6 Jumping Performance . . . 33
4.6.1 Narrow Design Space . . . 33
4.6.2 Wide Design Space . . . 35
4.6.3 Average Design Performance . . . 37
4.7 Conclusion . . . 38

V Concurrent Design of the Monopode System . . . 39
5.1 Concurrent Design Architecture . . . 39
5.2 Environment Definition . . . 40
5.2.1 Learning the Controller . . . 40
5.2.2 Learning the Design . . . 40
5.3 Deploying the Algorithm . . . 41
5.4 Mechanical Design Update . . . 41
5.4.1 Discrete vs. Continuous . . . 41
5.4.2 Averaging n Design Policies . . . 42
5.5 Mechanical Designs . . . 42
5.6 Controller Performance . . . 43
5.7 Conclusion . . . 44

VI Concurrent Design of a Two-Link Flexible-Legged Jumping System . 45
6.1 Jumping Cases . . . 45
6.2 Mechanical Designs: Simulation . . . 45
6.3 Controller Performance: Simulation . . . 45
6.4 Mechanical Designs: Sim-to-Real . . . 45
6.5 Controller Performance: Sim-to-Real . . . 45
6.6 Conclusion . . . 45

VII Appendix: Concurrent Design Algorithm . . . 46
7.0.1 Additional Hyperparameters . . . 46
7.0.2 Discrete vs. Continuous . . . 46

Bibliography . . . 51

Abstract . . . 55

Biographical Sketch . . . 56

List of Tables

Table 1. Monopode Model Parameters 11

Table 2. TD3 Training Hyperparameters 17

Table 3. TD3 Training Hyperparameters 32

Table 4. Learned Design Parameters 37

List of Figures

Figure 1. Flexible Robotics System 1

Figure 2. Rotary Style Series Elastic Actuator 2

Figure 3. Tendon Like Flexibility from [1] This is blurry af. 2

Figure 4. Reinforcement Learning Process 4

Figure 5. Twin Delayed Deep Deterministic Policy Gradient Block Diagram with Monopode as Environment 6

Figure 6. Monopode Jumping System 12

Figure 7. Example Single Jump 15

Figure 8. Example Stutter Jump 15

Figure 9. Reward vs Time Step During Training 17

Figure 10. Average and Standard Deviation Inputs to monopode 18

Figure 11. Average and Standard Deviation Heights of monopode 19

Figure 12. Height Reached vs Power Consumed of monopode 21

Figure 13. Optimal Inputs to monopode 22

Figure 14. Optimal Heights of monopode 23

Figure 15. Jumping Command 25

Figure 16. Resulting Actuator Motion 26

Figure 17. Decomposition of the Jump Command into a Step Convolved with an Impulse Sequence 26

Figure 18. Jumping Performance of Narrow Design Space 31

Figure 19. Jumping Performance of Broad Design Space 31

Figure 20. Reward vs. Episode for Learning Mechanical Design 32

Figure 21. Height Reached During Training 33

Figure 22. Spring Constant Selected During Training 34

Figure 23. Damping Ratio Selected During Training 34

Figure 24. Height Reached During Training 35

Figure 25. Spring Constant Selected During Training 36

Figure 26. Damping Ratio Selected During Training 36

Figure 27. Height vs Time of Average Optimal Designs 37

Figure 28. Designs Learned Using Update Rate = 1:500 Control Policy Updates 42

Figure 29. Average Input Performance 43

Figure 30. Average Jumping Performance 44

I Introduction and System Description

A legged locomotive robot can have many advantages over a wheeled or tracked one, particularly in regards to their ability to navigate uneven and unpredictable terrain [2, 3]. They can achieve this advantage because of the numerous movement types they can deploy. Abilities such as independently placing their feet within highly rigid terrain and jumping or bounding over obstacles have been shown to be effective ways of locomoting [4]. These advantages do not come at no cost, however. Legged systems are traditionally power inefficient compared to wheeled vehicles making them a less attractive option for applications where power conservation is required. Research has been conducted showing the usefulness of adding flexible components, like the legs seen on the robot in Figure 1, for combating efficiency and other issues [3, 5, 6]. The addition of these components in legged robots has been shown to increase system performance measures such as running speed, jumping capability and power efficiency [7]. However, the addition of flexible components creates a system that is highly non-linear, and thus requires a more complex control system.

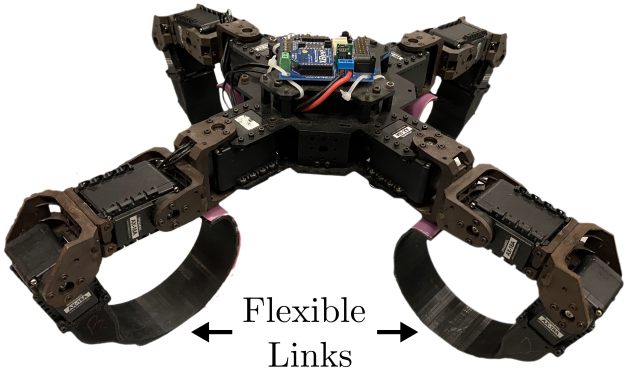


Figure 1. Flexible Robotics System

1.1 Improving Performance with Flexible Components

The addition of flexible components within robotic systems has been shown to be an effective way of improving performance metrics such as movement velocity and power efficiency [3, 7]. Of the different techniques that have be deployed the use of series elastic actuators (SEAs) has been shown to be very effective for increasing energy efficiency [8, 9]. Storing energy in the non-rigid parts of motor joints such as the spring seen in Figure 2 have proven to be an effective way in increasing efficiency. The addition of flexible joints is not the only technique that has been used to improve performance however; utilizing tendon like elastic members to connect actuators to links has also been shown to be an effective way of improving efficiency [1]. The use of tendons, being an example of replicating what is found in nature, is a common method

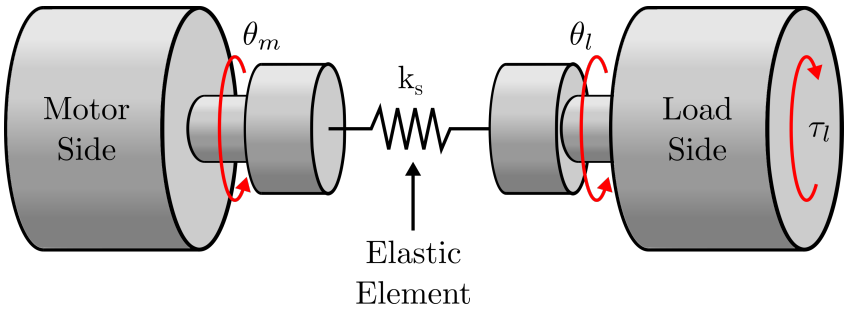
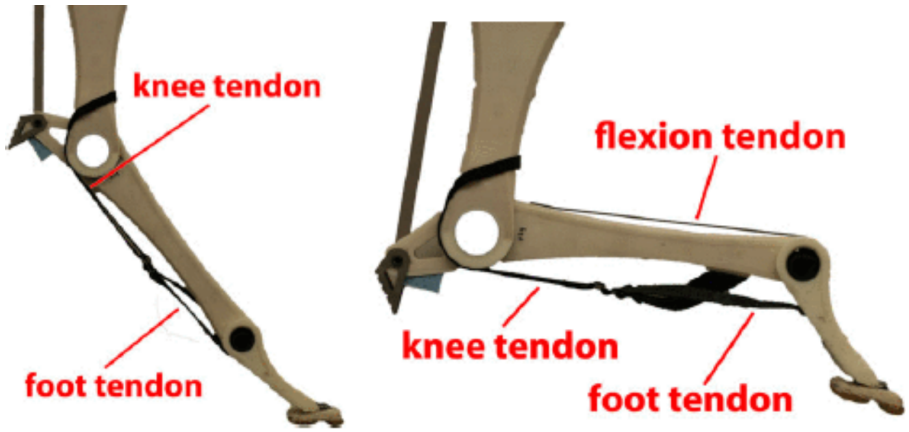


Figure 2. Rotary Style Series Elastic Actuator



(a) Load on leg (ground phase): knee tendon taut.

(b) No torque applied by knee parallel spring tendon due to ankle folding.

Figure 3. Tendon Like Flexibility from [1] This is blurry af.

of finding unique mechanical designs that perform well in the real world. An example of this type of design can be seen in Figure 3. Following a similar idea, research has also been conducted finding the usefulness of including flexibility in the the spine of of 2D running robots where the velocity of the robot was drastically increased [10]. Research studying the effects of flexible links, like the ones shown in Figure 1 is limited though, particularly in the realm of legged-robots. Still, it has been shown as a viable method of increasing performance in these types of robots [11].

1.2 Controlling Flexible Systems

Control methods developed for flexible systems have been shown to be effective for position control and vibration reduction [9, 12]. Because of the challenges seen in scaling the controllers, methods utilizing reinforcement learning are of interest. This method has been used in simple planar cases, where it was compared to a PD control strategy for vibration suppression and proved to be a higher performing method [13]. Additionally, it has also been shown to be effective at defining control strategies for flexible-legged locomotion. The use of actor-critic algorithms such as Deep Deterministic Policy Gradient [14] have been used to train running strategies for a flexible legged quadruped [15]. Much of the research is based in simulation, however, and often the controllers are not deployed on physical systems, which leads to the question of whether or not these are useful techniques in practice.

1.3 Concurrent Design

Defining an optimal controller for a system can be difficult due to challenges such as mechanical and electrical design limits. This is especially true when the system is flexible and the model is nonlinear. A solution to this challenge is to concurrently design a system with the controllers so that the two are jointly optimized. Defining the design process such that the robot's design results in a simple dynamic model has been shown to improve the performance of mechatronics systems [16]. Additionally, in more

recent work, the utilization of more complex deep learning methods have shown to be an effective strategy for finding optimal concurrent designs [17]. It has been used to find a concurrent designs for simulated legged robotic systems leading to improved performance in regards to movement velocity [18]. Some research has even been completed where the designs were deployed on physical hardware, validating that this area of research is an effective one for learning how best to define system/controller architectures [19]. Little research exists however, utilizing these techniques on legged-robotic systems, particularly ones that are flexible in nature.

1.4 Reinforcement Learning

With the recent successes seen in utilizing reinforcement (RL) learning to define control strategies, design parameters and concurrent designs for robot systems, it is of interest to apply this technique in a unique way to flexible-legged jumping systems. Firstly, it is important to understand what reinforcement learning is.

Reinforcement Learning is the process of training a policy to define a series of commands using an environment where those commands can be applied. A policy, often referred to as an agent, from a controls theory perspective, is synonymous with a controller. The environment the controller is deployed in, again from a controls theory perspective, is synonymous with a robotic system. Training the controller requires iteratively deploying the controller's commands, or actions, to the the environment and

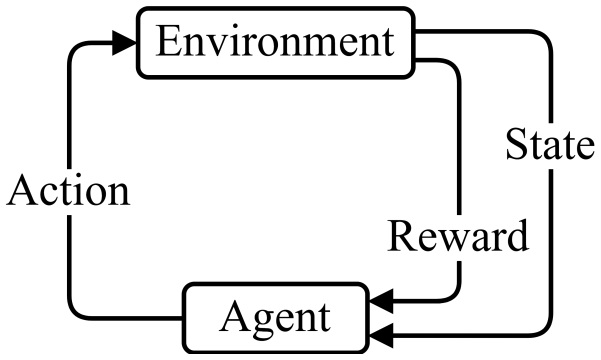


Figure 4. Reinforcement Learning Process

observing the results. The results are often in the form of the state of the environment and a reward resulting from the action that was applied. The reward is defined by the designer so that the controller is trained to accomplish a desired task. Other than the reward, the controller has no way to decern what commands are good when the environment is in some state. The iterative learning process is often described using the block diagram shown in Figure 4.

Learning an optimal control strategy to accomplish a task is completed by deploying a gradient decent based learning algorithm utilizing information such as the state of the environment and the reward. For general robotics applications, at each discrete time step t, the environment will be in a state $s \in \mathcal{S}$, the controller will select an action $a \in \mathcal{A}$ according to the current policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ and apply said action within the environment. The environment will transition to a new state s' and will generate a reward r based on the users definition. The return, being the value the algorithm is trying top optimize, is defined as a discounted sum or rewards, $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$, where γ is a discount factor for discerning between giving importance to near-term or long-term rewards.

The challenge of an RL algorithm is to optimize a policy, π_ϕ, with parameters, ϕ, such that actions generated at each time step will maximize the return. Ultimately an optimized policy will maximize the expected return, $J(\phi) = \mathbb{E}_{s_i \sim p_\pi, a_i \sim \pi}[R_0]$.

