University of California
Santa Barbara

Robotics, Reinforcement Learning, Markov Chains, Fractal Dimensions, and Other Buzz Words

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Sean Gillen

Committee in charge:

Professor Katie Byl, Chair
Professor Joao Hespanha
Professor Linda Petzold
Professor Yon Vissel

December 2021

The Dissertation of Sean Gillen is approved.

Professor Joao Hespanha

Professor Linda Petzold

Professor Yon Vissel

Professor Katie Byl, Committee Chair

December 2021

Robotics, Reinforcement Learning, Markov Chains, Fractal Dimensions, and Other Buzz Words

by

Sean Gillen

Dedication here

Acknowledgements

I acknowledge no one but myself.

SEAN GILLEN

sgillen@ucsb.edu | github.com/sgillen | linkedin.com/in/sgillen

EDUCATION

University of California Santa Barbara September 2017 - Present
MS/PhD Electrical And Computer Engineering GPA: 4.0

University of Maryland College Park September 2013 - May 2017
BS Electrical Engineering GPA: 3.5

TECHNICAL SKILLS

Languages Python, C, C++, MATLAB, Bash, Mathematica, Elisp, LaTeX
Libraries ROS/Gazebo, OpenCV, Pytorch, Tensorflow

WORK HISTORY

University of California Santa Barbara September 2017 - Present
Graduate Student Researcher

· Conducting research fusing traditional control theory with deep reinforcement learning (**Python, C++, Matlab**), focusing on the control of under-actuated dynamic systems, and legged locomotion in particular.

Veoneer June 2019 - February 2020
Robotics Consultant

· Wrote a device driver **C++** and **ROS** node for a new automotive camera system.
· Implemented camera authentication on an external msp430.

Northrop Grumman June 2016 - September 2016
Electrical Engineer

· System level integration of several sensor payloads in an unmanned underwater vehicle (UUV).
· Wrote software (**Python**) to collect, clean, and analyze sensor/log data from the UUV.

Horn Point Laboratory June 2014 - June 2015
Software Engineer

· Wrote software (**C++**) for a Remus 600 UUV to provide altitude control, user defined acoustic modem messages, augmented data simulation, and adaptive mission planning to the vehicle.

Naval Air Systems Command (NAVAIR) June 2015 - September 2015
Electrical Engineer

· Developed a prototype vision system capable of imaging in extremely turbid underwater environments.
· Heavily modified a commercial underwater vehicle to house and control aforementioned visioning system.

University Of Maryland September 2015 - June 2016
Undergraduate Student Researcher

· Implemented computer simulation of nematic liquid crystals using **Matlab** and **Mathematica**

SIDE PROJECTS

Project Qubo

- Collaborated with several other engineers to build an autonomous underwater octocoptor from scratch. The vehicle can autonomously navigate to various targets.
- Wrote majority of the vision (**OpenCV**), control (**C++**), autonomy (**Python**), and embedded (**Arduino**) software, using **ROS** as a middleware.

Compressed air engine

- Manufactured a small compressed air engine, using manual and CNC Machine tools.

PUBLICATIONS

Sean Gillen and Katie Byl. Mesh based analysis of low fractal dimension reinforcement learning policies. *International Conference on Robotics and Automation*, 2021

Sean Gillen and Katie Byl. Explicitly encouraging low fractional dimensional trajectories via reinforcement learning. *4th Conference of Robotic Learning*, 2020

Sean Gillen, Marco Molnar, and Katie Byl. Combining deep reinforcement learning and local control for the acrobot swing-up and balance task. *59th IEEE Conference on Decision and Control*, 2020

TEACHING

Intermediate C programming (Spring & Fall 2016)
Introduction to Circuits (Fall 2017)
Digital Control (Winter 2017)
Introduction to Probability & Statistics (Spring 2017)
Robotic Modeling & Control (Fall 2018)
Operating Systems (Spring 2019)
Compilers (Fall 2020)
Algorithms & Data Structures (Winter 2021, Spring 2021)

HONORS & AWARDS

Bodharamik Engineering scholarship (2016)

Sikorsky Corporate Partners scholarship (2016)

Outstanding TA award (2017)

Abstract

Robotics, Reinforcement Learning, Markov Chains, Fractal Dimensions, and Other Buzz Words

by

Sean Gillen

Abstract text.

Contents

Chapter 1

Introduction

Reinforcement learning has a long history, which I will tell you about at length.

Now in recent years research into deep reinforcement has exploded, with many impressive results advancing the state of the art in many fields. The focus of this work is on robotics, and again, a number of interesting results have been achieved with DRL in recent years.

However there are a number of glaring issues I see with the current state of the art in reinforcement learning. The first is that despite a number of problems that

Chapter 2

Background

2.1 Reinforcement Learning

Chapter 3

Switching

3.1 INTRODUCTION

Advances in machine learning have allowed researchers to leverage the massive amount of compute available today in order to better control robotic systems. The result is that modern model-free reinforcement learning has been used to solve very difficult problems. Recent examples include controlling a 47 DOF humanoid to navigate a variety of obstacles [**?**], dexterously manipulating objects with a 24 DOF robotic hand [**?**], and allowing a physical quadruped robot to run [**?**], and recover from falls [**?**].

Despite this, these algorithms can struggle on certain low dimensional problems from the nonlinear control literature. Namely the acrobot [**?**] and the cart pole pendulum. These are both under-actuated mechanical systems that have unstable fixed points in their unforced dynamics (see section 3.2.2). Typically, the goal is to bring the system to this fixed point and keep it there. In this paper we focus on the acrobot as we found less examples of model free reinforcement learning performing well on this task.

It is not uncommon to see some variation of these systems tackled in various reinforcement benchmarks, but we have found these problems have usually been artificially

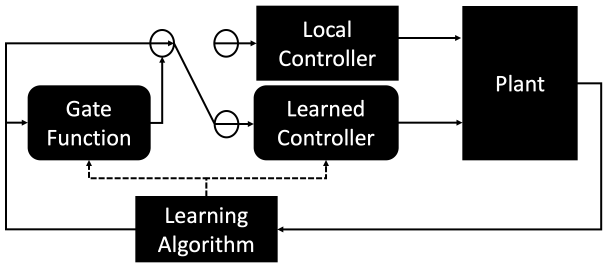


Figure 3.1: System diagram for the new technique proposed in this paper. Rounded boxes represent learned neural networks, squared boxes represent static, hand crafted functions. The local controller is a hand designed LQR, the swing-up controller is obtained via reinforcement learning, and the gating function is trained as a neural network classifier

modified to make them easier. For example the very popular OpenAI Gym benchmarks [**?**] includes an acrobot task. But the objective is only to get the system in the rough area of the unstable fixed point, and the dynamics are integrated with a fixed time-step of .2 seconds, which makes the problem much easier and unrepresentative of a physical system. We have found that almost universally, modern model free reinforcement learning algorithms fail to solve a more realistic version of the task. Notably, the Deep Mind control suite [**?**] includes the full acrobot problem, and all but one algorithm that they tested (the exception being [**?**]) learned nothing, the average return after training was the same as before training.

Despite this, there are many traditional model based solutions [**?**], [**?**], that can solve this problem well. In this work we do not seek to improve upon the model based solutions to this problem, but to extend to the class of problems that model free reinforcement learning methods can be used to solve. We believe the methods used here to solve the acrobot can be extended to other problems, such as making robust walking policies.

One of the primary reasons why this problem is difficult for RL is that the region of state space that can be brought to the unstable fixed point is very small, even with

generous torque limits. An untrained RL agent explores by taking random actions in the environment. Reaching the region of attraction is rare, we found that for our system, random actions will reach the basin of attraction for a well designed LQR in about 1% of trials. However an RL agent doesn't have access to a well designed LQR at the start of training, in addition to reaching the region where stabilization is possible, the agent must also stabilize the acrobot for the agent to receive a strong reward signal. This results in successful trials in this environment being extremely rare, and therefore training is in-feasibly slow and sample inefficient.

Our solution to add a predesigned balancing controller into the system, this is comparatively easy to design, and can be done with a linear controller. Our contribution is a novel way to combine this balancing controller with an algorithm that is learning the swing-up behavior. We simultaneously learn the swing-up controller, and a function that switches between the two controllers.

3.1.1 Related Work

Work done by Randolov et. al. [**?**] is closely related to our own. In that work they construct a local controller, an LQR, and combine it with a learned controller to swing-up and balance an inverted double pendulum (similar to the acrobot we study but with actuators at both joints). The primary differences between our work and theirs is that they hard code the transition between their two controllers. In contrast we learn our transition function online and in parallel with our swing-up controller.

Work done by Yoshimoto et. al. [**?**], like ours, learns the transition function between controllers in order to swing-up and balance an acrobot. However, unlike our work they limit the controllers they switch between to pre-computed linear functions. In contrast our work simultaneously learns a nonlinear swing-up controller and the transition between

a learned and pre-computed balance controller.

Wikledt et. al [**?**] too swing-up and balance an acrobot using a combined neural network and LQR. However they only learn to swing-up from a single initial condition, whereas our method learns to solve the task from any initial position.

Doya [**?**] also learns many controllers using reinforcement learning, and adaptively switches between them. However unlike our work, the switching function is not learned using reinforcement learning, but is instead selected according to which of the controllers currently makes best prediction of the state at the current point in state space. We believe our model free updates will avoid the model bias that can be associated with such approaches. Furthermore our work allows for combining learned controllers with hand designed controllers, such as the LQR.

3.2 Background

3.2.1 Nomenclature

We formulate our problem as a Markov decision process, $M = (S, A, R)$. At each time step t, Our agent receives the current state $s_t \in S$ and chooses an action $a_t \in A$. It then receives a reward according to the reward function $r_t = R(s_t, a_t, s_{t+1})$. The goal is to find a policy $\pi : S \rightarrow p(A = a|s)$ that satisfies:

$\pi^* = \arg\max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})\right]$

3.2.2 Acrobot

The acrobot is described in Figure 3.2. It is a double inverted pendulum with a motor only at the elbow. We use the parameters from Spong [**?**]:

tableMass and inertial parameters used in simulation

Parameter	Value	Units
m_1, m_2	1	Kg
l_1, l_2	1	m
l_{c1}, l_{c2}	.5	m
I_1	.2	Kg*m^2
I_2	1.0	Kg*m^2

fig/cdc2020/acrobot2.png

Figure 3.2: Diagram for the acrobot system

The state of this system is $s_t = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$. The action $a_t = \tau$, is the torque at the elbow joint. The goal we wish to achieve is to take this system from any random initial state, to the upright state $gs = [\pi/2, 0, 0, 0]$, which we will refer to as the goal state. To achieve this goal, we seek the maximum of the following reward function:

$\mathrm{r}_t = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$ This was motivated by the popular Acrobot-v1 environment [**?**], We found empirically that for our algorithm this reward signal led to the same solutions as the more typical $s_t - gs$. However we found that some of the other algorithms we compared to perform better with the sinusoidal reward function.

We implement the system in python (all source code is provided, see footnote on page one), the dynamics are implemented using Euler integration with a time-step of .01 seconds, and the control is updated every .2 seconds. We experimented with smaller timesteps and higher order integrators, generally we found these made the balancing task easier, but made the wall clock time for the learning much slower.

3.2.3 Soft Actor Critic

Soft actor critic (SAC) is an off policy deep reinforcement learning algorithm shown to do well on control tasks with continuous actions spaces [**?**]. To aid in exploration, rather than directly optimize the discounted sum of future rewards, SAC attempt to find a policy that optimizes a surrogate objective:

$\mathrm{J}^{\mathrm{soft}} = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R_t + \alpha H(\pi(\cdot|s_t)) \Big)\right]$

Where H is the entropy of the policy.

