Using Reinforcement Learning to Shrink Fractional Dimensions for Fun and Profit

Anonymous Author(s)
Affiliation
Address
email

Abstract: !! abstract still needs a rewrite, probably go heavy on curse of dimensionality, espcially if keeping the title

In fractal geometry, a fractional dimension is one of several statistical indices which roughly quantify the way in which the complexity of a set varies with the scale at which it is examined. In prior work [1] [**?**] mesh based tools have been used to design and analyze controllers for legged locomotion. (!!) In this context, it was found that the size of the mesh required to represent a system scales with a fractal dimension, specifically the Minkowski dimension, rather than the topological dimension. In this work, we show that the Minkowski dimension of trajectories induced by reinforcement leaning agents can be influenced by post processing the reward function. We present the results of varying this dimension on the popular Mujoco locomotion environments, and perform meshing analysis on the resulting policies, to demonstrate the desired scaling qualities. In addition, we present several novel improvements to the previous meshing algorithms.

Keywords: Fractal??, Locomotion, Learning

1 Introduction

The availability of compute 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 agents 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 [2], dexterously manipulating objects with a 24 DOF robotic hand [3], and allowing a physical quadruped robot to run [4], and recover from falls [5].

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 [4], 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 it's life. Nor can we necessarily say anything about the stability or robustness properties of the system. Recent work from our lab [**?**] has used so called mesh based tools to examine precisely these questions.

Submitted to the 4th Conference on Robot Learning (CoRL 2020). Do not distribute.

!! remainder of intro needs much work The meshed based tools we use are a discretized method, similar to a tabular learning setting, and as such they suffer from the curse of dimensionality.

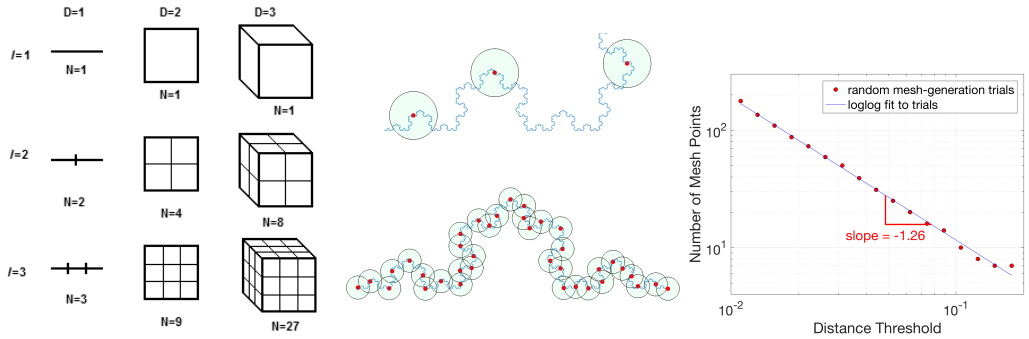
Fractal dimensionality of the system can be used to predict the resulting change in the mesh size. might need to talk about tabular learning???

How is RL able to learn anything at all in these high dimensional nonlinear systems? It clearly cannot be searching the space exhaustively. For on policy algorithms (!! need definition) we have a hypothesis that the policy must be finding a low dimensional manifold in the state space to move through. Intuitively we expect most walking gaits to have lower dimensional structure. Finding this structure is a neat trick, we suspect this is one of the reasons why trust region methods prove effective [**?**], [**?**] and as [**?**] shows, intuitive things like normalizing returns can also induce this sort of trust region behavior.

But sticking to a trust region isn't the same thing as finding a lower dimensional manifold to follow, can we encourage that explicitly? That's what we intend to do in this paper. To do this we introduce post processing into the reward function, which allows us to encourage properties of rollout trajectories as a whole, rather than individual states. In particular we examine the fractal dimensionality of trajectories, and encourage polices that find trajectories that fit into smaller fractal dimensions.