1.5 Twin Delayed Deep Deterministic Policy Gradient

There are many algorithms used to tain a network based controller in an RL application, some of which have shown their ability to learn high performing control strategies for robotics systems [20–22]. Of the different algorithms used in research today, the one selected and tested in this work is Twin Delayed Deep Deterministic Policy Gradient (TD3) [23]. This is an actor-critic type learning algorithm which is widely considered the predecessor to the popular and proven Deep Deterministic Policy

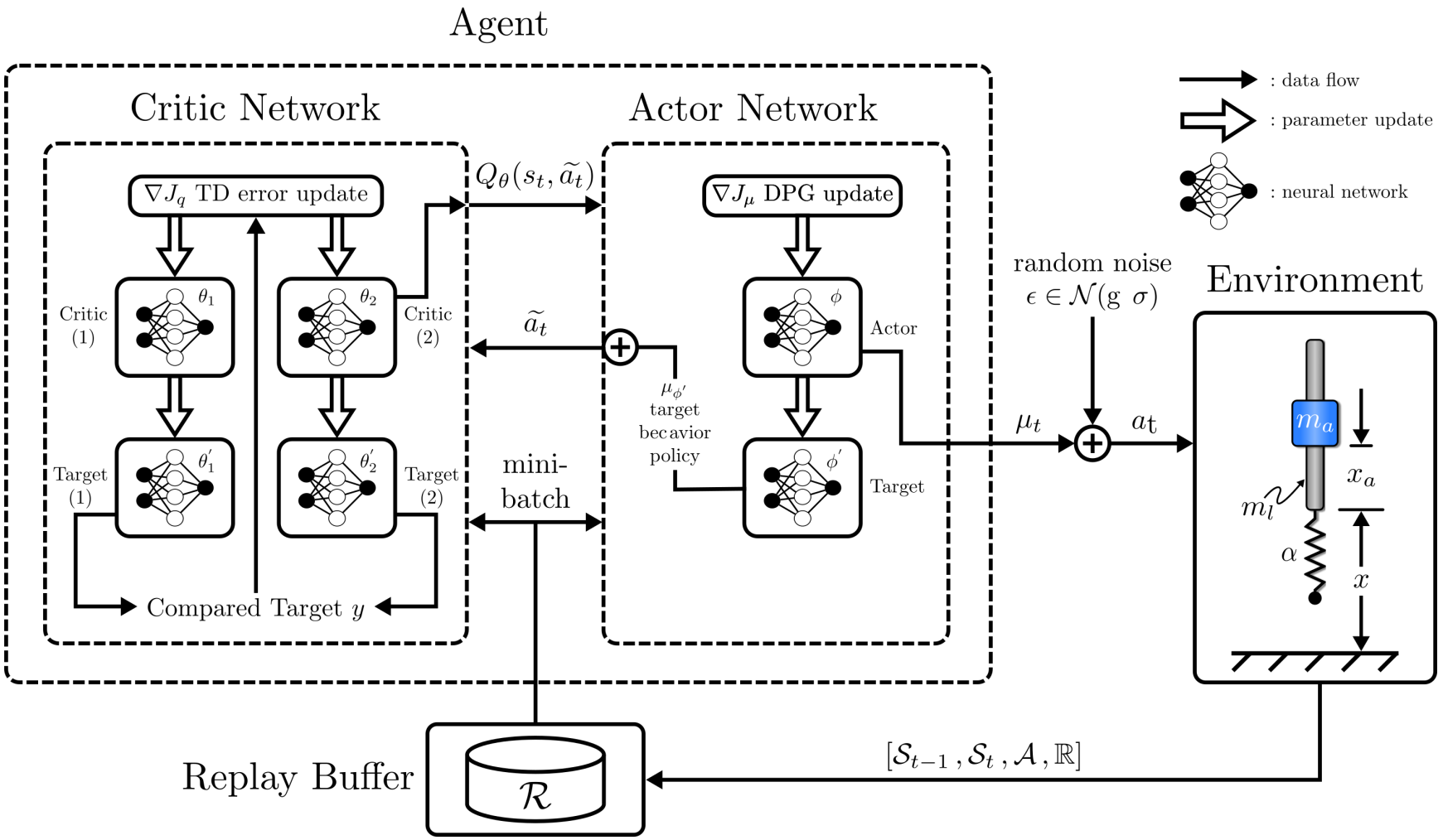


Figure 5. Twin Delayed Deep Deterministic Policy Gradient Block Diagram with Monopode as Environment

Gradient (DDPG) algorithm [14].

Figure 5 displays the flow in information for this algorithm. In general, this algorithm learns both a Q-function and a policy, being the *critic* and the *actor*. The Q-function is updated in an off-policy fashion using data stored in a replay buffer. A technique finding the temporal difference error between the target Q-function and the main Q-function is used to minimize a loss function. For algorithms such as TD3, the ultimate goal is to find a policy, π_θ, which maximizes the expected return:

$$\nabla_\phi J(\phi) = \mathbb{E}_{s\sim p_\pi}\left[\nabla_a Q^\pi(s,a)|_{a=\pi(s)} \nabla_\phi \pi_\phi(s)\right] \tag{1}$$

where $Q^\pi(s,a) = \mathbb{E}_{s_i\sim p_\pi, a_i\sim\pi}[R_t|s,a]$ is the value function and the critic in the case of the TD3 architecture. Updating the Q-function, again, is accomplished using temporal difference between the Q-function and a target Q-function [24, 25], both which are based on the Bellman Equation:

$$Q^\pi(s,a) = \underset{s'\sim P}{E}\left[r(s,a) + \gamma \max_{a'} Q^\pi(s',a')\right] \tag{2}$$

Using a differentiable function approximator, $Q^\pi(s,a)$ can be represented and estimated using by $Q_\phi(s,a)$, with parameters ϕ [26]. To maintain a fixed objective over multiple policy updates, the target Q-function approximator is instantiated separately as $Q_{\phi_{targ}}(s,a)$. The target does depend on the same parameters that are being trained, ϕ, where there exists an issue when trying to use it as a target. To solve this issue the target network is updated at a delayed pace following the main Q-function approximator by either matching the parameters or by polyak averaging, $\phi_{targ} \leftarrow \tau\phi + (1-\tau)\phi_{targ}$, where τ is a tunable hyperparameter.

Summing it up then, the critic side of the TD3 algorithm is responsible for minimizing the difference between the value of the current state/action pair using the main Q-function, and the reward of the current state/action pair plus the discounted value of the next state/action pair using the target Q-function. The loss function takes

the form:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_\phi(s,a) - \left(r(s,a) + \gamma\,(1-d)\,Q_{\phi_{targ}}(s', \pi_{\theta_{targ}}(s'))\right)\right)^2\right] \tag{3}$$

where $\pi_{\theta_{targ}}(s')$ is a target policy that, in a similar manner to the target Q-function, follows the main policy, π_θ, at a delayed pace either by directly copying the values or by polyak averaging. Additionally, d represents a boolean value which depends on the terminal status of the next state, s'.

As for updating the policy for the actor critic-type algorithm, this aspect is rather simple. Because DDPG, and therefore TD3, are built to accommodate only continuous action spaces, the Q-function is assumed to be differentiable with respect to action. Therefore to find optimal policy parameters, θ, for the policy, π_θ, the solution of the following equation must be found:

$$\max_\theta \underset{s\sim D}{E}\left[Q_\phi(s, \pi_\theta(s))\right] \tag{4}$$

The reason that TD3 is considered the predecessor to DDPG, is that there are some additional tricks deployed in addition to the description thus far. The first being the addition noise to the target policy. It can be seen in Equation 3 that the target policy, $\pi_{\theta_{targ}}$, is required to generate an action to evaluate the the target Q-function. Noise is added to the policy taking the form:

$$a'(s') = \text{clip}\left(\pi_{\theta_{targ}}(s') + \text{clip}(\epsilon, -c, c), a_{low}, a_{high}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{5}$$

where ϵ represents the noise sampled in some form that the user can specify. This method of adding noise was shown by the authors of the algorithm to reduce a known issue where the Q-function approximator can develop large peaks for certain actions.

The second trick the TD3 algorithm deploys is the addition of a second Q-function approximator and target Q-function approximator. A known potential issue of the Q-function is that is can suffer from overestimation of the value of action/state

pairs. This of course leads to the policy learning actions that the Q-function assumes are better than they actually are. To alleviate this issue, the authors suggest instantiating two Q-functions and two target Q-functions. Calculate the two target Q-functions:

$$y(r, s', d) = r(s, a) + \gamma\,(1 - d)\,Q_{\phi_{1,targ}}(s', \pi_{\theta_{targ}}(s')) \tag{6}$$

and take the lower of the two targets to update both the main Q-functions:

$$L(\phi_1, \mathcal{D}) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_{\phi_1}(s, a) - y(r, s', d)\right)^2\right] \tag{7}$$

$$L(\phi_2, \mathcal{D}) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_{\phi_2}(s, a) - y(r, s', d)\right)^2\right] \tag{8}$$

.

The last trick that the TD3 algorithm deploys is the addition of a delay between the update of the Q-functions and the policy. They found that in doing this, the Q-function was able to converge to a better solution before updating the policy. Ultimately, the addition of a policy update delay was done to reduce coupling between the Q-function and the policy. The recommended delay the authors suggest is updating the policy every two Q-function updates.

There are many implementations of the TD3 algorithm that are available, however the StableBaselines3 implementation is used to complete the work in this thesis [27]. StableBaselines3 is a widely popular library of RL algorithm implementations, and is composed of well written and understandable documentation for the supported implementations. The differentiable function approximators used to estimate the policies and Q-functions are built within StableBaselines3 using PyTorch [28], which is also a widely popular framework for machine learning and more specifically reinforcement learning.

1.6 Contributions

The purpose of the work presented in the remainder of the document is to evaluate and present the performance of a concurrent design architecture that utilizes

RL techniques, for flexible jumping systems. In doing this, the method will be divided into multiple sections wherein additional findings will be presented.

In the next Chapter, a RL based controller will be trained on a flexible jumping system to evaluate the effectiveness of training for efficient control. Power use is often considered when designing RL controllers for rigid systems, typically taking the form of a weighted negative reward when deploying an RL algorithm. It is of interest to evaluate if defining strategies, with efficiency being the primary objective, for flexible systems, if the resulting control strategy takes advantage of system flexibility. In Chapter 3, more traditional methods are used to evaluate the changes in control strategies when training a controller to consider efficiency.

In Chapter 4, a RL problem is defined in a unique way such that the environment the policy is deployed in is a simulation of an environment where the actions sent by the policy are mechanical design updates. It is of interest to evaluate if an RL based approach can be taken to learn mechanical design parameters of a flexible jumping system, such as flexibility. Futhermore, using a single fixed control input, it is of interest to determine if this technique can be used to define designs to accomplish multiple tasks. Finally, in Chapter 5, the methods of learning a control strategy and learning a design will be combined to create a concurrent design technique. The resulting designs and controller performance will be presented and the method will be evaluated.

II Learning Efficient Jumping Strategies for the Monopode System

Utilizing reinforcement learning to train a neural network based controller has been shown to be useful for controlling many robotic systems [21, 22]. Successful control of rigid legged robots both in simulation and on physical hardware has been shown to be highly effective [20, 29]. Reinforcement learning has been shown to be capable of defining more effective and efficient jumping techniques for a single-legged robot with SEAs [30]. It has also been shown to be an effective method for controlling multi-legged robots both in simulation and on physical hardware [31–33]. Furthermore it has been shown to be useful for defining energy efficient strategies for multi-legged robots that have been deployed on physical hardware [34]. However, the use of RL to train controllers for flexible systems is limited, particularly in regards to legged locomotive systems. In this Chapter, RL is deployed to define an energy efficient jumping strategy for the monopode jumping system described in the next Section. The purpose of this work is to validate the use of RL for defining the control aspect of a concurrent design architecture for flexible jumping systems.

Table 1. Monopode Model Parameters

Model Parameter	**Value**
Mass of Leg, m_l	0.175 kg
Mass of Actuator, m_a	1.003 kg
Spring Constant, $\alpha_{nominal}$	5760 N/m
Natural Frequency, ω_n	$\sqrt{\frac{\alpha}{m_l+m_a}}$
Damping Ratio, $\zeta_{nominal}$	1e-2 $\frac{\text{N}}{\text{m/s}}$
Gravity, g	9.81 m/s^2
Actuator Stroke, $(x_a)_{\max}$	0.008 m
Max. Actuator Velocity, $(\dot{x}_a)_{\max}$	1.0 m/s
Max. Actuator Acceleration, $(\ddot{x}_a)_{\max}$	10.0 m/s^2

2.1 Monopode Jumping System

The intent of the work completed in this thesis is to validate the use of RL to concurrently design a system/controller architecture for flexible-legged locomotive systems. To evaluate the methods used, a monopode system like the one shown in Figure 6 was used to represent a flexible jumping system. This system has been studied and has been proven to be an effective base for modeling the jumping gaits for many different animals [35].

The monopode is controlled by accelerating the actuator mass, m_a, along the rod mass, m_l, causing a hopping like motion. A nonlinear spring is modeled and represented by the variable α in the figure. Also included in the model is a damper parallel with the spring, having a damping coefficient of c, though it is not shown in the figure. Variables x and x_a represent the rod's global position and the actuator's local position with respect to the rod, respectively. The equations of motion for the system are:

$$\ddot{x} = \frac{\gamma}{m_t}\left(\alpha\, x + \beta\, x^3 + c\,\dot{x}\right) - \frac{m_a}{m_t}\,\ddot{x}_a - g \tag{9}$$

where x and $\dot{x}$ are position and velocity of the rod, respectively, the acceleration of the

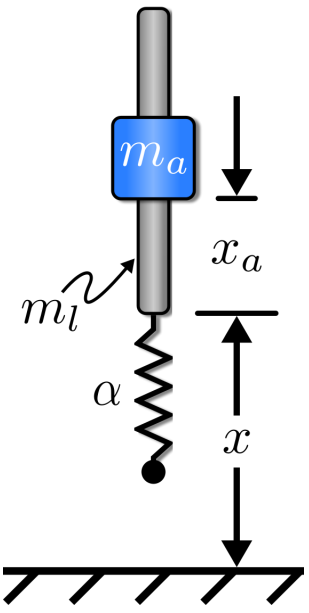


Figure 6. Monopode Jumping System

actuator, $\ddot{x}_a$, is the control input, and m_t is the mass of the complete system. Constants α and c represent the linear spring and damping coefficient, respectively, and constant β is set to $1e8$. Ground contact determines the value of γ, so that the spring and damper do not supply force while the leg is airborne:

$$\gamma = \begin{cases} -1, & x \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Additionally, the spring compression limit, or the systems position in the negative x direction, is limited to 0.008m. Additionally, the system is confined to move only vertically in regards to Figure 6 so that controlling ballance is not required.

2.2 Training Environment

In this Chapter, RL is used to find efficient control strategies for a flexible-legged robotic system. The monopode described in Ch. 1 was used as a test system to validate the effectiveness of the RL approach. An traditional RL environment aligning with the standards set by OpenAI for a Gym environment was created. [36]. The observation and action spaces were defined, respectively, as follows:

$$\mathcal{S} = [x_{a_t}, \dot{x}_{a_t}, x_t, \dot{x}_t] \tag{11}$$

$$\mathcal{A} = [\ddot{x}_{a_t}] \tag{12}$$

where x_t, $\dot{x}_t$ were the monopode's position and velocity at time t, and x_{a_t}, $\dot{x}_{a_t}$ and $\ddot{x}_{a_t}$ were the actuator's position, velocity and acceleration, respectively.

Two separate stopping conditions were defined for the environment to evaluate two different jump types and therefore two different input commands. The first was defined as the monopode's position being greater than zero than returning to zero once. The second was defined like the first but with the monopode's position being greater than zero and then less than zero twice. Two different jumps are create from these stopping conditions and they are highlighted in Section 2.4 below.

2.3 Efficient Control Strategies

Efficient control of a robotic system is often one of the most important aspects of a controller's design. Applications where a robotic system is deployed and relies on a limited power source, such as a mobile walking robot, will often require an efficient control strategy. Modern, traditional methods, such as model predictive control, have been shown to produce energy efficient locomotion strategies for wheeled and legged systems [37, 38]. It is of interest in this work to utilize RL, a modern neural network based control method, to find strategies which are designed with power efficiency as the primary objective.

Two different reward functions were designed to accomplish the task of determining how well RL learns efficient jumping strategies. The purpose of defining two different reward functions was to compare the input commands and resulting jumping shapes of the two controller types to determine if the efficient controller was learning to conserve power.