SAC introduces several neural networks for the training. We define a soft value function $V_\phi(s_t)$, a neural network defined by weights ϕ, which approximate J^{soft} given the current state. Next we define two soft Q functions $Q_{\rho_1}(s_t, a_t)$ and $Q_{\rho_2}(s_t, a_t)$ which approximate J^{soft} given both the current state and the current action. Using two Q networks is a trick that aids the training by avoiding overestimating the Q function. We must also define a target soft value function $V_{\overline{\phi}}(s_t)$, which follows the value function via polyak averaging:

$\mathrm{V}_{\overline{\phi}^+}(s_t) = c_{py} V_{\overline{\phi}}(s_t) + (1 - c_{py}) V_\phi$

With c_{py} a fixed hyper parameter. We also define Π_θ, a neural network that outputs $\mu_\theta(s_t)$ and $\log(\sigma_\theta(s_t))$ which define the probability distribution of our policy π_θ. The action is given by:

$\mathrm{a}_t = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t)\epsilon_t)$

where ϵ_t is drawn from $N(0, 1)$.

SAC also make use of a replay buffer D which stores the tuple (s_t, a_t, r_t) after policy rollouts. When it is time to update we sample randomly from this buffer, and use those samples to compute our losses and update our weights.

With this we can define the losses for each of these networks (originated from [**?**])

The loss for our two Q functions is:

$\mathrm{L}^{Q} = \mathbb{E}_{s_t,a_t\sim D}\left[\frac{1}{2}\left(Q_\rho(s_t,a_t) - \hat{Q}(s_t,a_t)\right)^2\right]$ where $(\mathrm{s}_t,a_t) = r(s_t,a_t)+\gamma\mathbb{E}_{s_{t+1}}\left[V_{\overline{\phi}}(s_{t+1})\right]$

Our policy seeks to minimize:

$\mathrm{L}^{\pi} = \mathbb{E}_{s_t\sim D,\epsilon_t\sim N(0,1)}\left[\log\pi_\theta(f_\theta(\epsilon_t,s_t)|s_t) - Q_{\rho_1}(s_t,f_\theta(\epsilon_t,s_t)\right]$

And our value function:

$\mathrm{L}^{\mathrm{V}} = \mathbb{E}_{s_t\sim D}\left[\frac{1}{2}\left(V_\phi(s_t) - \hat{V}_\phi(s_t)\right)^2\right]$

Where

$_\phi = \mathbb{E}_{a_t\sim\pi_\theta}\left[Q^{min}(s_t,a_t) - log\pi_\theta(a_t|s_t)\right]$

And $Q^{min} = \min(Q_{\rho_1}(s_t,a_t), Q_{\rho_2}(s_t,a_t))$

SAC starts by doing policy roll outs, recording the state, action, reward, and the active controller at each time step. It stores these experiences in the replay buffer. After enough trials have been run, we run our update step. We sample from the replay buffer, and use these sampled states to compute the losses above. We then run one step of Adam [**?**] to update our network weights. We repeat this update n_u times with different samples. Finally we copy our weights to our target network and repeat until convergence (or some other stopping metric).

3.3 Switched Soft Actor Critic

Our primary contribution is to extend SAC in two key ways, we call the modified algorithm switched soft actor critic (SSAC). The first modification is a change to the structure of the learned controller in order to inject our domain knowledge into the learning. Our controller consists of three distinct components. The gate function, the balancing controller, and the swing-up controller. The gate, $G_\gamma : S \to [0,1]$, is a neural network parameterized by weights γ which takes the observations at each time step and outputs a number g_t representing which controller it thinks should be active. $g_t \approx 1$ implies high confidence that the balancing controller should be active, and $g_t \approx 0$ implies

the swing-up controller is active. This output is fed through a standard switching hysteris function, to avoid rapidly switching on the class boundary, parameters given in the appendix. The swing-up controller can be seen as the policy network from vanilla SAC, the action then is determined by equation (3.2.3). The parameters for these networks are given in the appendix. The balancing controller is a linear quadratic regulator $C : S \to A$ about the acrobot's unstable equilibrium. We use the LQR designed by Spong [**?**]:

Using

$$Q = \begin{pmatrix} 1000 & -500 & 0 & 0 \\ -500 & 1000 & 0 & 0 \\ 0 & 0 & 1000 & -500 \\ 0 & 0 & -500 & 1000 \end{pmatrix}, R = \begin{pmatrix} .5 \end{pmatrix}$$

The resulting control law is:

$$u = -Ks$$

with

$$K = [-1649.8, -460.2, -716.1, -278.2]$$

These three functions together form our policy, π_θ. Algorithm 1 demonstrates how the action is computed at each timestep.

We learn the basin of attraction for the regulator by framing it as a classification problem, our neural network takes as input the current state, and outputs a class prediction between 0-1. A one implying that the LQR is able to stabilize the system, and a zero implying that it cannot. We then define a threshold function $T(s)$, as a criteria for what we consider a successful trial:

$\mathrm{T(s)} = \mathrm{s}_t - gs < \epsilon_{thr} \quad \forall t \in \{N_e - b, ..., N_e\}$

Here s is understood to be an entire trajectory of states, N_e is the length of each episode, e_{thr} and b hyper parameters with values given in the appendix. We are following the convention of a programming language here, (3.3) returns one when the inequality holds, and zero otherwise. To gather data, we sample a random initial condition, do a policy roll out using the LQR, and record the value of 3.3 as the class label.

To train the gating network we minimize the binary cross entropy loss:

$\mathrm{L}^{\mathrm{G}} = \mathbb{E}_\gamma - [c_w y_i \log(G_\gamma(s_i)) + (1 - y_i) \log(1 - G_\gamma(s_i))]$

Where y_i is the class label for the ith sample, c_w is a class weight for positive examples. we set $c_w = \frac{n_t}{n_p} w$ where n_t is the total number of samples, n_p is the number of positive examples, and w is a manually chosen weighting parameter to encourage learning a conservative basin of attraction. We found that the learned basin was very sensitive to this parameter, a value of .01 empirically works well. Note that unlike the other losses above, the data here is not computed over a sample but is instead computed over the entire replay buffer. We found the gate was prone to "forgetting" the basin of attraction early in the training otherwise. This also allows us to update the gate infrequently compared to the other networks, and so the total impact on wall clock time is modest.

The second extension is a modification of the replay buffer D. We do this by constructing D from two separate buffers, D_n and D_r. Only roll outs that ended in a successful balance (as defined by equation (3.3)) are stored in D_r. The other buffer stores all trials, the same as the unmodified replay buffer. Whenever we draw experience from D, with probability p_d we sample from D_n, and with probability $(1 - p_d)$ we sample from D_r. We found this to speed up learning dramatically, as even with the LQR and a decent gating function in place, the swing-up controller finds the basin of attraction only in a tiny minority of trials.

Algorithm 1 Do-Rollout(G_γ, Π_θ, K)

$s = r = a = g = r = \{\}$
Reset environment, collect s_0 **for** $t \in \{0, ..., T\}$ **do**
    $g_t = hyst(G_\gamma(s_t))$ **if** $(g_t) == 1$ **then**
        $a_t = -Ks_t$ **else**
        Sample ϵ_t from $N(0, 1)$
    $a_t = \beta \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t) * \epsilon_t)$

    Take one step using a_t, collect $\{s_{t+1}, r_t\}$
    $s = s \bigcup s_t$, $r = r \bigcup r_t$
    $a = a \bigcup a_t$, $g = g \bigcup g_t$

return s, a, r, g

Algorithm 2 Switched Soft Actor Critic

Initialize network weights $\theta, \phi, \gamma, \rho_1, \rho_2$ randomly
set $\overline{\phi} = \phi$ **for** $n \in \{0, ..., N_e\}$ **do**
    $s, r, a, g =$ Do-Rollout(G_γ, Π_θ, K) **if** $T(s)$ **then**
        Store s, r, a in D_n

    Store s, r, a in D_r
    Store $s, g, T(s)$ in D_g **if** *Time to update policy* **then**
        sample s^r, a^r, r^r from D
    $\hat{Q} \approx R + \gamma V_{\overline{\phi}}(S)$
    $Q^{min} = \min(Q_{\rho_1}(s^r, a^r), Q_{\rho_2}(s^r, a^r))$
    $\hat{V} \approx Q^{min} - \alpha H(\pi_\theta(A|S))$
    Run one step of Adam on $L^Q(s^r, q^r, r^r)$
    Run one step of Adam on $L^\pi(s^r)$
    Run one step of Adam on $L^V(s^r)$
    $\overline{\phi} = q\overline{\phi} + (1 - q)\phi$
    if *Time to update gate* **then**
        Run one step of Adam on L^G using all samples in D_g

3.4 Results

3.4.1 Training

To train SSAC we first start by training the gate exclusively, using the supervised learning procedure outlined in section 3.3 This allows us to form an estimate of the basin of attraction before we try to learn to reach it. We trained the gate for 1e6 timesteps, and then trained both in parallel using algorithm 2 for another 1e6 timesteps. The policy, value, and Q functions are updated every 10 episodes, and the gate every 1000. The disparity is because, as mentioned earlier, the gate is updated using the entire replay buffer, while all the other losses are updated with one sample batch from the buffer. Hyperparameters were selected by picking the best performing values from a manual search, which are reported in the appendix.

In addition to training on our own version of SAC and Switched SAC we also examined the performance of several algorithms written by OpenAI and cleaned up by the community [**?**]. We examine PPO and TRPO, two popular trust region methods. A2C was included to compare to a non trust region, modern policy gradient algorithm. We also include TD3, which has been shown in the literature to do well on the acrobot and cartpole problems [**?**]

Stable baselines includes hyperparameters that were algorithmically tuned for each environment. For algorithms where parameters for Acrobot-v1 were available we chose those, some algorithms were missing tuned Acrobot-v1 examples, and for those we used parameters for Pendulum-v0, simply because it is another continuous, low dimensional task. Note we don't expect the hyper-parameters to impact the learned policy's score in this case, only how fast learning occurs. Reported rewards are averaged over 4 random seeds. Every algorithm makes 2e6 interactions with the environment. Also note that this project was largely inspired by spending a large amount of time manually tuning these

parameters to work on this task (with no success better than what we see here). Figure 3.3 shows the reward curve for our algorithm and the algorithms from stable baselines. Table 3.4.1 shows the mean and standard deviation for the final rewards obtained by all algorithms.

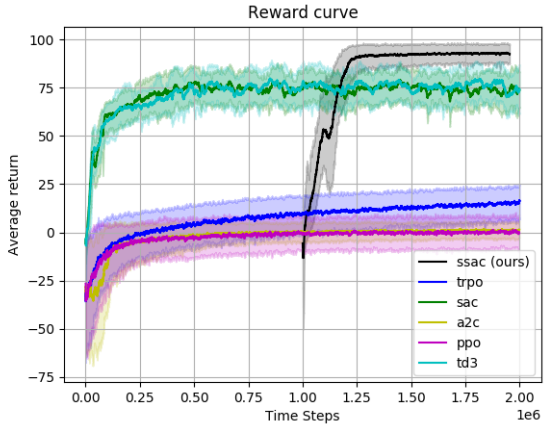


Figure 3.3: Reward curve for SSAC and the other algorithms we compare to. the solid line is the smoothed average of episode reward, averaged over four random seeds. The shaded area indicates the best and worst rewards at each epoch across the four seeds. SSAC is shown starting later to account for the time training the gating function alone.

Algorithm (implementation)	Mean Reward ± Standard Deviation
SSAC (Ours)	**92.12 ± 2.35**
SAC	73.01 ± 11.41
PPO	0.43 ± 8.89
TD3	78.67 ± 61.85
TRPO	17.63 ± 3.39
A2C	2.57 ± 3.63

tableRewards after training for across learning algorithms. This table shows results after 2 million environment interactions

As we can see, for this environment, with the number of steps we have allotted, our

approach outperforms the algorithms we compared to, with TD3 making it the closest to our performance. This is a necessarily flawed comparison. These algorithms are meant to be general purpose, so it is unfair to compare them to something designed for a particular problem. But that is part of the point we are making, that adding just a small amount of domain knowledge can improve performance dramatically.

3.4.2 Analyzing performance

To qualitatively evaluate the performance of our learned agent we examine the behavior during individual episodes. SSAC gives us a deterministic controller (we can set ϵ_t from 3.2.3 to zero). We chose the initial condition $s_0 = (-\pi/2, 0, 0, 0)$ and record a rollout. The actions are displayed in figure 3.4, and the positions in 3.5.