And that's what this work is all about, these meshing methods are able to work on relatively high dimensional systems because the reachable space grows like Koch's snowflake, at a rate that is much smaller than the exponential growth of the state space the system is embedded in. The meshing technique originally introduced in [1] was a way to iteratively find a tractable lower dimensional subspace that is still able to effectively control the system.

2 Meshing & Fractional Dimensions



(a) Scaling in different dimensions (b) A non uniform mesh (c) Calculating the mesh dimension

Figure 1: credit(a): [6], credit(b),(c): [**?**]

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 1(a) shows some examples of this, the line is broken into segments, the 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 1(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 1(b) and 1(c) illustrate. Figure 1(b) is an example of a (rather famous) 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.

$$D_f = -\lim_{k \to 0} \frac{\log N(k)}{\log k} \tag{1}$$

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 "fraction dimension".

As a practical matter, to calculate we use the slope of the log log plot of mesh sizes vs d, 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 [1] Saglam and Byl introduced a technique which 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 [7], 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 [8], to design agile gaits for a quadruped [9], and to analyze hybrid zero dynamics controllers [10]. There has also been recent work to use these tools to analyze policies trained by deep reinforcement learning [11]. 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.

2.1 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 of size d. We can then 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 the nonhierarchical bucket methods, which are well studied spatial data structure [12], 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.

2.2 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 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 can do things liek 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 tradeoff between accuracy and speed. Model free RL is predicated on having a huge number of rollouts to learn from, we would like for the 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 "normal" 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 conservative 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.

2.3 Variation estimators

As discussed, the computing the mesh dimension automatically is fraught with peril. But there are many different fractional dimensions. Gneiting et al [13] compare a number of these estimators, and

Algorithm 1: Create Box Mesh 2.1

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 2: Compute Box Mesh Dimension 2.2

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
Mesh Dim: fit Y = gX + b, **Return:** g
Conservative Mesh Dim: w = greatest slope in Y over X **Return:** w

submit that the variation estimators [14] offers a very good trade off between speed and robustness. To obtain this estimator, first define the power variation of order p as:

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

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

$$Dv_p(X) = 2 - \frac{\log P_p(X, 2) - \log P_p(X, 1)}{p \log 2} \tag{3}$$

The **madogram** estimator is the special case of (3) where p = 1, and the **variogram** is where p = 2.

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 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 $\pi_\theta : \mathbb{R}^n \to \mathbb{R}^m$

$$\arg\max_{\theta} \mathop{\mathbb{E}}_{\eta} \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \right] \tag{4}$$

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.

3.1 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 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:

$$R_*(\mathbf{s}, \mathbf{a}) = \frac{1}{D_*(\mathbf{s})} \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \tag{5}$$

Where $\mathbf{s}, \mathbf{a}$ are understood to be an entire trajectory of state action pairs, and D_* is some measure of fractional dimension. The variational estimators can be used here directly, we creatively call these the **madogram postprocessor** and **variogram postprocessor** respectively. The mesh dimensions require a little more care, we must first define a clipped dimension:

$$D_*^c = \text{clip}[D_*(\mathbf{s}_{t>Tr}), 1, D_t/2)] \tag{6}$$

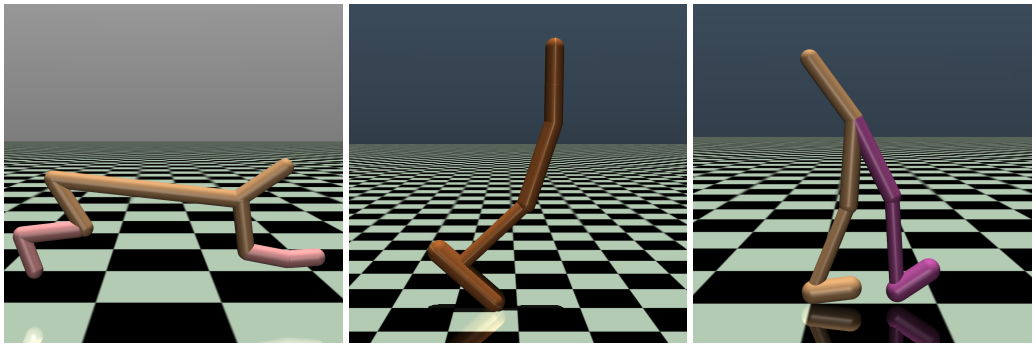
Wher D_t is the topological dimension, equal to the number of states in the system. Tr is a fixed timestep chosen to exclude the initial transients resulting from the systems moving from rest to into their gait. In this paper we set Tr = 200 for all experiments, for comparison the nominal episode length is 1000. The clipping is to ensure that the pathological trajectories that and RL agent sometimes generate 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 the systems. When the clipped mesh and conservative mesh dimension in 5, yeilds the **mesh and conservative mesh dimension postprocessors**. 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.