The first reward function was one that ignored power usage all together and focused solely on the height of the jump:

$$R = x_t \tag{13}$$

where x_t was the height of the monopode system at any given time step. The second reward function was one that was defined to accomplish the same task, but also consider power consumption and was defined as:

$$R = \frac{x_t}{\sum_{t=0}^{t} P_t} \tag{14}$$

where P_t was the power consumption of the monopode system at any given time step defined mechanically as the product of the actuator's acceleration, velocity and mass:

$$P_t = m_a \, \dot{x_a} \, \ddot{x_a} \tag{15}$$

where m_a was the mass of the actuator, and $\dot{x_a}$ and $\ddot{x_a}$ where the actuators velocity and acceleration, respectively.

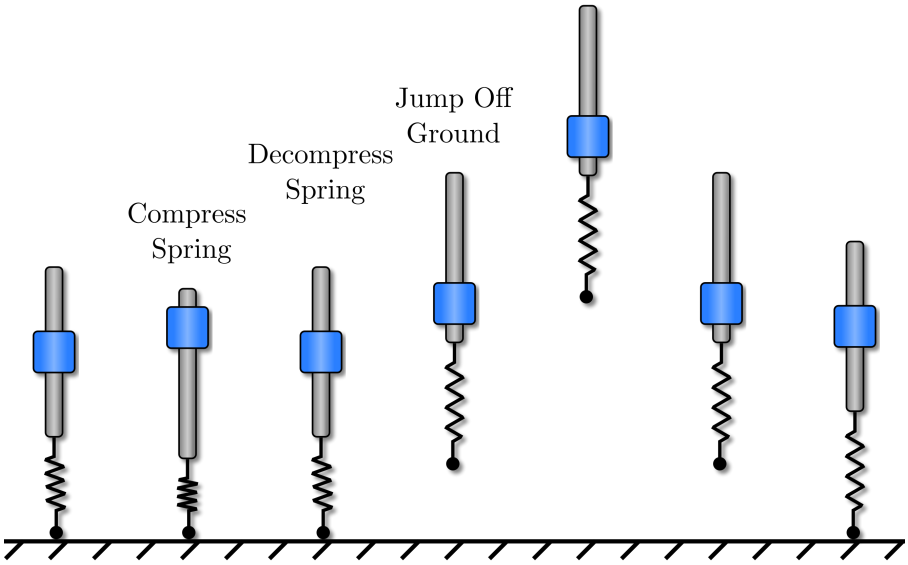


Figure 7. Example Single Jump

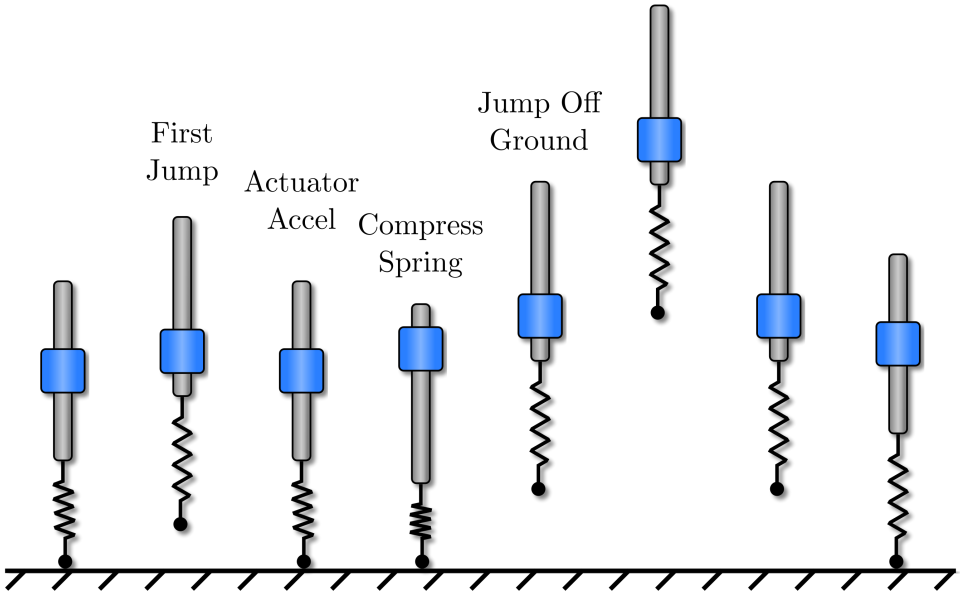


Figure 8. Example Stutter Jump

2.4 Input Complexity

Two different jumping types were also analyzed. The first was referred to as a single jump command, and the second a stutter jump command. The intent of utilizing two different jumping commands was to determine if a RL algorithm was more or less effective in learning differing strategies depending on the complexity of the desired command.

An example single jump can be seen in Figure 7. The intended command from the learned controller would be one that would jump the monopode once. This type of command would ideally compress the spring/damper by accelerating the actuator in the positive direction. This would allow the system to store some energy in the spring that could be used to jump the system. The actuator mass should then accelerate downward allowing the spring to decompress until it reaches its nominal length. At this point the system should be accelerating upwards and the actuator downwards such that the monopode leaves the ground completing a single jump.

An example stutter jump can be seen in Figure 8. The intended command from the learned controller would be one that would jump the monopode twice. This type of command would firstly complete an optimal single jump. Following that motion, the actuator should assume an acceleration direction to recompress the spring storing more energy with a farther compression. When the spring is compressed to its maximum value or the system's total acceleration reaches zero, the actuator mass should accelerate downwards allowing the spring to decompress until it again reaches its nominal length. At this point the system should be accelerating upwards and the actuator downwards, similar to the single jump, such that the monopode leaves the ground completing a stutter jump.

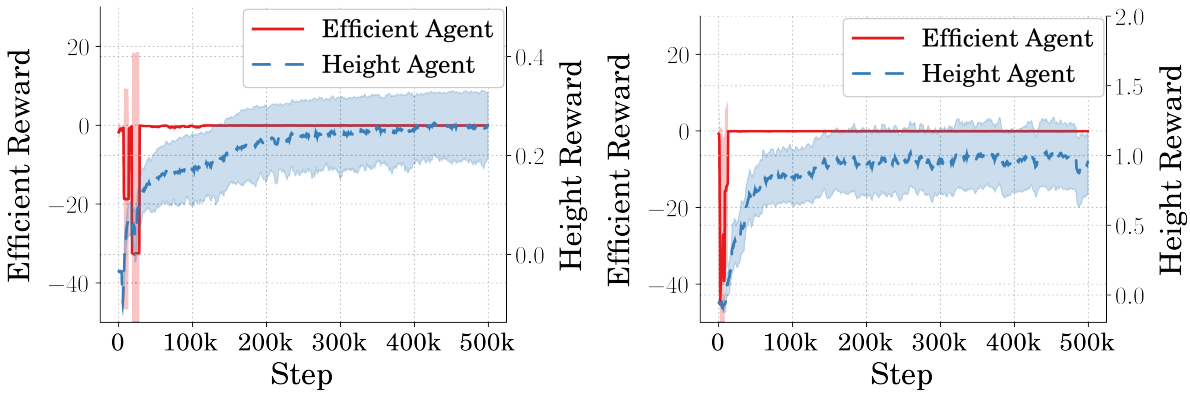
2.5 Deploying TD3

Because training an RL controller does not guarantee that the controller will learn an optimal strategy without finding local optima, training more than one controller is often practiced to evaluate performance. In this work, fifty different controllers where trained, each with a different random network initialization. Each controller was trained for a total 500k time steps. The remaining hyperameters set using the TD3 algorithm are defined in Table 2.

The rewards the TD3 algorithm received during training are presented in

Table 2. TD3 Training Hyperparameters

Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	1000 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip, c	0.5
Seed	50 Random Seeds



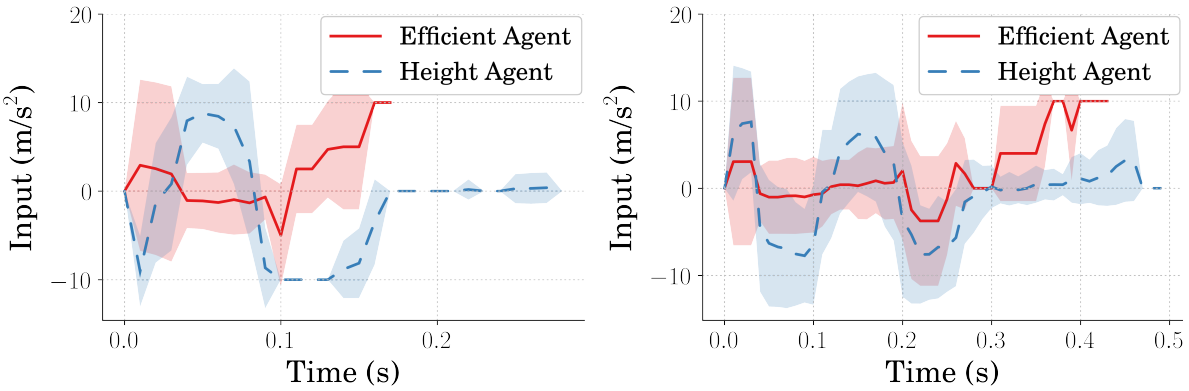
(a) Single Jumping Reward **(b)** Stutter Jumping Reward

Figure 9. Reward vs Time Step During Training

Figure 9. They represent the controllers being trained to accomplish their respective goals. Looking at Figure 9a, it is clear that the reward for the high jumping strategy is converging after 500k steps of training. The reward for the efficient strategy is less clear as it seems to have learned a solution quickly and not learned beyond this. The single jumping command, being a more simply command to learn can explain partially why the controller learns quickly. Additionally, the scale of the reward should be considered as the values recieved at the start of training are a result of wasting power therefore resulting in extremely low rewards. Looking at Figure 9b, it is also apparent that the height controller is learning a converging solution. The efficient controller can be seen to have suffered from the same errors as the single jumping command type.

2.6 Average Performance of Network Controller

2.6.1 Input Commands. Average learned input commands for both the single and stutter jumping cases are shown in Figure 10 to compare controllers that were trained to jump efficiently to those trained to jump high. At first glance, there are obvious differences regarding timing, magnitude and direction. There are also slight differences in variance seen between the two controller types. Starting with Figure 10a, which displays the input commands for the single jumping case, it is most obvious that

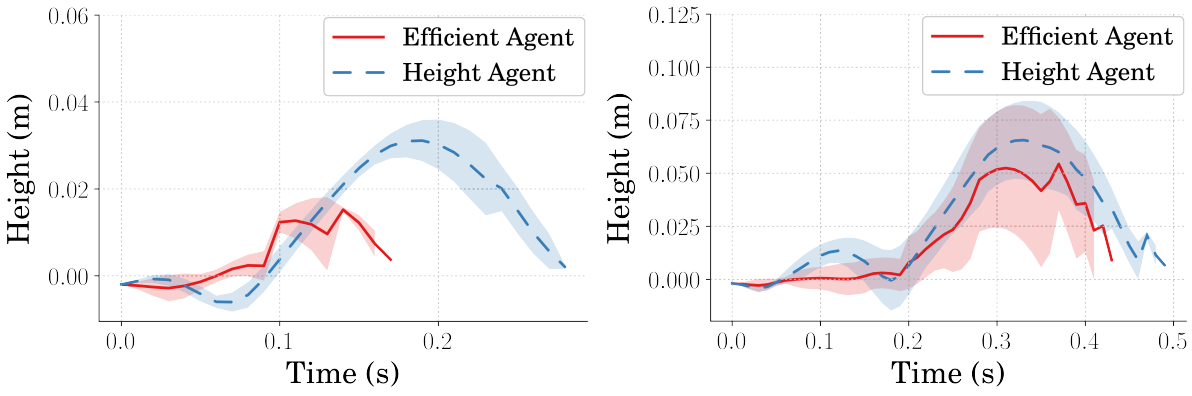


(a) Single Jumping Input **(b)** Stutter Jumping Input

Figure 10. Average and Standard Deviation Inputs to monopode

the direction for the initial acceleration of the actuator mass for the efficient controllers and height controllers differ. In the case where the controller is learning to jump high, an initial acceleration in the negative direction is learned, which contrasts the case where the controller is learning an efficient command. Further, the magnitude of the commands is drastically different which may be an indicator for conserving power. Looking now at Figure 10b, it is immediately apparent that the magnitudes of the commands differ greatly. They are however, more similar in regards to their timings and directions when comparing the stutter jumping command to the single jumping command. In both the single and stutter jumping cases, it can be seen that there is upward acceleration command towards the end of the jump, which again might be an indicator of a more efficient jumping strategy. Furthermore, it can be observed that the single jumping case, there exists more variance across different instances of the trained efficient controllers in comparison to the height controllers. This does not seem to be the case for the stutter jumping command type, though both cases do seem to generate controllers with high variance inputs across instances.

2.6.2 Jumping Height Performance. Average jumping performance resulting from the learned single and stutter jumping commands is displayed in



(a) Single Jumping Height **(b)** Stutter Jumping Height

Figure 11. Average and Standard Deviation Heights of monopode

Figure 11. In both the single and stutter jumping cases, there are differences seen in jumping ability when comparing the efficient and height controller types. At first glance it is apparent that when increasing the complexity of the command from a single jump to a stutter jump, the efficient controllers are better able to replicate the performance of the height controllers.

Starting with Figure 11a, it is most apparent that the height controllers learned a command input that was outperformed the efficient controllers in terms of jump height. The resulting motion from the input discussed in the previous section can also be seen in that the efficient controller learned to simply compress the spring, then jump the monopode. The height controllers, in contrast, disregarding power consumption, learned to decompress the spring from its nominal position, keeping it below the point of leaving the ground, then recompressing for a much higher jump. Summing up the single jumping commands, the efficient strategy trained controllers that jumped the monopode 104.53% lower than the high jumping strategy.

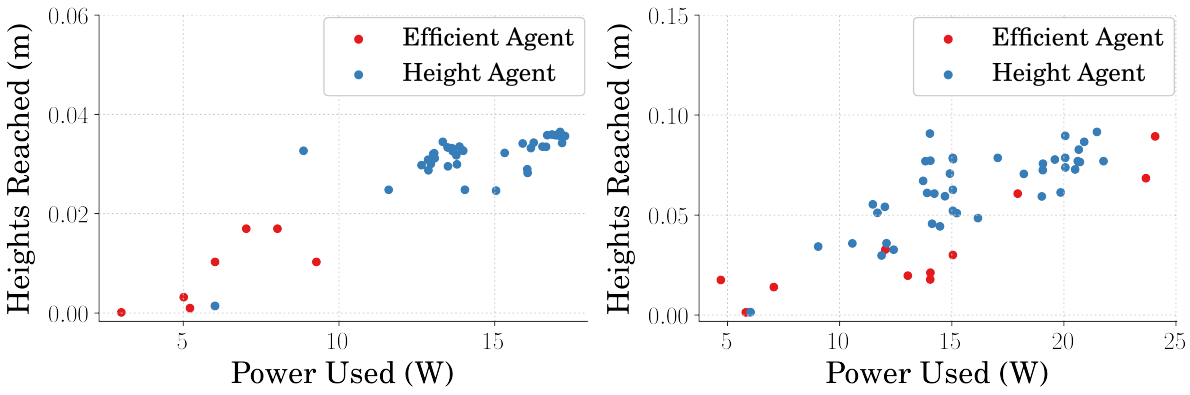
Figure 11b compares the jumping performance of the efficient and height strategies for the stutter jumping command. At first glance, they differ, but less drastically than the single jumping case. The likeness of the input command shapes propagate and the resulting jumping shapes share similar in form. The large differences seen in the stutter jumping shapes are similar those of the inputs that create them, in that they differ mostly regarding the magnitudes. Summing up the stutter jumping command, the efficient strategy resulted in a jumping command that jumped 20.69% lower than the high jumping strategy.