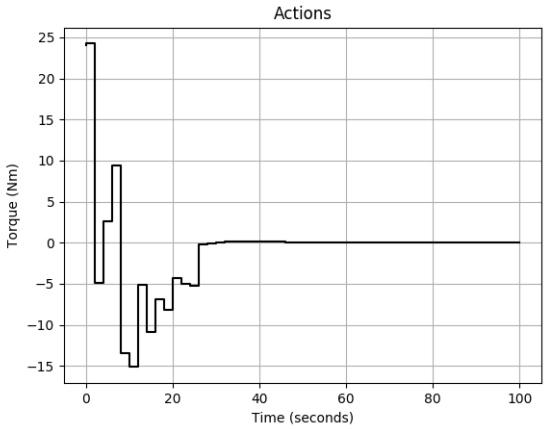


Figure 3.4: Torque exerted during the sampled episode

We have also found that despite achieving relatively high rewards, the other algorithms we compare to often fail to meet the balance criteria (11). We often see solutions where the first link is constantly rotating, with the second link constantly vertical. To demonstrate this, as well as to demonstrate our algorithms robustness, we run roll outs with the trained agents across a grid of initial conditions, recording if the trajectory

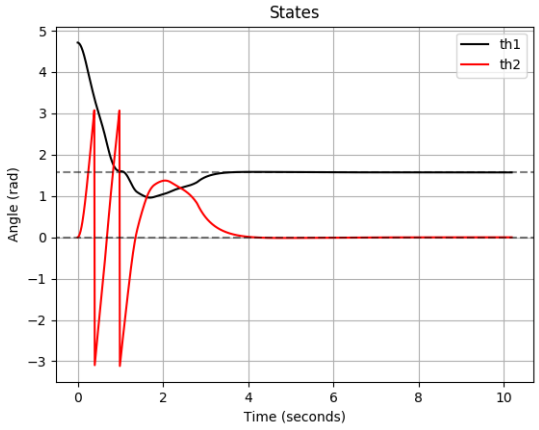


Figure 3.5: Observations during the sampled episode

satisfies (11) or not. We compare our method with TD3, which was the best performing model free method we could find on this task. Figure 3.6 show the results, **when these initial conditions were run for SSAC, it satisfied (11) for every initial condition**.

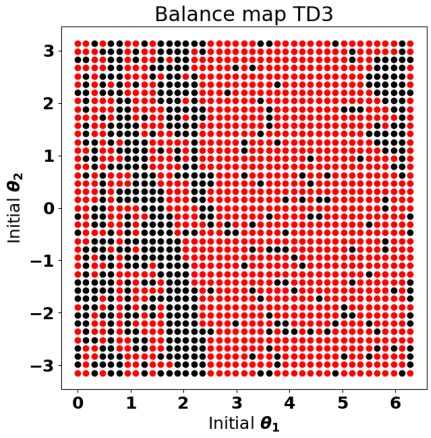


Figure 3.6: Balance map for TD3, X and Y indicate the initial position for the trial, a black dot indicates that the trial started from that point satisfies equation (3.3), and red indicates the converse. **when these initial conditions were run for SSAC, it satisfied (11) for every initial condition**

3.5 CONCLUSIONS

We have presented a novel control design methodology that allows engineers to leverage their domain knowledge, while also reaping many of the benefits from recent advances in deep reinforcement learning. In our case study we constructed a policy to swing-up and balance an acrobot while only needing to manually design a linear controller for the balancing task. We believe this method of control will be straightforward to apply to the double or triple cartpole problems, which to our knowledge no model free algorithm is reported as solving. We also think that this general methodology can be extended to more complex problems, such as legged locomotion. In that case the linear controller here could be a nominal walking controller obtained via trajectory optimization, and the learned controller could be a recovery controller to return to the basin of attraction of this nominal controller.

APPENDIX

Hyperparameters

Hyperparameter	Value
Episode length (N_e)	50
Exploration steps	5e4
Initial policy/value learning rate	1e-3
Steps per update	500
Replay batch size	4096
Policy/value minibatch size	128
Initial gate learning rate	1e-5
Win criteria lookback (b)	10
Win criteria threshold (ϵ_{thr})	.1
Discount (γ)	.95
Policy/value updates per epoch	4
Gate update frequency	5e4
Needle lookup probability p_n	.5
Entropy coefficient (α)	.05
Polyak constant (c_{py})	.995
Hysteresis on threshold	.9
Hysteresis off threshold	.5

Network Architecture

The policy, value, and Q networks are all made of four fully connected layers, with 32 hidden nodes and Relu activations. The gate network is composed of two hidden layers with 32 nodes each, also with Relu activations, the last output is fed through a sigmoid

to keep the result between 0-1.

Chapter 4

Mesh Dimensions

The availability of computation as a resource has been growing exponentially since at least the 1970s, and there is every indication that this resource will continue to become cheaper and more available well into the conceivable future. Researchers have been able to leverage the large amounts compute available to better control robotic systems, and advances in computational capacity and algorithmic development continue to open up new domains. One promising manifestation of this is model-free reinforcement learning, a branch of machine learning which allows an agent to interact with its environment and autonomously learn how to maximize some measure of reward. The promise here is to allow researchers to solve problems for systems that are hard to model, and/or that the user doesn't know how to solve themselves. Recent examples in the context of robotics include controlling a 47 DOF humanoid to navigate a variety of obstacles [**?**], dexterously manipulating objects with a 24 DOF robotic hand [**?**], and allowing a physical quadruped robot to run [**?**], and recover from falls [**?**].

In this paper we study legged locomotion. This class of problems is notoriously difficult, and as a result reinforcement learning is a popular tool to throw at it. We would argue that hand-designed, model based control still represents the state of the art

(a la Boston Dynamics), but RL has been a fruitful approach. There are examples of learned policies outperforming hand designed ones [**?**], and there is good reason to believe these learning methods will continue their current trajectory of increasing performance gains and ease of use. But these algorithms have a serious draw back in that they are mostly black boxes. It is an open challenge to figure out what exactly it is that your RL agent has learned. If all you know is that one of your agents achieved very high reward, it is not clear how to verify that this system is safe and sensible in all the regions of state space it will visit during its life. Nor can we necessarily say anything about the stability or robustness properties of the system. Recent work [**?**] has used so-called mesh-based tools to examine precisely these questions.

However, utility of any mesh-based tool to accurately discretize a state-space is limited, due to the curse of dimensionality. In practice, these methods are only able to work on relatively high dimensional systems if the reachable space grows at a rate that is much smaller than the exponential growth of the full state space the system within which it is embedded. To expand these methods to higher dimensional systems. We will need to find ways to keep the volume of visited states from expanding commensurately. One way to quantify this rate of growth is by using one of the several notions of "fractional dimensions" from fractal geometry.

In this work, we discuss an efficient meshing algorithm, which we call box meshing. We show that this approach makes calculating the so called mesh dimension feasible in the context of reinforcement learning. We also propose using other notions of fractional dimension from the literature as a proxy for the property we care about. We then show that reinforcement learning agents can be trained to shrink these measures by post processing their reward function. We present the results of this training, and finally present some brief analysis of the resulting structure for select policies.

4.1 Meshing & Fractional Dimensions

Let's say we have a continuous set S that we want to approximate by selecting a discrete set M composed of regions in S. We will call this set M a mesh of our space. Figure **??**(a) shows some examples of this: a line is broken into segments, a square into grid spaces, and so on. The question is: as we increase the resolution of these regions, how many more regions N do we need? Again, figure **??**(a) shows us some very simple examples. For a D dimensional system, if we go from regions of size d to d/k, then we would expect the number of mesh points to scale as $N \propto k^D$. But not all systems will scale like this, as **??**(b) and **??**(c) illustrate. Figure **??**(b) is an example of a curve embedded in a two dimensional space. The question of how many mesh points are required must be answered empirically. Going backwards, we can use this relationship to assign a notion of "dimension" to the curve.

$$D_f = -\lim_{k \to 0} \frac{\log N(k)}{\log k} \tag{4.1}$$

What we are talking about is called the Minkowski–Bouligand dimension, also known as the box counting dimension. This dimension need not be an integer, hence the name "fractional dimension". As a practical matter, we use the slope of the log-log plot of mesh sizes over d to calculate this, rather than taking a limit. This is one of many measures of "fractional dimension" that that emerged from the study of fractal geometry. Although these measures were invented to study fractals, they can still be usefully applied to non-fractal sets.

In [**?**], Saglam and Byl introduced a technique that is able to simultaneously build a non-uniform mesh of a reachable state space while developing robust policies for a bipedal walker on rough terrain. Having a discrete mesh allows for value iteration over several

candidate controllers, which found a robust control policy. In addition this mesh allows for the construction of a state transition matrix, which was used to calculate the mean first passage time [**?**], a metric that quantifies the expected number of steps a meta-stable system can take before falling.

Since its introduction, meshing in this fashion has been used for designing walking controllers robust to push disturbances [**?**], to design agile gaits for a quadruped [**?**], and to analyze hybrid zero dynamics (HZD) controllers [**?**]. There has also been recent work to use these tools to analyze policies trained by deep reinforcement learning [**?**]. A long term goal and motivation for this work is to take a high performance controller obtained via reinforcement learning, and extract from it a mesh-based policy that is both explainable and amenable to analysis.

4.1.1 Box Meshing

Our primary improvement to the prior work on meshing is to introduce something we call box meshing. Prior, a new mesh point could take any value in the state space. To determine if a new state is already in the mesh, we would compute a distance metric to every point in the mesh, and check if the minimum was below our threshold. Thus, building the mesh was an $O(n)^2$ algorithm. By contrast, in box meshing we apriori divide the space uniformly into boxes with side length d. We identify any state s with a key obtained by: key = round$(\frac{s}{d})d$, where round performs an element-wise rounding to the nearest integer. We can then use these keys to store mesh points in a hash table. Using this data structure, we can still store the mesh compactly, only keeping the points we come across. However, insertion and search are now $O(1)$, and so building the mesh is $O(n)$. This is very similar to non-hierarchical bucket methods, which are well studied spatial data structure [**?**], although we are using them for data compression here. In

the prior meshing work, this sort of speedup would be minor, the run-time is dominated by the simulator or robot. However, this speedup does open some new possibilities: most poignantly, it makes calculating the mesh dimension during reinforcement learning plausible.

4.1.2 Algorithmic Box Mesh Dimension

The "mesh dimension" is the quantity extracted from the slope of the log log plot of mesh sizes vs d values. For this paper, it is assumed that the mesh algorithm being used for this calculation is the box mesh. Automatically computing the mesh dimension of a data set generated from learning agent with speed, accuracy, and robustness is very challenging. A single trajectory provides only a small amount of data, which adds significant noise to the mesh sizes. Agents might do things like fall over and generate extremely short trajectories, or learn a trajectory that "stands in place", which can lead to numerical errors. Finally, every decision is a trade-off between accuracy and speed. Model-free RL is predicated on having a huge number of rollouts to learn from, and we would like for any mesh-dimension quantification algorithm to be fast enough so as to not dominate the total learning time. With these factors in mind, we introduce two box mesh dimensions. The lower mesh dimension does the linear fit, but intentionally errs on the side of including flat parts of the graph, and therefore tends to underestimate the true mesh dimension. We then have the upper mesh dimension, which takes the largest slope in the log log relationship, thus tending to overestimate the true mesh dimension. Neither of these measures are correct, but taken together they can bound the mesh dimension, and as we will see they can be useful on their own.