3.2 Environments

We examine a subset of the popular OpenAI Mujoco locomotion environments introduced in [15]. In particular, we evaluate our work on HalfCheetah-v2, Hopper-v2, and Walker2d-v2, which can be seen in figure 2. These environments were chosen because they have a relatively high dimensionality (11-17 DOF) but which we beleieve 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.

3.3 Augmented Random Search

In [16] 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



(a) HalfCheetah-v2 (b) Hopper-v2 (c) Walker-v2

Figure 2: The Mujoco locomotion environments

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.

3.4 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 [16] 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 [16]. We train on 10 each postprocessor on 10 random seeds, the evaluation metrics are averages over 5 rollouts from each seed, 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. The variational postprocessors are trained for 750 epochs each. We found that the mesh postprocessors were getting very poor performance when trained from a blank policy. However we found that we saw good results when these trials were initialized with a working policy. To that end we used the results from the variational trials as the initial policies for the mesh trials. The mesh policies were then trained for an additional 250 epochs and the results reported.

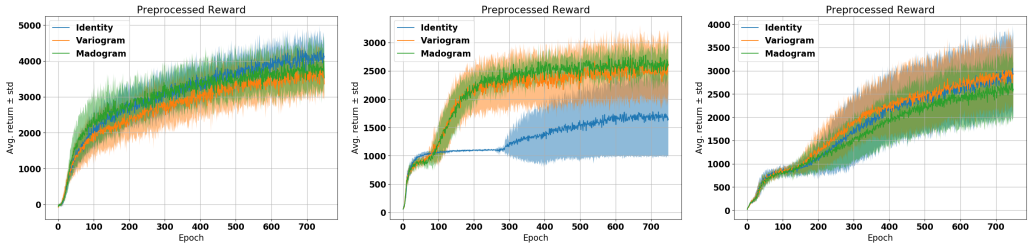
4 Results

4.1 Variational post processors

It seems that the variation postprocessors had a modest effect on the measures they were training for, and low standard deviation in the resulting polices is encouraging. These policies were very consistent in the variation dimensions they found, which may say something interesting about the underlying environment, though we're not sure what (??). We can also see that the variogram and madogram also got higher performance on the hopper task, it may be that these processors lower the sensitivity of ARS to hyper parameters, but without running many more trials and hyper parameter sweeps, that's not a claim that can be substantiated. These experiments show that 1) these measures for fractional dimension can be influenced without adversly effecting the reward, and 2) that it is possible for an agent to learn the variogram and madogram without a large impact on its mesh dimension.

4.2 Mesh dimension postprocessors

The effects from these trials are far more pronounced, for all environments the mesh post processor had a significant impact on the resulting mesh dimension, and the conservative post processor had a significant effect on the conservative mesh dimensions. Also every case there is a significant decrease in the unprocessed rewards, In the case of walker, several seeds (4 for the conservative