In both the single and stutter jumping cases, the upward acceleration from efficient controllers toward the end of the command can be seen in that the position of the monopode creates a psuedo plato. Additionally, regarding variance, the single jumping controllers seems to produce jumping shapes with similar levels of variance. As for the stutter jumping case, though similar in high levels of input variance, the

jumping height variance for the efficient controllers is noticeably higher that that of the height controller.

2.6.3 Height Reached vs. Power Used. Height reached versus power consumed data, for both the single jumping and stutter jumping cases is shown in Figure 12. It is firstly apparent that in both the single and stutter jumping cases, the efficient controllers utilize less power and therefore suffer regarding jump height. This matches what was seen in the previous sections regarding input commands and jumping shapes. In the single jumping case, shown in Figure 12a, there is an apparent separation between the two controller types where the height controllers are mostly clustered in the upper-right part of the dataset. On average, for the single jumping command type, the efficient jumping strategy learned jumping commands that were 126.27% more efficient. As for the stutter jumping case, show in Figure 12b, the difference is less obvious though still present. The variance of the two controller types, being quite high, matches what is seen in the previous sections and results in more mixing of the data. On average, for the stutter jumping command type, the efficient strategy learned commands that were 101.45% more efficient.

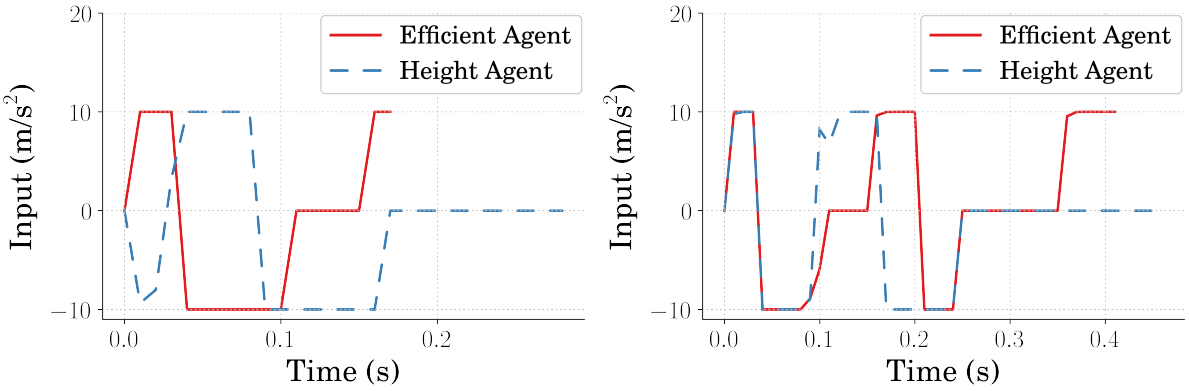
2.7 Optimal Performance of Network Controller



(a) Single Jumping Efficiency **(b)** Stutter Jumping Efficiency

Figure 12. Height Reached vs Power Consumed of monopode

2.7.1 Input Commands. Taking the best of the fifty different controllers trained for both the single and stutter jumping cases and comparing the efficient and height controller's performance can show what is possible with a properly defined RL problem. Figure 13, shows the differences in the input commands generated when selecting the highest performing controller in terms of reward received. At first glance, it can be seen that there are less differences between the the efficient and height controllers in comparison to the average results from Section 2.6. A major similarity being that magnitudes are similar across all cases such that the controllers utilize the actuator's maximum acceleration. Looking at Figure 13a, which compares the efficient and height controllers for the single jumping input, the major difference are of course the timing and direction of the command. This is similar to the average performance evaluation, from Section 2.6.1, in that the efficient controller does not take advantage of the slight decompression of the spring before the monopode leaves the ground. Because of this, the efficient controller learns a different timing for a single jump. As for the stutter jumping case, shown in Figure 13b, the differences between the efficient and height controller is less drastic. The initial timing is largely the same as both controller types learn to utilize the decompression of the spring. The differences begin when decompressing the spring a second time and completing the first jump. The efficient

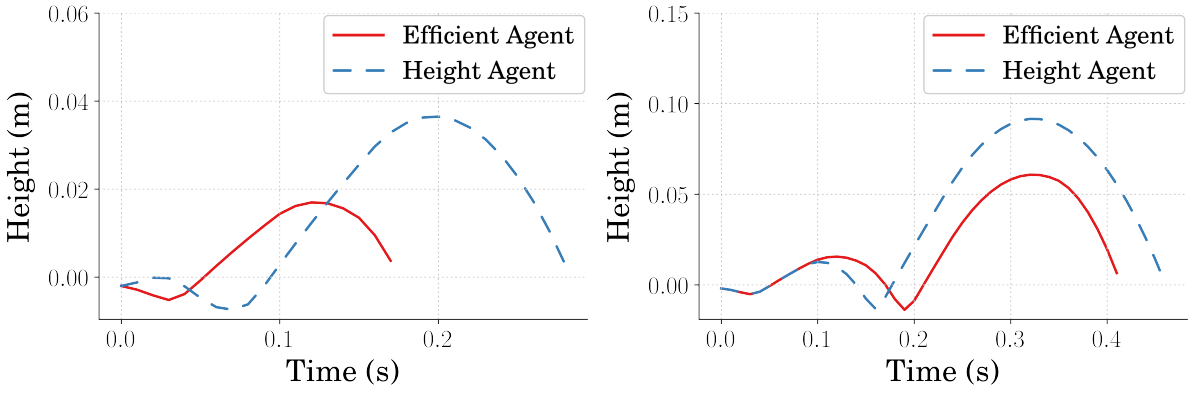


(a) Single Jumping Input **(b)** Stutter Jumping Input

Figure 13. Optimal Inputs to monopode

controller learns a command similar in form to a bang-coast-bang command, where in contrast the height controller learns a command similar to that of a bang-bang shaped input.

2.7.2 Jumping Height Performance. In line with the previous section, it is of interest to evaluate the jumping performance curves of the best controllers trained. Figure 14 displays the jumping performance for both jump types as well as both controller types when utilizing the inputs shown in Section 2.7.1. These curves validate what was shown in the previous section and display that the efficient controllers do not generate commands that jump the monopode as high. Figure 14a, which compares the efficient and height controllers for the single jump, verifies that the efficient controller does not learn to utilize the slack in the spring. In Figure 14b, it can be seen that when utilizing a command more similar in form to a bang-coast-bang command, like the efficient controller learned, the timing of the jump sequence is shifted and the resulting final height is less than the height controller who's command is more similar to a bang-bang shaped command.



(a) Single Jumping Height **(b)** Stutter Jumping Height

Figure 14. Optimal Heights of monopode

2.8 Conclusion

Two different controller types where trained to generate two different jumping commands for the simplified monopode jumping system described in Section 2.1. The first type of controller was one that would command the monopode system to jump high where the reward was based on nothing other than system height. The second type of controller was one which controlled the monopode to jump high but at the cost of power consumed, such that high jumps that consumed high amount of power where less desirable than high jumps that consumed less power. It has been shown that the rewards past to RL algorithms that are training controllers can be manipulated so that the the learned input commands take advantage of the spring/damper that exists within the monopode jumping system. Furthermore, the timing of the commands, the input magnitude and direction are all affected when defining a reward strategy that seeks to increase power efficiency. When considering the average performance of the different control strategies, for both the single and stutter jumping cases, the heights reached were less for the efficient strategies. However, they were significantly more efficient, particularly when scaling the complexity of the command from the single jump to the stutter jump. It should be the concluded then, that RL might serve as a useful method for defining control strategies for flexible-legged jumping systems, particularly when energy efficiency is of interest. Additionally, when considering more complex control strategies, which might be difficult to define efficiently, RL might serve as a useful method for effective efficient strategies.

III Using Input Shaping to Validate RL Controller

Discussion on the use of input shaping to find optimal jumping strategies for jumping systems. How can one use these techniques to validate RL controllers? Do RL algorithms generate robust controllers when trained robustly. ...

Include some discussion of: 1. Most RL having black boxes 2. Need for interpratable RL 3. Input shaping as a way to interpret RL

3.1 Input Shaping Controller Input

Information from the paper DV wrote and some other resources found from the sources in that paper. Will need some resources from DV to fill in some input shaping information.

3.2 RL Controller Input

Discussion and figures from the inputs defined by the RL algorithms for the monopode system.

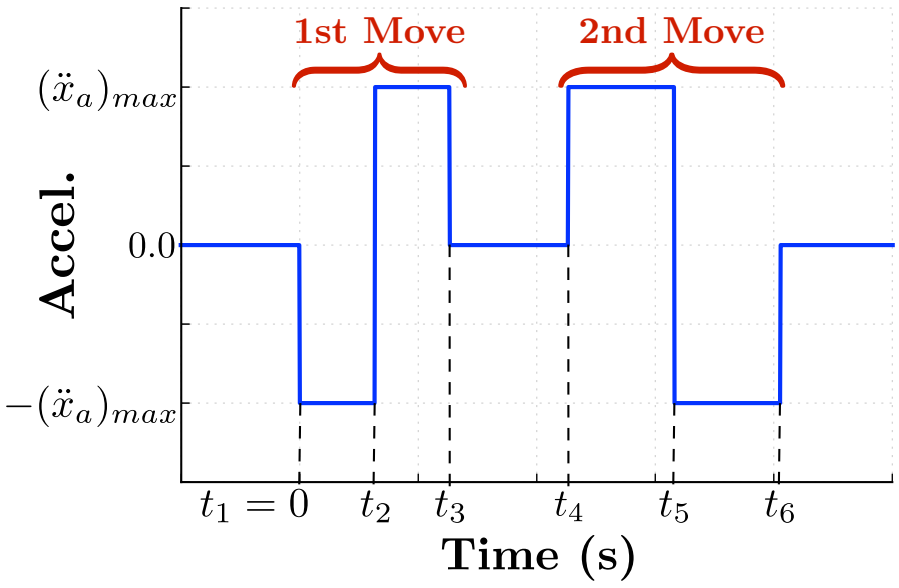


Figure 15. Jumping Command

3.3 Training a Robust Controller

How are robust controllers trained using RL algorithms? How did we do ours? Show the results.

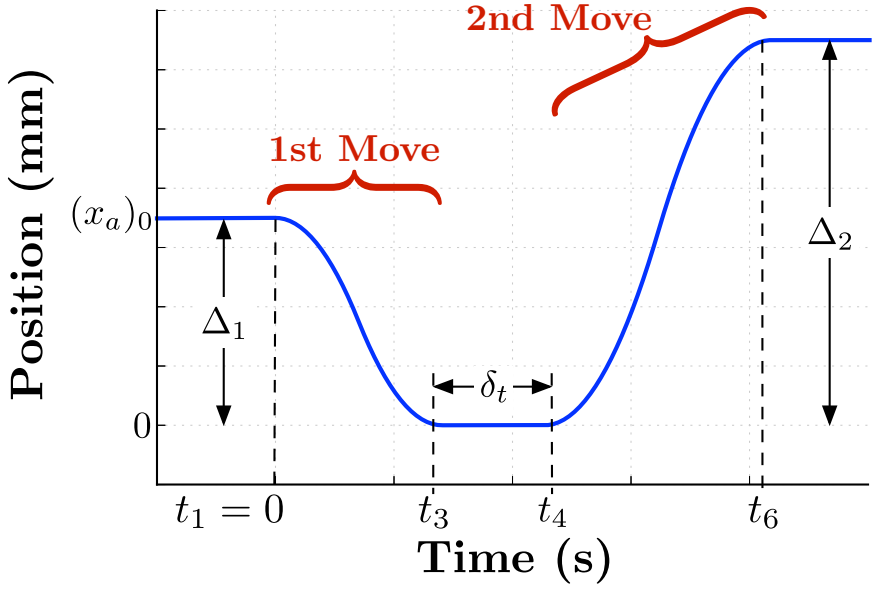


Figure 16. Resulting Actuator Motion

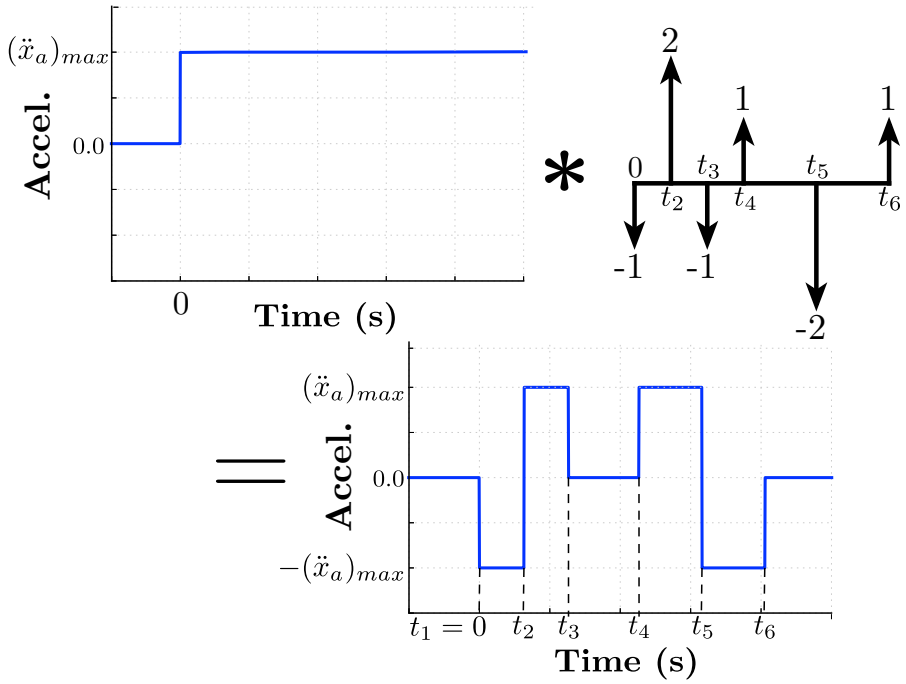


Figure 17. Decomposition of the Jump Command into a Step Convolved with an Impulse Sequence

3.4 Conclusion

Discuss the results.