Algorithm 3 Create Box Mesh, see section 4.1.1

Input: State set S, box size d.
Output: Mesh size m.
Initialize: Empty hash table M.
for $s \in S$ **do**
  $\bar{s}$ = Normalize(s)
  key = round($\bar{s}$ / d)d
  if $s \in M$ **then**
    M[key]++
  else
    M[key]=1
  end
end
Return: M

Algorithm 4 Compute Box Mesh Dimension, see section 4.1.2

Input: State set S.
Output: Mesh M.
Hyperparameters: scaling factor f, initial box size d_0.
Initialize: Empty list of mesh sizes H, empty list of d values D.
m = Size(CreateBoxMesh(S, d_0))
d = d_0
Append m to H, append d to D.
while $m <$ *size(S)* **do**
  d = d/f
  m = Size(CreateBoxMesh(S, d))
  Prepend m to H, prepend d to D.
end
while $m \neq 1$ **do**
  d = d*f
  m = Size(CreateBoxMesh(S,d))
  Append m to H, append d to D.
end
X = log d
Y = -log m
Lower Mesh Dim: fit Y = gX + b, **Return:** g
Upper Mesh Dim: w = greatest slope in Y over X **Return:** w

4.2 Reinforcement Learning

The goal of reinforcement learning is to train an agent acting in an environment to maximize some reward function. At every timestep $t \in \mathbb{Z}$, the agent receives the current state $s_t \in R^n$, uses that to compute an action $a_t \in \mathbb{R}^b$, and the receives the next state s_{t+1}, which is used to calculate a reward $r : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. The objective is to find a policy $\pi_\theta : \mathbb{R}^n \to \mathbb{R}^m$

$$\arg\max_{\theta} \mathop{\mathbb{E}}_{\eta} \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \right] \tag{4.2}$$

Where $\theta \in \mathbb{R}^d$ is a set that parameterizes the policy, and η is a parameter representing the randomness in the environment. This includes the random initial conditions for episodes.

4.2.1 Post Processing Rewards

In order to influence the dimensionality of the resulting policies, we introduce various postprocessors, which act on the reward signals before passing them to the agent. These obviously modify the problem: in some sense the postprocessed environment is a completely different problem from the original. However our meta-goal is to train agents that achieve reasonable rewards in the base environment, while simultaneously exhibiting reduced dimensionality we are looking for. These postprocessors take the form:

$$R_*(\mathbf{s}, \mathbf{a}) = \frac{1}{D_*(\mathbf{s})} \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \tag{4.3}$$

Where $\mathbf{s}, \mathbf{a}$ are understood to be an entire trajectory of state action pairs, and D_* is some measure of fractional dimension. Some measures of dimensionality can be inserted here directly (See 4.5.1). However the mesh dimensions computed by algorithm 4 require

a little more care. We must first define a clipped dimension:

$$D_*^c = \text{clip}[D_*(\mathbf{s}_{t>Tr}), 1, D_t/2)] \tag{4.4}$$

where D_t is the topological dimension, equal to the number of states in the system. Tr is a fixed timestep chosen to exclude the initial transients resulting from a system moving from rest to into a quasi-cyclical "gait". In this paper we set $Tr = 200$ for all experiments. For comparison, the nominal episode length is 1000. The clipping is to ensure that the pathological trajectories that and RL agent sometimes generates don't interfere with the training. It will also clip trajectories that terminate early, to prevent agents learning to fall over immediately to "game the system". Half of the topological dimension proved to be a decent upper bound for the worst case dimensionality of each system in practive. The **mesh dimension postprocessors** use the clipped dimension. Finally, when $D_* = 1$ is used, we call the result is the **identity post processor**, since in this case the total reward is completely unchanged.

4.2.2 Environments

We examine a subset of the popular OpenAI Mujoco locomotion environments introduced in [**?**]. In particular, we evaluate our work on HalfCheetah-v2, Hopper-v2, and Walker2d-v2. These environments were chosen because they have a relatively high dimensionality (11-17 DOF), yet we believe can be made feasible for meshing based approaches. The state space consists of all joint / base positions and velocities, with the x (the "forward") position being held out, because we want a policy that is invariant along that dimension.

4.2.3 Augmented Random Search

In [?] Mania et al introduce Augmented Random Search (ARS) which proved to be efficient and effective on the locomotion tasks. Rather than a neural network, ARS used static linear policies, and compared to most modern reinforcement learning, the algorithm is very straightforward. The algorithm operates directly on the policy weights, each epoch the agent perturbs it's current policy N times, and collects 2N rollouts using the modified policies. The rewards from these rollouts are used to update the current policy weights, repeat until completion. The algorithm is known to have high variance; not all seeds obtain high rewards, but to our knowledge their work in many ways represents the state of the art on these benchmarks. Mania et al introduce several small modifications of the algorithm in their paper, our implementation corresponds to the version they call ARS-V2t.

4.2.4 Training

To keep things simple, we wanted to find one set of hyper parameters for all environments and post processors. These parameters were chosen by hand, with the parameters reported in Table 9 from [?] as a starting point. We tuned until our unprocessed learning achieved satisfactory results across all tasks. Again, ARS is known to have high variance between random seeds, and some seeds never learn to gather a large reward. The parameters we found are able to consistently solve the cheetah and walker; for the hopper, the algorithm learns a policy with high reward around half the time. This seems consistent with the performance reported in [?]. We train each postprocessor on 10 random seeds, the evaluation metrics are averages over 5 rollouts from each seed, and for the dimension metrics we use extended episodes of length 10,000 to get a more accurate measurement. The reported returns, and the training, both use the normal 1,000 step episodes. We

found that the mesh postprocessors were getting very poor performance when trained from a random policy. However, we found that we saw good results when these trials were initialized with a working policy. Therefore we trained agents for 750 epochs without post processing, and used that to initialize the mesh dimension policies. The mesh policies were then trained for an additional 250 epochs and the results reported.

4.3 Results

4.3.1 Mesh Dimension Postprocessors

Environment	Postprocessor	Lower Mesh Dim.	Upper Mesh Dim.	Return
HalfCheetah-v2	Identity	2.31 ± 0.71	7.34 ± 1.56	5469 ± 823
	Lower Mesh Dim	**0.66 ± 0.51**	**2.55 ± 1.52**	4962 ± 598
	Upper Mesh Dim.	**1.06 ± 1.13**	**2.83 ± 1.27**	4432 ± 539
Hopper-v2	Madogram*	1.62 ± .27	4.68 ± 0.82	3461 ± 119
	Lower Mesh Dim.	1.13 ± .02	3.54 ± 0.96	2941 ± 538
	Upper Mesh Dim.	1.27 ± .50	2.98 ± 1.48	3020 ± 337
Walker2d-v2 (walking seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Lower Mesh Dim.	1.21 ± 0.06	4.09 ± 1.03	3339 ± 887
	Upper Mesh Dim.	1.89 ± 0.42	3.10 ± 0.93	3359 ± 903
Walker2d-v2 (all seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Lower Mesh Dim.	1.04 ± 0.53	4.45 ± 1.19	3034 ± 1086
	Upper Mesh Dim.	1.48 ± 0.67	2.27 ± 0.95	2556 ± 1378

Table 4.1: Mesh dimensions and returns for trajectories after training. See 4.2.4 for details

For all environments the mesh post processors had a significant impact on the mesh dimensions. It's important to remember here, the dimensions reported represent lower and upper bounds for the actual mesh dimensions. There was also a corresponding and significant decrease in the unprocessed rewards. However with our meta goal of training agents that have acceptable reward but which are more amenable to meshing, this is a more than acceptable trade. In the case of walker, several seeds (4 for the upper dim., 3

for the lower mesh dim.), "forget how to walk", and learn a policy that stands in place. This certainly has a low dimensionality, but is not very useful, to be complete we include statistics from the seeds that learned a gait, and for all 10 seeds, including the standing policies.

4.4 Analysis

We now examine the learned behavior for one of the more notable policies. By far the most dramatic effect from the tables above was the mesh dimension postprocessors on the cheetah. Both measures of dimension shrunk by 2-4 times. Figure 7 presents data for this case.

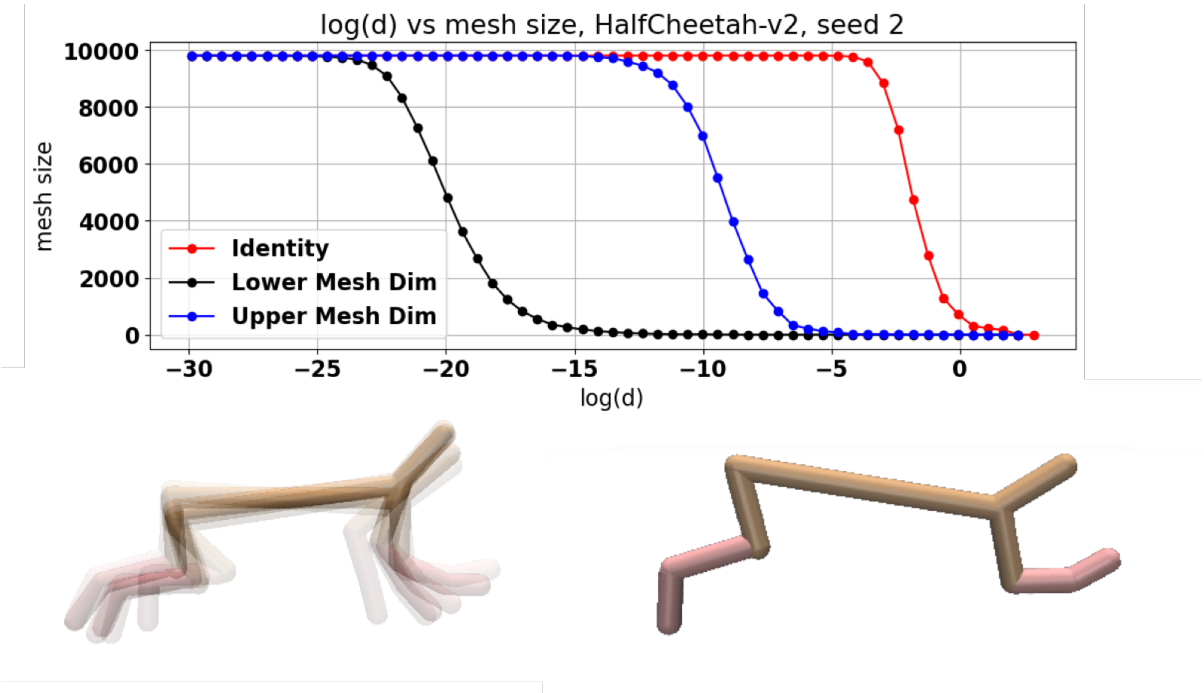


Figure 4.1: Top: mesh sizes vs log of the box size for the cheetah environment. Lower Left: Every five frames overlaid for the an identity policy on the cheetah. Lower Right: Every five frames of cheetah after the lower mesh dimension training.

Toward more intuitively understanding this data, a few comments are worth making,

first. We have discussed the mesh dimension rather abstractly so far. In visualizing what this really means, imagine two different gait cycles. In one case, there is a general pattern to the motion, but it wanders in a noisy-looking way, like a "signature" that does not quite match up, cycle after cycle. As motions become closer to being exact limit cycles, there is a more clear pattern of repetition, exactly analogous to re-tracing the same path, again and again, within the state space. Such a more tightly-structured limit cycle nature in turn results in a significantly lower-dimensional set of states being visited.

We can see from the mesh size curves in Figure 7 that there is an overwhelming difference in the mesh sizes between the lower mesh dimension post processor and the other two. To put this in perspective, before the extra 250 epochs of training, if given a box size of .01, the agent would need a unique mesh point for every single state in the 10,000 state trajectory. After the additional training, the agent can represent all 10,000 points with just 5 mesh points! In this case it appears both agents learned a quasi periodic gait with period 5. In figure 4.1 we present an overlay of the agents rendered every 5 steps. The results show us that the mesh agent has learned an extremely tight limit cycle. It's a bit of a strange limit cycle, being only 5 timesteps long, but nonetheless we think this is interesting and surprising behavior.

The behavior displayed in figure 4.1 is clearly something that can only happen in a noiseless simulation, so we also measured the mesh dimensions of our policies when subjected to noise during rollouts using the noise values from the robustness experiments (see table 4.3)to inform parameter values. The difference in fractal dimension is less pronounced than the no noise case but is still a clear improvement. Furthermore we expect that if we were to add noise at training time, that the learning may be able to find ways to lower the mesh dimension that is more robust to noise.