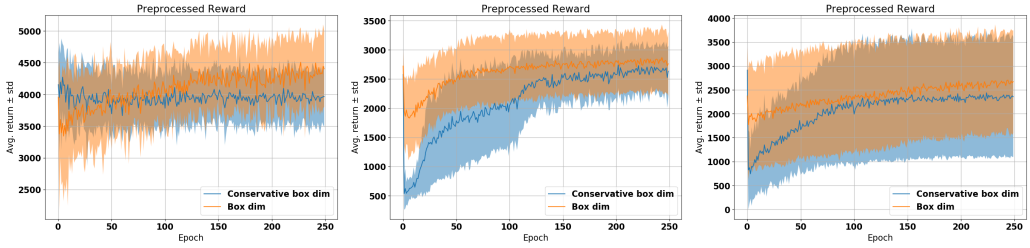


(a) HalfCheetah (b) Walker (c) Walker

Figure 3: Reward curves for the variation postprocessors

Environment	Postprocessor	Variogram	Madogram	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

Figure 4: Mesh dimensions and returns for trajectories after training. See 3.4 for details
* This includes policies which learned to "stand still", which lowers the average mesh dimension considerably see discussion



(a) HalfCheetah (b) Walker (c) Walker

Figure 5: Reward curves for mesh dimension postproccesor runs.

Environment	Postprocessor	Mesh Dim.	Cons. Mesh Dim.	Return
HalfCheetah-v2	Identity	2.31 ± 0.71	7.34 ± 1.56	5469 ± 823
	Mesh Dim	**0.66 ± 0.51**	**2.55 ± 1.52**	4962 ± 598
	Cons. Mesh Dim.	**1.06 ± 1.13**	**2.83 ± 1.27**	4432 ± 539
Hopper-v2	Madogram*	1.62 ± .27	4.68 ± 0.82	3461 ± 119
	Mesh Dim.	1.13 ± .02	3.54 ± 0.96	2941 ± 538
	Cons. 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
	Mesh Dim.	1.21 ± 0.06	4.09 ± 1.03	3339 ± 887
	Cons. 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
	Mesh Dim.	1.04 ± 0.53	4.45 ± 1.19	3034 ± 1086
	Cons. Mesh Dim.	1.48 ± 0.67	2.27 ± 0.95	2556 ± 1378

Figure 6: Mesh dimensions and returns for trajectories after training. See 3.4 for details
* Because our instance of ARS does not consistently perform well on the hopper, we instead use madodiv for the seed policies.
** See 4.2

dim., 3 for 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.

5 Analysis

Let's examine one of these policies closer, first we take a look at the Halfcheetah with the mesh dimension post processor, since it seems to have a dramatic .66 mesh dimension when measured.

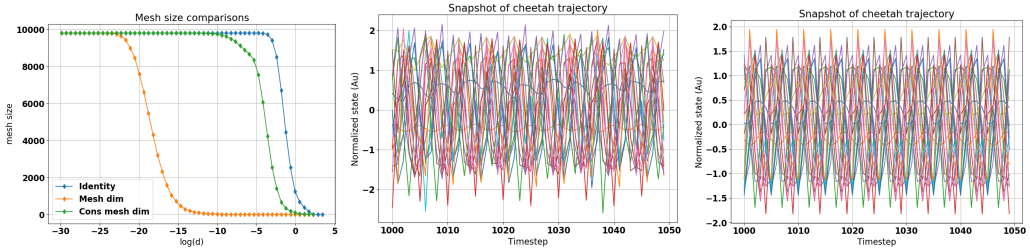


Figure 7: Left: HalfCheetah trajectory after normal ARS training, used as the inital policy for the mesh dim training. Right: trajectory after refining with the mesh dim training

Wow that's dramatic! what happened here. well, it seems that the cheetah was encouraged to refine its mesh, since doing so resulted in a smaller dimension. Shweet.

6 Future Work

!!! We've shown that it's possible to decrease the mesh size needed to analyze policies by post processing the rewards. So the next step is to use this to recreate some of the earlier work from our lab. Ideally these techniques will allow our mesh based methods to scale to even higher dimensional systems. There is also clear improvements that can be made to the meshing algorithm. and a lot more of data to dig through...