IV Mechanical Design of a the Monopode Jumping System

Often it is the goal of a controls engineer to design a controller to accommodate and manipulate systems according to the system description provided. However, research has been conducted showing the value of studying the manipulation of mechanical design parameters in order to achieve a desired system behavior [16]. In this chapter, reinforcement learning is shown to be useful as a tool to learn mechanical designs given a predefined system controller for the monopode jumping system. RL has been shown to be an effective strategy for finding optimal concurrent designs for many different types of systems [18, 39, 40]. It has even shown it's ability to define designs that are successfully deployed on physical hardware [17]. It is comparable to work where evolutionary type algorithms are deployed to optimize physical parameters of systems for improved energy efficiency [41, 42]. Here, RL is used to define an optimal design for the monopode jumping system described in Chapter 2 using a predefined control input.

4.1 Controller Input

Bang-bang based jumping commands like the one discussed in Figure 15 of the Chapter 3 are likely to result in a maximized jump height [43]. For these command types, regarding the monopode jumping system, the actuator mass travels at maximum acceleration within its allowable range, pauses, then accelerates in the opposite direction. Commands designed to complete this motion are bang-bang in each direction, with a selectable delay between them. The resulting motion of the actuator along was rod is shown in Figure 16. Starting from an initial position, x_{a_0}, the actuator moves through a motion of stroke length Δ_1, pauses there for δ_t, then moves a distance Δ_2 during the second portion of the acceleration input.

This bang-bang-based profile can be represented as a step command convolved with a series of impulses, as was shown in Figure 17 [44]. Using this decomposition, input-shaping principles and tools can be used to design the impulse sequence [45, 46].

For the bang-bang-based jumping command, the amplitudes of the resulting impulse sequence are fixed, $A_i = [-1, 2, -1, 1, -2, 1]$. The impulse times, t_i, can be varied and optimal selection of them can lead to a maximized jump height of the monopode system [43]. Commands of this form will often result in a stutter jump like what was shown in Figure 8 of Chapter 2, where the small initial jump allows the system to compress the spring to store energy to be used in the final jump. This jumping command type was used as the input for the monopode during the simulation phase of training.

...

I think a bit of this belongs in the previous chapter?

4.2 Environment Definition

To allow the agent to find a mechanical design, a reinforcement learning environment conforming to the OpenAI Gym standard [36] was created for the monopode model described in Chapter 2, including a fixed controller input based on the algorithm described in Section 4.1. Unlike the common use case for RL, which is tasking the agent with finding a control input to match a design, the agent in this work was tasked with finding mechanical parameters to match a control input. The mechanical parameters the agent was tasked with optimizing were the spring constant and the damping ratio of the monopode system. At each episode during training, the agent selected a set of design parameters from a distribution of available designs. The actions applied, $\mathcal{A}$, and transitions saved, $\mathcal{S}$, from the environment were defined as follows:

$$\mathcal{A} = \{\{a_\alpha \in \mathbb{R} : [-0.9\alpha, 0.9\alpha]\}, \{a_\zeta \in \mathbb{R} : [-0.9\zeta, 0.9\zeta]\}\} \tag{16}$$

$$\mathcal{S} = \left\{ \sum_{t=0}^{t_f} x_t, \sum_{t=0}^{t_f} \dot{x}_t, \sum_{t=0}^{t_f} x_{at}, \sum_{t=0}^{t_f} \dot{x}_{at} \right\} \tag{17}$$

where α and ζ are the nominal spring constant and damping ratio of the monopode, respectively; x_t and $\dot{x}_t$ are the monopode's rod height and velocity steps, and x_{a_t} and

$\dot{x}_{a_t}$ are the monopode's actuator position and velocity steps, all captured during simulation.

4.3 Rewards for Learning Designs

The RL algorithm was utilized to find designs for two different reward cases. Time series data was captured during the simulation phase of training and was used to evaluate the designs performance through these rewards. The first reward case used was:

$$R_1 = \left(\sum_{t=0}^{t_f} x_t \right)_{max} \tag{18}$$

where x_t was the monopode's rod height at each step during simulation. The goal of the first reward was to find a design that would cause the monopode to jump as high as possible.

The reward for the second case was:

$$R_2 = \frac{1}{\frac{|\mathbb{R}_1 - x_s|}{x_s} + 1} \tag{19}$$

where x_s was the desired jump height, which was set to 0.01 m. The second case was utilized to test RL's ability to find a design that minimized the error between the maximum height reached and the desired maximum height to reach.

4.4 Ability to Learn a Design

Figures 18 and 19 represent the heights the monopode could reach for two different design spaces. The design space provided for the first case, shown in Figure 18, represents a space where the allowable damping ratio was limited to a fairly narrow range. This limits the solution space, making it less likely that the agent will settle to a locally optimal value. The design space provided for the second case, shown in Figure 19, represents a space where a wider range of damping ratios are allowed. This wider range of possible values makes it more likely that the agent will settle to a local maxima.

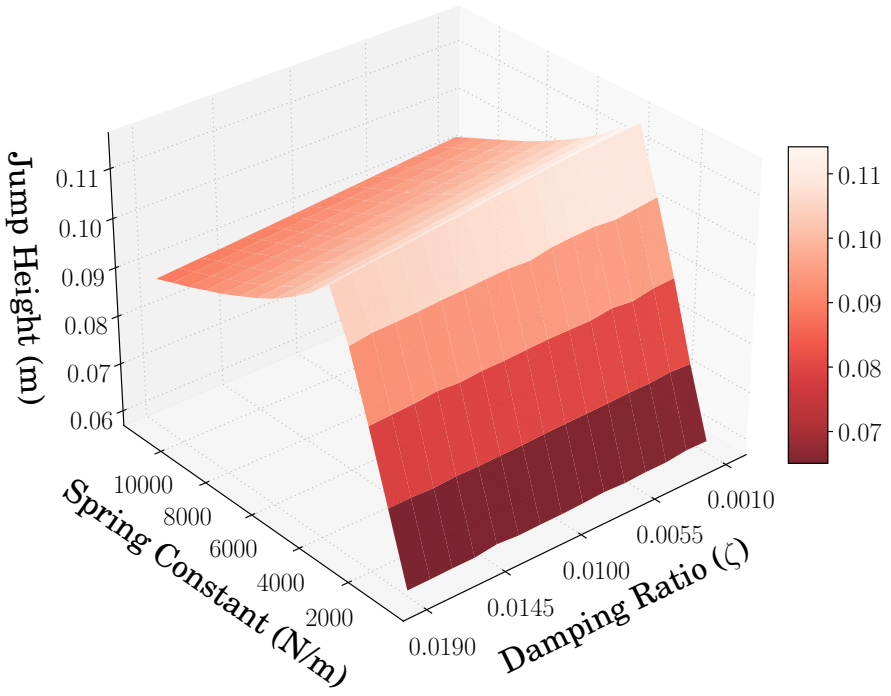


Figure 18. Jumping Performance of Narrow Design Space

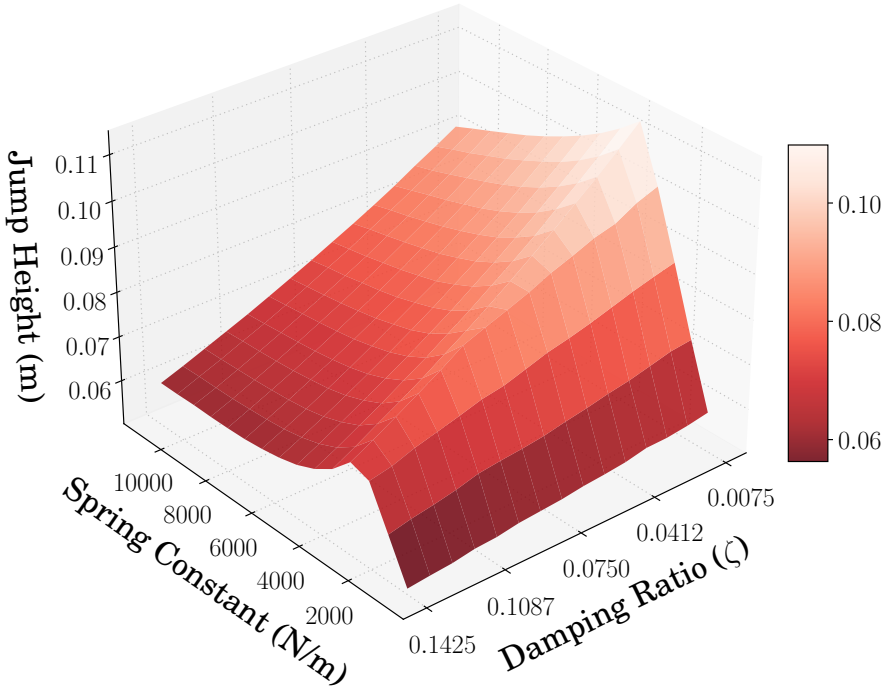


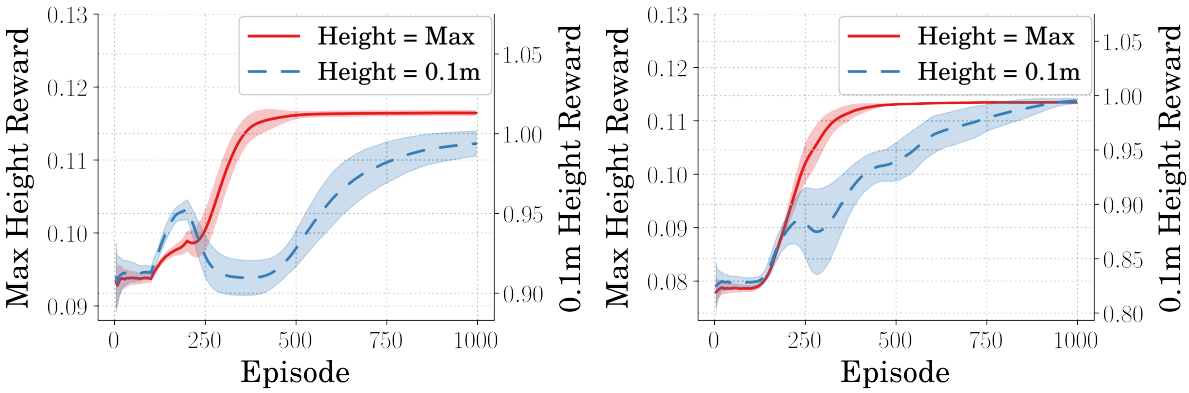
Figure 19. Jumping Performance of Broad Design Space

4.5 Deploying TD3

The training hyperparameters were selected based on TD3's author recommendations and Stable Baselines3 [27] experimental findings and are displayed in Table 3. All of the hyperparameters, with the exception of the rollout (Learning Starts) and the replay buffer, were set according to Stable Baselines3 standards. The rollout setting was defined such that the agent could search the design space at random, filling the replay buffer with enough experience to prevent the agent from converging to a design space that was not optimal. The replay buffer was sized proportional to the

Table 3. TD3 Training Hyperparameters

Hyperameter	Value
Learning Rate, α	0.001
Learning Starts	100 Steps
Batch Size	100 Transitions
Tau, τ	0.005
Gamma, γ	0.99
Training Frequency	1:Episode
Gradient Steps	$\propto$ Training Frequency
Action Noise, ϵ	None
Policy Delay	1 : 2 Q-Function Updates
Target Policy Noise, ϵ	0.2
Target Policy Clip, c	0.5
Seed	100 Random Seeds



(a) Reward vs. Episode: Narrow Design Space **(b)** Reward vs. Episode: Wide Design Space

Figure 20. Reward vs. Episode for Learning Mechanical Design

number of training steps due to system memory constraints.

The average rewards for both the narrow and the wide design space agents are shown in Figure 20. They represent the agents learning a converging solution to the problem of finding optimal design parameters. Looking at Figure 20a, it is apparent that given a more narrow design space, both the high and the specified jumping agents were still able to learn a converging solution. It can also be observed that there exists more variance for the specified height agent type compared to the height agents. Looking at Figure 20b, it is also apparent that the agents given a broader design space where both able to learn a converging design solution. Though given more designs to choose from, it appears the specified height agents are taking longer to search the design space, and towards the end, finding a designs with less variance.

4.6 Jumping Performance

4.6.1 Narrow Design Space. Figure 21 shows the height achieved by the learned designs for the agents given the narrow range of possible damping ratio values. For the agents learning designs to maximize jump height, Figure 21 can be compared

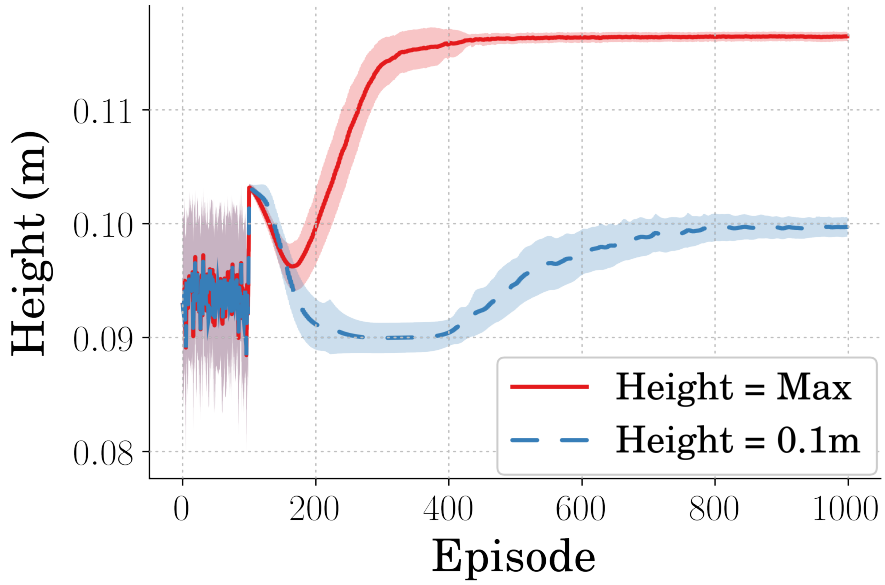


Figure 21. Height Reached During Training

with Figure 18 showing that the agent learned a design nearing one which would achieve maximum performance. Additionally, looking at the agents learning designs to jump to the specified 0.1 m, the designs learned accomplish this with slightly more variance than that of the maximum height case.