We also performed analysis on the robustness properties of the resulting polices. We tested the failure rate for agents in the presence of noise and disturbances, figure 4.3 shows

Environment	Postprocessor	Lower Mesh Dim.	Upper Mesh Dim.	Return
HalfCheetah-v2	Identity	2.38 ± 0.43	6.65 ± 1.90	5404 ± 1015
	Lower Mesh Dim	1.51 ± 0.13	3.03 ± 1.09	4952 ± 572
	Upper Mesh Dim.	1.76 ± 0.53	3.54 ± 1.27	4222 ± 803
Hopper-v2	Madogram*	1.63 ± .14	4.49 ± 0.75	3438 ± 185
	Lower Mesh Dim.	1.67 ± .22	3.71 ± 0.89	2943 ± 535
	Upper Mesh Dim.	1.64 ± .16	3.01 ± 1.36	3019 ± 337
Walker2d-v2 (walking seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Lower Mesh Dim.	1.83 ± 0.34	2.73 ± 0.75	3511 ± 872
	Upper Mesh Dim.	1.60 ± 0.33	4.01 ± 1.18	3384 ± 903
Walker2d-v2 (all seeds)**	Identity	2.10 ± 0.34	4.42 ± 1.00	3743 ± 1034
	Lower Mesh Dim.	1.68 ± 0.70	4.19 ± 1.25	3048 ± 1071
	Upper Mesh Dim.	1.48 ± 0.38	2.98 ± 0.86	2558 ± 1373

Table 4.2: Mesh dimensions and returns for trajectories subject to zero mean Guassian noise. Standard deviation of .001 and .01 was added to all actions and observations respectively. See 4.2.4 for details
Because ARS with our chosen hyper parameters does not consistently produce 10 seeds that perform well on the hopper, we instead use madodiv (see the 4.5.1) for the seed policies.
** See 4.3.1

the results. It's worth noting at this point that in practice the lower mesh dimension seems to work better in practice than the upper one. We found that when computing the mesh dimension by hand (by hand fitting a line to a set of carefully obtained mesh size data) that the hand picked value was generally much closer to the lower mesh dimension, at least for the three systems we studied. Training with the lower mesh dimension also resulted in agents that were more robust and achieved higher reward compared to the upper dimension.

We tested three different cases, adding zero mean Gaussian noise to the actions and observations, and adding a force disturbance to the center of mass of the agents during their rollouts. For the push disturbances we have two parameters, the rate of disturbances, and the mangitude of the force applied. At every step we sample uniformly from [0,1], if the result is less than the rate parameter, then a force is applied at that timestep. The force is applied at a random angle in the xz plane, with the fixed magnitude from

	Identity	Lower mesh dim.	Action std
Cheetah	0.24	0.05	.05
Hopper	0.19	0.10	.05
Walker	0.28	0.03	.15
	Identity	Lower mesh dim.	Observation std
Cheetah	0.20	0.02	.005
Hopper	0.20	0.25	.02
Walker	0.18	0.10	.03
	Identity	Lower mesh dim.	Magnitude, Rate
Cheetah	0.21	0.03	3, .2
Hopper	0.17	0.10	1, .2
Walker	0.20	0.00	1, .2

Table 4.3: Failure rates for agents under various noise and push disturbances

the magnitude parameter. For each type of disturbance we did a grid search over the parameters and report the parameter for which the identity post processor failed in around 20 percent of cases. Failure is defined as an early termination of an episode.

4.5 Conclusion

In this work, we introduced a technique to influence the fractional dimension of the closed-loop dynamics of a system through the use of novel, dimensionality-based modifications to the cost functions for reinforcement learning policies. We demonstrate this technique on several benchmark tasks, and we briefly analyze a resulting policy to verify the outcome, demonstrating a much smaller mesh dimension without a large loss in reward or function.

Hyper Parameters

ARS: For all environments $\alpha = .02$, $\sigma = .025$, $N = 50$, $b = 20$.

MeshDim: f = 1.5, d_0 = 1e-2

4.5.1 Variation Estimators

As discussed, computing the mesh dimension automatically is fraught with peril, in many practical scenarios. But there are also many other, different metrics one might consider, to give various approximations to the fractional dimension we seek to estimate. Gneiting et al. [**?**] compare a number of these estimators, and submit that the variation estimator [**?**] offers a very good trade off between speed and robustness. To obtain this estimator, first define the power variation of order p as:

$$P_p(X, l) = \frac{1}{(2n - l)} \sum_{i=l}^{n} |X_i - X_{i-l}|^p \tag{4.5}$$

Then, we define the variation estimator of order p as:

$$Dv_p(X) = 2 - \frac{\log P_p(X, 2) - \log P_p(X, 1)}{p \log 2} \tag{4.6}$$

The **madogram** estimator is the special case of (4.6) where p = 1, and the **variogram** is where p = 2.

4.5.2 Variational Postprocessors

It seems that the variational postprocessors had a modest effect the variational dimension, but that does not seem to correlate to a smaller mesh dimension, despite what our preliminary tests had led us to believe. The hopper and walker did have remarkable consistency in the variation dimensions they found; possibly this could be used to lower the variance in ARS. The fact that the variogram and madogram also got higher performance on the hopper task could support this claim. However without running many more trials and hyper parameter sweeps, that's not a claim that can be substantiated.

Environment	Postprocessor	Variogram	Madogram	Lower Mesh Dim.	Return
HalfCheetah-v2	Identity	1.71 ± .03	1.42 ± .05	2.36 ± .61	5545 ± 593
	Variogram	1.68 ± .01	1.36 ± .02	2.06 ± .60	5136 ± 851
	Madogram	1.65 ± .02	1.31 ± .04	2.09 ± .64	5234 ± 950
Hopper-v2	Identity*	1.61 ± .14	1.22 ± .28	1.03* ± .71	2063 ± 1052
	Variogram	**1.51 ± .02**	**1.03 ± .04**	1.58 ± .54	3299 ± 711
	Madogram	**1.51 ± .002**	**1.02 ± .004**	1.57 ± .36	3449 ± 146
Walker2d-v2	Identity	1.68 ± .35	1.36 ± .71	2.14 ± .29	3742 ± 1038
	Variogram	**1.54 ± .07**	**1.07 ± .01**	1.85 ± .54	3779 ± 894
	Madogram	**1.53 ± .01**	**1.06 ± .02**	1.99 ± .53	3414 ± 1025

Table 4.4: Mesh dimensions and returns for trajectories after training. See 4.2.4 for details
* This includes policies which learned to "stand still", which lowers the average mesh dimension considerably see discussion

These experiments show that 1) measures for fractional dimension can be influenced without adversely effecting the reward, and 2) that it is possible for an agent to shrink it's variogram and madogram dimensions without a large impact on its mesh dimension.

4.5.3 Mesh Dimension Examples

Figure **??** illustrates two examples of the curves used to compute the mesh dimension. Recall that to compute the mesh dimension, we choose several values for d, the box length, and for each d construct a mesh using that box size. The x axis of these plots represents the log of the box length used, the y axis represents the log of size of the mesh created. For each curve, we display the lower bound and upper bound for the dimension as computed by algorithm 2, as well as several hand fits of the data. We hope that figure **??**a makes clear what a close to ideal situation looks like, and provide intuition as to why the upper and lower mesh dimension bound the quantity we are trying to measure. Figure **??**b serves to illustrate some of the problems with making an algorithmic measure of the dimension. There is much less data to work with due to performance constraints, which causes a large amount of noise on the estimate of the mesh dimension. Indeed even

fitting this data by hand becomes a challenge and we provide two fits which can both be argued to be "correct". Which of these two represents the quantity we care about depends on the exact system being used and the purpose of the meshes we want to build with the resulting policy.

4.5.4 Implementation Details

For performance reasons, the mesh dimension algorithm does not actually create meshes until the mesh size equals the total data size, but rather until the mesh size is 4/5 the total data size. Figure 8a shows a typical mesh curve, and we can see the long tail of values with mesh sizes close to the maximum value. Not much useful information is gained from this and it is wasting time, so we stop early. We do not place the same limitation on the lower size of the mesh, since typically the mesh size hits one much more rapidly, again figure 8a illustrates this. In addition implement a minimum size for d, set to 1e-9 in this work to avoid numerical errors.

The normalization done during box creation uses a running mean and standard deviation of all states seen so far during training. These stats are saved and used for evaluation as well, we found that the upper mesh dimension is very sensitive to the normalization used, but that the other metrics where not.

4.6 INTRODUCTION

Legged robots have clear potential to play an important role in our society in the near future. Examples include contact-free delivery during a pandemic, emergency work after an environmental disaster, or as a logistical tool for the military. Legged robots simply expand the reach of robotics when compared to wheeled systems. However, compared to

wheeled systems, designing control policies for legged systems is a much more complex task, especially in the presence of disturbances, noise, and unstructured environments.

The increasing availability of massive quantities of computation has led to a resurgence of reinforcement learning (RL) in recent years. RL provides a promising approach for complex, under-actuated, hybrid control problems, such as those involved in designing control for legged locomotion. Recent examples in the context of robotics include controlling a 47 DOF humanoid to navigate a variety of obstacles [**?**], dexterously manipulating objects with a 24 DOF robotic hand [**?**], and allowing a physical quadruped robot to run [**?**], and recover from falls [**?**].

Despite the obvious promise of RL approaches, several problems need to be resolved before these systems are ready for real world applications. One of the biggest problems is that the resulting policy is typically a complete black box, there are no good ways to make theoretical, or even empirical guarantees about the resulting policies. Prior work has used so called mesh based techniques to this end [**?**]. Broadly, these techniques take a continuous system and approximate it with a discrete set of states. This allows us to model the system as a Markov chain, these systems are arguably easier to reason about, and it opens up a new box of tools we can bring to bear on the problem. For example we can use value iteration to switch between several controllers to improve the robustness [**?**] or agility [**?**] of the system. We can also perform eigen-analysis on the Markov chain's transition matrix, which provides us insights on the stability of the system [**?**]. These techniques could both be used for policy refinement, and/or for verification and analysis of existing policies.

However, these methods suffer from the "curse of dimensionality", because the number of possible states in our mesh grows exponentially with the degrees of freedom in our system. That is, if we make a change to the volume of continuous space represented by each discrete state, the number of states in our new mesh will grow exponentially with

respect to the change in discrete state size. However for virtually all plausible walking controllers, the reachable state space is a small fraction of the total state space. Although the scaling for meshes of the reachable state space also scale exponentially as we increase the mesh resolution, the rate of scaling is typically much smaller. The scaling factor for the reachable mesh can be seen as a **fractal dimension**, which is elaborated upon in section 4.7.1.

In previous work [**?**], we introduced a modified reward function for on-policy reinforcement learning algorithms. The reward explicitly encourages policies which induce trajectories which have a smaller fractal dimension. It's worth noting that although each individual trajectory was encouraged to have a smaller fractal dimension, this does not obviously extend to properties of the entire reachable state space for the system, which is what was used for the previous mesh based analysis of RL policies.

In this work we take the next step and construct reachable state space meshes of agents trained with and without our modified reward. Our primary contribution is showing that these modified policies result in significantly smaller reachable meshes for a given box size, and in smaller fractal dimensions for the reachable state space. We then use the modified policies to construct a much finer mesh than would be possible otherwise. We use this mesh to compute a quantity called the mean first passage time (MFPT), and validate the obtained MFPT with Monte Carlo trials. Finally we use our mesh to produce interesting visualizations of failure states, which motivates future work.

4.7 BACKGROUND

In this section we introduce fractional dimensions, meshing, reinforcement learning, and our test environment. The environment is a hopping robot, coupled with a specific reward function. Reinforcement learning is used to train a control policy for this system

which attempts to maximize the given reward function. In previous work we used a fractional dimension to modify the reward function, this modified reward results in policies that can be meshed significantly more efficiently.