7 Conclusion

!!! In this work, we introduced a technique to influence the fractional structure of reinforcement learning policies. We show that our method does in fact result in policies with smaller fractional dimension for several test problems. We further used meshing techniques to verify that we did in fact get smaller meshes, great!

8 APPENDIX

ARS Hyper Parameters

For all environments $\alpha = .02$, $\sigma = .025$, $N = 50$, $b = 20$. This is the same notation used in [16]

Box Mesh Hyper Parameters

Our scaling parameter was chosen to be f = 1.5.

Implementation Details

Implementation matters [**?**], we have included our code, and in this section we also want to cover some implementation details that we think will be most important to those trying to replicate this work.

To get an accurate mesh, it is important to cut out the transients from your trajectory, for this work we chose an ad hoc number of 200 timesteps for transients to settle down. We chose this number after observing many trajectories generated by vanilla ARS.

When calculating the mesh dimension, we attempt to only look at the ”linear region” of the X Y plot. For large and small meshes, the slope we are trying to measure flattens out. This is why we have an upper bound for the mesh, going much higher gets into flatland. The number chosen here is ad hoc too, ultimately the best way to determine the mesh dimension is to look at the data yourself.

Finally, the results presented here are very sensitive to the choice of normalization. During training, we keep a running average μ and standard deviation σ of all states encountered so far, and the norm is naturally $\bar{s} = (s - \mu)/\sigma$. We found that if we trained with this norm, but at test time used the mean and standard deviation for the current trajectory, that the measured mesh dimensions were signifigantly different, usually much larger.

References

[1] C. Oguz Saglam and K. Byl. Robust Policies via Meshing for Metastable Rough Terrain Walking. In *Robotics Science and Systems*, 2015. doi:10.15607/rss.2014.x.049.

[2] 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. URL http://arxiv.org/abs/1707.02286. arXiv: 1707.02286.

[3] 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. URL http://arxiv.org/abs/1808.00177. arXiv: 1808.00177.

[4] 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(26): eaau5872, Jan. 2019. ISSN 2470-9476. doi:10.1126/scirobotics.aau5872. URL http://robotics.sciencemag.org/lookup/doi/10.1126/scirobotics.aau5872.

[5] J. Lee, J. Hwangbo, and M. Hutter. Robust Recovery Controller for a Quadrupedal Robot using Deep Reinforcement Learning. *arXiv:1901.07517 [cs]*, Jan. 2019. URL http://arxiv.org/abs/1901.07517. arXiv: 1901.07517.

[6] Brendan Ryan / Public domain. Fractal Dimension Example, 2020.

[7] K. Byl and R. Tedrake. Metastable walking machines. *International Journal of Robotics Research*, 28(8):1040–1064, 2009. ISSN 02783649. doi:10.1177/0278364909340446.

[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:1860–1866, 2019. ISSN 07431619. doi:10.23919/acc.2019.8815226.

[9] K. Byl, T. Strizic, and J. Pusey. Mesh-based switching control for robust and agile dynamic gaits. *Proceedings of the American Control Conference*, pages 5449–5455, 2017. ISSN 07431619. doi:10.23919/ACC.2017.7963802.

[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(June):5718–5725, 2015. ISSN 10504729. doi:10.1109/ICRA.2015.7140000.

[11] N. Talele and K. Byl. Mesh-based Tools to Analyze Deep Reinforcement Learning Policies for Underactuated Biped Locomotion. 2019. URL http://arxiv.org/abs/1903.12311.

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

[13] T. Gneiting, H. Ševčíková, and D. B. Percival. Estimators of fractal dimension: Assessing the roughness of time series and spatial data. *Statistical Science*, 27(2):247–277, 2012. ISSN 08834237. doi:10.1214/11-STS370.

[14] X. Emery. Variograms of order ω: A tool to validate a bivariate distribution model. *Mathematical Geology*, 37(2):163–181, 2005. ISSN 08828121. doi:10.1007/s11004-005-1307-4.

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

[16] 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(NeurIPS):1800–1809, 2018. ISSN 10495258.