The average and standard deviation of the spring constant and damping ratio

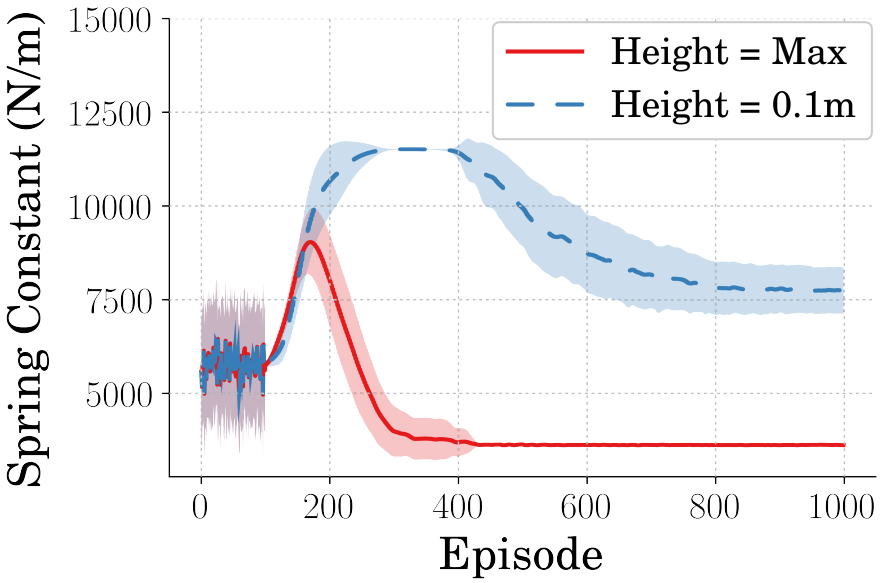


Figure 22. Spring Constant Selected During Training

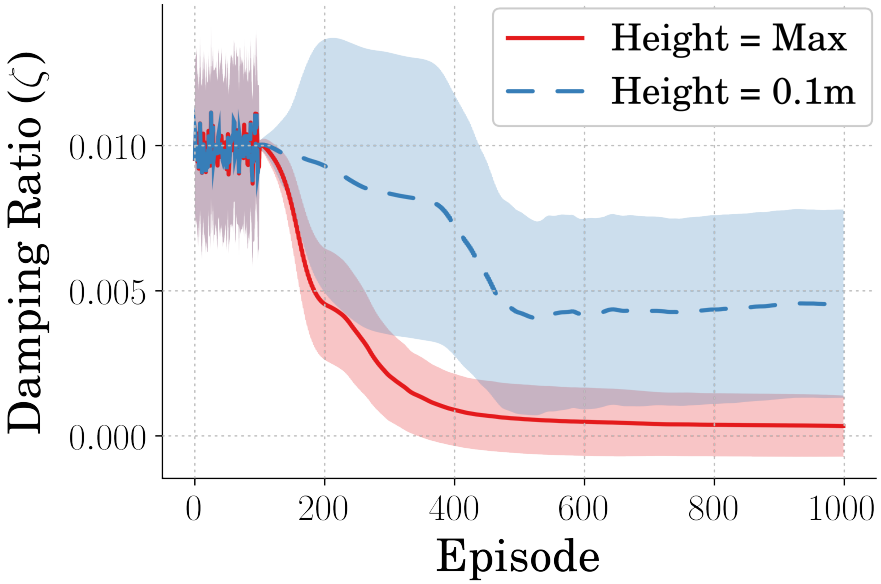


Figure 23. Damping Ratio Selected During Training

design parameters the agents selected during training are shown in Figs. 22 and 23. These plots represent the learning curves for the agents learning design parameters to maximize jump height and the agents learning design parameters to jump to 0.01 m. There is a high variance in both the spring constant and the damping ratio found for the agents that learned designs to jump to a specified height. The agents which were learning designs which maximized height found designs with very little variance in terms of spring constant and significantly less variances in terms of damping ratio.

4.6.2 Wide Design Space. Figure 24 shows the height achieved by the learned designs for the agents given a wider range of damping ratios. For the agents learning designs to maximize jump height, Figure 24 can be compared with Figure 19 showing that the agents learned a design nearing one which would achieve maximum performance. Additionally, looking at the agents learning designs to jump to the specified 0.1 m, the designs learned accomplish this, only with slightly more variance than what is seen in the maximum height agents.

The average and standard deviation of the spring constant and damping ratio

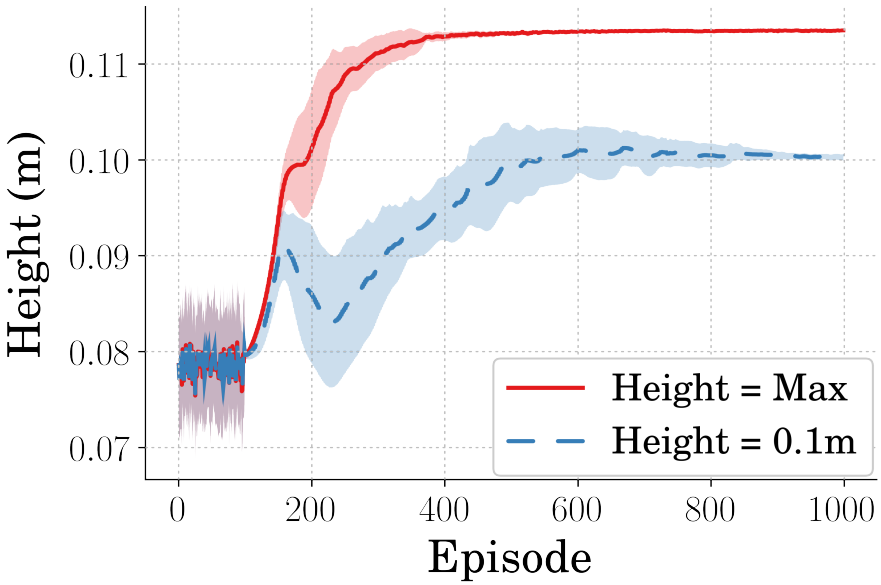


Figure 24. Height Reached During Training

design parameters the agents selected during training are shown in Figures 25 and 26. For the agents that learned designs to jump to a specified height, it can be seen that there is a high variance in spring constant throughout training. However, the majority of agents converge to a specific design, lowering the variance. The same can be seen in the damping ratio; however, the variance is mitigated significantly earlier in training.

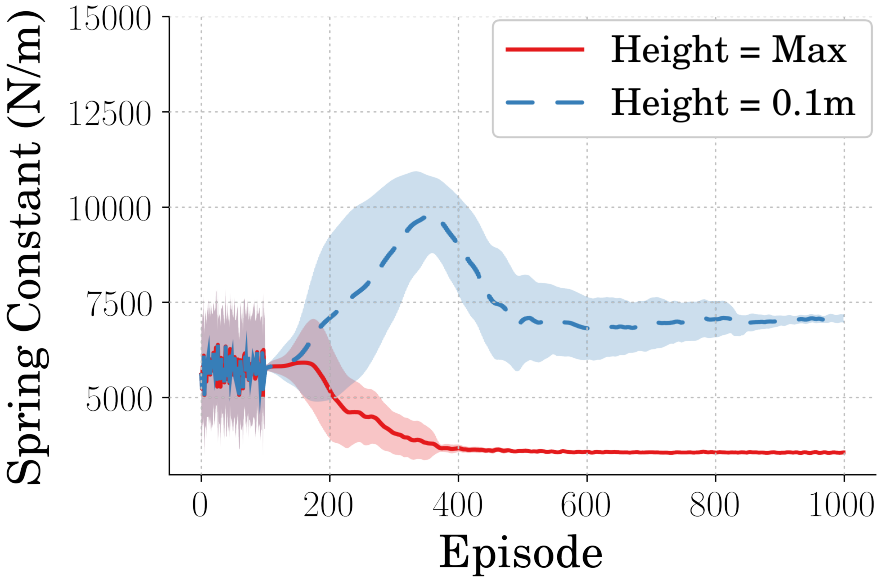


Figure 25. Spring Constant Selected During Training

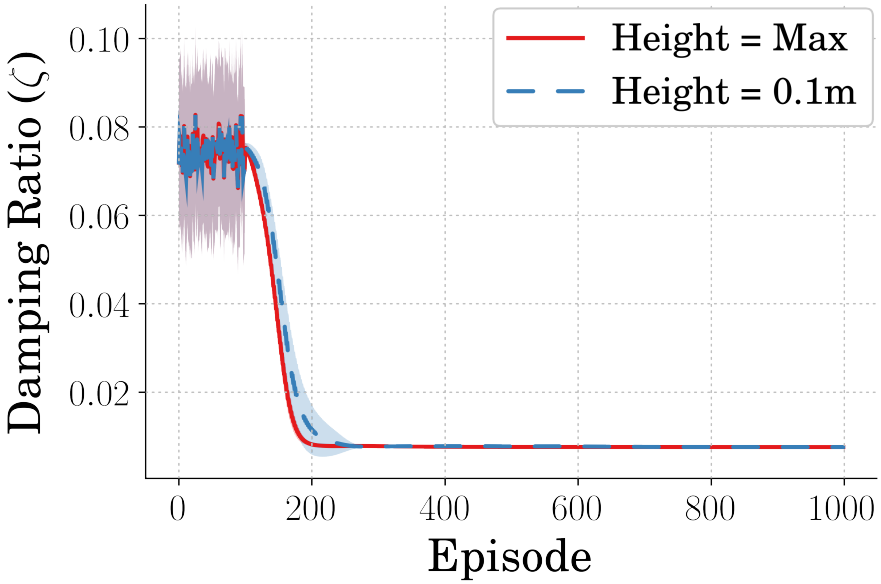


Figure 26. Damping Ratio Selected During Training

The agents which were learning designs that maximized height found them with very little variance in terms of spring constant and damping ratio.

4.6.3 Average Design Performance. The final mean and standard deviation of the design parameters for the two different cases are presented in Table 4. Figure 27 shows the jumping performance of the mean designs learned for both cases tested. The agents tasked with finding designs to jump to the specified 0.1 m, did so with minimal error. The difference seen in maximum height reached between the two

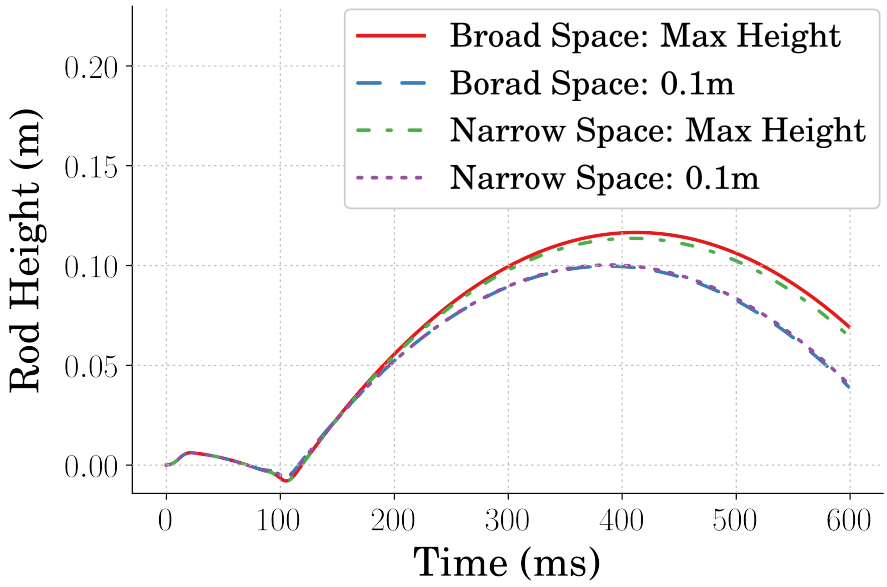


Figure 27. Height vs Time of Average Optimal Designs

Table 4. Learned Design Parameters

Training Case		Design Parameter	Mean	STD
Narrow Design Space	Max Height	Spring Constant	3.62e03	3.82e01
		Damping Ratio	3.37e-04	2.11e-03
	Specified Height	Spring Constant	7.74e03	1.24e03
		Damping Ratio	4.55e-03	6.49e-03
Broad Design Space	Max Height	Spring Constant	3.55e03	4.86e01
		Damping Ratio	7.53e-03	8.86e-06
	Specified Height	Spring Constant	7.07e03	2.16e02
		Damping Ratio	7.54e-03	3.27e-05

cases represents the difference in the damping ratio design space the agents had access to. The peak heights achieved can be compared again to Figures 18 and 19 to show that the agents learned designs nearing those achieving maximum performance.

4.7 Conclusion

The monopode model was used in conjunction with a predetermined control input to determine if a reinforcement learning algorithm (TD3) could be used to find optimal performing design parameters regarding jumping performance. This work was done in part to determine if reinforcement learning could be used as the mechanical design learner for an intelligent concurrent design algorithm. It was shown that when providing an agent with a design space that was smaller in size, the agents performed well in finding design parameters which met the performance constraints. The designs found were high in design variance, however. It was additionally shown that when provided with larger design space, the agents excelled at finding design parameters which were lower in design variance but still met the design constraints. It should be concluded ultimately that utilizing an RL algorithm, such as TD3, for the mechanical design aspect of a concurrent design method, is a viable solution.

V Concurrent Design of the Monopode System

Finding a control architecture for a mechanically defined system is often the the workflow for generating a controlled robotic system. However, the mechanical system is not always a simple one and generating a controller for it may require a more complex workflow. It is of interest as well to allow the mechanical parameters of the system, and therefore the system description, to be fluid, allowing for a more optimal mesh between controller and system. Designing the system and control input in unison has been researched and is ofter referred to as concurrent design. This strategy has been used to develop better performing mechatronics systems [16]. More recent work has used advanced methods such as evolutionary strategies to define robot design parameters [41]. In addition to evolutionary strategies, reinforcement learning has been shown to be a viable solution for concurrent design of 2D simulated locomotive systems [40]. This is further shown to be a viable method by demonstrating more complex morphology modifications in 3D reaching and locomotive tasks [18]. However, these techniques have not yet been applied to flexible systems for locomotive tasks. In this Chapter, a novel definition of a concurrent design architecture is purposed to find an optimal design and controller for the monopode jumping system defined in Chapter 2.

5.1 Concurrent Design Architecture

To define a concurrent design process utilizing RL, an algorithm is proposed which utilizes two instances of the TD3 algorithm creating an inner and an outer loop. The first instance, being responsible for learning the control policy, will be instantiated in a similar fashion to the what is seen in Chapter 2. It is the outer loop of the concurrent design process. The second instance, being responsible for learning the mechanical design, is instantiated within the outer loop and in a similar fashion to what is seen in Chapter 4. The second instance is the inner loop of the concurrent design

process. A key aspect of the inner loop instantiation is that rather than using a pre-defined control input, like what was shown in Chapter 4, the simulation will be using the input being trained by the outer loop instantiation. The algorithm is presented in Appendix 7 in greater detail.