4.7.1 Meshing and Fractal Dimensions

Let's say we have a continuous set S that we want to approximate by selecting a discrete set M composed of regions in S. We will call this set M a mesh of our space. Figure **??**(a) shows some examples of this: a line is broken into segments, a square into grid spaces, and so on. The question is: as we increase the resolution of these regions, how many more regions N do we need? Again, Figure **??**(a) shows us some very simple examples. For a D dimensional system, if we go from regions of size d to d/k, then we would expect the number of mesh points to scale as $N \propto k^D$. But not all systems will scale like this, as Figure **??**(b) illustrates. Figure **??**(b) is an example of a curve embedded in a two dimensional space. The question of how many mesh points are required must be answered empirically. Going backwards, we can use this relationship to assign a notion of "dimension" to the curve.

$$D_f = -\lim_{k \to 0} \frac{\log N(k)}{\log k}. \tag{4.7}$$

This quantity is known as the Minkowski–Bouligand dimension, also called the box counting dimension. This dimension need not be an integer, hence the name "fractional" or "fractal" dimension. This is one of many measures of fractional dimensionality that emerged from the study of fractal geometry. Although these measures were invented to study fractals, they can still be usefully applied to non-fractal sets. For non fractal sets, we use the slope of the log-log relation of mesh sizes to d to compute the dimension, rather than taking a limit.

4.7.2 Box Meshing

In this work, we identify any state s with a key obtained by:

$$s_k = \frac{s - \mu_s}{\sigma_s}$$
$$\text{key} = \text{round}(\frac{s_k}{d_{thr}})d_{thr}. \tag{4.8}$$

where μ_s and σ_s are the mean and standard deviation of all the states seen by the policy of interest during training. The round function here performs an element-wise rounding to the nearest integer. We can then use these keys to store mesh points in a hash table. Using this data structure, we can store the mesh compactly, only keeping the points we come across, and lookups are done in constant time. The parameter d_{thr} is called the **box size**. Geometrically we can think of this operation as dividing the state space into a uniform grid of hypercubes, each with a side length of d_{thr}

4.7.3 Reinforcement Learning

The goal of reinforcement learning is to train an agent acting in an environment to maximize some reward function. At every timestep $t \in \mathbb{Z}$, the agent receives the current state $s_t \in R^n$, uses that to compute an action $a_t \in \mathbb{R}^b$, and receives the next state s_{t+1}, which is used to calculate a reward $r : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. The objective is to find a policy $\pi_\theta : \mathbb{R}^n \to \mathbb{R}^m$ that satisfies:

$$\arg\max_\theta \mathop{\mathbb{E}}_\eta \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \right]. \tag{4.9}$$

Where $\theta \in \mathbb{R}^d$ is a set that parameterizes the policy, and η is a parameter representing the randomness in the environment. This includes the random initial conditions for

episodes.

In [**?**], we introduced a modified reward function:

$$\arg\max_{\theta} \mathop{\mathbb{E}}_{\eta} \left[\frac{1}{D_m(s)} \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \right] \tag{4.10}$$

where D_m is the "lower mesh dimension" explained in detail in [**?**], which is an estimate of equation 4.7 dimension for our non fractal set.

4.7.4 Environment

Our model system is openAI gym's Hopper-v2 environment introduced in [**?**]. This environment is part of a popular and standardized set of benchmarking tasks for reinforcement learning algorithms. The system is a 4 link, 6 DOF hopper constrained to travel in the XZ plane, seen in Figure **??**. The observation space for the agent has 11 states, the position in the direction of motion is held out, since we seek a policy that is invariant to forward progress. The actions in this case are commanded joint torques. The reward function for this environment is simply forward velocity minus a small penalty to actions. Successful controllers in this environment must execute a dynamic hopping motion to move robot along the x axis as quickly as possible. This is clearly a toy problem, but it captures many of the challenges of legged locomotion. The system is highly nonlinear, under-actuated, and must interact with friction and ground contacts to maximize it's reward.

4.8 MESHING

In [**?**] we used meshes of individual trajectories to calculate fractional dimensions. However the previous work that has used meshing for analysis of RL policies [**?**] instead

examines meshes of the reachable state space of a system. In particular [**?**] examines the reachable state space with a fixed control policy subject to a given set of disturbances. We will now outline the process for this style of meshing.

We are interested in the set of states that our system can transition to with a fixed policy and a given set of push disturbances. We first introduce a failure state to the mesh. The failure state is assumed to be absorbing, once the robot falls it is assumed to stay that way. For our hopper, any state where the COM falls below .7m is considered to have failed, which works well in practice. This is also the failure condition of the environment during training, and therefore the agent is never trained in regions of the state space that satisfy the failure condition.

In addition to the reachable set of states, we want to construct a state to state transition map. That is, for a given initial state, we wish to know which state we transition to for every disturbance in our disturbance set. It's worth emphasizing that this map is completely deterministic.

To make this concrete, recall that we manifest our mesh as a hash table. The key for any given state is obtained by 4.8. When we insert a new key into our hash table, the value we place is a pair with a unique state ID (which is simply the number of keys in the table at the time of insertion), and an initially empty list of all mesh states which are reachable after one step from the key state. This data structure will provide both the reachable set, and the transition mapping.

For the hopper in particular, the system transitions from its initial standing position to a stable long term hopping gait. After letting the system enter its gait, we start detecting states on the Poincaré section by selecting the state corresponding to the peak of the base link's height in every ballistic phase. These states are then collected as the initial states to seed the mesh with. Throughout this paper, we seed the mesh with trajectories from 10 initial conditions.

For each snapshot, we initialize the system in the snap-shotted state. For each disturbance in our fixed disturbance set, we simulate the system forward subject to that disturbance. If the system does not fail, then the next Poincaré snapshot is captured, this state is then checked for membership in our mesh. If the new state is already in our mesh, then we simply append the new state to the list of states that the initial state can transition to. If the new state is not already in our mesh, then we expand our mesh to include the new state, and append this new state to the transition list of the initial state. If the system does fail, then we simply append the failure state to the transition list of the initial state, and no new state is added to the mesh.

For every new state added to the mesh, we repeat this process until every state has been explored. Algorithm 5 details this process in pseudo code.

Algorithm 5 createMesh

Input: Initial states S_i, Disturbance set D
Output: Mesh M.
Q $\leftarrow S_i$ (excluding the failure state)
while *Q not empty* **do**
    pop q from Q
    for $d \in D$ **do**
        Initialize system in state q
        Run system for one step subject to disturbance d
        Obtain final state x
        if $x \notin M$ **then**
            M[x] = List()
            Push x onto Q
        end
        Append x to M[q]
    end
end
Return: M

4.8.1 Stochastic Transition Matrix

The stochastic transition matrix $\mathbf{T}$ is defined as follows:

$$\mathbf{T}_{ij} = \Pr(id[n+1] = j \mid id[n] = i) \tag{4.11}$$

where $id[n]$ is the index in our mesh data structure of the state at step n. For some intuition, consider the transition matrix as the adjacency matrix for a graph. There is one row/column for every state in our mesh, for a given row i, each entry j is the probability of transitioning from state i to state j. Every row will sum to one, but the sum for each column has no such constraint. After constructing a mesh using algorithm 5, it is straightforward to create the stochastic transition matrix by iterating through every transition list in our mesh.

4.8.2 Mean First Passage Time

We wish to use our mesh based methods to quantify the stability of our system. To do this we estimate the average number of steps the agent will take before falling, subject to a given distribution of disturbances. To do this we will use the so called Mean First Passage Time (MFPT) which in this case will describe expected number of footsteps, rather than the number of timesteps to failure. First recall that our assumption is that our failure state is an absorbing state in our Markov chain approximation, and this implies that the largest eigenvalue of $\mathbf{T}$ will always be $\lambda_1 = 1$. In [**?**] Byl showed that when the second largest eigenvalue λ_2 is close to unity, the MFPT is approximately equal to:

$$MFPT \approx \frac{1}{(1-\lambda_2).} \tag{4.12}$$

4.9 TRAINING

In [?] Mania et al introduce Augmented Random Search (ARS) which proved to be efficient and effective on the locomotion tasks. Rather than a neural network, ARS used static linear policies, and compared to most modern reinforcement learning, the algorithm is very straightforward. The algorithm is known to have high variance; not all seeds obtain high rewards, but to our knowledge their work in many ways represents the state of the art on the Mujoco benchmarks. Mania et al introduce several small modifications of the algorithm in their paper, our implementation corresponds to the version they call ARS-V2t, hyper parameters are provided in the appendix.

The training process is done in episodes, each episode corresponds to 1000 policy evaluations played out in the simulator. At the start of each episode, the system is initialized in a nominal initial condition offset by a small amount of noise added to each state. During each episode we fix a static policy to let the the system evolve under, we collect the observed state, the resulting action, and the resulting reward at each timestep. This information is then used to update the policy for the next episode.

We compare four different sets of agents trained in different conditions, for each training condition we use training runs across 10 different random seeds. As mentioned ARS is a very high variance algorithm, so a common practice is to run many seeds in parallel and choose the highest performing one. The standard environment has two sources of randomness which are set by the random seed. The first is a small amount of noise added to a the nominal initial condition at the beginning of each episode. The second is noise added to the policy parameters as part of the normal ARS training procedure. Using ARS in the unmodified Hopper-v2 environment will be called the **standard** training procedure. In addition to this, we have a second set of agents which are initialized with the standard training, and then trained for another 250 epochs with

the fractal reward function used in equation 4.10, these are called the **fractal agents**. Using the standard training agents as the initial policies for the fractal reward was also used in [**?**], please see that manuscript for more details.

In addition to standard training, we repeat this standard / fractal setup but with the addition of a small amount of zero mean Gaussian noise added to both the states and actions at training time. For brevity we will call these the **Standard noise** and **Fractal noise** scenarios. Hyper parameters for ARS and noise values are reported in the appendix.

4.10 Results

4.10.1 Mesh Sizes Across All Seeds

First we wish to compare the reachable state space mesh sizes obtained for these four different training regiments. For this we assume a disturbance profile consisting of 25 pushes equally spaced between -15 and 15 Newtons, applied for 0.01 seconds along the x axis at the apex of each jump. The goal for this particular exercise is to get an idea of the relative mesh sizes among the different agents across box sizes. Table 4.5 shows these results. We can see that across all box sizes, adding noise at run time decreases the mesh sizes slightly, and that adding the fractal reward training decreases the mesh size even further. The combination of adding noise and the fractal reward seems to perform best at reducing the mesh size.

4.10.2 Larger Meshes

With the general trend established, we now take the best performing seed from the noisy training for further study. We chose the seed that had the smallest mesh size from

Training	$d_{thr} = .4$	$d_{thr} = .3$	$d_{thr} = .2$	$d_{thr} = .1$
Standard	64.9	129.0	289.2	2975.2
Standard Noise	40.7	73.3	231.6	2133.3
Fractal	26.0	41.8	67.7	684.4
Fractal Noise	15.1	24.6	45.1	297.2

Table 4.5: Mesh sizes across all seeds for a disturbance profile of 25 pushes. All values are the average mesh size across 10 agents trained with different seeds.

both the standard noise and fractal noise agents.

For this next experiment, we consider a richer distribution of 100 randomly generated push disturbances. These disturbances have a magnitude drawn from a uniform distribution between 5-15 Newtons. This force is applied in the xz plane with an angle drawn from a uniform distribution between 0 and 2π. The number of forces was chosen by increasing the number of forces sampled until the mesh sizes between two random sets did not change. The magnitude of the pushes was chosen arbitrarily, in principle one can use these methods for any distribution of disturbance they expect their robot to encounter during operation.

We then construct meshes for different box sizes. For each agent we construct 10 meshes. We vary the box size between 0.1 and 0.01 for the fractal noise agent. For the standard noise agent we instead vary the box size between 0.1 and 0.02 because the mesh sizes for the standard agent were proving to be too large at the smaller box sizes. Figure **??** shows the comparison, We can see clearly that at the very least, the exponential blowup in mesh size starts at much more accurate mesh resolutions for the fractal agent.

We are also interested in the exponential scaling factor in the mesh size as the box gets smaller, which is captured by the fractal dimension discussed in section 4.7.1. As mentioned before, in previous work our modified reward signal resulted in agents with a smaller fractal dimension with respect to individual trajectories. We now ask if this

carries over to meshes of the reachable state space obtained by the procedure from algorithm 5. Table 4.6 shows the results, we can see that indeed, the fractal training does seem to reduce the mesh dimensionality for the reachable state space meshes.

Training	Trajectory Mesh Dim.	Reachable Mesh Dim.
Standard Noise	1.38	3.83
Fractal Noise	1.16	3.16

Table 4.6: Mesh dimensions for the best performing seed from the standard with noise training, and the fractal with noise training, given the same disturbance profile of 100 pushes. For reference the state space for our system has 12 dimensions.

4.10.3 Validating the Mean First Passage Time

We emphasize that the reward function for the hopper environment is simply to move forward with the highest velocity possible, no attempts were made to make the system robust to disturbances. Perhaps because of this, the mean first passage time for these systems are relatively small, on the order of 100 foot steps. For this small number of steps, we can validate the mean first passage time with Monte Carlo trials. It's worth noting that the eigen estimate of the mean first passage time is much more valuable for more robust systems. This is because this estimate becomes more accurate as the system becomes more stable, and because the cost of calculating the MFPT with Monte Carlo trials grows much more expensive for more stable systems. In previous works [**?**] it was used to quantify robustness for systems with a MFPT as high as 10^{15}.

To do this, we compare the mean first passage time as estimated by equation 4.12 to the value computed by looking at many Monte Carlo rollouts. For the rollouts we apply a random action drawn from the same distribution described above. Instead of sampling 100 pushes though we sample a new push every time we need a new disturbance. During the rollouts we still apply the push at the apex height of the ballistic phase.

Figure **??** shows the convergence of the MFPT as we expand the size of the mesh, and compares it to the mean steps to failure obtained with Monte Carlo trials. We can see that it does look like the MFPT is converging to the Monte Carlo result. Although at the largest mesh we tried, the eigen analysis gives an estimate of 110.2 steps to failure, while the Monte Carlo trials tell us that an average of 85 steps are taken before failure. It's worth noting that the distribution of failure times has a large variance with a standard deviation of 80 steps.

4.10.4 High Resolution Mesh

We now use the fractal agent and construct an even more accurate mesh. Figure **??** show the sparsity pattern for the state transition matrix for the fractal noise agent with a box size of 0.005. Recall that in the process for creating the mesh, we start with a small number initial seed states. After that every new state that we add is added in order we find them to the mesh. So if we are expanding state # 2, and there are currently 100 states in the mesh, if we transition to an unseen state, that state will be labeled # 101. So although it may seem like it is not possible for states in the top right quadrant to visit states later in the mesh, this is really an artifact of how we construct our mesh and label our points.

We note that there are a smaller set of states that make up most of the transitions. In fact we can see from Figure **??** that 20% of the states in our mesh account for about 90% of all transitions seen during the mesh construction.

One of the advantages of having a discrete set of states is that it opens up new tools and visualizations, for example we can apply Principle Component Analysis (PCA). Figure **??** shows a projection of our mesh states on the top 3 principle components. We note that these three states account for more than 97% of the variance, we also note

that our analysis reveals that states in red are where 99% of all failures occur. The visualization reveals that at least in PCA space, all the trouble states are clustered in one spot. A promising direction for future work is to introduce a policy refinement step that attempts to avoid these states. Additionally, if we were designing a real robot this may give us insights into design changes that could be made.

4.11 CONCLUSIONS

In this work, we apply previously developed tools that create discrete meshes for the reachable state space of a system. These tools were applied to policies obtained with a modified reinforcement learning reward function which was previously shown to encourage small mesh dimensions for individual trajectories not subject to any disturbances. We showed that these modified policies have a smaller average reachable mesh size across all random seeds for coarse meshes and a small number of disturbances. We then showed a clear difference in mesh sizes and mesh dimensions for the top performing seeds on a richer set of disturbances and finer mesh sizes. We also validated our use of the MFPT as a tool by comparing it to Monte Carlo trials. Finally, we constructed a high fidelity mesh at a resolution that would not have been feasible with standard ARS policies. In addition, we created visualizations with this mesh that revealed insights about the contracting nature of the policy, and which point to future applications of this approach. Taken together, these results show two things. First, it further validates the utility of the fractal dimension reward, which we have shown transfers it's desirable quality of having a more compact state space to a setting with external disturbances. These results are also a credit to the mesh based tools, because it shows that the fractal training can be used to extend the reach of these tools to higher dimensional systems or higher resolution meshes than would have otherwise been possible.

APPENDIX

Hyper Parameters

ARS: (from [**?**]) $\alpha = 0.02$, $\sigma = 0.025$, $N = 50$, $b = 20$.

MeshDim: (from [**?**]) f = 1.5, d_0 = 1e-2

Noise During Training

Zero mean Gaussian noise with std = 0.01 added to policy actions before being passed to the environment, for reference all actions from the policy are between -1 and 1. Zero mean Gaussian noise with std = 0.001 added to observations before being passed to the policy.

[b]0.45

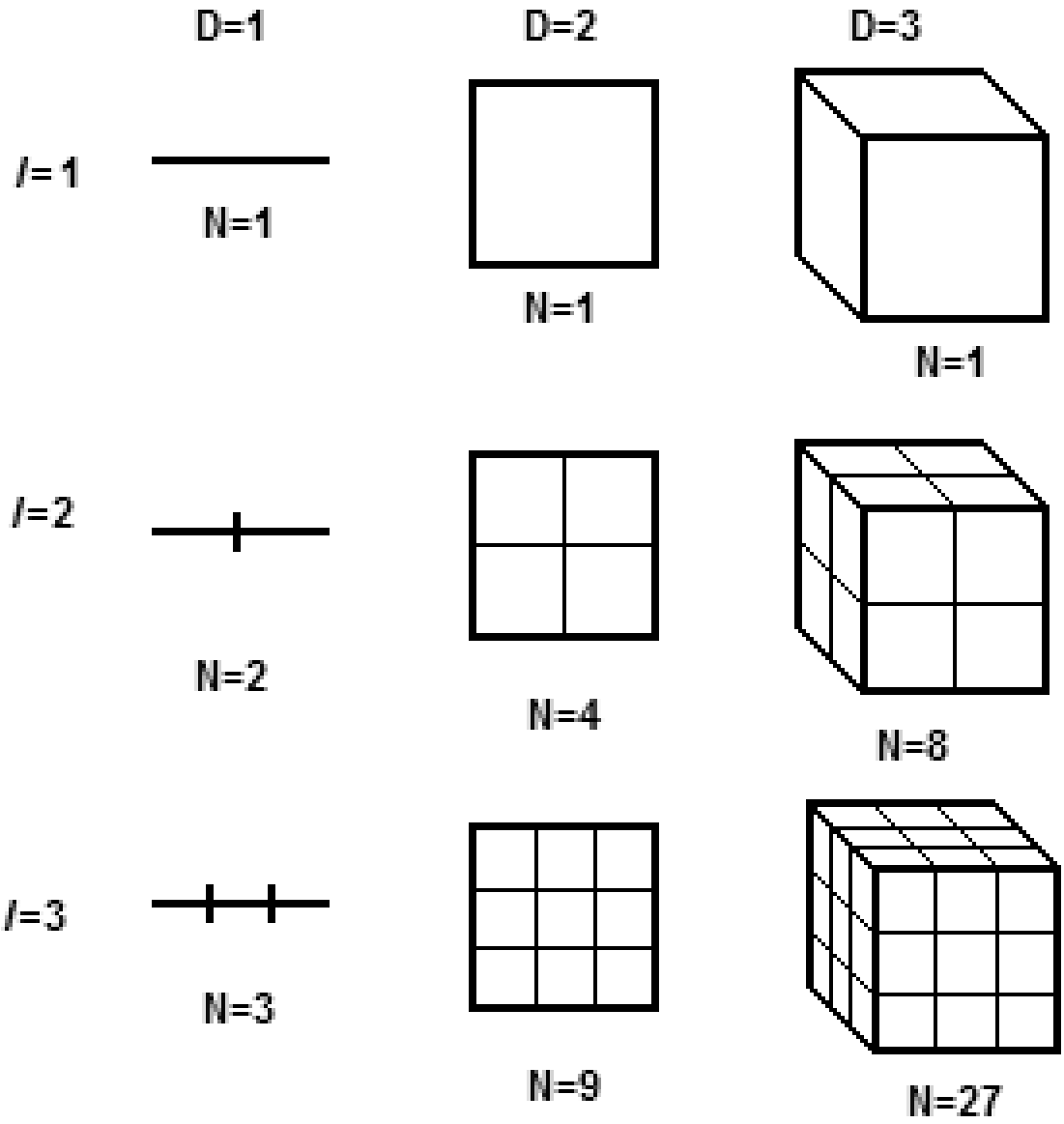
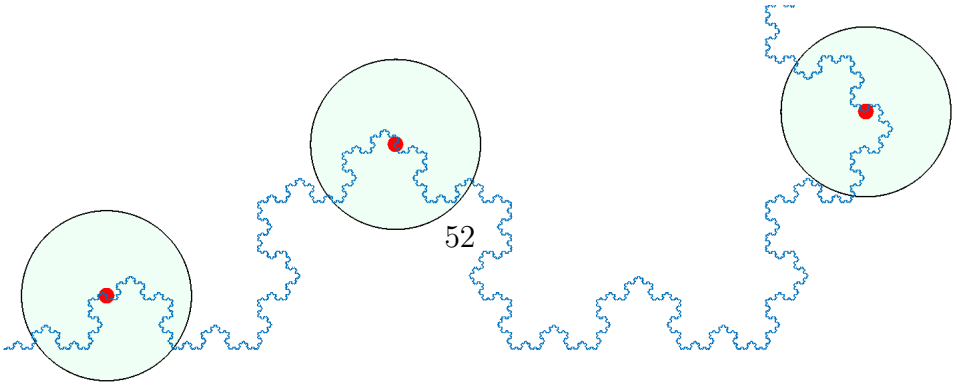


Figure 4.2: Scaling in different dimensions
[b]0.45



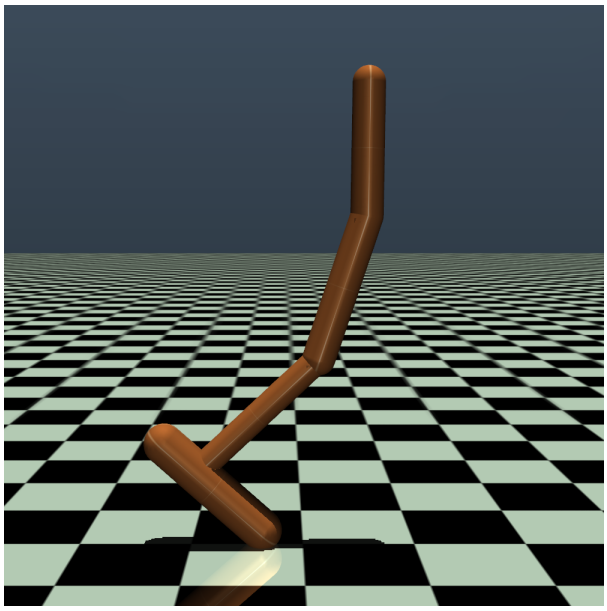


Figure 4.5: A render of the hopper system studied in this work.