5.2 Environment Definition

5.2.1 Learning the Controller. Similar to what was shown in Chapter 2, a traditional RL environment aligning with the standards set by OpenAI for a Gym environment was created. [36]. The monopode, also described in Chapter 2 was used to evaluate the methods discussed in this Chapter. The observation and action spaces were defined, respectively, as follows:

$$\mathcal{S} = [x_{a_t}, \dot{x}_{a_t}, x_t, \dot{x}_t] \tag{20}$$

$$\mathcal{A} = [\ddot{x}_{a_t}] \tag{21}$$

where x_t, $\dot{x}_t$ were the monopode's position and velocity at time t, and x_{a_t}, $\dot{x}_{a_t}$ and $\ddot{x}_{a_t}$ were the actuator's position, velocity and acceleration, respectively.

Differing from Chapter 2, only the stutter jumping command was evaluated. Therefore the stopping conditions for the environment were either the time step limit or the monopode completing two jumps. The time step limit was set to 400 steps at 0.01 seconds per step. The specifics of the stutter jump command are further discussed in Chapter 2. The values of the nominal constants are shown in Table 1 within Chapter 2.

5.2.2 Learning the Design. To allow an RL algorithm to find a mechanical design within the outer control loop, a second reinforcement learning environment conforming to the OpenAI Gym standard [36] was created in a similar fashion to what was discussed in Chapter 4. The control input, rather than being fixed, is captured from the outer loop and used to evaluate the performance of different design choices.

The mechanical parameters the algorithm was tasked with optimizing were the spring constant and the damping ratio of the monopode system.

At each episode during training, the agent selected a set of design parameters from a distribution of available designs. The actions applied, $\mathcal{A}$, and transitions saved, $\mathcal{S}$, from the environment were defined as follows:

$$\mathcal{A} = \{\{a_\alpha \in \mathbb{R} : [-0.9\alpha, 0.9\alpha]\}, \{a_\zeta \in \mathbb{R} : [0, 0.01]\}\} \tag{22}$$

$$\mathcal{S} = \left\{ \sum_{t=0}^{t_f} x_t, \sum_{t=0}^{t_f} \dot{x}_t, \sum_{t=0}^{t_f} x_{at}, \sum_{t=0}^{t_f} \dot{x}_{at} \right\} \tag{23}$$

where α is the nominal spring constant the monopode, x_t and $\dot{x}_t$ are the monopode's rod height and velocity steps, and x_{a_t} and $\dot{x}_{a_t}$ are the monopode's actuator position and velocity steps, all captured during simulation. Again, the values of the nominal constants are shown in Table 1 within Chapter 2.

5.3 Deploying the Algorithm

...

Describe the training setup.

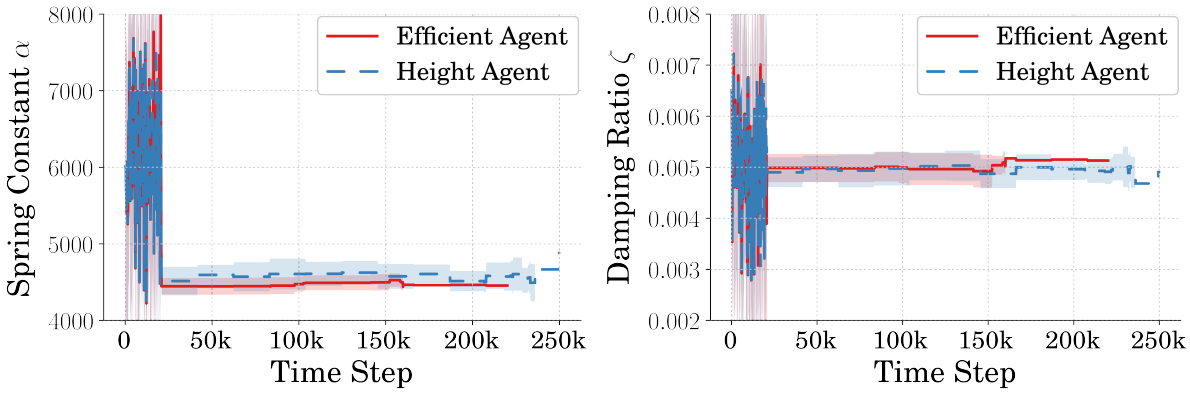
5.4 Mechanical Design Update

5.4.1 Discrete vs. Continuous. As is further discussed in Appendix 7, two different methods for implementing the inner loop of the concurrent design algorithm. The first is called the *discrete* method, where at each environment design update, the model learning the design is instantiated fresh and learns a design from scratch. The second method is called *continuous*, where at each environment update the model learning the design is saved and reloaded so that the model that is learning a design is the same over the course of the controller model training.

5.4.2 Averaging n Design Policies. To ensure that the mechanical design inner loop did not suffer from a single policy finding a locally optimal design, n number of design policies were instantiated at each design update. This was done to better replicate the results found in Chapter 4 where the results shown were averages of multiple design policies. This methodology was applied to both the discrete and the continuous methods of mechanical design update. In this work, n was set as 10 as it proved to resolve a policy finding a locally optimal design.

5.5 Mechanical Designs

Of the many different approaches taken, the one highlighted here is a discrete method where the design parameters were updated every 500 control policy updates. Figure 28 displays the average learning of the designs during the training of the control policy for both the efficient and high jumping control types. This data was captured at every 500 control policy update interval, after the environment's parameters were updated per the learned design. It is apparent that the design policy can learn a design early in training, even utilizing a loosely defined control policy. It is also apparent that the design learned early in the training process is one that is close to optimal for the control policy even after the control policy is better defined. Figure 28a showing the



(a) Spring Constant Selected During Training **(b)** Damping Ratio Selected During Training

Figure 28. Designs Learned Using Update Rate = 1:500 Control Policy Updates

learning of the spring constant shows that the efficient controller types learn a lower value to increase efficiency. Figure 28b being the learning of the damping ratio, shows little difference between the two control types.

5.6 Controller Performance

Ten different control policies were trained and each of them was trained to be optimal with a specific design. Therefore, to evaluate the average performance of the concurrent design method discussed in this Chapter, each of the policies was evaluated on its respective optimal design and the performance of them were averaged.

Figure 29 shows the average input learned from the control policies for the efficient and high jumping control types. Similar to what was seen in Chapter 2, it is apparent that there are distinct differences in the control techniques from the efficient and high controllers. Like what was discovered within the single jump command from Chapter 2, the efficient controller type learns a policy that in its first acceleration command accelerates in the positive direction. This does not take advantage of the space the spring has to decompress but also uses less power.

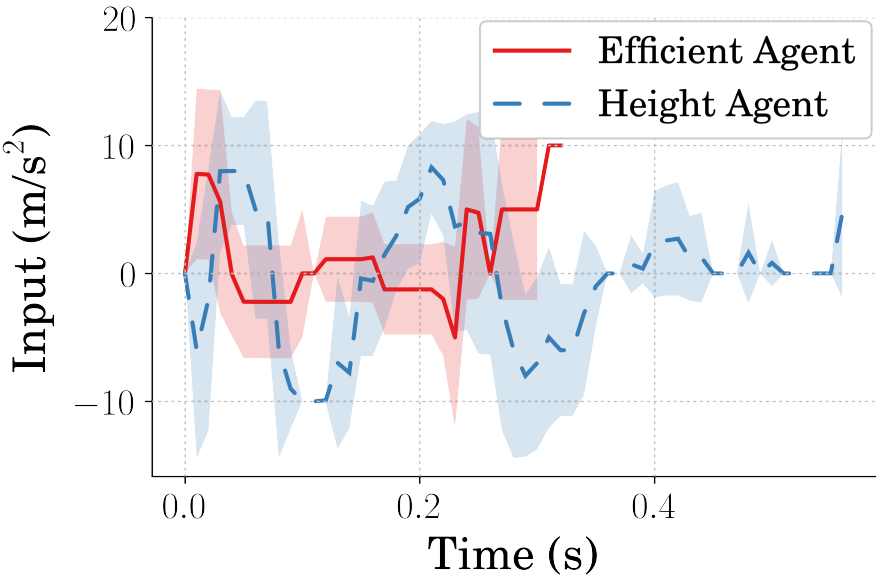


Figure 29. Average Input Performance

Average jumping performance for the efficient and high command types are shown in Figure 30. It is firstly obvious that the high jumping controllers learned a command that jumped the monopode significantly higher. It can be seen as well, as the input suggested, the efficient controllers learned a policy that did not utilize the spring decompression available.

5.7 Conclusion

The conclusion.

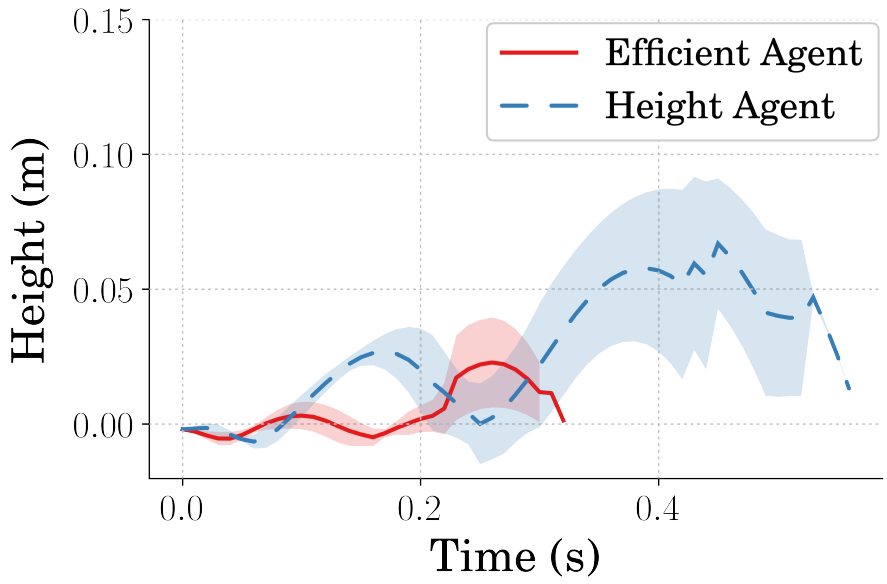


Figure 30. Average Jumping Performance

VI Concurrent Design of a Two-Link Flexible-Legged Jumping System

Discussion on concurrent design for robotic systems. How does this relate to our system and work? Overview of the system.

- Figure: system
- Table: System parameters

6.1 Jumping Cases

The cases which the agents were learning designs and controllers for.

6.2 Mechanical Designs: Simulation

This is the data basically.

6.3 Controller Performance: Simulation

- Figure: the performance of the controllers with nominal designs
- Figure: the performance of the controllers with learned designs

6.4 Mechanical Designs: Sim-to-Real

This is the data basically.

6.5 Controller Performance: Sim-to-Real

- Figure: the performance of the controllers with nominal designs
- Figure: the performance of the controllers with learned designs

6.6 Conclusion

The conclusion.

VII Appendix: Concurrent Design Algorithm

The algorithm presented, is an simple terms, an instantiation of the TD3 algorithm within an instantiation of the TD3 algorithm. Line 18, highlighted with green text, denotes the start of the inner loop and is responsible for learning a mechanical design similar to what is shown in Chapter 4. This inner loop runs before the control policy is updated depending on a hyperparameter that is related to how often the mechanical design should be updated. What wraps the green text, is considered the outer loop, and is responsible for learning a control policy.

The inner and outer loop are concurrent in that the inner loop takes the policy being trained in the outer loop and uses it to simulate the environment to learn an optimal mechanical design. Once the design has been learned, the inner loop passes the design back to the outer loop so that it can modify the environment which it is using to train a control policy.

7.0.1 Additional Hyperparameters. This algorithm presents an additional hyperparameter on top of the ones already present, being the rate at which the design is updated within the outer loop (see blue text). Testing this hyperparameter has shown ...

What happens when we change this hyperparameter?

Additionally, there are n, number of design updating policies instantiated within the inner loop. This is to better replicate the results found in Chapter 4 and to avoid issues where a single policy might find a local optimal design. In this work, n was set to 10 as it provided enough variation to keep a single bad policy from effecting the average design learned.

7.0.2 Discrete vs. Continuous. Additionally, there are two methods for implementing the inner loop. The first being at every instantiation, the policy

parameters learning a design (line 20) are initialized from nothing having no built intuition for good designs. To accomplish this, load would be removed from the description on line 20 of the algorithm. As for the second method, rather than starting with an untrained policy every design update, the policy is saved and then reloaded the next time the design is updated. During the first design update, n policies are created and learn a design. They are then saved along with their replay buffers so they can be reloaded to continue updating the mechanical design. ...

Test this and get some results.

1: Input: initialize policy parameters θ_{ctr}, Q-function parameters $\phi_{1,ctr}$ and $\phi_{2,ctr}$ and empty replay buffer, $\mathcal{D}_{ctr}$

2: Set target parameters equal to main parameters: $\theta_{ctr,targ} \leftarrow \theta$, $\phi_{1,ctr,targ} \leftarrow \phi_{1,ctr}$, $\phi_{2,ctr,targ} \leftarrow \phi_{2,ctr}$

3: **while** Not Converged **do**

4: Observe system state s_{ctr} and select action

$a_{ctr} = \text{clip}(\mu_{\theta_{ctr}}(s_{ctr}) + \epsilon, a_{ctr,low}, a_{crt,high}), \quad \epsilon \sim \mathcal{N}$

5: Execute the action a in the environment

6: Observe the next state s'_{ctr} and the reward r_{ctr} (verify if the state s'_{ctr} is a terminal state d_{ctr})

7: Store $(s_{ctr}, a_{ctr}, r_{ctr}, s'_{ctr}, d_{ctr})$ in the replay buffer $\mathcal{D}_{ctr}$

8: **if** s'_{ctr} is terminal **then**

9: Reset environment

10: **end if**

11: **if** Update Parameter % Update Frequency **then**

12: **for** j in range number of updates **do**

13: Sample random batch of transitions from buffer $\mathcal{R}_{ctr}$

14: Compute target actions:

$$a_{ctr} = \text{clip}(\mu_{\theta_{ctr,targ}}(s'_{ctr}) + \text{clip}(\epsilon, -c, c), a_{ctr,low}, a_{ctr,high}), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

Compute targets:

$$y(r_{ctr}, s'_{ctr}, d_{ctr}) = r_{ctr} + \gamma\,(1 - d_{ctr}) \min_{i=1,2} Q_{\phi_{ctr,targ,i}}(s'_{ctr}, a_{ctr}(s'_{ctr}))$$

Update the Q-function by way of gradient decent:

$$\nabla_{\theta_{ctr}} \frac{1}{|B|} \sum_{(s_{ctr}, a_{ctr}, r_{ctr}, s'_{ctr}, d_{ctr}) \epsilon B} \left(Q_{\phi_{ctr,i}}(s_{ctr}, a_{ctr}) - y(r_{ctr}, s'_{ctr}, d_{ctr})\right)^2 \quad \text{for } i = 1, 2$$

if j % Policy Delay is 0 **then**

if Update Design % Update Frequency **then**

for i in range n instantiations of a mechanal design policy **do**

Input: initialize/load policy parameters θ_{des}, Q-function parameters $\phi_{1,des}$ and $\phi_{2,des}$ and empty replay buffer, $\mathcal{D}_{des}$

Set target parameters equal to main parameters: $\theta_{des,targ} \leftarrow \theta$, $\phi_{1,des,targ} \leftarrow \phi_{1,des}$, $\phi_{2,des,targ} \leftarrow \phi_{2,des}$

while Not Converged **do**

Observe system state s_{des} and select a design $a_{des} = \text{clip}(\mu_{\theta_{des}}(s_{des}) + \epsilon, a_{des,low}, a_{crt,high}), \quad \epsilon \sim \mathcal{N}$

Simulate the design a in the environment using $\pi_{\theta_{ctr}}$

Observe the simulation s'_{des} and the reward r_{des} (verify if the state s'_{des} is a terminal state d_{des})

Store $(s_{des}, a_{des}, r_{des}, s'_{des}, d_{des})$ in the replay buffer $\mathcal{D}_{des}$

if s'_{des} is terminal **then**

Reset environment

end if

if Update Parameter % Update Frequency **then**

for j in range number of updates **do**

Sample random batch of transitions from buffer $\mathcal{R}_{des}$

Compute target actions:

$$a_{des} = \text{clip}(\mu_{\theta_{des,targ}}(s'_{des}) + \text{clip}(\epsilon, -c, c), a_{des,low}, a_{des,high}), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

Compute targets:

$$y(r_{des}, s'_{des}, d_{des}) = r_{des} + \gamma\,(1 - d_{des}) \min_{i=1,2} Q_{\phi_{des,targ,i}}(s'_{des}, a_{des}(s'_{des}))$$

Update the Q-function by way of gradient decent:

$$\nabla_{\theta_{des}} \frac{1}{|B|} \sum_{(s_{des}, a_{des}, r_{des}, s'_{des}, d_{des}) \epsilon B} (Q_{\phi_{des,i}}(s_{des}, a_{des}) - y(r_{des}, s'_{des}, d_{des}))^2 \quad \text{for } i = 1, 2$$

if j % Policy Delay is 0 **then**

Update policy by one step of gradient decent:

$$\nabla_{\theta_{des}} \frac{1}{|B|} \sum_{s_{des} \epsilon B} Q_{\phi_{des,1}}(s_{des}, \mu_{\theta_{des}}(s_{des}))$$

Update target networks by:

$$\phi_{des,targ,i} \leftarrow \rho \phi_{des,targ,i} + (1 - \rho)\phi_{des,i} \quad \text{for } i = 1, 2$$

$$\theta_{targ} \leftarrow \rho \theta_{targ} + (1 - \rho)\theta$$

end if

end for

end if

end while

Capture the final design that was learned as d_i

end for

Compute the average of the designs:

$$\mathbb{D} = \frac{\sum_{i=1}^{n} \mathrm{d}_i}{n}$$

end if

Update the environment with the learned design $\mathbb{D}$

Update policy by one step of gradient decent:

$$\nabla_{\theta_{ctr}} \frac{1}{|B|} \sum_{s_{ctr} \epsilon B} Q_{\phi_{ctr,1}}(s_{ctr}, \mu_{\theta_{ctr}}(s_{ctr}))$$

Update target networks by:

$$\phi_{ctr,targ,i} \leftarrow \rho \phi_{crt,targ,i} + (1 - \rho)\phi_{ctr,i} \quad \text{for } i = 1, 2$$

$$\theta_{crt,targ} \leftarrow \rho \theta_{ctr,targ} + (1 - \rho)\theta_{ctr}$$

end if

51: **end for**

52: **end if**

53: **end while**

Bibliography

[1] Folkertsma, G. A., Kim, S., and Stramigioli, S., "Parallel stiffness in a bounding quadruped with flexible spine," *IEEE International Conference on Intelligent Robots and Systems*, pp. 2210–2215, 2012.

[2] Park, H. W., Wensing, P. M., and Kim, S., "High-speed bounding with the MIT Cheetah 2: Control design and experiments," *International Journal of Robotics Research*, vol. 36, no. 2, pp. 167–192, 2017.

[3] Seok, S., Wang, A., Chuah, M. Y., Hyun, D. J., Lee, J., Otten, D. M., Lang, J. H., and Kim, S., "Design principles for energy-efficient legged locomotion and implementation on the MIT Cheetah robot," *IEEE/ASME Transactions on Mechatronics*, vol. 20, no. 3, pp. 1117–1129, 2015.

[4] Blackman, D. J., Nicholson, J. V., Pusey, J. L., Austin, M. P., Young, C., Brown, J. M., and Clark, J. E., "Leg design for running and jumping dynamics," *2017 IEEE International Conference on Robotics and Biomimetics, ROBIO 2017*, vol. 2018-Janua, pp. 2617–2623, 2018.

[5] Sugiyama, Y. and Hirai, S., "Crawling and jumping of deformable soft robot," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 4, no. c, pp. 3276–3281, 2004.

[6] Galloway, K. C., Clark, J. E., Yim, M., and Koditschek, D. E., "Experimental investigations into the role of passive variable compliant legs for dynamic robotic locomotion," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1243–1249, 2011.

[7] Hurst, J., "The Role and Implementation of Compliance in Legged Locomotion," *The International Journal of Robotics Research*, vol. 25, no. 4, p. 110, 2008.

[8] Pratt, G. A. and Williamson, M. M., "Series elastic actuators," *IEEE International Conference on Intelligent Robots and Systems*, vol. 1, pp. 399–406, 1995.

[9] Ahmadi, M. and Buehler, M., "Stable control of a simulated one-legged running robot with hip and leg compliance," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 1, pp. 96–104, 1997.

[10] Kani, M. H. H. and Ahmadabadi, M. N., "Comparing effects of rigid, flexible, and actuated series-elastic spines on bounding gait of quadruped robots," pp. 282–287, 2013.

[11] Horigome, A., Qudsi, Y., Fisher, E., and Vaughan, J., "Robot Jumping with Curved-Beam Flexible Legs Orange marker Blue marker Tether,"

[12] Luo, Z.-h., "Direct Strain Feedback Control of Flexible Robot Arms: New Theoretical and Experimental Results," vol. 38, no. 11, 1993.

[13] He, W., Gao, H., Zhou, C., Yang, C., and Li, Z., "Reinforcement Learning Control of a Flexible Two-Link Manipulator: An Experimental Investigation," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–11, 2020.

[14] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D., "Continuous control with deep reinforcement learning," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[15] Dwiel, Z., Candadai, M., and Phielipp, M., "On Training Flexible Robots using Deep Reinforcement Learning," *IEEE International Conference on Intelligent Robots and Systems*, pp. 4666–4671, 2019.

[16] Li, Q., Zhang, W. J., and Chen, L., "Design for control - A concurrent engineering approach for mechatronic systems design," *IEEE/ASME Transactions on Mechatronics*, vol. 6, no. 2, pp. 161–169, 2001.

[17] Chen, T., He, Z., and Ciocarlie, M., "Hardware as Policy: Mechanical and computational co-optimization using deep reinforcement learning," *arXiv*, no. CoRL, 2020.

[18] Schaff, C., Yunis, D., Chakrabarti, A., and Walter, M. R., "Jointly learning to construct and control agents using deep reinforcement learning," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2019-May, pp. 9798–9805, 2019.

[19] Whitman, J., Bhirangi, R., Travers, M., and Choset, H., "Modular Robot Design Synthesis with Deep Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 06, pp. 10418–10425, 2020.

[20] Zhao, W., Queralta, J. P., and Westerlund, T., "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: A Survey," *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020*, pp. 737–744, 2020.

[21] Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T., and Riedmiller, M., "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards," pp. 1–10, 2017.

[22] Plappert, M., Andrychowicz, M., Ray, A., McGrew, B., Baker, B., Powell, G., Schneider, J., Tobin, J., Chociej, M., Welinder, P., Kumar, V., and Zaremba, W., "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research," pp. 1–16, 2018.

[23] Fujimoto, S., Van Hoof, H., and Meger, D., "Addressing Function Approximation Error in Actor-Critic Methods," *35th International Conference on Machine Learning, ICML 2018*, vol. 4, pp. 2587–2601, 2018.

[24] Sutton, R. S. and Matheus, C. J., "Learning Polynomial Functions by Feature Construction," *Machine Learning Proceedings 1991*, pp. 208–212, 1991.

[25] Watkins, C., "Learning From Delayed Rewards," 1989.

[26] Mnih, V. and Silver, D., "Playing Atari with Deep Reinforcement Learning," pp. 1–9.

[27] Dormann, A. R., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Noah, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, vol. 22, no. 22, pp. 1–8, 2021.

[28] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and Desmaison, Alban and Kopf, Andreas and Yang, Edward and DeVito, Zacha, S., *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates, Inc., 2019.

[29] Ha, S., Xu, P., Tan, Z., Levine, S., and Tan, J., "Learning to walk in the real world with minimal human effort," *arXiv*, no. CoRL, pp. 1–11, 2020.

[30] Fankhauser, P., Hutter, M., Gehring, C., Bloesch, M., Hoepflinger, M. A., and Siegwart, R., "Reinforcement learning of single legged locomotion," *IEEE International Conference on Intelligent Robots and Systems*, pp. 188–193, 2013.

[31] Xiao, Q., Cao, Z., and Zhou, M., "Learning locomotion skills via model-based proximal meta-reinforcement learning," *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, vol. 2019-Octob, pp. 1545–1550, 2019.

[32] Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., and Levine, S., "Learning to Walk Via Deep Reinforcement Learning," 2019.

[33] Tsounis, V., Alge, M., Lee, J., Farshidian, F., and Hutter, M., "DeepGait: Planning and control of quadrupedal gaits using deep reinforcement learning," *arXiv*, no. 1, pp. 1–8, 2019.

[34] Da, X., Xie, Z., Hoeller, D., Boots, B., Anandkumar, A., Zhu, Y., Babich, B., and Garg, A., "Learning a Contact-Adaptive Controller for Robust, Efficient Legged Locomotion," vol. 1, no. c, 2020.

[35] Blickhan, R. and Full, R. J., "Similarity in multilegged locomotion: Bouncing like a monopode," *Journal of Comparative Physiology A*, vol. 173, no. 5, pp. 509–517, 1993.

[36] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W., "OpenAI Gym," pp. 1–4, 2016.

[37] Harper, M. Y., Nicholson, J. V., Collins, E. G., Pusey, J., and Clark, J. E., "Energy efficient navigation for running legged robots," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2019-May, pp. 6770–6776, 2019.

[38] Pace, J., Harper, M., Ordonez, C., Gupta, N., Sharma, A., and Collins, E. G., "Experimental verification of distance and energy optimal motion planning on a skid-steered platform," *Unmanned Systems Technology XIX*, vol. 10195, p. 1019506, 2017.

[39] Luck, K. S., Amor, H. B., and Calandra, R., "Data-efficient co-adaptation of morphology and behaviour with deep reinforcement learning," *arXiv*, no. CoRL, 2019.

[40] Ha, D., "Reinforcement learning for improving agent design," *Artificial Life*, vol. 25, no. 4, pp. 352–365, 2019.

[41] Wang, T., Zhou, Y., Fidler, S., and Ba, J., "Neural graph evolution: Towards efficient automatic robot design," *arXiv*, pp. 1–17, 2019.

[42] Hu, S., Yang, Z., and Mori, G., "Neural fidelity warping for efficient robot morphology design," *arXiv*, 2020.

[43] Vaughan, J., "Jumping Commands For Flexible-Legged Robots," 2013.

[44] Sorensen, K. L. and Singhose, W. E., "Command-induced vibration analysis using input shaping principles," *Autom.*, vol. 44, pp. 2392–2397, 2008.

[45] Singer, N. C. and Seering, W. P., "Preshaping Command Inputs to Reduce System Vibration," *Journal of Dynamic Systems, Measurement, and Control*, vol. 112, no. 1, pp. 76–82, 1990.

[46] Singhose, W., Seering, W., and Singer, N., "Residual Vibration Reduction Using Vector Diagrams to Generate Shaped Inputs," *Journal of Mechanical Design*, vol. 116, no. 2, pp. 654–659, 1994.

Albright, Andrew. Bachelor of Science, North Carolina State University, Spring 2020;
Masters of Science, University of Louisiana at Lafayette, Spring 2022
Major: Mechanical Engineering
Title of Thesis: Mechanical Design and Control of Flexible-Legged Jumping Robots
Thesis Director: Dr. Joshua E. Vaughan
Pages in Thesis: 125; Words in Abstract: 185

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus nec tellus eget elit aliquet accumsan sit amet in lacus. In hac habitasse platea dictumst. Ut sit amet elit odio. Aenean lobortis mollis metus, sed consequat neque tristique in. Curabitur nec hendrerit metus. Praesent non scelerisque urna, vitae iaculis diam. Aliquam nisl est, imperdiet eu nulla sed, bibendum pulvinar arcu. In ultricies purus purus, vulputate congue justo volutpat ut. Donec nunc magna, rutrum nec turpis et, viverra efficitur lorem. In hac habitasse platea dictumst. Vestibulum maximus lobortis nisl, eget molestie sem sollicitudin nec. Mauris ut enim eu ipsum auctor rhoncus ac vel eros.

Vivamus tincidunt, tortor eu rutrum dapibus, orci turpis porta metus, ac iaculis quam eros sollicitudin nisl. Nam id massa elementum, commodo mi at, lobortis nisl. Fusce vestibulum eu lorem non aliquam. Morbi eleifend tortor id metus elementum, ac tincidunt lorem commodo. Pellentesque vestibulum, erat in tempus vehicula, ex urna auctor leo, ut lobortis eros mauris nec erat. Aliquam erat volutpat. Sed sed pretium risus.

Biographical Sketch

Forrest Montgomery was born in Lafayette, Louisiana for all intents and purposes. He began his academic career at the University of Louisiana with an internal struggle between majoring in Mechanical Engineering or Industrial Design. This thesis is evident of the choice he made. After earning his Bachelor's degree at the University of Louisiana at Lafayette in the Spring of 2015, he joined the CRAWLAB and conducted research in dynamics, controls, and robotics under the tutelage of Dr. Joshua Vaughan. This research culminated with earning a Master's degree in Mechanical Engineering again at the University of Louisiana at Lafayette in the Summer of 2017.