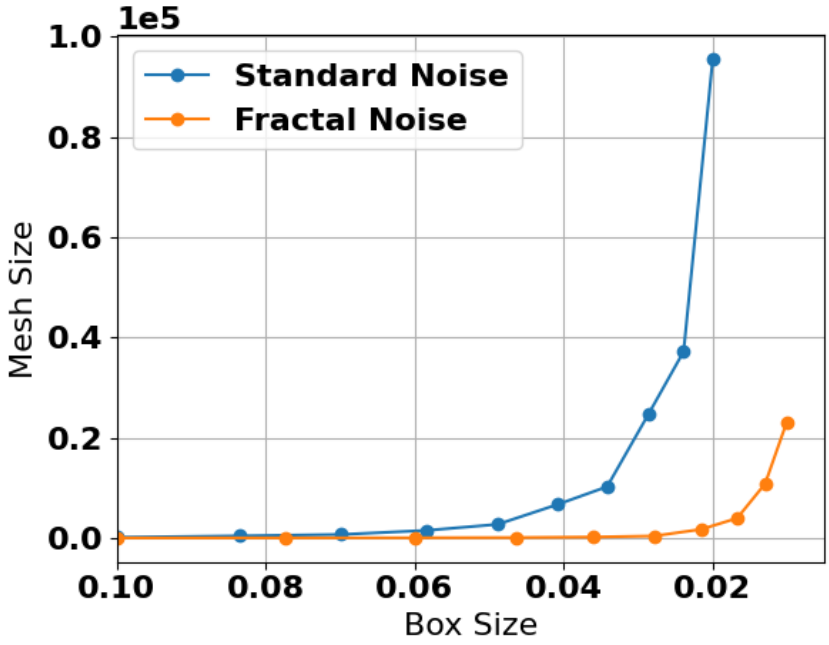


Figure 4.6: Mesh sizes for the top performing standard noise and fractal noise agents.

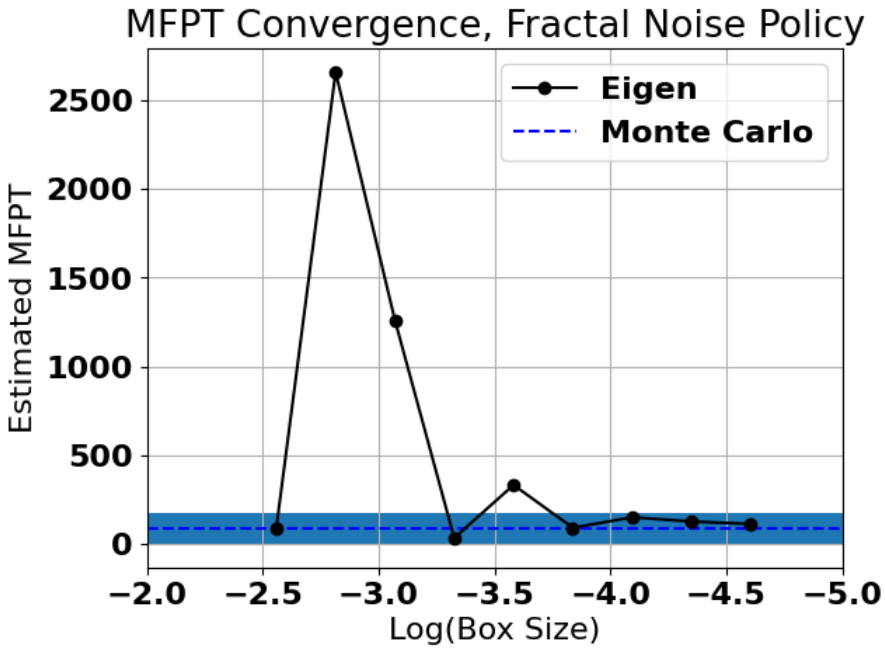


Figure 4.7: Estimated mean first passage time computed from 4.12 compared to a Monte Carlo estimate. The blue dashed line and shaded region are the mean and standard deviation of the steps to failure for 2500 Monte Carlo rollouts.

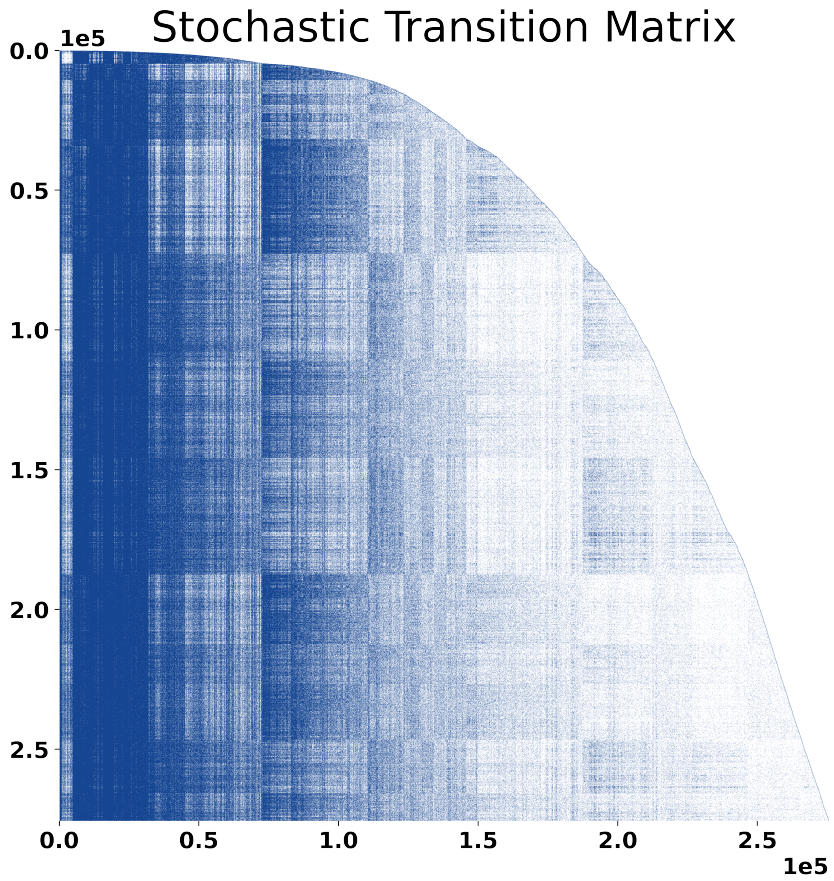


Figure 4.8: Visualization of the stochastic transition matrix for the top performing fractal noise agent. All non zero values are shown with equal size and coloration. Recall that each entry in T_{ij} tell us the probability of transitioning to state j after one step if we start in state i.

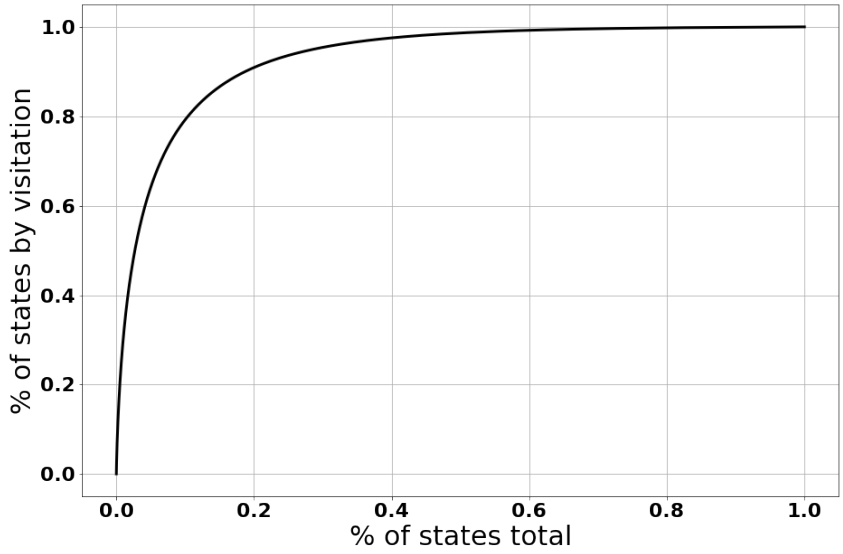


Figure 4.9: Cumulative sum of probability mass excluding the failure state. We take the sum of each column of T, and sort it in descending order, then report the cumulative sum of probability. Each point on the curve tells us that x% of states make up y% of all state transitions.

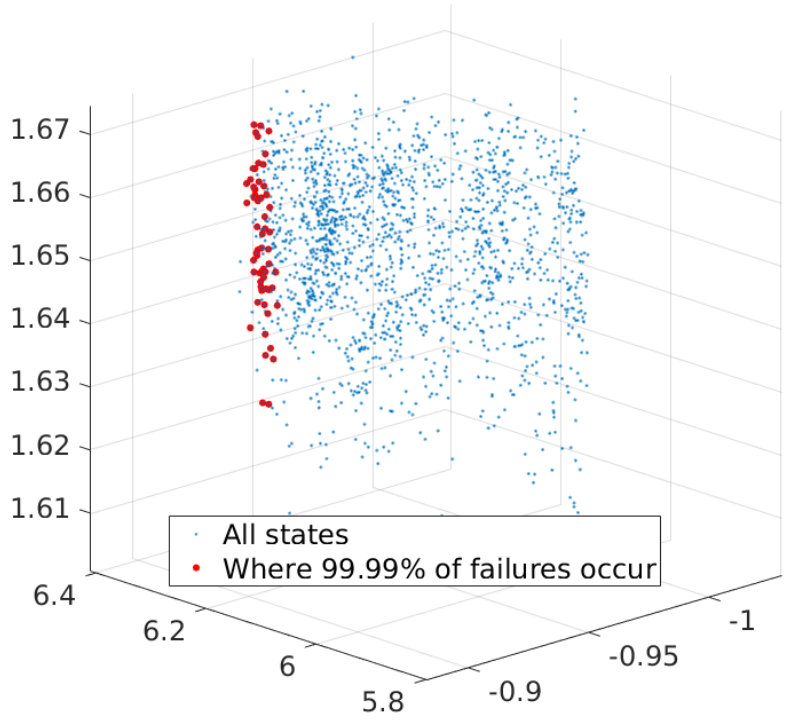


Figure 4.10: View of the first 3 principle components of the mesh for a fractal noise policy.

Appendix A

Appendix Title

A.1 Section Title

Appendicitis

Bibliography

[1] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, *Emergence of Locomotion Behaviours in Rich Environments*, *arXiv:1707.02286 [cs]* (July, 2017). arXiv: 1707.02286.

[2] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, *Learning Dexterous In-Hand Manipulation*, *arXiv:1808.00177 [cs, stat]* (Aug., 2018f). arXiv: 1808.00177.

[3] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, *Learning agile and dynamic motor skills for legged robots*, *Science Robotics* **4** (Jan., 2019) eaau5872.

[4] J. Lee, J. Hwangbo, and M. Hutter, *Robust Recovery Controller for a Quadrupedal Robot using Deep Reinforcement Learning*, *arXiv:1901.07517 [cs]* (Jan., 2019). arXiv: 1901.07517.

[5] N. Talele and K. Byl, *Mesh-based Tools to Analyze Deep Reinforcement Learning Policies for Underactuated Biped Locomotion*, arXiv:1903.1231.

[6] C. Oguz Saglam and K. Byl, *Robust Policies via Meshing for Metastable Rough Terrain Walking*, in *Robotics Science and Systems*, 2015.

[7] K. Byl and R. Tedrake, *Metastable walking machines*, *International Journal of Robotics Research* **28** (2009), no. 8 1040–1064.

[8] N. Talele and K. Byl, *Mesh-based methods for quantifying and improving robustness of a planar biped model to random push disturbances*, *Proceedings of the American Control Conference* **2019-July** (2019) 1860–1866.

[9] K. Byl, T. Strizic, and J. Pusey, *Mesh-based switching control for robust and agile dynamic gaits*, *Proceedings of the American Control Conference* (2017) 5449–5455.

[10] C. O. Saglam and K. Byl, *Meshing hybrid zero dynamics for rough terrain walking*, *Proceedings - IEEE International Conference on Robotics and Automation* **2015-June** (2015), no. June 5718–5725.

[11] H. Samet, *The design and analysis of spatial data structures.pdf*. Addison-Wesley, 1990.

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016.

[13] H. Mania, A. Guy, and B. Recht, *Simple random search of static linear policies is competitive for reinforcement learning*, *Advances in Neural Information Processing Systems* **2018-December** (2018), no. NeurIPS 1800–1809.

[14] T. Gneiting, H. Ševčíková, and D. B. Percival, *Estimators of fractal dimension: Assessing the roughness of time series and spatial data*, *Statistical Science* **27** (2012), no. 2 247–277, [arXiv:1101.1444].

[15] X. Emery, *Variograms of order ω: A tool to validate a bivariate distribution model*, *Mathematical Geology* **37** (2005), no. 2 163–181.